NORTH-HOLLAND

WELL-FOUNDED ORDERED SEARCH: GOAL-DIRECTED BOTTOM-UP EVALUATION OF WELL-FOUNDED MODELS

PETER J. STUCKEY AND S. SUDARSHAN

▷ There have been several evaluation mechanisms proposed for computing query answers based on the well-founded semantics, for programs with negation. However, these techniques are costly; in particular, for the special case of modularly stratified programs, Ordered Search is more efficient than the general-purpose techniques. However, Ordered Search is applicable only to modularly stratified programs. In this paper, we extend Ordered Search to compute the well-founded semantics for all (non-floundering) programs with negation. Our extension behaves exactly like Ordered Search on programs that are modularly stratified, and hence pays no extra cost for such programs. © Elsevier Science Inc., 1997 ◁

1. INTRODUCTION

In the recent past, much attention has been paid to the semantics and evaluation of programs that use negation. To handle programs that combine the use of negation with recursion, three-valued semantics, which allow the truth status of some facts to be undefined, have been proposed. If negation is used in conjunction with recursion, it is nontrivial to provide semantics to all programs based purely on logical implication. Early techniques to work around this problem (e.g., [1, 16, 21]) restricted the class of programs for which semantics (and correspondingly evaluation mechanisms) were defined. These semantics were two-valued, in that each fact (ground atom) is either true or it is false. For the general case of programs with

Address correspondence to Peter J. Stuckey, Department of Computer Science, University of Melbourne, Parkville 3052, Australia, E-mail: pjs@cw.mu.oz.au; S. Sudarshan, Computer Science and Engineering Department, Indian Institute of Technology, Bombay 400076, India.

Received February 1995; accepted August 1996.

recursion and negation, two-valued semantics were found to be inadequate in many situations. For example, with a rule $p :- \neg p$, it is not clear whether p should be true or false. If it is false, it would imply that it is true. But there is no *basis* for deducing it to be true. More recently, three-valued semantics were proposed that allow the truth value of facts to be undefined. In the case of the rule $p :- \neg p$, a three-valued semantics can leave p undefined (if this is the only rule defining p), thereby solving the problem of whether to make p true or false.

The well-founded semantics [26] is the leading candidate among the three-valued semantics that have been proposed. The well-founded semantics is nontrivial to compute; in particular, it is nontrivial to make the computation "goal-directed," that is, given a query on a program, to make sure that intermediate facts are generated only if they are relevant to answering the query. Early evaluation mechanisms, such as the alternating fixpoint technique of [25], were not goal-directed. Other techniques, such as that of Ross [20], were goal-directed, but (as with Prolog) could repeat computation of subgoals multiple times and, worse, were noneffective (i.e., could loop) even for DATALOG programs.

For situations where the cost of recomputation is high (as when computation goes into a loop), memoing evaluations, which remember subgoals and avoid recomputation, are important. For the simple case of programs without negation, several memoing evaluation techniques have been proposed [2, 17, 24, 27]. Several attempts have been made at extending some of these for computing the well-founded semantics. These past attempts have the problem that either the computation is not completely goal-directed [11–13, 15] since some facts that are irrelevant to the computation may be generated, or they compute only relevant facts, but may compute some of them multiple times [9]. We present more details on related work in Section 7. But, in particular, for the important special case of modularly stratified programs [21], these techniques are less efficient than special-purpose techniques such as Ordered Search [18].

Although Ordered Search is more efficient than the general-purpose evaluation techniques proposed in the past, as described in [18] it applies only to modularly stratified programs, and not to the class of all programs with recursive negation. In this paper, we extend the Ordered Search evaluation algorithm to compute the well-founded semantics for all (nonfloundering) programs with negation. We call our technique Well-Founded Ordered Search. Our technique has the benefits of performing memoization of facts and being goal-directed.

For the case of modularly stratified programs, our technique reduces to the original Ordered Search algorithm, thereby reaping the cost benefits of Ordered Search. For the general case, our technique has important advantages over evaluation techniques proposed in the past. Recently, Chen, Swift, and Warren [10, 8] have developed a goal-directed technique for computing the well-founded model. Our technique was developed independently of theirs. Their technique and ours each have advantages and disadvantages with respect to the other; we present details in Section 7.

The rest of this paper is organized as follows. In Section 2, we present some background material. In Section 3, we present a short background on Ordered Search [18]. We then present details of the extended Ordered Search algorithm, Well-Founded Ordered Search in Section 4. The correctness of the algorithm is shown in Section 5, and in Section 6, we discuss extensions. Finally, in Section 7, we discuss related work.

2. BACKGROUND

We assume familiarity with logic programming terminology (see [14]). Familiarity with Magic Templates rewriting [17], and with semi-naive bottom-up evaluation [2], would help in reading this paper, but we provide some background information. For the purposes of this paper, a *program* is a set of normal clauses which possibly include negative literals in the rule body. We assume that the programs we evaluate are *nonfloundering*, i.e., any subgoal set up on a negative literal is ground. In the context of deductive databases, this restriction is not severe, as most programs are *allowed*, that is, they satisfy a syntactic condition called *allowedness* which ensures that they do not flounder. A program is allowed if, in each clause, every variable appearing in the clause appears in a positive body literal.

As is standard in the deductive database literature, we differentiate between the *extensional database* which consists of the facts for relations that are explicitly stored in the database, and the *intensional database* which consists of the predicates that are defined using rules. Predicates in the extensional database are called *EDB predicates*, and predicates in the intensional database are called *IDB predicates*. We make the standard assumption that the set of EDB predicates is disjoint from the set of IDB predicates.

2.1. The Well-Founded Semantics

The well-founded semantics [26] is generally viewed as the desired choice of semantics of programs with negation from a deductive database point of view, because it extends the iterated model semantics [1] for stratified programs to arbitrary normal programs and gives a unique model to any such program.

We extend the definition of the usual consequence operator T_P for definite programs, to infer information from normal rules using a fixed set M of information about negative literals. Let M be a set of atoms:

$$T_P(M)(I) = \{a \mid \text{where there is a ground instance of a clause in } P$$
$$a :- q_1, \ldots, q_n, \neg p_1, \ldots, \neg p_r$$
$$\text{such that } \forall 1 \leq i \leq n \quad q_i \in I \quad \text{and } \forall 1 \leq j \leq r, \quad p_j \notin M\}.$$

Essentially, we do not infer new negative information using T_P, but we allow the use of fixed negative information, the complement of M, in inferring positive information.

For successor ordinals $\beta + 1$, $T_P(M) \uparrow (\beta + 1)(I)$ is defined as $T_P(M)(T_P(M) \uparrow \beta(I))$, and for limit ordinals β, $T_P(M) \uparrow \beta(I)$ is defined as $\bigcup_{\alpha < \beta} T_P(M) \uparrow \alpha(I)$.

It is straightforward to show that $T_P(M)(I)$ is monotonic and continuous on I for all M, and $T_P(M) \uparrow \omega(\emptyset)$ (usually written $T_P(M) \uparrow \omega$) is the least fixpoint of $T_P(M)$.

If A is a set of atoms, then let $\neg \cdot A$ be the set of literals $\{\neg a \mid a \in A\}$. Given a program P, its *well-founded semantics*, denoted W_P^*, is defined using an alternating fixpoint formulation as below:

$$F_P(T) \overset{\text{def}}{=} T_P(T) \uparrow \omega(\emptyset),$$

$$F_P^2(T) \overset{\text{def}}{=} F_P(F_P(T)),$$

$$W_P^* \overset{\text{def}}{=} lfp(F_P^2) \cup \neg \cdot (HB_P - gfp(F_P^2)),$$

where *lfp* and *gfp* denote the least and greatest fixpoints, respectively, and HB_P denotes the Herbrand base of program P. (The above formulation is adapted from the alternating fixpoint formulation in [25], and is similar to that of [5].) We shall denote the true, false, and undefined atoms in the well-founded model of a program P as $T[P]$, $F[P]$, and $U[P]$, respectively.

The alternating fixpoint determines a method of computing the well-founded model of a program P (see [25, 11]), by computing the sets $F_P(\emptyset), F_P^2(\emptyset) = F_P^2 \uparrow 1(\emptyset), F_P(F_P^2 \uparrow 1(\emptyset)), F_P^2 \uparrow 2(\emptyset), \ldots$. The computation terminates with the two sets: $lfp(F_P^2) = F_P^2 \uparrow \alpha(\emptyset)$ for some α, representing all the *true* atoms of the program, and $gfp(F_P^2) = F_P(F_P^2 \uparrow \alpha(\emptyset))$, representing all the *true* and *undefined* atoms of the program (the complement of the *false* atoms). In general, α could be transfinite, but so long as the program is a finite DATALOG program with finite relations, the fixpoint terminates with a finite α. The actual set of false facts (which is typically much larger than the number of true or undefined facts) is never directly computed.

Define an *unfounded set* (of P) with respect to $T \cup \neg \cdot F$ as a set of atoms A such that, for each $a \in A$ and each ground instance of a rule in P of the form

$$a :\!-\, q_1, \ldots, q_m, \neg p_1, \ldots, \neg p_r,$$

either (i) there exists $q_i \in F$ or $p_j \in T$, or (ii) there exists $q_i \in A$. The original formulation of well-founded semantics was in terms of unfounded sets; the intuition is that given any unfounded set (with respect to the set of known true and false facts) at any point, all facts in the unfounded set can be inferred to be false in the well-founded semantics. The alternating fixpoint formulation of the well-founded semantics is better for our purposes, although we occasionally use the idea of unfounded sets to provide extra intuition.

2.2. *Query-Restricted Bottom-Up Evaluation*

Query optimization transformations for bottom-up evaluation of programs (e.g., [17]) restrict computation to facts that are "interesting" to the query by calculating the set of queries that the original query "depends on." They were originally defined only for positive programs, and most such transformations are incorrect when applied to programs with negation since their notion of "depends on" is not applicable if negation is used (see [12]). We provide some background on bottom-up evaluation using the Magic-Templates transformation.

The bottom-up approach to answering queries consists of a two-part process. First, the program-query pair is rewritten in a form so that the bottom-up fixpoint evaluation of the program will be more efficient; next, the fixpoint of the rewritten program is computed by bottom-up iteration. Section 2.3 describes the initial rewriting, while Section 2.4 investigates the computation of the fixpoint of the rewritten program.

