Journal of the ACM, Vol. 41, No. 6, 1994, pp.1216–1266

Modular Stratification and Magic Sets for Datalog Programs with Negation

Kenneth A. Ross*
Columbia University
kar@cs.columbia.edu

Abstract

A class of "modularly stratified" logic programs is defined. Modular stratification generalizes stratification and local stratification, while allowing programs that are not expressible as stratified programs. For modularly stratified programs the well-founded semantics coincides with the stable model semantics, and makes every ground literal true or false. Modularly stratified programs are weakly stratified, but the converse is false. Unlike some weakly stratified programs, modularly stratified programs can be evaluated in a subgoal-at-a-time fashion. An extension of top-down methods with memoing that handles this broader class of programs is presented. A technique for rewriting a modularly stratified program for bottom-up evaluation is demonstrated, and extended to include magic-set techniques. The rewritten program, when evaluated bottom-up, gives correct answers according to the well-founded semantics, but much more efficiently than computing the complete well-founded model. A one to one correspondence between steps of the extended top-down method and steps during the bottom-up evaluation of the magic-rewritten program is exhibited, demonstrating that the complexity of the two methods is the same. Extensions of modular stratification to other operators such as set-grouping and aggregation, which have traditionally been stratified to prevent semantic difficulties, are discussed.

Categories and Subject Descriptors: D.3.1 **Programming Languages**: Formal definitions and theory — semantics. F.4.1 **Mathematical Logic and Formal Languages**: Mathematical logic — logic programming. H.2.3 **Database Management**: Languages — Query languages. H.2.4 **Database Management**: Systems — Query processing. I.2.3 **Artificial Intelligence**: Deduction and theorem proving — deduction, logic programming, nonmonotonic reasoning and belief revision.
General Terms: Algorithms, Languages, Theory.
Additional Key Words: Deductive databases, well-founded semantics, stratification, modular stratification, magic sets, rule rewriting.

*This research was performed while the author was at Stanford University, and was supported by NSF grant IRI-87-22886, a grant from IBM corporation, and by AFOSR under contract number 88-0266. A preliminary abstract of this paper appeared at the Ninth ACM Symposium on Principles of Database Systems [28].

1 Introduction

Much recent work has concerned defining the semantics of negation in deductive databases. The "perfect model semantics" [22] 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). A stratified program is one in which, effectively, there is no predicate that depends negatively on itself.

Recent work [17] has shown that there are interesting logic programs that are not stratifiable but for which a natural, unambiguous semantics exists. The well-founded semantics [35] and the stable model semantics [14] are two (closely related) proposals for defining the semantics of logic programs, whether stratified or not. For stratified programs they both coincide with the perfect model semantics.

The well-founded semantics is a three-valued semantics. Literals may be *true*, *false* or *undefined*. The stable model semantics is also a three-valued semantics in the sense that the meaning of the program is, in general, determined by a *set* of (two-valued) models rather than a single model.

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 [21] 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.

Kemp and Topor, and Seki and Itoh have proposed (similar) extensions of the QSQR/SLD

top-down query evaluation procedure of Vieille [36, 37] from Horn programs to the class of stratified programs [15, 32]. We further generalize their methods, called QSQR/SLS-resolution in [15], 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 method without memoing by the use of magic sets [33]. Several authors have considered the problem of extending the magic sets method to stratified programs.

The approach of [1, 2] 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 [3].

The approach of [7] 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 [16]. They define a property called weak stratification[1] and demonstrate that the magic rewriting of a stratified program must be weakly stratified. Weakly stratified programs are evaluated using a nested fixpoint technique that computes the fixpoint of rules from lower strata after each application of a rule from a higher stratum.

In [10], 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, and at the present time, no full version is available. Only after the present paper was written did we become aware of the details of Bry's work [9]. Although the class of constructively consistent programs is incomparable with the class of modularly stratified programs, it does also generalize the class of stratified programs, and constructively consistent programs have a two-valued semantics. Bry's method handles negative dependencies by storing revised rules rather than atoms. Rules are unfolded, leaving negative literals in the bodies, and these unfolded rules effectively control the order of rule execution.

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 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 [34] for a discussion of why bottom-up methods are likely to be preferable to top-down methods.

[1] This concept is different from Przymusinska and 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.

2 Terminology

We consider normal logic programs without function symbols[2] [18], also known as "Datalog" programs with negation.

Definition 2.1: A *term* is either a variable or a constant symbol. 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.

A *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 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 rule, 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 *program* is a finite set of rules. □

Logical variables begin with a capital letter; constants, functions, and predicates begin with a lowercase letter. The word *ground* is used as a synonym for "variable-free."

If a predicate is defined only by facts, then we say that the predicate is an *extensional database* (EDB) predicate; otherwise the predicate is an *intensional database* (IDB) predicate.

A *query* or *goal* is a conjunction of literals. We may sometimes write a query as ?-Q where Q is a conjunction of literals. When we perform resolution of a rule head with an atom appearing in a goal, we will assume the goal has been negated so that the polarity of the two literals being resolved is complementary.

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 shall also make the assumption that programs are *range restricted*, i.e., every variable occurring in the head of a rule or in a negative literal in the body also occurs in a positive literal in the body. Such programs have also been called *allowed* or *safe*.

We do allow one piece of syntactic sugar in subgoals, namely the "don't care" variable. For example, the rule

$$p(X) \leftarrow q(X, _, _)$$

is a shorthand for

$$p(X) \leftarrow q(X, Y, Z).$$

We assume that finite universe $\mathcal{U}$ is given. $\mathcal{U}$ 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. When we talk about "instantiated" atoms and rules, we mean that values from $\mathcal{U}$ are substituted for all variables in the atom or rule. The choice of $\mathcal{U}$

[2]The definition of modular stratification in Section 3 does not depend on the assumption of function-freeness, although both our QSQR/SLS procedure of Section 4.3 and the magic sets method presented in Section 5.3 may not terminate for programs with function symbols.

can be problematic, in that in some cases it can affect the semantics of the program under consideration. Such issues are beyond the scope of this paper, and the reader is referred to [35] and the references therein for further discussion. For *range restricted* programs, the choice of $\mathcal{U}$ does not affect what is true according to their well-founded semantics.

If P is a program, then we define $\mathcal{B}_P$ to be the set of atoms whose predicates appear in P and whose arguments (the number of which matches the arity of the predicate) are in $\mathcal{U}$.

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.

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.

2.1 The Well-Founded Semantics

We now give a brief presentation of the well-founded semantics; for a more complete presentation with examples see [35].

Definition 2.2: Let P be a program and let H be the set of ground instances of predicates from P with respect to $\mathcal{U}$. Let I be a consistent set of ground literals whose atoms are in H. 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. The complement of some literal in the body of r is in I.
2. Some positive literal in the body of r is in A. □

Definition 2.3: The *greatest unfounded set* of P *with respect to I*, denoted by $U_P(I)$, is the union of all sets that are unfounded with respect to I. (The "greatest unfounded set" is easily seen to be an unfounded set.) □

Definition 2.4: Transformations T_P, U_P and W_P from sets of literals to sets of literals are defined as follows.