2.3. *The Magic Templates Rewriting Algorithm*

We present below a simplified version of the Magic Templates rewriting algorithm [17].[1] The idea is to compute an auxiliary predicate *query* that stores subgoals generated on predicates in the program. A fact of the form $query(p(\bar{t}))$ denotes that $?p(\bar{t})$ is a subgoal generated on p. In the fact $query(p(\bar{t}))$, p is formally treated as a function symbol, rather than a predicate, since the language is first order. We thus have a predicate and a function symbol of the same name—they are distinguished based on where they occur in the rule.

The rules in the program are then modified by attaching a literal to the rule body that uses the *query* predicate to act as a filter that prevents the rule from generating irrelevant facts when evaluated bottom-up. Further, the rewriting generates rules that define how to generate a query fact for a body literal, given a query fact on the head literal. For efficiency, *query* facts are only generated for intensional database (IDB) relations, those defined by rules, and not for extensional database (EDB) relations, defined by sets of facts.

Definition 2.1. The Magic Templates Algorithm. Let P be a program, and $?q(\bar{c})$ a query on the program. We construct a new program P^{mg}. Initially, P^{mg} is empty.

1. For each rule in P, add the *modified version* of the rule to P^{mg}. If rule r had head, say, $p(\bar{t})$, the modified version is obtained by adding the literal $query(p(\bar{t}))$ to the body.
2. For each rule r in P with head, say, $p(t)$, and for each occurrence of an IDB literal $q_i(\bar{t}_i)$ in its body, add a *query rule* to P^{mg}. The head is $query(q_i(\bar{t}_i))$. The body contains the literal $query(p(\bar{t}))$, and all literals that precede $q_i(\bar{t}_i)$ in the rule.
3. Create a *seed* fact $query(q(\bar{c}))$ from the query on the program.

We refer to the rules defining the *query* predicate as *query rules*. We sometimes refer to query rules as *magic rules*, and the query predicate as the *magic predicate*, when we need to be consistent with the terminology used in [4, 6, 17].

The rewriting has the important effect of mimicking Prolog in that (modulo optimizations such as tail recursion optimization and intelligent backtracking, and modulo some inefficiencies when nonground facts are generated) only goals and facts generated by Prolog are generated.

Example 2.1. Consider the following program (in this program, *sg* stands for "same generation"):

$$R1\colon sg(X,Y) \quad :- flat(X,Y).$$

$$R2\colon sg(X,Y) \quad :- up(X,U), sg(U,V), down(V,Y).$$

$$? - sg(john,Z).$$

[1] As described in [6, 17], the initial rewriting of a program and query is guided by a choice of *sideways information passing strategies*, or SIPS. For each rule, the associated SIP determines the order in which the body literals are evaluated. The version we present is tailored to the case that SIPS correspond to left-to-right evaluation with all arguments considered "bound" (perhaps to a free variable), as in Prolog.

The Magic Templates algorithm rewrites it as follows:

$sg(X,Y)$:– $query(sg(X,Y)), flat(X,Y)$. [Mod. Rule R1]
$sg(X,Y)$:– $query(sg(X,'Y)), up(X,U),$
$sg(U,V), down(V,Y)$. [Mod. Rule R2]
$query(sg(U,V))$:– $query(sg(X,Y)), up(X,U)$. [Query Rule]
$query(sg(john,Z))$. [Seed Query]

The first two rules above are the original rules, modified by adding filters. The third rule defines how to generate queries on the body of the second rule (in the original program), given queries on its head predicate. The last rule is a fact that corresponds to the original query on the program, and it is called the *seed query* fact.

The following theorem ensures the soundness and completeness of the transformed program P^{mg} with respect to the query on the original program P.

Theorem 2.1 [17]. If P is a definite clause program without negation, P is equivalent to P^{mg} with respect to the set of answers to the query.

Magic Templates is often presented along with an adornment rewriting that annotates predicates with a string composed of characters "f" and "b," with one character for each argument. This step, along with a modification of Magic Templates rewriting that projects out of query predicates those arguments that have an "f" adornment, is used to ensure that the rewritten program generates only ground facts if the original program generated only ground facts. The benefit of generating only ground facts is achieved at the possible cost of some redundant computation, but is important since it permits the use of database systems that handle only ground facts. For simplicity, we omit this step.

2.4. *Iterative Fixpoint Evaluation*

The fundamental step in iterative fixpoint evaluation is a derivation. A *derivation* generates a fact f from a rule R and a substitution θ, given a set of already known facts W, where

1. the fact f generated by the derivative is the head of $R[\theta]$, and
2. for each body literal $p_i(\overline{t_i})$ in R, there is a fact in W that subsumes $p_i(\overline{t_i})[\theta]$, and
3. θ is the most general such substitution.

Given a set of facts W, a *rule application* generates all facts that can be inferred from the W using a derivation.

A naive evaluation of the fixpoint of a program performs iterations, with each iteration generating all facts that can be derived using the program rules, base facts, and the facts derived in earlier iterations. Iteration proceeds until a fixpoint is reached. In such a naive evaluation of the fixpoint, each iteration repeats all derivations made in earlier iterations.

Semi-naive evaluation (see, e.g., [3, 2]) is an incremental version of naive fixpoint evaluation. Semi-naive evaluation avoids the repetition of derivations by performing in each iteration an incremental computation using facts generated in the previous iteration. That is, it only carries out derivations that use at least one fact generated for the first time in the previous iteration. Any other derivations must have been performed before and are not repeated. Semi-naive evaluation maintains *differential relations* corresponding to each relation in the program, to keep track of when each fact in the relation was generated (before the last iteration, in the last iteration, or in the current iteration).

2.5. *The Depends On Relationship*

Magic Templates rewriting does not work correctly under the well-founded semantics. The problem is its notion of relevance, which says that a subgoal is relevant only if there is an instantiated rule prefix whose last literal is the subgoal, and all literals before the subgoal are satisfied. With the well-founded semantics, even if the truth of a rule body literal is undecided, it may be necessary to check if a later literal rule body is definitely false.

The following definition gives the formal meaning of "depends on," and is applicable to the well-founded semantics. Here we assume, as we do throughout the paper, a complete left-to-right order on generation of subgoals.

Definition 2.2 (Depends On). Let P be a given program. We say a query $?p(\bar{t})$ *directly depends on* $?q_i(\bar{b}_i)$ if there is a rule instance

$$p(\bar{a}) :- q_1(\bar{b}_1), \ldots, q_i(\bar{b}_i), \ldots, q_n(\bar{b}_n)$$

where each $q_i(\bar{b}_i)$ is a positive or negative literal, such that $p(\bar{a})$ is an instance of $p(\bar{t})$, and each literal $q_j(\bar{b}_j)$, $1 \le j \le i-1$, is either true or undefined in the well-founded model of P.

We define *depends on* as the transitive closure of "directly depends on."

The definition essentially says that in order to solve the query $?p(\bar{t})$, answers to the subquery $?q_i(\bar{b}_i)$ are relevant. In the case of two-valued models, the definition reduces to the regular definition of "depends on" [18] based on which relevance of facts is defined [17].

Intuitively, the importance of *depends on* is this: to correctly compute the answers to query $?(p(\bar{t})$ with respect to (w.r.t.) W_P^*, we only require the correct answers (w.r.t. W_P^*) of each of the queries $?q_i(\bar{b}_i)$ that $?p(\bar{t})$ depends on. (This is shown implicitly in the course of the correctness proofs of our technique.) Hence we would like to restrict computation to only those queries that $?p(\bar{t})$ depends on. This is not possible because the *depends on* relationship is known only once the well-founded model is computed. In general, we must use a superset of the queries that $?p(\bar{t})$ depends on. Minimizing this set is one of the main aims of this work.

3. ORDERED SEARCH

We now describe the Ordered Search evaluation method [18], which is applicable to modularly stratified programs. In the next section, we describe our extension to the technique to handle the general case. This technique generates subgoals and answers to subgoals asynchronously, as in bottom-up evaluation, but orders the use of generated subgoals in a manner reminiscent of top-down evaluation, and is in a sense a hybrid between pure (tuple-oriented) top-down evaluation and pure (set-oriented) bottom-up evaluation. The Ordered Search evaluation algorithm [18] has two phases. The first rewrites the program at compile time. The second evaluates the rewritten program. Unlike the case for programs without negation (Theorem 2.1), the rewritten program is not equivalent to the original program, and ordinary bottom-up evaluation of the rewritten program does not yield the correct set of answers to the query. Rather, it is equivalent in the sense that under a special evaluation mechanism, described below, the correct set of answers to the query is generated by the rewritten program.

3.1. Modified Magic Templates Rewriting

We describe the rewriting phase using an example rule. Suppose we have the following rule in a program:

$$p(X) :- r(X,Y), \neg q(Y), s(Y).$$

The modified Magic Templates rewriting [18] of the rule generates the following rules:

$$\begin{aligned}
&p(X) && :- query(p(X)), r(X,Y), done(q(Y)), \neg q(Y), s(Y).\\
&query(r(X,Y)) && :- query(p(X)).\\
&query(q(Y)) && :- query(p(X)), r(X,Y).\\
&query(s(Y)) && :- query(p(X)), r(X,Y), done(q(Y)), \neg q(Y).
\end{aligned}$$

The first rule is basically the original rule, but with two modifications. First, as in Magic Templates, a literal $query(p(X))$ has been inserted, which ensures that an "answer" fact for the predicate p is generated only if there is a corresponding query fact. This is done to avoid generating irrelevant facts. Second, a literal $done(q(Y))$ has been added to the rule to guard the $\neg q(Y)$ literal; this is an extension to Magic Templates, introduced by Ordered Search. A fact $done(q(a))$ is created when Ordered Search decides that all answers to the query $?q(a)$ have been generated.

We then use a modification of semi-naive evaluation where a ground negative literal $\neg p(\bar{a})$ is satisfied if $p(\bar{a})$ is not known to be true or undefined. Without the guard literal $done(q(Y))$, the rule could potentially be used in a semi-naive evaluation to make an inference, assuming $\neg q(a)$ is true even if a fact $q(a)$ is indeed generated later. The guard literal ensures that such a derivation is made only when $done(q(a))$ is present; by means of inserting facts $done(\ldots)$ at appropriate times, Ordered Search ensures the soundness of derivations.

The next three rules specify how to generate subgoals on the three body literals, given a subgoal on the head literal. These subgoals need to be solved in order to answer the subgoal on the head literal. For example, the second rule, read declaratively, says that if there is a subgoal $?p(X)$, then a subgoal $?r(X,Y)$ is generated. The third rule says that if there is a subgoal $?p(X)$ and an answer $r(X,Y)$, then a subgoal $?q(Y)$ is generated.

The modified Magic Templates rewriting of a program is the union of the modified Magic Templates rewriting of all the rules in the program.

3.2. *Ordered Search Evaluation*

The second phase of the Ordered Search algorithm evaluates the rewritten rules. We present an intuitive description of the evaluation algorithm here, but refers the reader to [18] for details. The algorithm makes inferences from the rewritten rules, and is built on top of the semi-naive evaluation technique. But unlike normal semi-naive evaluation, it orders the use of generated subgoals in a manner somewhat like Prolog. Unlike Prolog, Ordered Search performs duplicate elimination on subgoals and answers. It is, in a sense, a hybrid between pure (tuple-oriented) top-down evaluation and pure (set-oriented) bottom-up evaluation.

The central data structure used by Ordered Search, the *Context*, is used to preserve "dependency information" between subgoals. The *Context* is a sequence of *Context Nodes*. Each *ContextNode* has an associated set of query facts and each query fact is associated with a unique *ContextNode*.

The *Context* behaves somewhat like a stack in that, for the most part, nodes are either added to its end or removed from its end. However, other operations such as collapsing together nodes are also performed on the *Context*. In the rest of this paper, when we use adjectives like "earlier," "later," etc. to refer to *ContextNodes* in *Context*, we mean their position in the sequence and not the time at which these nodes were inserted in *Context*.

The Ordered Search evaluation algorithm is summarized below.

```
Algorithm Ordered Search
Input: Rewritten Program $P^{mg\_mod}$ (without the seed query fact), and query $?q(\bar{t})$.
Output: Answers to $?q(\bar{t})$.
  1. Initialize Context to consist of a single context-node containing the
       (unmarked) seed fact $query(q(\bar{t}))$.
  2. Repeat
       Repeat
         Evaluate the rules of the program using semi-naive evaluation.
         However for each newly generated query fact, call it $query(q(\bar{a}))$,
           instead of inserting it into the query relation
           2(a) insert $query(q(\bar{a}))$ in Context (as described later) and
           2(b) perform duplicate elimination on $query(q(\bar{a}))$ (as described
                  later).
           /* $query(q(\bar{a}))$ is not made visible to the evaluation yet */
       Until no new derivations can be made
  3. Make facts from the context visible (as described later)
  4. Until there is no change in the set of visible facts.
     /* At this stage Context is empty, and there are no hidden facts. */
```

Newly generated facts other than *query* facts are inserted in the differential relations, and made available as usual to the semi-naive evaluation. When query facts are first inserted into *Context*, they are "hidden," that is, they are not made available to the evaluation. The Ordered Search algorithm makes each query fact "visible" to the evaluation later; when a query fact that is in the *Context* is made available to the evaluation, the copy in the *Context* in *marked.* A *ContextNode* is said to be *marked* if any fact associated with the *Context* is *marked.* A *ContextNode* is said to be *marked* if any fact associated with the *ContextNode* is marked.

We now describe some of the context manipulation operations performed in Steps 2 and 3 of the above algorithm in more detail:

2(a). *Insertion*: When a new query fact *query*($q(\bar{a})$) is inserted in *Context*, it is inserted in a new *ContextNode*. Let *query*($q(\bar{a})$) be a query fact derived from query fact *query*($p(\bar{b})$).

(i) If *done*($q(\bar{a})$) is present do not insert *query*($q(\bar{a})$) in *Context* (since it has been fully evaluated already).

(ii) Else, *query*($p(\bar{b})$) must be in the *Context* and must be marked since it is visible and has just been used to derive *query*($q(\bar{a})$). Insert *query*($q(\bar{a})$) in a new unmarked *ContextNode* immediately before the next marked *ContextNode* following the marked node containing *query*($p(\bar{b})$). (If there is no such marked *ContextNode*, *query*($q(\bar{a})$) is inserted as the last *ContextNode* in the *Context*.) Some subsection of the initial *Context* is shown at the top of Figure 1, where nodes marked A and Z are unmarked and the next marked *ContextNode* contains *query*($r(\bar{c})$). The resulting subsection after the insertion is illustrated in the bottom of Figure 1.

2(b). *Duplicate Elimination*: Duplicate elimination is performed on *query*($q(\bar{a})$) in the *Context* to ensure that there is at most one copy of it in *Context*. If there is more than one unmarked copy of *query*($q(\bar{a})$) in *Context* at this stage, only the last copy of *query*($q(\bar{a})$) is retained, and the rest deleted. If there is a marked copy of *query*($q(\bar{a})$) in *Context*, i.e., if *query*($q(\bar{a})$) has already been made available to the evaluation, there are two possibilities:

(i) If the marked copy of *query*($q(\bar{a})$) occurs after the unmarked copy, only the marked copy of *query*($q(\bar{a})$) is retained in *Context*.

(ii) If the unmarked copy of *query*($q(\bar{a})$) occurs after the marked copy, *query*($q(\bar{a})$) depends on itself. We have thus detected a cyclic dependency between the set of all marked facts in *Context* between the two occurrences of *query*($q(\bar{a})$). Ordered Search deletes the unmarked copy of *query*($q(\bar{a})$) and collapses the above set of marked facts into the node of the marked copy of *query*($q(\bar{a})$) in *Context*.

3. *Making Query Facts Visible*: This step makes query facts in the *Context* visible to the evaluation when no new facts can be computed using the set of available facts. Intuitively, this is done as follows:

(i) If the last *ContextNode* contains at least one unmarked query fact, Ordered Search chooses one such unmarked fact, marks it, and makes

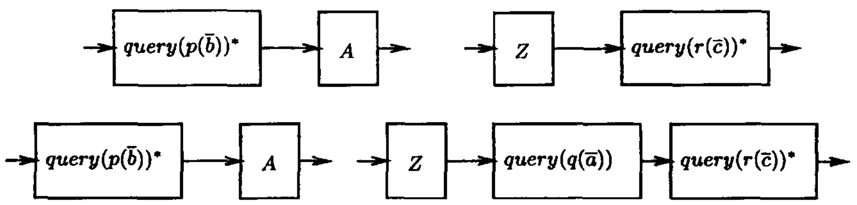

FIGURE 1. Inserting *query*($q(\bar{a})$) in the *Context*.

it available to the evaluation by inserting it in the corresponding differential relation. (Note that this fact still remains in the *Context*.)

(ii) If all query facts in the last *ContextNode* are marked, all the facts in the last *ContextNode* can be considered to be completely evaluated *in the case of* Ordered Search. Then the node is removed from *Context*, and for each subgoal *query*($q(\bar{a})$) in the node, a fact *done*($q(\bar{a})$) is created and made available to the semi-naive evaluation.

A major difference between Ordered Search and Well-Founded Ordered Search, which we describe in Section 4, is in Step 3.

In the above, we consider variants of a fact (i.e., facts that are equal, up to a renaming of the variables, to the given fact) as being the same as the fact. The algorithm can be easily extended to perform subsumption checking, and details are presented in [19]. The insertion step (2(a)) ensures that facts on *Context* are stored in an ordered fashion, such that if query fact Q_1 depends on the query fact Q_2, then Q_2 is stored after or along with Q_1 in the *Context*. But, unlike the stack of subgoals in Prolog evaluation, cyclic dependencies are handled gracefully by means of collapsing nodes together. Each subgoal in a node depends on all the other subgoals in the node, and hence we cannot in general deduce that we have found all answers for one until we are convinced we have found all answers for the others. In Step 2(b), on detecting a cyclic dependency between subgoals on the *Context*, the associated *ContextNodes* are collapsed into one *ContextNode*, and all the facts associated with these *ContextNodes* are now kept together. Thus we have the following property:

- If a subgoal *query*($q(\bar{a})$) depends on another subgoal *query*($p(\bar{b})$), then either *query*($p(\bar{b})$) is completely evaluated before *query*($q(\bar{a})$) is made available to evaluation (i.e., marked on *Context*) or at some point in the evaluation *query*($p(\bar{b})$) is in a node in *Context* above a node containing a marked version of *query*($q(\bar{a})$).

The above property is used to show that when a query is declared to be completely evaluated (i.e., a corresponding *done* fact is created), all answers to it have indeed been generated.

The Ordered Search algorithm also satisfies the following property:

- Each marked subgoal in the context sequence depends directly on the following marked subgoal in the *Context*, and on each unmarked subgoal that lies between it and the following marked subgoal in the sequence.

The above property is used to show that no false dependencies between query facts are introduced by the algorithm. The full dependence relation known at any stage can be computed by a transitive closure on the immediate dependencies. It is clear that each marked subgoal depends (transitively) on all marked subgoals later in the context.

Example 3.1. We now give an example of the Ordered Search procedure in action. Consider the following program, which determines a winning position for games such as checkers where each player alternately makes a move, and the winner is the person who makes the last move. Sometimes a player may make extra moves.

Board positions are encoded as simple letters:

$win(X) :- move(X,Y), \neg win(Y).$

$win(X) :- extramove(X,Y), win(Y).$

$move(a,b).$

$move(a,d).$

$move(b,c).$

$extramove(a,e).$

$extramove(e,a).$

For simplicity, we will consider the *move* and *extramove* relations to be in the EDB and not determine *query* facts for them. The Magic Templates rewriting is

$win(X) \quad :- query(win(X)), move(X,Y), done(win(Y)), \neg win(Y).$

$win(X) \quad :- query(win(X)), extramove(X,Y), win(Y).$

$query(win(Y)) :- query(win(X)), move(X,Y).$

$query(win(Y)) :- query(win(X)), extramove(X,Y).$

Given the query $?win(a)$, Ordered Search evaluation starts by adding $query(win(a))$ to the *Context*; $query(win(a))$ is not made available for inferences yet. Nothing more can be derived, and hence Step 3(i) marks the fact and makes it available for making inferences. Using this fact, facts $query(win(b))$, $query(win(d))$, and $query(win(e))$ then get derived, and each is added to a new node at the end of *Context*. First, $query(win(e))$ is marked and made available for inferences. This derives the fact $query(win(a))$ which is initially placed at the end of the *Context*. We have discovered a cyclic dependency and the two marked nodes are collapsed together. The *Context* now looks like $\{query(win(a))^*, query(win(e))^*\}$ $\{query(win(b))\}$ $\{query(win(d))\}$.