- $p \in T_P(I)$ if and only if there is some instantiated rule r of P such that r has head p and each literal in the body of r is in I.
- $U_P(I)$ is the greatest unfounded set of P with respect to I, as in Definition 2.3.
- $W_P(I) = T_P(I) \cup \neg \cdot U_P(I)$. □

It is straightforward to show that W_P is monotonic, and so has a least fixpoint. We call this least fixpoint the well-founded (partial) model of P.[3] Note that the well-founded model is, in general, a "three-valued model." A ground atom A may appear positively, negatively or not at all in the well-founded model.

[3]For a justification that it is a partial model see [35].

2.2 Weak Stratification

We now present the concept of weak stratification from [21]. In the definitions below, the program P is assumed to be ground. In order to apply these definitions to a program with variables, one must first take the instantiation of the program with respect to the universe $\mathcal{U}$.

Definition 2.5: (Dependency graph) Let P be a ground program. The vertices of the dependency graph G_P are the atoms appearing (possibly negated) in P. The edges of G_P are directed and labelled either *positive* or *negative* or both. For every rule

$$H \leftarrow B_1, \cdots, B_n$$

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

We write $B \leq_P A$ if there is a directed path from B to A in G_P. We write $B <_P A$ if there is a directed path from B to A in G_P passing through a negative edge. □

Definition 2.6: Let $\sim_P$ be the equivalence relation between ground atoms defined as follows:

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

We shall refer to the equivalence classes induced by $\sim_P$ simply as "classes." (In [21] they are called *components*.) We say a class is *trivial* if it consists of one element, say A, and $A \not<_P A$.

Let C_1 and C_2 be classes. We define

$C_1 \prec_P C_2$ if and only if $C_1 \neq C_2$ and there exist $A_1 \in C_1$ and $A_2 \in C_2$ such that $A_1 <_P A_2$.

A class C is *minimal* if there is no class C' such that $C' \prec_P C$. Define $S(P)$ to be the union of all minimal classes with respect to $\prec_P$. Define $L(P)$ to be the set of rules from P whose heads belong to $S(P)$. □

The relation $\sim_P$ denotes either equality or mutual negative dependence. That $\prec_P$ is a partial order is shown in [21].

Definition 2.7: Let M be a partial interpretation for P, i.e., a consistent set of literals whose atoms are in $\mathcal{B}_P$. The *reduction* of P modulo M is a new program P' obtained from P by performing the following operations:

1. Delete from P all rules which contain a subgoal whose complement is in M.
2. Delete from P all rules whose head belongs to M.
3. Remove from all the remaining rules those subgoals that are members of M.
4. From the program resulting from the above operations, delete all rules with nonempty bodies whose heads appear as unit facts. □

Definition 2.8: (Weak Stratification) Let P be a program, let $P_0 = P$, and let $M_0 = \emptyset$. Suppose that $\alpha > 0$ is a countable ordinal such that programs P_δ and partial interpretations M_δ have been already defined for all $\delta < \alpha$. Let

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

and let $P_\alpha =$ the reduction of P with respect to N_α.

- If the program P_α is empty, then the construction stops and the program is weakly stratified. N_α is a two-valued[4] model of P.
- Otherwise, if the bottom stratum $S(P_\alpha)$ of P_α is empty or if it contains a nontrivial class, then the construction stops: P is not weakly stratified.
- Otherwise, the partial interpretation M_α is defined as the least model (restricted to literals whose atoms are in $S(P_\alpha)$) of the "bottom layer rules" $L(P_\alpha)$ of P_α and the construction continues. □

In [21] it is shown that if a program is weakly stratified, then the model N_α given in the construction above is the 2-valued well-founded model.

Example 2.1: Consider the following program:

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

Suppose that our universe $\mathcal{U}$ consists only of the symbols a and b; let P be the instantiation of this program with respect to $\mathcal{U}$. Then $N_1 = \emptyset$, and P_1 is the reduction of P with respect to $\emptyset$, namely

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

P_1 contains no rules with head $p(b)$, since $p(b)$ appears as a unit fact. The dependency graph for P_1 is given in Figure 1. The t atoms in the graph and $p(b)$ each form a minimal class, and so $S(P_1)$ is the union of these t atoms with $p(b)$. $L(P_1)$ consists of the three unit rules $\{t(a,b,a), t(a,a,b), p(b)\}$, whose least model M_1 is

$$\{t(a,b,a), t(a,a,b), p(b), \neg t(a,a,a), \neg t(a,b,b)\}.$$

The reduction of P with respect to $N_2 = M_1$ is empty. The two valued model thus constructed is $\{t(a,b,a), t(a,b,b), p(b)\}$.

[4]Any ground atom that appears neither positively nor negatively in N_α is assumed false.

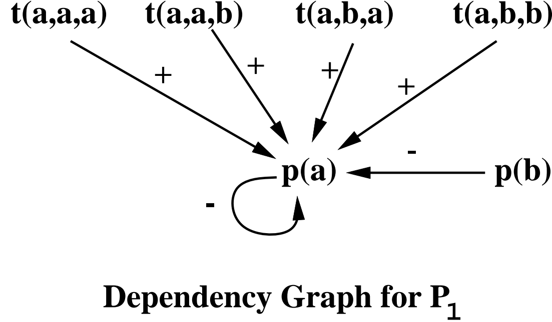

Figure 1: Dependency graph for Example 2.1.

Note that, were we to delete the fact $p(b)$, the program would not be weakly stratified. We would eventually be left with the program

$$p(a) \leftarrow \neg p(a)$$

which has no nontrivial classes. □

3 Modularly Stratified Programs

Example 3.1: 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 [17] 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 $[\neg w(Y)]$.

P is not stratified or even locally stratified. In fact Kolaitis [17] 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 shall not be able to 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, i.e., when the game cannot have repeated positions. □

Definition 3.1: We say a predicate p *depends upon* a predicate q if there is a sequence of rules $r_0, \ldots, r_{n-1}$ with predicates $p_0, \ldots, p_{n-1}$ in the head, respectively, such that

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 r_{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 $r_0, \ldots, r_{n-1}$, respectively, are negative. We say p depends *negatively* on q if p depends on q through at least one negation. A predicate p is *mutually recursive with* a predicate q if p depends upon q and q depends upon p. □

Definition 3.2: Let F be a component (i.e., a subset of the rules) of a logic 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 p are in F, and
- if p is mutually recursive with a predicate q, then all rules in P with head 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. □

If we say that predicates are mutually recursive with themselves, then mutual recursiveness 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 [34]) 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 is such a condition, and as we shall see, all modularly stratified programs are weakly stratified.

3.1 Subgoal-at-a-Time Evaluation

There is another important issue, apart from having a 2-valued semantics, namely the ordering of literals in the body of rules. As observed in [30] 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 3.2: Consider the program

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

from Example 2.1 whose well-founded semantics is total, the (2-valued) well-founded model being $\{p(b), t(a,b,a), t(a,a,b)\}$. In fact, as shown in Example 2.1, this program is weakly stratified.

Suppose we try to answer the query $?-p(a)$ top-down using a subgoal-at-a-time method such as SLDNF-resolution [18]. 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.

We consider a left-to-right ordering of subgoals.[5] 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. (The programmer or database optimizer places the "easier" subgoals to the left of the "harder" ones.) Example 3.2 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 3.3: (Reduction of a component) Let F be a program component, and let S be the set of predicates used by F. Let M be a two-valued interpretation over the universe $\mathcal{U}$ for the predicates in S.

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 Q whose predicate in S, but for which Q is false in M. From the remaining rules, delete all (both positive and negative) subgoals having predicates in S (these subgoals 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 3.3: Let F be the component

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

[5] See the end of Section 3.2 for a discussion of other sideways information passing strategies.

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

$$p(a,d) \leftarrow \neg q(d,e)$$
$$q(b,c) \leftarrow p(c,c)$$

together with all possible instantiations of the rule

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

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

This definition of reduction is similar to the definition of reduction in Section 2.2 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 rules whose heads appear elsewhere in the program as unit facts.

The significance of the second restriction is that we do not consider it sufficient that an atom p have a successful derivation if some other attempted derivation for p leads to infinite recursion through negation. We require that *all* derivations for p yield finite recursion through negation. This restriction is necessary to ensure subgoal-at-a-time evaluation.

In what follows we shall be referring to the notion of reduction from Definition 3.3 and not that of Section 2.2. We are now in a position to describe the class of programs we allow.

Definition 3.4: (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 3.4: Let P be the program consisting of the component F from Example 3.3, 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 3.3 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 transformation of Gelfond and Lifschitz [14]. 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 3.2 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 3.1 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 [21].

Theorem 3.1: 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 [21], we can show by induction on the level of components that modularly stratified programs are weakly stratified. ▮

The converse of Theorem 3.1 is false, as illustrated by Example 3.2. Also, every stratified program is modularly stratified, as is every locally stratified program.

Corollary 3.2: 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 [21]. ▮

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 [22] and hence a total well-founded model that coincides with the perfect model. The "lowest" components must be locally stratified; compute their perfect model M. The next lowest components are locally stratified when reduced modulo M; compute the perfect 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 3.1 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 [8], 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.

Example 3.5: Consider the following program component F, inspired by the events of 1890–1893; it is designed to identify even-numbered presidents of the United States.[6] (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.)

$$even_president(X) \leftarrow predecessor(X, Y), \neg even_president(Y)$$

[6] Note that this program is just a re-interpretation of the symbols in Example 3.1 above.

$predecessor(X,Y)$ 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 $even_president(Y) <_F even_president(X)$ 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 arise. $R_M(F)$ 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.[7] (Despite the loop, *even_president(cleveland)* is independently derivable.) However, if Cleveland's first presidency were odd-numbered, then the well-founded 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 subsequent presidents 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. $\square$

Example 3.6: 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. We adopt the convention that a mechanism is not a component of itself — we are only interested in smaller, simpler components. 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

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

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 any ordering $<_F$ satisfying

$$working(X) <_F has_suspect_part(X)$$

for all X, and

$$has_suspect_part(X) <_F working(Y)$$

when $part(X, Y)$ holds. □

3.2 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 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 stratifiable" are passed to the negative subgoals whose predicates belong to F.

To see what can go wrong, consider the game program from Example 3.1 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 non-memoing 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 apply the notion of local stratifiability to the reduction of the *prefixes* of the rules, to take account of the left-to-right order of evaluation.

Definition 3.5: 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 3.6: (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 3.7: 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. This program 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 value. 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 3.3: 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 subgoals 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 prefixes 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 3.3. On the other hand, rearranging the subgoals as above may cause a left-to-right evaluation method to try to evaluate a negative subgoal containing unbound variables. An evaluation method is said to *flounder* if it selects for expansion a negative literal with one or more unbound variables. There are several semantic difficulties associated with floundering; the interested reader is referred to [18] for further discussion.

Example 3.8: 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 is 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 3.7: A program P is *permissible* if it is range-restricted, modularly stratified from left to right, and 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 [11, 24] into the evaluation mechanisms proposed in this paper. 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 [6]. Alternative strategies can be put into our framework by applying Definition 3.6 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 paper.

4 Top-Down Evaluation

4.1 Global SLS-Resolution

In this section we present a variant of the top-down method called "global SLS-resolution" from [30]. A similar method was independently proposed in [23]. The version presented here is slightly different from the one appearing in [30]. In this paper 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 methods 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.

Recall that 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 context where we can expand negative literals ahead of positive ones.

Definition 4.1: (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 rules whose heads unify with L. The children of Q are obtained by resolving Q with (a variant of) each of the rules in U_L over the literal L using most general unifiers. If U_L is empty, then Q 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, Q 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. $\Box$

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 4.1: Consider the game program from Example 3.1 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 4.2: 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 (step 3 of Definition 4.1), A is a child of H. □

Example 4.2: For the program in Example 4.1, $N_{w(a)}$ 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 [30]. We shall show that for the class of programs that we are considering in this paper, this simplified version is both sound and complete.[8]

Definition 4.3: (Global SLS-resolution) Global SLS-resolution is the top-down process of finding all answer substitutions for a goal G by constructing the SLP-tree for G as in Definition 4.1. □

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.

[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 4.3.

Theorem 4.1: Global SLS-resolution (as defined in Definition 4.3) is both sound and complete for non-floundering programs with finite-depth negation trees.
Proof: The proof follows from the corresponding results in [30]. 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 [30], 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 4.2: 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 4.3 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 infinitely many 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.)

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 4.1, 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 4.2 does not hold for programs having function symbols, as illustrated by the program

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

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. The status of an SLP-tree as successful, failed or indeterminate depends only upon the *leaves* of the tree. A branch without leaves will not affect this status. 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.

4.2 Towards Termination

We now show how one can modify Global SLS-resolution to ensure termination for permissible programs. This modification preserves soundness and completeness, and motivates the memoing method to be introduced in Section 4.3.

Suppose that for all possible atoms, we have simultaneously constructed their SLP-trees as in Section 4.1. There are finitely many such trees if we disallow variants of the same atom.

Suppose that the goal G given by

$$L_1, L_2, \cdots, L_n$$

is a child of the root node R in one of these SLP-trees. Suppose that L_1 is positive. It seems that the SLP-tree for L_1 is replicated within the SLP-tree for R, with appropriate instantiations of $L_2, \cdots, L_n$ appended to all such goals. (This follows because it is always the leftmost literal that is selected.)

Rather than repeating the expansion of L_1, let us consult the SLP-tree for L_1 to find all answer substitutions, and use those substitutions directly, so that the children of G will have various instances of $L_2, \cdots, L_n$ as children. If L_2 is positive, then we may do the same for those instances of L_2, and so on, thus minimizing repeated computation.

To be precise, once we perform the modifications outlined above, we are no longer talking about "SLP-trees." In Section 4.3 we shall define the modified trees more precisely as "derivation trees." For now we ignore this distinction and refer to the transformed trees also as SLP-trees.

Lemma 4.3: For permissible programs, the transformation of SLP-trees outlined above yields a method that remains sound and complete with respect to the well-founded semantics.
Proof: The main observation is that for permissible programs, success and failure are the only possible results for a derivation; indeterminate derivations do not exist. The proof is by induction. We say an SLP- tree is *correct* if and only if the answer substitutions subsume exactly those that are correct with respect to the well-founded semantics. Our induction hypothesis states that after n transformations, all SLP-trees are correct.

The base case follows from Theorem 4.1.