Now $query(win(d))$ is marked and made available for inferences. No inferences can be made; hence, using Step 3(ii), we add a fact $done(win(d))$ and the *Context* node is removed. We have thus determined $\neg win(d)$. Now we generate the facts $win(a)$ and $win(e)$. The last *Context* node is now $query(win(b))$; this is marked and the fact $query(win(c))$ is derived and placed on the end of the context and, as before, gets marked and made available for making inferences. Similarly to the $win(d)$ case, we add $done(win(c))$ (inferring $\neg win(c)$ since $win(c)$ is absent) and remove the *Context* node. We now derive the fact $win(b)$, before $done(win(b))$ is derived and the $query(win(b))$ node is deleted. Finally, the last remaining *Context* node is $\{query(win(a))^*, query(win(e))^*\}$. All possible facts upon which these facts depend have been investigated. The last *Context* node is deleted and the facts $done(win(a))$ and $done(win(e))$ are added. This is the end of computation.

Ordered search is correct for this program because there are no loops through negation, but if we add the single extra fact $move(d,a)$, Ordered Search is no longer applicable.

4. WELL-FOUNDED ORDERED SEARCH

We now describe *Well-Founded Ordered Search* (WF-OS for short), our extension to Ordered Search. A one-sentence summary (for the expert) of the idea behind

WF-OS is that it combines Ordered Search with the alternating fixpoint technique for evaluating the well-founded semantics, and manages to use the (costly) alternating fixpoint technique on subregions of the program rather than on the entire program. As with Ordered Search, we split the description of WF-OS into two parts. The first part describes the extended magic rewriting, and the second part describes the actual WF-OS evaluation technique.

In the case of a cycle subgoals, Ordered Search keeps track of the cycle, and when no more subgoals and no more answers can be generated from subgoals in the cycle, Ordered Search decides that all answers for subgoals in the cycle have been obtained. If a cycle of subgoals containing a negative subgoal is found, Ordered Search concludes that the program is not modularly stratified and proceeds no further. However, to compute the well-founded semantics for all programs, one cannot stop at a point where a negative cycle has been found.

Well-Founded Ordered Search extends Ordered Search by the actions that are taken in Step 3 (the "Making Facts Visible" step) of the Ordered Search algorithm, in the case that a negative cycle is present in the last node of the *Context*. The actions are described in more detail later in this section, but the intuition behind our extension is as follows. There are two parts to the extension—generating more subgoals, and performing "local" alternating fixpoints rather than performing a single "global" alternating fixpoint.

We describe the intuition for each extension below.

Let us consider the motivation for the first part of the extension. Consider (for simplicity) a ground rule, with a subgoal that unifies with the head of the rule. In order to answer the subgoal on the head, subgoals have to be generated on body literals. In Ordered Search, the left-to-right subgoal generation mechanism generates a subgoal on a literal only if all preceding literals are true (i.e., for positive literals $p(\bar{a})$, it is known that $p(\bar{a})$ is true, and for a negative literal $\neg p(\bar{a})$, it is known that $p(\bar{a})$ is false). In order to compute the well-founded semantics, we may need to know if a literal later in the rule is true or false, even if the truth value of a literal earlier in the rule is not known [11]. Hence, to extend Ordered Search to compute well-founded models, we may need to generate a subgoal on a later literal even in cases where the truth value of earlier literals is not known.

In this respect, WF-OS differs from Ordered Search; in the restricted context of modularly stratified programs, using Ordered Search one can generate only subgoals that the original query depends on, directly or indirectly. In the general case handled by WF-OS, we may have to generate a superset of these subgoals.

The first part of our extension to Ordered Search is to generate extra subgoals when required. When WF-OS finds a negative cycle, it starts off the computation of "possibly true" facts (rather than just true facts) by considering negative literals that form part of the cycle as "possibly true." This computation ensures that a superset of all required subgoals is generated. Further, the computation generates a set of "possibly true" facts that contains the set of true facts.

Note that new subgoals that are generated as above may be added to the end of the *Context*, and the node with the negative cycle may no longer be the last node. But eventually the nodes added above it will be removed, and it will become the last node again. More new subgoals may then be added, and the cycle repeats. But eventually a stage is reached when no new subgoals can be added as above. At this stage, the last node in *Context* has a negative cycle, and all subgoals on which subgoals in the node depend have already been generated, and have either been solved or are in the node, and the "possibly true" facts are a superset of the true and undefined facts for subgoals in the last *ContextNode*.

The second part of our extension of Ordered Search is applied when a stage as above is reached. The subgoals in the last node define a subpart of the program. Intuitively, WF-OS applies the alternating fixpoint technique [25, 11] for computing the well-founded semantics (in a non-goal-directed fashion) to this subpart of the program. (Since all relevant subgoals are generated and have been taken into account in defining the subpart of the program, goal-directed evaluation need not be used for this subpart of the program.) The alternating fixpoint technique (and other techniques for computation of the well-founded semantics) can be quite costly, and by applying it only to well-chosen subparts of the full program, we are able to reduce the cost of evaluation considerably.

4.1. The Undef Magic Templates Rewriting

We now give the intuition behind the *Undef Magic Rewriting*, our extension of Magic Templates rewriting [17] which we use in WF-OS. In order to compute the well-founded semantics, we may need to know if a literal later in the rule is true or false, even if the truth value of a literal earlier in the rule is not known [11]. For example, with a rule $r :- \neg r, s$, and no rule defining s, the truth value of s is needed in order to determine that r is false; a subgoal $?s$ must be generated to find the truth status of s, at a point when the truth status of $\neg r$ is not known.

To do so, we use an extended Magic Templates rewriting, which we call Undef Magic Templates rewriting, which can generate "possibly true" facts (rather than just true facts) when provided appropriate "seed facts." Undef Magic Templates rewriting generates facts of the form $un(p(\bar{a}))$ and $un(\neg q(\bar{a}))$.[2] These facts respectively indicate that $p(\bar{a})$ is possibly true (i.e., has not been shown to be false), and $q(\bar{a})$ is possibly false (i.e., has not been shown to be true). Facts of the form $un(\ldots)$ are used to represent information about the truth value of a fact as of some point in the evaluation, and unlike other facts, may be present at some point of an evaluation but absent later. However, a fact $un(p(\bar{a}))$ is always present when $p(\bar{a})$ is known to be true (and similarly $un(\neg q(\bar{a}))$ is always present when $q(\bar{a}))$ is known to be false). We say a fact $p(\bar{a})$ is *possibly undefined* if a fact $un(p(\bar{a}))$ is present.

We say "possibly" since the fact may not actually be undefined in the well-founded semantics; it could be true, undefined, or even false. Such facts are needed to compute an overestimate of what (relevant) facts are true (resp., false).

We consider again the rule used to describe Ordered Search:

$$p(X) :- r(X,Y), \neg q(Y), s(Y).$$

Undef Magic rewriting of this rule generates the following rules:

$$\begin{aligned}
&query(r(X,Y)) :- query(p(X)).\\
&query^{\neg}(q(Y)) \ :- query(p(X)), un(r(X,Y)).\\
&query(s(Y)) \quad :- query(p(X)), un(r(X,Y)), un(\neg q(Y)).
\end{aligned}$$

[2] In an abuse of notation, we treat the negation symbol $\neg$ as an uninterpreted function symbol when it occurs inside an *un* fact.

$$un(p(X)) \quad :- query(p(X)), un(r(X,Y)), un(\neg q(Y)), un(s(Y)).$$

$$p(X) \quad :- query(p(X)), r(X,Y), done(q(Y)), \neg un(q(Y)), s(Y).$$

Further, for every predicate $p(\bar{X})$, we generate rules

$$un(p(\bar{X})) \quad :- p(\bar{X}).$$

$$un(\neg p(\bar{X})) :- done(p(\bar{X})), \neg p(\bar{X}).$$

The intuition behind the above rules is as follows. The first three rules generate subgoals, but differ from the rewriting used in Ordered Search in that they can generate a subgoal on a literal not only when earlier literals are true, but also when they are possibly undefined (i.e., corresponding *un*(...) facts have been generated). Another difference is illustrated in the second rule, where the generated query fact is tagged with a superscript $^{\neg}$. The tag is used in *Context* to recognize that the subgoal is generated from a negative literal. We treat the predicates $query^{\neg}(\ldots)$ and *query*(...) as separate facts in the *Context* but as synonymous for the purposes of semi-naive evaluation. The tag is used by the WF-OS evaluation algorithm. The fourth rule in the rewritten program generates an *un*(...) fact for the head predicate in case each literal in the body is possibly undefined. The last rule generated from the original rule derives answer facts that are definitely true. The purpose of the two other rules shown above is to make sure a literal is possibly undefined if it is true.

The general case of the rewriting is as follows.

Definition 4.1. The Undef Magic Templates Algorithm. Let P be a program, and $?q(\bar{c})$ a query on the program. We construct a new program *MagUnd*(P). Initially, *MagUnd*(P) is empty.

1. For each rule in P, add the *modified version* of the rule to *MagUnd*(P). If rule r has head, say, $p(\bar{t})$, the modified version is obtained by adding the literal $query(p(\bar{t}))$ to the body, and for each negative literal $\neg q(\bar{s})$ in the body where q is an IDB relation, adding the literal $done(q(\bar{s}))$ before the literal $\neg q(\bar{s})$, and replacing $\neg q(\bar{s})$ by $\neg un(q(\bar{s}))$.
2. For each rule in P, add the *undefined version* of the rule to *MagUnd*(P). If rule r has head, say, $p(\bar{t})$, the undefined version is obtained by adding the literal $query(p(\bar{t}))$ to the beginning of the body, and for each IDB relation literal in the rule (including the head) $q(\bar{s})$ or $\neg q(\bar{s})$, wrapping it with *un*( ), i.e., $un(q(\bar{s}))$ or $un(\neg g(\bar{s}))$.
3. For each rule r in P with head, say, $p(\bar{t})$, and for each occurrence of an IDB literal $q_i(\bar{t}_i)$ (or $\neg q_i(\bar{t}_i)$) in its body, add a *query rule* to *MagUnd*(P). The head is $query(q_i(\bar{t}_i))$ (resp., $query^{\neg}(q_i(\bar{t}_i))$). The body contains all literals that preceded $un(q_i(\bar{t}_i))$ in the undefined version of r.
4. For each IDB relation p in the program, add the rules

   $$un(p(\bar{X})) \quad :- p(\bar{X}).$$

   $$un(\neg p(\bar{X})) :- done(p(\bar{X})), \neg p(\bar{X}).$$

   to *MagUnd*(P).
5. Create a *seed* fact $query(q(\bar{c}))$ from the query on the program.