We inductively replace a redundant copy of an SLP-tree for the leftmost literal L_1 (if positive) within a larger SLP-tree T by a simple lookup of the answer substitutions in the separate tree for L_1. The separate SLP-tree for L_1 is correct by the induction hypothesis. (The "separate" copy may be T itself if the tree contains itself as a subtree.) Since we have obtained exactly those substitutions for L_1 that are correct with respect to the well-founded semantics, and since all substitutions not subsumed by these yield failed branches, the resulting SLP-tree remains correct. ▮

The first benefit of performing this rudimentary form of memoing is that SLP-trees are now finite. Infinite positive derivations do not occur, since there is at most one expansion for each positive literal. For example, if the subgoal $p(X)$ appears leftmost in a node of the SLP-tree for (a variant of) the same goal, say $p(X')$, then the subgoal $p(X)$ is not recursively expanded; substitutions for X' generated by other branches of the tree may be used as bindings for X and the derivation may proceed.

Given these observations, there are several ways that we could make such a procedure more efficient:

1. Construct trees only when needed.
2. Omit any tree whose root is an instance of a previously constructed tree.
3. Expand nodes of the tree incrementally, using either newly generated answer substitutions, or newly generated nodes of a tree.

These optimizations are included in the algorithm of Section 4.3 below.

In the discussion above we assumed that all SLP-trees had been simultaneously constructed. When computing the corresponding derivation trees, we must be careful to maintain negative dependencies properly.

Suppose that, at some intermediate stage of the computation when all trees have not yet been fully constructed, we select a negative literal L_1. We look at the SLP-tree for the complement of L_1, let us say L. If L succeeds, i.e., the tree for L has a successful leaf, then G has no children. However, we can conclude that L fails (and hence L_1 succeeds) only if both

- The SLP-tree for L has no success leaves, and
- The SLP-tree for L, and all subsidiary SLP-trees have been fully expanded.

The second condition above is needed because it may be possible that a partially expanded SLP-tree may develop a successful leaf at a later stage of the computation. By subsidiary SLP-trees, we mean both the recursively constructed trees for negative subgoals, and the SLP-trees examined as above for positive subgoals.

An SLP-tree for a goal R is "fully expanded" when

- All possible resolution steps for positive subgoals have been performed, and
- All children of R in the negation tree for R are themselves fully expanded.

The concept of the negation tree is the same as before. Since we have memoed parts of the expansion, we may have to trace through several trees, rather than looking at branches of a single SLP-tree, in order to identify the structure of the negation tree. These conditions are analogous to the part of Definition 4.1 where we recursively construct the SLP-tree for the complement of a leftmost negative literal. This recursive construction must be complete in order to apply the definition.

4.3 Top-Down with Memoing

Global SLS-resolution may get lost down an infinite positive branch and hence not terminate. Several authors [13, 36, 37] 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 [15] and, independently, Seki and Itoh [32] have generalized these proposals to the class of stratified programs. Our method applies to a larger class of programs, namely those programs that have finite negation trees, and still retains soundness, completeness and termination properties. In particular, our method applies to permissible programs.

Kemp and Topor called their method the "QSQR/SLS-procedure." We now present our extension, which we also call the QSQR/SLS-procedure. While our method is similar to [15], there are several differences that we shall explain after giving the definitions. Our version of the QSQR/SLS procedure, given in Algorithm 4.1, is presented in such a way that the comparison with bottom-up evaluation will be clearer later on.

Our method is motivated my the memoing technique of Section 4.2. 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 [15], the set of "r-predicates" is the set of all IDB predicates. Following [15], we shall refer to literals that have been derived at an intermediate stage of the computation as *lemmas.*

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.

Before we define the concept of derivation trees, we introduce the concept of a *restriction* of a substitution.

Definition 4.4: Let r_j be a rule, and let i be the number (counting from left to right) of a subgoal in the body of r_j. Let θ be a substitution. We say ϕ is the *restriction of* θ *to* r_j *at subgoal* i 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 the paper. For example, consider the program

$$\begin{aligned}&p(X) \leftarrow q(X,Y), r(X,Z)\\&q(c,a)\\&q(c,b)\end{aligned}$$

In a corresponding derivation tree there will be two branches leading to the subgoal $r(X,Z)$, one with $\{X|c,\ Y|a\}$ and the other with $\{X|c,\ Y|b\}$. We would prefer to expand only one of these branches, since it seems that otherwise work would be repeated. To ensure that we notice the similarity of the two branches above, we project the computed substitution onto the set of relevant variables. Since Y appears neither in the head, nor in the subgoal

$r(X, Z)$, it is not relevant. Once the substitutions are projected onto the variable X, the similarity of the two branches above becomes apparent.

Definition 4.5: 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. The children of the root node, which we call *level-one* nodes, are formed by resolving the root node with the head of a rule r from the program; that rule becomes the associated rule for the corresponding level-one node. Descendants of level-one nodes have the same associated rule as their level-one ancestor. The associated substitution of a level-one node G is the restriction to r at subgoal 1 of the most general unifier used in resolving the root with the corresponding rule.

Non-root goals G contain (partially instantiated) subgoals from the rule r associated with G. Children G' of a non-root goal G are formed by resolving the leftmost literal in G with lemmas, yielding child goals (with one less literal) that have been instantiated according to the most general unifier θ used in the corresponding resolution. Suppose that the leftmost literal in G' was originally the ith subgoal in r. If G has associated substitution ϕ, then the associated substitution at G' is the restriction of $\phi\theta$ to r at subgoal i.

A *derivation forest* is a collection of derivation trees. □

In a derivation tree it is not necessary for all children of a node to have been constructed. Thus a derivation tree corresponds to a possibly partial expansion according to the rules above. Also, Definition 4.5 depends on the set of lemmas. In our context, the set of lemmas will be increasing as Algorithm 4.1 below proceeds, and so the set of legal derivation trees will grow.

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 descendant 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. It may happen that an atom P depends (positively or negatively) on an atom Q even if Q is not in the derivation tree with root P, since the root of the derivation tree with Q may be an atom that is more general than P.

Definition 4.6: 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 goal G in F with associated rule r_j, leftmost literal R, associated substitution θ, and $P = P'\theta$, where P' is the head of r_j; or

- For some S, P depends positively on S, there is some goal G in F with associated rule r_j, leftmost literal R', associated substitution θ, S and $S'\theta$ unify with most general unifier ϕ, and $R = R'\phi$, where S' is the head of r_j.

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 goal G in F with associated rule r_j, that has leftmost literal $\neg Q$, associated substitution θ, and $P = P'\theta$, where P' is the head of r_j; or

- For some S, P depends positively on S, there is some goal G in F with associated rule r_j, that has leftmost literal $\neg Q$, associated substitution θ, and S and $S'\theta$ are unifiable, where S' is the head of r_j.

We say P is *settled* if some atom depends negatively on P, and 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 4.6.

We now define the QSQR/SLS procedure.

Algorithm 4.1: The algorithm of Figures 2 and 3 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. A goal, lemma or root atom is *new* at a given step of the computation if it did not exist when that step was last executed. (Everything is new on the first iteration.) $C_{j,i}$ denotes the set of goals that have associated rule r_j, and whose leftmost literal is (an instance of) the ith subgoal of r_j. □

Example 4.3: 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

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 {

Perform inner loop steps 1 to 5

} **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, 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.

Figure 2: Algorithm 4.1: The QSQR/SLS procedure's outer loop.