In practice, we would use a variant of the above rewriting that generates "supplementary rules," to factor out common subexpressions in a manner similar to Supplementary Magic rewriting [6]. We omit details for simplicity. The rewriting and evaluation mechanisms contain some redundancies, such as generating *un* facts even when it is obvious that they are not needed (e.g., for programs without negation). Such inefficiencies can be removed fairly easily; but for simplicity, we describe only the unoptimized but less complicated algorithms.

4.2. *Intuition behind the Well-Founded Ordered-Search Algorithm*

An inspection of the rules in *MagUnd*(P) indicates that a fact of the form $un(p(\bar{a}))$ can be generated using the rules only if there is already a fact $p(\bar{a})$. However, there is another mechanism to generate facts of the form $un(\ldots)$—the WF-OS evaluation algorithm described in the next section. Such facts are generated in order to bypass negative literals so as to generate subgoals on later literals in a rule, in case cycles containing negative subgoals are encountered.

WF-OS proceeds like Ordered Search, except for ignoring negative cycles of subgoals, until all subgoals in the top node of context have been made visible. At this stage, WF-OS starts off the computation of "possibly true" facts (rather than just definitely true facts) by considering negative literals that form part of the cycle as "possibly true" (these constitute the "seed facts"). This process eventually ensures that a superset of all required subgoals [12] is generated.

Eventually, a stage is reached when no new subgoals can be added as above. At this stage, the last node in *Context* has a negative cycle, and all subgoals on which subgoals in the node depend have already been generated, and have either been solved or are in the node. At this stage, the $un(\ldots)$ facts are a superset of the true and undefined facts for subgoals in the last *ContextNode*. The subgoals in the last node define a subpart of the program. Intuitively, WF-OS now applies the alternating fixpoint technique [25, 11] for computing the well-founded semantics (in a non-goal-directed fashion) to *this subpart* of the program, rather than to the whole program. The alternating fixpoint technique (and other techniques for computation of the well-founded semantics) can be quite costly, and by applying it only to well-chosen subparts of the full program, we are able to reduce the cost of evaluation considerably.

4.3. *The Well-Founded Ordered Search Algorithm*

We now present some details of the WF-OS algorithm. The algorithm is basically the same as the Ordered Search algorithm presented in Section 3.2, except that (a) the Undef Magic rewriting is used instead of Magic rewriting, and (b) Steps 2(b) and 3 of the evaluation algorithm are modified to be as follows:

2(b). *Duplicate elimination*: Unmarked copies of $query(q(\bar{a}))$ and $query^{\neg}(q(\bar{a}))$ are treated as distinct facts, and only the latest unmarked copy of each is retained. It is important to note that no dependency information is lost thus—a direct dependency is replaced by an indirect dependency.

If there is a marked copy and an unmarked copy of $query^{[\neg]}(q(\bar{a}))$ (with or without tag "$\neg$") in *Context*, there are two possibilities:

(i) If the marked copy of $query^{[\neg]}(q(\bar{a}))$ occurs after the unmarked copy, only the marked copy of $query^{[\neg]}(q(\bar{a}))$ is retained in *Context* if they are both tagged "$\neg$" or both untagged; otherwise, they are both retained.

(ii) If the unmarked copy (tagged or untagged) of $query^{[\neg]}(q(\bar{a}))$ occurs after the (tagged or untagged) marked copy, we have detected a cyclic dependency involving $query^{[\neg]}(q(\bar{a}))$ and all marked facts in *Context* in between the two occurrences of $query^{[\neg]}(q(\bar{a}))$. The unmarked copy of $query^{[\neg]}(q(\bar{a}))$ and the above set of marked facts are collapsed into the node of the marked copy of $query^{[\neg]}(q(\bar{a}))$ in *Context*. If one of the facts collapsed into this node has a negative tag, then the node is marked as a NEGLOOP. If $query^{\neg}(q(\bar{a}))$ and $query(q(\bar{a}))$ are both present and one is marked, the other is marked as well.

3. *Making Query Facts Visible*

(i) While the last node in *Context* contains at least one unmarked query fact,

```
    Choose an unmarked fact from the last node
    Perform duplicate elimination using the fact (Step 2(b)(ii));
    If no marked (tagged or untagged) copy of the fact was found,
    break;
If an unmarked fact was found above, mark it and make it available to
the evaluation by inserting it (without tag) in the corresponding
differential relation.
```

(ii) Otherwise, all facts in the last *ContextNode* are marked. If the node is not marked NEGLOOP, the node has been completely evaluated. The node is removed from *Context*, and for each (tagged or untagged) fact $query^{[\neg]}(p(\bar{a}))$ in the node, a fact $done(p(\bar{a}))$ is created.

Otherwise, executive Procedure Add_Undefined. If no new facts are added by Add_Undefined, execute Procedure Local_Alternation.

The intuition behind the above is that even if we find a cycle with negative subgoals, we proceed with other subgoals that are generated from subgoals in the cycle since they may not be recursive with those in the cycle. When we can proceed no further, we are at a stage where we have to bypass some of the negative subgoals in order to compute the well-founded model. This is done by means of Procedure Add_Undefined, which lets the left-to-right subgoal generation order skip over negative literals that are in the last node in *Context*, by introducing facts of the form $un(\neg\, q(\bar{a}))$.

```
Procedure Add_Undefined
  /* We are at a local fixpoint and there is a negative cycle. */
  For every fact $query^{\neg}(q(\bar{a}))$ in the last ContextNode,
    if neither $done(q(\bar{a}))$ nor $q(\bar{a})$ is present
      Add $un(\neg\, q(\bar{a}))$ to the set of facts.
```

In case some new $un(\ldots)$ facts are added by Add_Undefined, evaluation continues as in Ordered Search. Further subgoals may be generated. If they do not depend on the goals in the negative cycle, they get solved independently. If there is a dependency, they get collapsed into the node containing the negative cycle.

Eventually, a stage is reached where all negative literals whose subgoals are in the last node of *Context* are noted as undefined (and thus bypassed), and no further subgoals can be generated. At this stage, all relevant subgoals have been generated. These subgoals define a subprogram that contains a cycle with a negative subgoal. To compute the well-founded model for this subprogram, WF-OS evaluation starts an alternating fixpoint evaluation [25, 11] using Procedure Local_Alternation, shown below. Alternating fixpoint computation by itself is not goal directed, and if used on the entire program would generate a potentially large number of irrelevant facts. However, the alternating fixpoint performed in Procedure Local_Alternation is "local" in that it only involves answers for the subgoals in the last node of *Context*. By restricting the alternating fixpoint to a subprograms containing "relevant" facts, we can reduce the time cost of computation considerably.

```
Procedure Local_Alternation
1.  Repeat
2.    For every query fact $query^{\neg}(q(\bar{a}))$ in the last $ContextNode$,
3.      If $un(q(\bar{a}))$ is not present   /* $q(\bar{a})$ is definitely false */
4.        Add $done(q(\bar{a}))$ to the set of facts.
5.      If $q(\bar{a})$ is present   /* $q(\bar{a})$ is true */
6.        Add $done(q(\bar{a}))$ to the set of facts.
7.        Remove $un(\neg q(\bar{a}))$
8.    If there is no change in the set of facts Then
9.      Break;  /* Last node in $Context$ has been fully evaluated */
10.   Else  /* Restart to find new upper-bound */
11.     For every fact $un(q(\bar{a}))$ that matches a tuple $query(q(\bar{b}))$ in the last
          $ContextNode$, and does not match any fact $done(q(\bar{c}))$
12.       Remove $un(q(\bar{a}))$.
13.     /* Note: Facts $un(\neg q(\bar{a}))$ are not removed at this step. */
14.     Apply all rules that define un-predicates in the last $ContextNode$.
15.     Do semi-naive evaluation on all rules until fixpoint.
16. Forever;
17. /* Local alternating fixpoint has terminated; Clean up and pop node */
18. Pop the last node from $Context$.
19. For every fact $query(q(\bar{a}))$ in the node,
20.   Add a fact $done(q(\bar{a}))$ to the set of facts.
```

Procedure Local_Alternation tightens the set of *un*(...) and *un*(¬ ...) facts by removing those whose truth status has been determined to be true or false, and recomputing the set of *un*(...) facts while keeping the *un*(¬ ...) facts fixed. The recomputation (lines 14–15) begins by firing all the rules that can produce *un*(...) facts, and these are used as the differential relations for the semi-naive evaluation. Our technique, like other techniques that compute the well-founded semantics in a goal-directed fashion, generates some queries that may not actually be relevant, but during the evaluation it is not possible to make out whether or not they are relevant. Specifically, we generate *query* facts from *un* facts that may be retracted later.

WF-OS behaves nearly identically to OS on left-to-right modularly stratified programs. In particular, Procedure Add_Undefined is never invoked. The only difference is that for each fact $[\neg]p(\bar{a})$ generated by OS, a fact $un([\neg]p(\bar{a}))$ is also generated by WF-OS. This does not result in any change in complexity. The difference between OS and WF-OS shows up on programs that are not left-to-right modularly stratified.

Example 4.1. To exemplify the relationship between the WF-OS procedure and Ordered Search, we now give an example of the WF-OS procedure in action on the *win* program from Example 3.1, with a database of moves that makes the program no longer modularly stratified:

$$
\begin{array}{l}
win(X) :- move(X,Y),\ \neg\, win(Y).\\
win(X) :- extramove(X,Y), win(Y).\\
move(a,b).\\
move(a,d).\\
move(b,c).\\
move(d,a).\\
extramove(a,e).\\
extramove(e,a).
\end{array}
$$

The Undef Magic rewriting is

$$
\begin{array}{ll}
win(X) & :- query(win(X)), move(X,Y), done(win(Y)),\\
 & \quad \neg(win(Y)).\\
un(win(X)) & :- query(win(X)), move(X,Y), un(\neg\, win(Y)).\\
win(X) & :- query(win(X)), extramove(X,Y), win(Y).\\
un(win(X)) & :- query(win(X)), extramove(X,Y), un(win(Y)).\\
un(win(X)) & :- win(X).\\
un(\neg\, win(X)) & :- done(win(X)), \neg\, win(X).\\
query^{\neg}(win(Y)) & :- query(win(X)), move(X,Y).\\
query(win(Y)) & :- query(win(X)), extramove(X,Y).
\end{array}
$$

The computation starts as in Example 3.1 using Ordered Search. $query(win(a))$ is added to the *Context*, marked, and the facts $query^{\neg}(win(b))$, $query^{\neg}(win(d))$, and $query(win(e))$ are derived; each is added to a new node at the end of *Context*. First, $query(win(e))$ is marked and made available for inferences. This derives the fact $query(win(a))$ and the *Context* is collapsed to become

$$\{query(win(a))^{*}, query(win(e))^{*}\} \quad \{query^{\neg}(win(b))\} \quad \{query^{\neg}(win(d))\}$$

Now, $query(win^{\neg}(d))$ is marked and made available for inferences. It derives the fact $query^{\neg}(win(a))$, which is placed on the end of the context; then this node and the marked node $\{query(win^{\neg}(d))^{*}\}$ are collapsed back into the first node, which is now marked as a NEGLOOP. The *Context* is now

$$\{query(win(a))^*, query(win(e))^*, query^{\neg}(win(d))^*, query^{\neg}(win(a))^*\},$$
$$\{query^{\neg}(win(b))\}$$

Execution then proceeds (basically) as in Ordered Search (Example 3.1) marking $query^{\neg}(win(b))$, and adding $query(win(c))$, $done(win(c))$, $win(b)$, and $done(win(b))$. In addition, the facts $un(\neg\, win(c))$ and $un(win(b))$ are derived.

Finally, the last remaining *Context* node is

$$\{query(win(a))^*, query(win(e))^*, query^{\neg}(win(d))^*, query^{\neg}(win(a))^*\}$$

Nothing more can be derived now, and all facts in the node are marked. Since the node is marked NEGLOOP, there is a negative query in a cycle. Hence, Step 3(ii) calls Add_Undefined, which adds the facts $un(\neg\, win(d))$ and $un(\neg\, win(a))$ to the negative query facts. Now facts $un(win(a))$, $un(win(e))$, and $un(win(d))$ are derived. (In general, new queries may be generated and evaluated at this stage.) Finally, we enter Local Alternation. Because $un(win(a))$, $un(win(d))$, and $un(win(e))$ are present and $win(a)$, $win(d)$, and $win(e)$ are not present, no change is made to the set of facts. Hence, we immediately exit the loop and pop the last *Context* node adding $done(win(a))$, $done(win(e))$, and $done(win(d)$.

The WF-OS procedure terminates having determined that $win(b)$ is true, $win(c)$ is false, and $win(a)$, $win(d)$, and $win(e)$ are undefined.

Example 4.2. The above example does not fully illustrate WF-OS. In this example, we see how Add_Undefined and Local_Alternation interact with the Ordered Search part of the procedure. Given the initial program

$$\begin{aligned}
&r(X) &&:- \neg\, s(X).\\
&s(X) &&:- q(X,Y), \neg\, r(Y), t(Y).\\
&q(X,a) &&:- \neg\, r(X).
\end{aligned}$$

the Undef Magic rewriting is

$$\begin{aligned}
&r(X) &&:- query(r(X)), done(s(X)), \neg\, un(s(X)).\\
&s(X) &&:- query(s(X)), q(X,Y), done(r(Y)), \neg\, un(r(Y)), t(Y).\\
&q(X,a) &&:- query(q(X,a)), done(r(X)), \neg\, un(r(X)).\\
&un(r(X)) &&:- query(r(X)), un(\neg\, s(X)).\\
&un(s(X)) &&:- query(s(X)), un(q(X,Y)), un(\neg\, r(Y)), un(t(Y)).\\
&un(q(X,a)) &&:- query(q(X,a)), un(\neg\, r(X)).\\
&un(r(X)) &&:- r(X).\\
&un(s(X)) &&:- s(X).\\
&un(q(X,a)) &&:- q(X,a).\\
&un(\neg\, r(X)) &&:- done(r(X)), \neg\, r(X).\\
&un(\neg\, s(X)) &&:- done(s(X)), \neg\, s(X).\\
&un(\neg\, q(X,a)) &&:- done(q(X,a)), \neg\, q(X,a).
\end{aligned}$$

$$\begin{aligned}
&query^{\neg}(s(X)) &&:- query(r(X)).\\
&query(q(X,Y)) &&:- query(s(X)).\\
&query^{\neg}(r(Y)) &&:- query(s(X)), un(q(X,Y)).\\
&query(t(Y)) &&:- query(s(X)), u(q(X,Y)), un(\neg r(Y)).\\
&query^{\neg}(r(X)) &&:- query(q(X,a)).
\end{aligned}$$

Given the query $r(a)$, WF-OS evaluation starts by adding $query(r(a))$ to the *Context*; $query(r(a))$ is not made available for inferences yet. Nothing more can be derived, and hence Step 3(a) marks the fact and makes it available for making inferences. Using this fact, $query^{\neg}(s(a))$ then gets derived, added to a new node at the end of *Context*, and as before, gets marked and made available for making inferences. Similarly, a fact $query(q(a,Y))$ is derived and inserted. Using this query fact, $query^{\neg}(r(a))$ is derived. Hence a cycle is detected and the nodes in the cycle (all the nodes in *Context* in this case) are collapsed into a single node containing $\{query(r(a)), query^{\neg}(s(a)), query(q(a,Y)), query^{\neg}(r(a))\}$. Because the marked facts $query^{\neg}(s(a))$ and $query^{\neg}(r(a))$ are collapsed back into the node, it is marked as a NEGLOOP.

Nothing more can be derived now, and all facts in the node are marked. Since the node is marked NEGLOOP, there is a negative query in a cycle. Hence Step 3(ii) calls Add_Undefined, which adds the fact $un(\neg s(a)), un(\neg r(a))$ corresponding to the negative query facts. Now facts $un(q(a,a)$, $un(r(a))$, and $query(t(a))$ get derived. To determine that $s(a)$ is false, we must examine the subgoal $t(a)$; this is why we skip over the undetermined literals $q(a,a)$, $\neg r(a)$.

The new query fact $query(t(a))$ is placed in a new *Context* node and, after marking, provides nothing new. Step 3(ii) removes the node from the *Context* and adds $done(t(a))$. Nothing more can be derived, and we are back at Step 3(ii) with the NEGLOOP marked node as the last in the *Context*, so we execute Local_Alternation.

Line	Action	Facts
4	Add	$done(s(a)))$
12	Delete un-facts	$un(q(a,a)), un(r(a))$
15	Fixpoint	$un(q(a,a)), r(a), un(r(a))$
6	Add	$done(r(a)))$
7	Remove	$un(\neg(r(a)))$
12	Delete un-facts	$un(q(a,a))$
15	Fixpoint	$\{\ \}$

Since nothing further is produced, we remove the *Context Node* and add the fact $done(q(a,Y))$. The results for the queries facts $r(a)$, $\neg s(a)$, $\forall Y \neg q(a,Y)$, $\neg t(a)$ agree with the well-founded model of the original program.

5. CORRECTNESS

The correctness of the method relies on two key observations: first, the query facts set up are large enough so that all the computations are correct, and second, a number of invariants hold throughout the computation. For simplicity, we do not consider any special treatment of EDB relations in this section; every relation is assumed to be IDB. EDB literals present no difficulties since they have a fixed two-valued model. Let W be the ground instances of the set of facts present at any stage in the computation.

We use a set of invariants to describe correctness properties of the program. The invariants are shown formally below, but first we consider the intuitive meaning of the invariants. Invariant 1 ensures that (a) when a *done* fact is generated, all true facts in the well-founded model that match the done fact have been generated, and (b) every true fact generated is true in the well-founded model. Invariant 2 ensures that when a *done* fact is generated, among those facts that match the *done* fact, all and only those facts that are not false in the well-founded model have been generated as possibly undefined. Thus, when a *done* fact is generated, by Invariants 1 and 2, the facts that match the *done* fact, and are generated as possibly undefined but not generated as true, are exactly those that are indeed undefined in the well-founded model. Hence, Invariants 1 and 2 together help ensure the soundness of the computation with respect to the well-founded model.

Invariant 3 is a technical condition ensuring that (a) when we have generated a true fact, it will have a corresponding *un* fact, and (b) for each fact $q(\bar{a})$ whose truth value has been determined (i.e., $done(q(\bar{a})) \in W$), the two indicators that it is possibly false $q(\bar{a}) \notin W$ and $un(\neg(q(\bar{a}))) \in W$ are either both present or both absent. Invariant 4 ensures that the *Context* maintains correct dependency information.

Invariant 1. (True facts) (a) $done(q(\bar{a})) \in W \rightarrow (q(\bar{a}) \in W \leftrightarrow q(\bar{a}) \in T[P])$, and (b) $q(\bar{a}) \in W \rightarrow q(\bar{a}) \in T[P]$.

Invariant 2. (False facts) $done(q(\bar{a})) \in W \rightarrow (un(q(\bar{a})) \notin W \leftrightarrow q(\bar{a}) \in F[P]$.

Invariant 3. At each fixpoint (that is, step 3 of WF-OS and step 15 of Local_Alternation) (a) $q(\bar{a}) \in W \rightarrow un(q(\bar{a})) \in W$, and (b) $done(q(\bar{a})) \in W \rightarrow (q(\bar{a}) \notin W \leftrightarrow un(\neg\, q(\bar{a})) \in W)$.