$Z < X$. (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 $?\text{–}p(18)$. The forest constructed by the QSQR/SLS method is given in Figure 4. 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 $p(3)$ and $p(2)$ of positive polarity are not expanded because the expansion has been done in the corresponding trees of negative polarity (this check occurs in step 1a). The $\bullet$ symbol indicates an empty goal node. $\square$

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. This effectively prunes infinite branches.

Steps 1 to 4 of the inner loop correspond directly to an incremental expansion of derivation trees in a way analogous to the expansion of SLP-trees.

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.

The "depends negatively" relationship of Algorithm 4.1 represents a partial expansion of the negation tree for the query. A depends negatively on B at some stage of the computation only if B is a child of A in the negation tree for the query, and the status of B is unresolved at that point.

Consider the state of Algorithm 4.1 after it terminates. (We shall prove termination in Section 4.4.) At that point an atom A may depend negatively on another B only if B is a child of A in the negation tree for A. If the negation tree is finite, the atoms at the leaves of the negation tree must be settled, since they do not depend negatively on anything. Hence the parents of the leaves must also be settled. We may proceed inductively up the

1a: For each new root atom R 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 to get a collection of goals G_j for each $r_j \in S$. Insert each G_j as a child of R, as long as the associated substitution at G_j would not be subsumed by a substitution at any other node previously in $C_{j,1}$. }

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 new empty goal G do {

> Let θ be the associated substitution of G. Suppose the associated rule of G is r_j, with head Q. Add the lemma $Q\theta$ to M, 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 new nonempty non-root goal G do {

> Let L be the leftmost literal in G. 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 new nonempty non-root goal G do {

> Let L be the leftmost literal in G. 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, to get a set of new goals G_k. Add as children of G those goals G_k whose associated substitution would not be subsumed by a substitution at any other node previously in $C_{j,i}$. }

4b: For each new lemma L 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 to get new goals $G'_1, \ldots, G'_n$ respectively. Insert each G'_k as a child of G_k if the associated substitution at G'_k would not be subsumed by a substitution at any other node previously in C_{j_n,i_n}. }

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.

Figure 3: Algorithm 4.1: The QSQR/SLS procedure's inner loop.

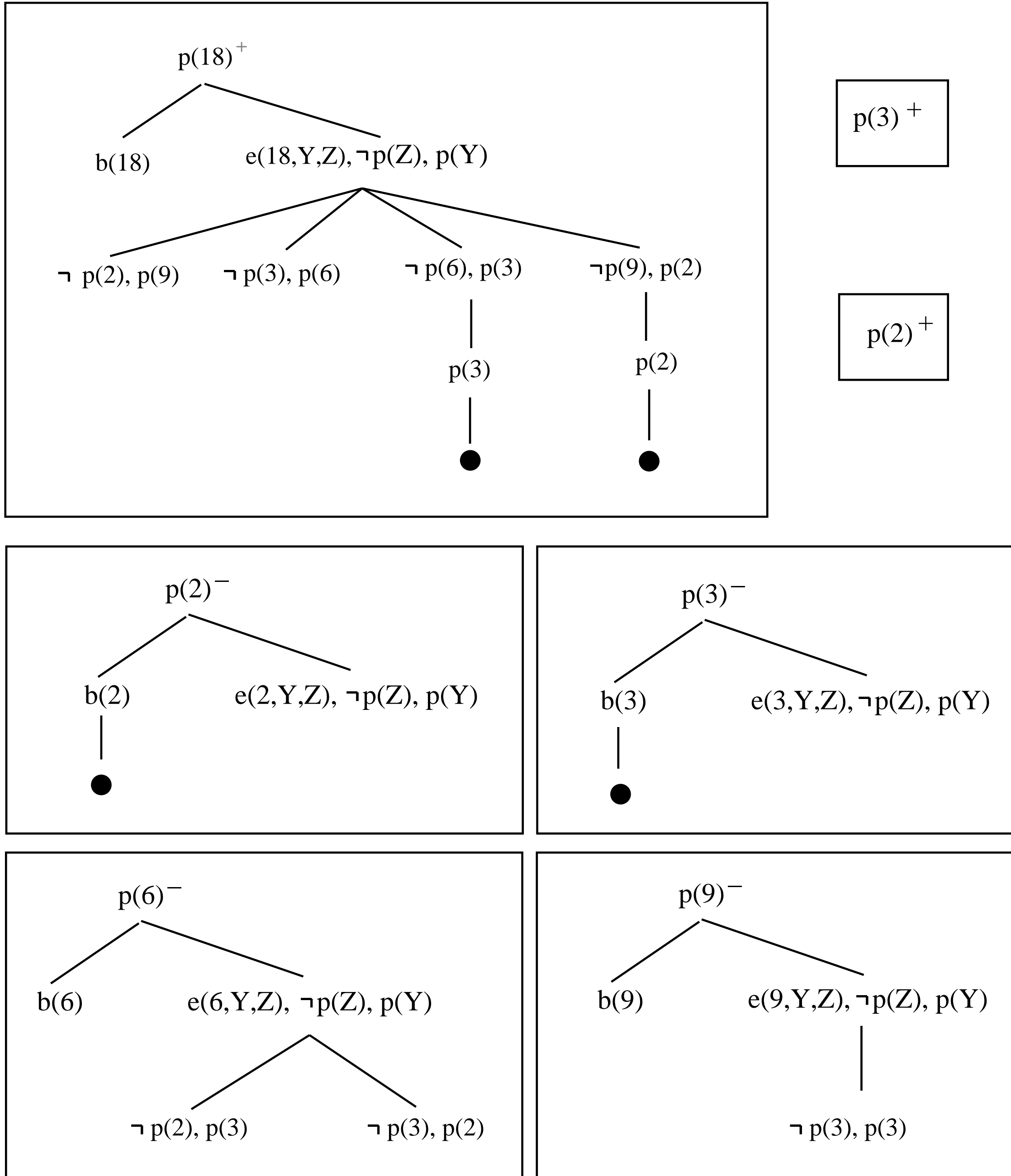

Figure 4: Forest generated in Example 4.3.

negation-tree for the (atom of) the initial query, showing that all such atoms are settled, and hence either true or false. Thus there are no "depends negatively" facts remaining after Algorithm 4.1 is run on a program with finite negation trees.

We now explain the major differences between Algorithm 4.1 and the algorithms presented in [15] and [32].

The first difference is in how the lemmas are established. In [15], "proof segments" of SLD-type trees are employed rather than derivation trees. In [32] 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 4.1 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" for deciding when to create a subsidiary tree for the leftmost atom in a goal. The less redundant of the two compares the selected atom with all previously visited atoms *in the current tree only.* 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 step 3 of Algorithm 4.1.

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 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 [15] and [32], 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 [15] and other similar work on top-down methods with memoing [31, 32] 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 since the root atom may contain some irrelevant variables.

4.4 Correctness of the Memoing Method

Lemma 4.4: For all (nonfloundering) function-free programs, Algorithm 4.1 terminates.
Proof: Since there are finitely many possible distinct lemmas, rules and roots of derivation trees. ▮

Lemma 4.5: For Datalog programs without negation, Algorithm 4.1 is correct with respect to the well-founded semantics.
Proof: See the proof of Theorem 4.6 below. ▮

Lemma 4.5 is a special case of a result of Vieille [37]. Vieille shows that top-down with memoing (with a restricted, but not necessarily left-to-right order of expansion of subgoals) is correct with respect to the least Herbrand model. For programs without negation, the least Herbrand model is the two-valued well-founded model [35].

Theorem 4.6: (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 4.1 terminates such that for every ground substitution θ for L:

$L\theta$ belongs to the (total) well-founded model for P if and only if $L\theta$ is output.

Proof: By analogy with Global SLS-resolution, using Lemma 4.3. The lemmas computed are exactly the answer substitutions applied to the root atom in the corresponding trees. Algorithm 4.1 performs the following optimizations in addition to the rudimentary memoing technique of Section 4.2:

1. If there are different branches leading to the same substitution for relevant variables, then only one of them is further expanded.
2. A derivation tree whose root is an instance of a previously constructed tree is omitted.
3. Nodes of the tree are generated incrementally, using either newly generated answer substitutions (lemmas), or newly generated nodes of a tree.

We claim that these transformations preserve correctness. Item 1 preserves correctness since this step just removes branches that are effectively duplicates.

Item 2 preserves correctness, since if an atom A has a derivation tree T, then any atom more general than A will have a derivation tree that includes (a more general version of) T, and hence all answer substitutions for A will be subsumed by answer substitutions in the more general tree.

Item 3 states that we restrict steps of the computation in such a way that they operate on at least one item of data that was not seen on the previous iteration of that step. If no data items are new, then nothing is lost by omitting the step since the step must have been performed on a previous iteration. ▮

4.5 Comparison with SLDNF-Resolution

SLDNF-resolution is a top-down resolution-based method that does not perform any memoing [18]. Hence it will be informative to compare our memoing method with SLDNF-resolution.

In Example 4.3 we have seen how QSQR/SLS-resolution would answer the query $?{-}p(18)$, i.e., "Is 18 the product of an odd number of primes?" The final derivation trees are given in Figure 4. SLDNF-resolution would still give the correct answer, but would repeat a lot of computation. In particular, the subgoal $p(3)$ would be expanded four times and the subgoal $p(2)$ three times, assuming that all branches are searched. By comparison, QSQR/SLS-resolution expands each of these atoms once.

There are (stratified) programs for which SLDNF-resolution is not complete with respect to the well-founded semantics. Such programs usually involve infinite positive recursion, as in the complement of the transitive closure of a relation with cycles, for example. QSQR/SLS-resolution is sound and complete with respect to the well-founded semantics for all permissible programs.

SLDNF-resolution is not guaranteed to terminate. In particular, it may get into an infinite loop with recursive programs. QSQR/SLS-resolution always terminates for function-free programs.

There are some situations where SLDNF-resolution is *more efficient* than QSQR/SLS-resolution. The following example is taken from [29].

Example 4.4: Let P be

$$\begin{array}{l} p(X,Z) \leftarrow e(X,Y), p(Y,Z) \\ p(n,X) \leftarrow t(X) \\ e(1,2) \\ \vdots \\ e(n-1,n) \\ t(1) \\ \vdots \\ t(m) \end{array}$$

SLDNF-resolution would construct the following 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 $\Theta(m+n)$. By contrast, QSQR/SLS-resolution would memo all the intermediate subgoals $p(i, X)$ for $i = 2, \ldots n$, computing m lemmas for each. Hence QSQR/SLS-resolution computes $\Theta(mn)$ tuples, and is less efficient to SLDNF-resolution on this example. □

In [29], improvements to the magic sets method are given that make magic-sets competitive on examples like this. As we shall see later, there is a very close relationship between QSQR/SLS-resolution and magic sets, and improvements corresponding to those suggested in [29] could be applied to QSQR/SLS-resolution.

5 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 [34] for further discussion on the relative merits of top-down versus bottom-up methods.

5.1 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 [34]. 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 we must perform something more sophisticated than duplicate elimination. (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.)

Suppose we have a set of (possibly nonground) tuples S, and wish to add some tuples from a set T to S. 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. In contrast, the result of "incremental" duplicate elimination, i.e., successively adding elements of T to S as long as they are not subsumed by the current set of tuples, may depend on the order in which tuples are taken from T.

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 $r(S, T)$.

5.2 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 evaluation 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 [5]. One of these is magic sets [4, 6, 25], 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 [33].

In Section 5.3 we describe a magic set method for evaluating modularly stratified programs bottom-up. In particular, this method also works for stratified programs. Our method extends the magic templates method of [25] rather than the original magic sets proposals.

Before we introduce the magic set 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. Meta-variables, usually denoted by Q and R, that may unify with any atom.

2. The *depends* predicate (which we abbreviate to d), that maintains a record of which atoms depend negatively on which other atoms.

3. 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 $d'(p,q)$.

4. The *currently-depends* predicate (which we abbreviate to dd), which is precisely the relation defined by the difference $d - d'$.

5. An extra modal operator $\Box$. $\Box p(a)$ holds if and only if, at the most recent fixpoint, neither $p(a)$ nor any atom of the form $dd(p(a), X)$ had been deduced.

Definition 5.1: (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 $\Box q(\vec{X})$.

- 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.

Definition 5.2: (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
   $$d(p(\vec{X}), Q) \quad \leftarrow \quad q_1(\vec{X}_1), \ldots, q_{i-1}(\vec{X}_{i-1}), dd(q_i(\vec{X}_i), Q)$$
   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 5.1: Consider the program from Example 4.3 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.

$p(X)$	$\Box p(X)$	$d(p(X), p(Z))$	$d'(p(X), p(Z))$
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: Order of inference. Each horizontal line corresponds to a fixpoint.

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. $\Box$

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 5.3 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 in Section 5.3.

Note that *WFR(P)* may itself be not range restricted (although it must be non-floundering). For example, if P contains the rule

$$p(X, Y, Z) \leftarrow t(X, Y), u(Y, Z)$$

then *WFR(P)* contains the rule

$$d(p(X,Y,Z),Q) \leftarrow d(t(X,Y),Q)$$

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 5.1: (Confluence) 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 5.2: 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 5.3: 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: See Appendix A. ∎

To see where this method breaks down when the program does not have a finite negation tree, consider the weakly stratified program from Example 3.2. The first rule yields the rewritten rules

$$\begin{array}{l} p(X) \leftarrow t(X,Y,Z), \Box p(Y), \Box p(Z) \\ d(p(X),p(Y)) \leftarrow t(X,Y,Z) \\ d(p(X),p(Z)) \leftarrow t(X,Y,Z), \Box p(Y) \end{array}$$

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 3.7. While the program of Example 3.7 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 3.2.

5.3 Magic Sets

Recall how QSQR/SLS-resolution answered the query $?\!-\!p(18)$ from Example 4.3. 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 or $\neg p$ holds for *all* numbers in the range $1, \ldots, n$.

We shall present a magic set rewriting 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 [25] that is also described in [34].

The magic set method presented in the preliminary version of this paper [28] 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 this section we introduce a magic sets method 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 name for each IDB predicate 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. 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 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 a fixpoint. 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 [12].

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 [34].