Invariant 4. When we reach step 3(ii), if $query(q(\bar{a}))$ appears marked in the last node of *Context*, and depends on $query(p(\bar{b}))$, then either

- (a) $query(p(\bar{b}))$ is also in the last node of *Context*, or
- (b) $query(p(\bar{b}))$ was on *Context* earlier and was popped from *Context*, and a corresponding fact $done(p(\bar{b}))$ is present, and $query(p(\bar{b}))$ does not depend on $query(q(\bar{a}))$.

We define notation for referring to the definitely true, false, and undefined facts given by W:

- $T[W] = \{p(\bar{b}) \mid p(\bar{b}) \in W\}$
- $F[W] = \{p(\bar{b}) \mid done(p(\bar{b})) \in W \wedge un(p(\bar{b})) \notin W\}$
- $U[W] = \{p(\bar{b}) \mid done(p(\bar{b})) \in W \wedge p(\bar{b}) \notin W \wedge un(p(\bar{b})) \in W\}$

An outline of the proof is as follows: Lemma 5.1 is a technical lemma required for Lemma 5.2. Lemma 5.2 shows that every time computation reaches the first line of Local_Alternation, the *un* facts are a superset of the true and undefined facts of the well-founded model (restricted to those in the queries of interest). This means that if $un(p(\bar{a}))$ is not present, then it is false in the well-founded model. This is used in Lemma 5.3 to show that the invariants are maintained throughout the repeat loop of Local_Alternation. Lemma 5.4 is the main lemma of the proof. It shows how Local_Alternation computes the alternating fixpoint of the subprogram of interest (all facts which have a query fact in the last *ContextNode*). This result is used in Lemma 5.5 to show that the last lines of Local_Alternation maintain the invariants. The theorem follows straightforwardly from Lemma 5.5.

Lemma 5.1. Suppose the invariants hold at a point when evaluation reaches the first line in Local_Alternation. *Let W denote the set of ground instances of all facts present at that point. Suppose query*$(p(\bar{b}))$ *is an instance of a fact in the last ContextNode, and consider any ground instance of a rule in P with head* $p(\bar{b})$. *Then, either*

(*a*) *the rule instances is made false by information in W* (*i.e., there is a negative literal* $\neg s(\bar{e})$ *such that* (*s.t.*) $s(\bar{e}) \in T(W)$, *or there is a positive literal* $s(\bar{e})$ *s.t.* $s(\bar{e}) \in F(W)$), *or*
(*b*) *the query facts for every literal in the body are in W and un*$(p(\bar{b})) \in W$, *or*
(*c*) *there is a positive literal* $r(\bar{c})$ *in the rule such that query*$(r(\bar{c}))$ *is an instance of a fact in the last ContextNode and un*$(r(\bar{c})) \notin W$.

Furthermore, un$(p(\bar{b})) \in W$ *only if there is a rule for p that is in category* (*b*) *above.*

The proof of the lemma is based on the Undef Magic rewriting presented earlier, Step 3 of WF-OS, and on Procedure Add_Undefined. Details are presented in the Appendix.

Given a set of facts S, and a set M of query facts, define S/M as follows:

$$S/M \overset{\text{def}}{=} \{p_i(\overline{a_i})\alpha \mid p_i(\overline{a_i}) \in S,\ query(p_i(\overline{b_i})) \in M,\ \alpha \text{ is a grounding substitution } s.t.\ p_i(\overline{a_i})\alpha = p_i(\overline{b_i})\alpha\}.$$

Whenever evaluation reaches the first line of Local_Alternation, evaluation has reached a fixpoint; let the ground instance of the set of facts present at the point be W. Based on Lemma 5.1, we can show that at any such point, if a fact *un*$(p(\bar{a}))$ is absent, either all rule instances defining it have at least one literal that is false based on $T(W)$ and $F(W)$, or (based on Condition (c) of Lemma 5.1) there is a set of positive literals that forms an unfounded set. Hence we have the following lemma.

Lemma 5.2. Suppose the invariants are satisfied before the start of Local_Alternation. *Every time execution reaches the first line in* Local_Alternation *for every atom* $q(\bar{a}) \in T[P] \cup U[P]$, *if query*$(q(\bar{a}))$ *is an instance of a fact in the last ContextNode, then un*$(q(\bar{a})) \in W$, *where W is the set of ground instances of facts present at that time.*

The proof is by induction on the stage of the alternating fixpoint computation when the fact is derived. Details are presented in the Appendix.

Lemma 5.3. Suppose the invariants hold at the time of a call to Local_Alternation. *During the repeat loop of procedure* Local_Alternation, *the invariants are maintained and no new query or un facts are generated* (*hence the Context does not change throughout the procedure*).

Details of the proof are presented in the Appendix.

Lemma 5.4. Suppose the invariants are satisfied before a call to Local_Alternation. *Let M be the query facts in the last ContextNode, and let N be the union of M together with all computed query facts, i.e., where query($q(\bar{a})$) and done($q(\bar{a})$) are both present.*

Let W_i be the ground instances of the set of facts present at the $(i+1)$th time evaluation reaches the first line of Local_Alternation *during the call to* Local_Alternation. *Let $T_0 = T[W_0]/(N-M)$, $U_0 = (HB_P - F[W_0])/(N-M)$, and $F_0 = F[W_0]$. Let $T_1 = T_P(HB_P - F_0)\uparrow \omega(T_0)$ and $U_1 = T_P(T_0)\uparrow \omega(U_0)$, and let $T_{i+1} = T_P(U_i)\uparrow \omega(T_0)$, $i > 0$, and let $U_{i+1} = T_P(T_i)\uparrow \omega(U_0)$, $i > 0$.*

For $n \geq 0$, $q(\bar{a}) \in W_n/N$ iff $q(\bar{a}) \in T_{n+1}/N$, and $un(q(\bar{a})) \in W_n/N$ iff $q(\bar{a}) \in U_{n+1}/N$.

The above lemma proves the main results that are needed to show the soundness of our technique. The proof is by induction on the sequence of derivations (for the "only if" direction), and by induction on the stage of alternating fixpoint at which a fact is derived (for the "if" direction). Details are presented in the Appendix.

Corollary 5.1. Suppose the invariants are satisfied before a call to Local_Alternation. *At the end of a call to* Local_Alternation, *for every fact $p(\bar{b})$ such that query($p(\bar{b})$) is in the ContextNode that is popped at the end of the procedure, $p(\bar{b})$ is present iff $p(\bar{b}) \in T[P]$, and un($p(\bar{b})$) is present iff $p(\bar{b}) \in T[P] \cup U[P]$.*

PROOF. Clearly, at the fixpoint n, T_n/M and U_n/M are the restriction of the well-founded model of $P \cup T_0 \cup \neg \cdot F_0$ to M. By invariants 1, 2, 3, and 5, $T_0 \subseteq W_P^*$ and $\neg \cdot F_0 \subseteq W_P^*$, and the result follows. □

The following lemma essentially follows from the above corollary, and from the earlier lemmas.

Lemma 5.5. Invariants 1, 2, 3, *and* 4 *are maintained by* Local_Alternation.

PROOF. Invariant 1 follows from Corollary 5.1 and Lemma 5.3; the second part follows by induction. Invariant 2 similarly follows from Corollary 5.1 and Lemma 5.3. Invariant 3 is a simple property of the rewritten program, and the definition of Local_Alternation. Invariant 4 follows trivially since the set of *query* facts does not change. □

We have not discussed the maintenance of invariants throughout the remainder of WF-OS, in particular when in Step 3 the last *ContextNode* is not marked NEGLOOP. In this case, we can easily see each of the above lemmas holds (perhaps in a vacuous manner). In effect, if Local_Alternation were applied, it would immediately terminate; hence, the invariants are maintained. The operations on *Context*, such as insertion and duplicate elimination, maintain Invariant 4, and do not affect Invariants 1 and 2. Invariant 3(a) is a simple property of the rewritten program, while Invariant 3(b) is unaffected because no *done* facts are added.

We show, based on the invariants, that WF-OS evaluation is sound. We also show partial completeness—if evaluation terminates, all facts in the well-founded model are generated, and for the case of DATALOG programs with finite base relations, evaluation does terminate.

Theorem 5.1. Given any nonfloundering program P and a terminating query $?q(\bar{t})$, WF-OS *evaluation is sound and partially complete w.r.t. the well-founded seman-*

tics of P. That is,

1. $q(\bar{t})[\theta] \in T[P]$ *iff* $q(\bar{t})[\theta]$ *is a ground instance of a fact derived by* WF-OS.
2. $q(\bar{t})[\sigma] \in U[P]$ *iff* $un(q(\bar{t}))[\sigma]$ *is a ground instance of a fact derived by* WF-OS, *and* $q(\bar{t})[\sigma]$ *is not an instance of any fact that is derived.*
3. $q(\bar{t})[\gamma] \in F[P]$ *iff* $un(q(\bar{t}))[\gamma]$ *does not unify with any fact derived by* WF-OS.

PROOF. The result holds because invariants 1, 2, and 3 are maintained throughout the operation of WF-OS, and when the procedure terminates, the *Context* is empty and thus *done*$(q(\bar{t}))$ is in the set of facts. □

6. EXTENSIONS

We presented a simple version of WF-OS for ease of exposition. Straightforward improvements include not generating *un* facts when it is clear that they are not needed (e.g., for programs without negation). A number of other improvements are discussed below.

Subsumption checking on query facts in the *Context* can be used instead of duplicate elimination, as described in [18]. In the case of Ordered Search, subsumption checking was done in "one direction" in order to maintain exact dependencies: if a query fact in a new node in *Context* subsumes a marked query fact lower in *Context*, a collapse operation is initiated. If the subsumption is in the order direction, the collapse operation is not initiated. In Ordered Search, collapsing *Context* nodes in such a situation can create spurious negative cycles in left-to-right modularly stratified programs, which cannot be handled by the evaluation. With WF-OS, the spurious negative cycles do not affect soundness or completeness, and merely affect efficiency. Hence subsumption checking can be performed in *both* directions without affecting correctness, only affecting efficiency.

Procedure Well-Founded Ordered Search is not set-oriented in making generated subgoals available for further use (although it is set-oriented in generating subgoals and answers to subgoal). The procedure can be made more set-oriented by marking a whole set of subgoals at a time (in Step 3), and collapsing the corresponding nodes in *Context* together. Unlike in Ordered Search, we can indiscriminately apply this procedure without affecting soundness or completeness, because Local_Alternation is a safe method for computing the well-founded model of any (query-closed) fragment of the program. Marking sets of facts at a time leads to more set-oriented evaluation but can significantly decrease efficiency by creating apparent negative cycles where none exist, or making the query sets to which Local_Alternation is applied larger than necessary. The trade-off between efficiency of set-oriented evaluation versus more Local_Alternation suggests marking sets of facts at a time is only worthwhile when the subprogram is positive or stratified.