Definition 5.3: (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 4.3 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. $\Box P \leftarrow magic(P, -), \forall Q(dn(P, Q) \Rightarrow dn'(Q)), \sim P.$

□

Strictly-speaking, the formula $\forall Q(dn(P, Q) \Rightarrow dn'(Q))$ is not a literal, and hence is not a subgoal in our language. Nevertheless, we do admit this subgoal here. The symbol $\Rightarrow$ is embedded implication: the formula holds for P if whenever P depends negatively on Q, the truth value of Q has been settled. There is no semantic difficulty in evaluating this formula in our context, and so the only objection to using universal quantification would be on the basis of complexity. We shall describe suitable data structures for evaluating this subgoal below.

If one is uncomfortable with variables as atoms, then the rules 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 involving IDB predicates that appear negatively in the program.

Algorithm 5.1: (Evaluation of *SMR(P)*.) The rules from Definition 5.3 above are evaluated according to the following procedure:

```
P0;
iterate {  iterate{ P1 };
           P2; }
```

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 5.2: We rewrite the program of Example 4.3 using the supplementary magic rewriting as follows.

P_0 contains

$$magic(p(18), +)$$

P_1 contains

$$sup_{1.0}(X) \leftarrow magic(p(X), _)$$
$$sup_{2.0}(X) \leftarrow magic(p(X), _)$$

$$p(X) \leftarrow sup_{1.1}(X)$$
$$p(X) \leftarrow sup_{2.3}(X)$$

$$magic(p(Z), -) \leftarrow sup_{2.1}(X, Y, Z)$$
$$magic(p(Y), +) \leftarrow sup_{2.2}(X, Y)$$

$$sup_{1.1}(X) \leftarrow sup_{1.0}(X), b(X)$$
$$sup_{2.1}(X, Y, Z) \leftarrow sup_{2.0}(X), e(X, Y, Z)$$
$$sup_{2.2}(X, Y) \leftarrow sup_{2.1}(X, Y, Z), \Box p(Z)$$
$$sup_{2.3}(X) \leftarrow sup_{2.2}(X, Y), p(Y)$$

$$dp(p(X), p(Y)) \leftarrow magic(p(X), -), sup_{2.2}(X, Y)$$
$$dp(P, p(Y)) \leftarrow dp(P, p(X)), sup_{2.2}(X, Y)$$

$$dn(p(X), p(Z)) \leftarrow magic(p(X), -), sup_{2.1}(X, Y, Z)$$
$$dn(P, p(Z)) \leftarrow dp(P, p(X)), sup_{2.1}(X, Y, Z)$$

$$dn'(Q) \leftarrow magic(Q, -), Q$$
$$dn'(Q) \leftarrow magic(Q, -), \Box Q$$

P_2 contains

$$\Box P \leftarrow magic(P, -), \forall Q(dn(P, Q) \Rightarrow dn'(Q)), \sim P.$$

Consider how Algorithm 5.1 would proceed. The rule in P_0 will give the tuple $magic(p(18), +)$. P_1 would then be evaluated to a fixpoint; omitting the supplementary relations, the tuples generated at this stage would be

$$magic(p(9), -),\ magic(p(6), -),\ magic(p(3), -),\ magic(p(2), -),\ p(2),\ p(3),$$
$$dn(p(9), p(3)),\ dn(p(6), p(3)),\ dn(p(6), p(2)),\ dn'(p(3)),\ dn'(p(2)).$$

Then P_2 will be executed, giving $\Box p(9)$ and $\Box p(6)$. P_1 will be executed once more, giving

$$magic(p(3), +),\ magic(p(2), +),\ p(18)$$

and nothing new will be generated after this step. □

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 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 [34], we can further optimize the rewriting in several ways. For example, since 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 predicate should be renamed apart if they have different binding patterns; also, subgoal rectification is necessary for subgoals with repeated variables. See [34] for details of how renaming and 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 [26].

One may ask whether there is significant overhead involved in using the magic sets method presented here rather than standard magic sets for programs *without* negation. One can verify that for programs without negation, Algorithm 5.1 produces no tuples of the form $magic(P, -)$. As a result, there are rules (namely those for *dp*, dn' and $\Box P$) that never fire. If one was to delete those rules, the remaining rules constitute rules isomorphic to the standard magic templates transformed rules [25]. Thus the overhead in Algorithm 5.1 consists entirely of repeatedly attempting to join an empty relation with another relation in the body of the rules mentioned above. Ideally, a compiler would notice that a program does not contain negation and would hence delete these redundant rules or, equivalently, use the standard magic templates method.

We could demonstrate the correctness of Algorithm 5.1 by comparison with the methods of the previous section. However we shall show a closer connection between Algorithm 5.1 and Algorithm 4.1 that identifies a one-to-one correspondence between steps of each method. In addition to demonstrating the correctness of Algorithm 5.1, the correspondence also shows that the top-down and bottom-up methods will have the same complexity.

6 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 4.1 and the semi-naive evaluation of Algorithm 5.1. 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" [25, 31]. Informally, a bottom-up method is *sip-optimal* if it infers no more facts about any predicate from the original program that a top-down method would. 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 6.1: 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 4.1.

In the evaluation of Algorithm 5.1, 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 5.3. 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 6.1: (Correspondence theorem) Let $n = (\alpha, \beta, \gamma)$ be an arbitrary stage later that ϵ, 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: See Appendix A ▮

Corollary 6.2: Algorithm 5.1 is correct with respect to the well-founded semantics for range-restricted nonfloundering programs having finite negation trees.
Proof: By Theorem 6.1 and Theorem 4.6. ▮

In particular, Corollary 6.2 holds for all permissible programs.

Corollary 6.3: The semi-naive bottom-up computation of *SMR(P)* has the same time complexity as QSQR/SLS. ▮

Seki [31] demonstrates a similar correspondence to that of Theorem 6.1 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 6.1, 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.

Our algorithms work for programs that are not range restricted, as long as the program has finite negation trees and does not flounder, and Theorem 6.1 holds in this case.

7 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. (Again, we adopt the convention that we are only interested in smaller, simpler subparts.) 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. We can solve the parts-explosion problem using the following program.

$$in(X, Y, null, N) \leftarrow part(X, Y, N)$$
$$in(X, Y, Z, N) \leftarrow part(X, Z, P), contains(Z, Y, M), N = P * M$$
$$contains(X, Y, N) \leftarrow N = \sum P : in(X, Y, _, P)$$

(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[9] with only stratified aggregation. Recall the game program from Example 3.1, which was not expressible by a stratified program because an *unbounded* number of recursions through

[9] We ignore here the possibility of using the integers, multiplication and summation to encode Turing machines.

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 paper generalize to the class of programs with function symbols. However, if function symbols are recursively applied, then termination cannot be guaranteed.

With a slight change of syntax, the rewritten program of Algorithm 5.1 may be interpreted as statements in the database programming language Glue [19, 20]. In fact, Algorithm 5.1 is the basis of a compiler that has been implemented by the author and others as part of the NAIL! system at Stanford University. The class of modularly stratified programs has also been adopted for use in the CORAL deductive database programming language [27].

Acknowledgements

I would like to thank Shuky Sagiv, Rodney Topor and Jeff Ullman for comments on earlier versions of this paper. The anonymous referees made many helpful suggestions to improve the presentation of this paper. Thanks also to Ashish Gupta for his contributions to the compiler that was implemented as part of the NAIL! system, and to Geoff Phipps for his help with Glue.

A Long Proofs

Proof of Theorem 5.3.

The proof will be by induction, comparing the evaluation of *WFR(P)* with an "implementation" of global SLS-resolution from Section 4.1.

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 4.1 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:

[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.

	Global SLS	*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 nor 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.

(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 nor 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 $\Box A$ 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 4.1. Since only finitely many tuples can be generated in the evaluation of *WFR(P)*, it follows that this bottom-up evaluation terminates. ∎

Proof of Theorem 6.1.

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 most general elements from a set of atoms, we assume that the same choice is made by both QSQR/SLS and magic sets.[11] We also assume that the bottom-up evaluation employs regional duplicate elimination, as described in Section 5.1.

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

[11] 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.

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

$$p(\vec{Z}) \leftarrow sup_{j.s_j}(\vec{Y})$$

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 4.4), 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 4.1, 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. ∎

References

[1] Balbin, I., Meenkashi, K., and Ramamohanarao, K. An efficient labelling algorithm for magic set computation on stratified databases. Tech. Rep. 88/1, Dept. of Computer Science, University of Melbourne, 1988.

[2] Balbin, I., Port, G. S., and Ramamohanarao, K. Magic set computation for on stratified databases. Tech. Rep. 87/3, Dept. of Computer Science, University of Melbourne, 1987.

[3] Balbin, I., Port, G. S., Ramamohanarao, K., and Meenkashi, K. Efficient bottom-up computation of queries on stratified databases. *Journal of Logic Programming 11* (1991), 295–344.

[4] Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J. D. Magic sets and other strange ways to implement logic programs. In *Proceedings of the Fifth ACM Symposium on Principles of Database Systems* (1986).

[5] Bancilhon, F., and Ramakrishnan, R. An amateur's introduction to recursive query processing strategies. In *1986 ACM-SIGMOD Conf. on Management of Data* (1986), pp. 16–52.

[6] Beeri, C., and Ramakrishnan, R. On the power of magic. *Journal of Logic Programming 10* (1991), 255–300. Preliminary version appeared in the 6th ACM Symposium on Principles of Database Systems, 1987.

[7] Beeri, C., Ramakrishnan, R., Srivastava, D., and Sudarshan, S. Magic implementation of stratified logic programs. (manuscript), 1989.

[8] Brodsky, A., and Sagiv, Y. Inference of monotonicity constraints in Datalog programs. In *Proceedings of the Eighth ACM Symposium on Principles of Database Systems* (1989).

[9] Bry, F., Dec. 1989. (personal communication).

[10] Bry, F. Logic programming as constructivism: A formalization and its application to databases. In *Proceedings of the Eighth ACM Symposium on Principles of Database Systems* (1989).

[11] Chan, D. Constructive negation based on the completed database. In *Proc. Fifth International Conference and Symposium on Logic Programming* (1988).

[12] Chen, W., Kifer, M., and Warren, D. S. HiLog: A first order semantics for higher-order logic programming constructs. In *Proc. North American Logic Programming Conference* (1989).

[13] Dietrich, S., and Warren, D. S. Dynamic programming strategies for the evaluation of recursive queries. Tech. Rep. 85/31, Computer Science Department, State University of New York at Stony Brook, 1985.

[14] Gelfond, M., and Lifschitz, V. The stable model semantics for logic programming. In *Proc. Fifth International Conference and Symposium on Logic Programming* (1988).

[15] Kemp, D. B., and Topor, R. W. Completeness of a top down query evaluation procedure for stratified databases. In *Proc. Fifth International Conference and Symposium on Logic Programming* (1988).

[16] Kerisit, J. M., and Pugin, J. M. Efficient query answering on stratified databases. In *Proceedings of the International Conference on Fifth Generation Computer Systems* (1988).

[17] Kolaitis, P. G. The expressive power of stratified programs. *Information and Computation 90* (1991), 50–66.

[18] Lloyd, J. W. *Foundations of Logic Programming*, 2nd ed. Springer-Verlag, New York, 1987.

[19] Phipps, G. Glue: A deductive database programming language. Tech. Rep. TR-CS-90-14, Kansas State University, 1990. Proceedings of the NACLP'90 Workshop on Deductive Databases.

[20] Phipps, G., Derr, M., and Ross, K. A. Glue-Nail: A deductive database system. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data* (1991).

[21] Przymusinska, H., and Przymusinski, T. C. Weakly stratified logic programs. *Fundamenta Informaticae 13* (1990), 51–65. Preliminary version appeared in Proc. Fifth International Conference and Symposium on Logic Programming, 1988.

[22] Przymusinski, T. C. On the declarative semantics of deductive databases and logic programs. In *Foundations of Deductive Databases and Logic Programming* (Los Altos, CA, 1988), J. Minker, Ed., Morgan Kaufmann, pp. 193–216.

[23] Przymusinski, T. C. Every logic program has a natural stratification and an iterated fixed point model. In *ACM Symposium on Principles of Database Systems* (1989).

[24] Przymusinski, T. C. On constructive negation in logic programming. In *Proceedings, North American Conference on Logic Programming* (1989).

[25] Ramakrishnan, R. Magic templates: A spellbinding approach to logic programs. *Journal of Logic Programming 11* (1991), 189–216.

[26] Ramakrishnan, R., Srivastava, D., and Sudarshan, S. Rule ordering in bottom-up fixpoint evaluation of logic programs. In *Proceedings of the International Conference on Very Large Databases* (1990).

[27] Ramakrishnan, R., Srivastava, D., and Sudarshan, S. Coral: A deductive database programming language. In *Proc. 18th Int. Conf. on Very Large Databases* (Vancouver, Canada, Aug. 1992).

[28] Ross, K. A. Modular stratification and magic sets for Datalog programs with negation. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems* (1990).

[29] Ross, K. A. Modular acyclicity and tail recursion in logic programs. In *Proceedings of the Tenth ACM Symposium on Principles of Database Systems* (1991).

[30] Ross, K. A. A procedural semantics for well-founded negation in logic programs. *Journal of Logic Programming 13*, 1 (1992), 1–22. Preliminary version appeared in the Proceedings of the Eighth ACM Symposium on Principles of Database Systems, 1989.

[31] Seki, H. On the power of alexander templates. In *Proceedings of the Eighth ACM Symposium on Principles of Database Systems* (1989).

[32] Seki, H., and Itoh, H. A query evaluation method for stratified programs under the extended CWA. In *Proc. Fifth International Conference and Symposium on Logic Programming* (1988).

[33] Ullman, J. D. Bottom-up beats top-down for datalog. In *Proceedings of the Eighth ACM Symposium on Principles of Database Systems* (1989).

[34] Ullman, J. D. *Principles of Database and Knowledge Base Systems.* Computer Science Press, Rockville, MD, 1989. (Two volumes).

[35] Van Gelder, A., Ross, K. A., and Schlipf, J. S. Unfounded sets and well-founded semantics for general logic programs. *JACM 38*, 3 (1991), 620–650.

[36] Vieille, L. Recursive axioms in deductive databases: The query-subquery approach. In *Proc. First International Conference on Expert Database Systems* (1986).

[37] Vieille, L. A database-complete proof procedure based on SLD-resolution. In *Proc. Fourth International Conference on Logic Programming* (1987).