Throughout the paper, we have concentrated on evaluating programs with left-to-right complete SIPS. The results easily extend to arbitrary SIPS, because query facts depend on *un* facts rather than the original predicates. Ordered Search is restricted to left-to-right SIPS since other SIP orderings may produce negative loops not present in the left-to-right order.

We presented our algorithms based on the Undef Magic Templates rewriting. Supplementary Magic Templates rewriting [6, 17] is a variant of Magic Templates rewriting, which essentially factors our subexpressions that are common to a (modified) original rule and the query rules derived from that rule. The Undef Supplementary Magic Templates rewriting is a straightforward modification of the Undef Magic Templates Rewriting that factors out common subexpressions in the query rules and *un* rules. The supplementary predicates created correspond to successive increasing prefixes of the (modified) original rule. As a result of supplementary magic rewriting, we lose the direct connection we had between the subgoals on the head of the rule and the subgoals generated for the body literals. Details of how to modify Supplementary Magic rewriting to keep track of the dependencies of subgoals can be done in a manner similar to that described in the full version of [18].

To do a well-founded ordered search using Undef Supplementary Magic Templates, we need to store with each supplementary fact the subgoal on the rule head that resulted in the generation of the fact. It is an easy modification to the well-founded ordered search algorithm to insert this information for the first supplementary fact, and to propagate the information along derivations of facts for supplementary predicates further down the rule. Given the modifications described above, Procedure Well-Founded Ordered Search can be used along with Undef Supplementary Magic Templates rewriting.

Procedure Local_Alternation is roughly equivalent to the magic-sets-based alternating fixpoint technique of [12] applied to a small part of the program. We can use the optimization of [12] suggested by [15], which permits some *query* facts to be discarded if they are found to be irrelevant due to some facts earlier (temporarily) assumed undefined being found to be either true or false. There is some extra work to recompute the set of query facts (the "magic sets"), but the set is decreasing within the alternating fixpoint, and this may save some irrelevant computation.

Another possibility for optimization is illustrated by the following example.

Example 6.1. WF-OS generates some *un* facts that are later retracted. These *un* facts are used in the context of the local alternating fixpoint on the last node of *Context*. In the following program, *query* facts are generated unnecessarily using *un* facts that are later retracted:

$r :- p, u.$
$p :- \neg q, p.$
$q :- p, s.$

The WF-OS evaluation of the above program for a query ?$r(\ )$ generates fact *query*(p), which leads to the derivation of *query*$^{\neg}(q)$, which in turn results in the derivation of *query*(p). A local fixpoint is reached, with a negative loop in the last node of *Context* containing *query*(p) and *query*$^{\neg}(q)$. At this stage, Add_Undefined adds fact *un*($\neg q$), which leads to the derivation of *un*(p), and *query*(p) (which is already present). The fact *un*(p) leads to the derivation of *query*(s) using a rule instance

$$query(s) :- query(q), un(p).$$

but also of *query*(u) using a rule instance

$$query(u) :- query(r), un(p).$$

Of these, *query*(q) is part of the cycle, and the fact *query*(s) is required in order to solve *query*(q). But r is not part of the cycle and there is no need at this point to generate *query*(u) to solve *query*(r). In terms of *Context*, the node containing *query*(r) is before the node containing *query*$^{\neg}(q)$ and *query*(p), and hence *query*(r) can be evaluated after *query*$^{\neg}(q)$ and *query*(p) are completely evaluated. Indeed, evaluation proceeds, and q is determined to be false, and so is p. However, the subgoal *query*(u) has been generated already and will be solved.

We can avoid unnecessary derivations of the above kind by delaying derivations where the *query*$(\ldots)$ fact used in the body is not part of the last *Context* node. (Rules in the Undef Magic rewritten program have at most one *query* literal in the body.) In practice, in order to coexist with semi-naive evaluation, it is easier to find that the derivation can be made, and note it without generating the fact, and to recheck the derivation when the relevant query node becomes the last node in *Context*.

While the well-founded model is not recursively enumerable in general, when restricted to DATALOG programs it has size polynomial in the size of the EDB [26], since all predicates are of fixed arity, and the only elements of the Herbrand universe are those that are explicitly in the EDB. Actually, the above argument shows that the Herbrand base of the program itself is of size polynomial in the size of the EDB. Further, the Magic rewritings can result in at most a polynomial increase in size of the Herbrand base, since they increase the number of predicates by at most a constant factor, and each new predicate has arity no more than existing ones.

We can use this to argue that WF-OS executes in a polynomial time in the size of the EDB on DATALOG programs. Between any two calls to alternating fixpoint, there are only a polynomial number of steps, since each step computes a fixpoint on the rules, or makes a new fact visible. Each step can be seen to take polynomial time. Finally, there are only a polynomial number of calls to alternating fixpoint since each all results in the addition of some *done* facts. And because of the polynomial data complexity of the whole well-founded model, the execution of a single alternating fixpoint on some subpart of the program itself takes polynomial time.

Theorem 6.1. For a fixed DATALOG program P, WF-OS *runs in polynomial time in the size of the EDB.*

Note that the *Undef Magic Rewriting* as described translates DATALOG programs to non-DATALOG programs, but this can be avoided by introducing new predicates *query_p*, *query_neg_p*, *un_p*, and *un_neg_p* and replacing the constructions *query*$(p(\bar{t}))$, *query*$^{\neg}(p(\bar{t}))$, *un*$(p(\bar{t}))$, and *un*$(\neg p(\bar{t}))$ by *query_p*$(\bar{t})$, *query_neg_p*$(\bar{t})$, *un_p*$(\bar{t})$, and *un_neg_p*$(\bar{t})$, respectively.

7. RELATED WORK

The most closely related work to that presented in this paper is SLG resolution (Chen and Warren [10] and Chen, Swift, and Warren [8]). Our work is independent of theirs, and in fact the two techniques approach the problem from different directions; while WF-OS is based on bottom-up evaluation made query directed, SLG is based on top-down evaluation made memoing. Their technique maintains instantiated rules and answers that may contain "delayed" literals. Their "delaying" step for a negative literal $\neg p(\bar{a})$ corresponds to a step where we introduce a fact $un(\neg p(\bar{a}))$. The answers with delayed literals correspond roughly to our *un* facts, but maintain dependency information.

There are three interesting differences between our techniques. The first is that when they delay a negative literal, they remove the negative dependencies that are introduced by the literal, in effect dynamically moving the literal back in the SIP order. They are thus able to relate positive cycles in unfounded sets directly to positive cycles in their dependency information. Since we do not update dependency information at the time of our equivalent to delaying, we cannot make this connection. They also optimize some of their actions by incrementally maintaining dependency information. By combining the above optimizations, they avoid using the alternating fixpoint technique. We can incorporate some of these optimizations in our technique as well, but it is not clear how we can avoid the alternating fixpoint technique since we do not maintain exact dependency information within a node of *Context* (if the program is not modularly stratified). Equally interesting is the question of whether their technique is always better than (local) alternating fixpoint or not.

The second difference is that their technique does not use exact dependency information even in the case of modularly stratified programs—a sequence of strongly connected components (SCCs) in the "depends on" relation may be merged and viewed as if it were a single SCC. This has bad consequences in cases where the need to maintain the separation of SCCs is important, as may be the case if the technique is to be extended to aggregation (even on modularly stratified programs). Equally importantly, since they do not have exact SCC information, they may delay a negative literal that is not really in a negative cycle, but appears to be in a negative cycle due to the merging of SCCs. We maintain the separation of SCCs, and are thus able to avoid "delaying literals" in some cases where they delay the literal. Thus there are cases where we compute fewer facts than they do. Recent extensions to their technique to recover exact dependency information in the case of modularly stratified programs are discussed in [23].

The third difference is that, using the optimization of [12] proposed by [15], we can recognize that some queries are irrelevant and delete them in the course of the alternating fixpoint, as we noted in Section 6. In the technique of Chen et al., once a query is generated it is never deleted, even if it is irrelevant.

Our technique performs better than that of [12] and its optimization [15] since it is able to restrict the alternating fixpoint to a subpart of the program. In parts of the program where there are no cyclic dependencies, WF-OS is able to determine the status of a fact before using it, and thereby avoid unnecessary computation caused by treating it as undefined. As a special case of the above, for modularly stratified programs, WF-OS reduces to Ordered Search, and performs no irrelevant computation and repeats no computation. Our technique is better than WELL! [7] and QSQR/SLS resolution [22] since both perform repeated computation even for programs without negation. Unlike XOLDTNF [9], our technique is able to share answers to subgoals effectively; XOLDTNF repeats computation even for modularly stratified programs. The technique of [13] is not goal directed, although they mention that they can use a restricted version of Magic sets (where no negative literals are used in query rules).

8. CONCLUSIONS AND FUTURE WORK

We extended the Ordered Search technique to handle well-founded negation. The extension essentially uses Ordered Search to find dependencies, and when a circular dependency is found, it applies the alternating fixpoint technique to compute the well-founded model for the subgoals that are involved in the cycle. Thus we are able to use the (costly) alternating fixpoint technique only if it is required. Since implementations of Ordered Search and of the alternating fixpoint technique are already available, it should be relatively straightforward to combine them.

The implementation of SLG resolution described in [8] and WF-OS have advantages and disadvantages over each other in different cases. It would be interesting to see if the benefits of both techniques can be combined. Another interesting extension would be to see if the alternating fixpoint technique can be replaced by some other technique that is more efficient (possibly by exploiting information that is generated during Ordered Search).

APPENDIX: PROOFS OF LEMMAS FROM SECTION 5

Lemma 5.1. Suppose the invariants hold at a point when evaluation reaches the first line in Local_Alternation. *Let W denote the set of ground instances of all facts present at that point. Suppose $query(p(\bar{b}))$ is an instance of a fact in the last ContextNode, and consider any ground instance of a rule in P with head $p(\bar{b})$. Then, either*

- (*a*) *the rule instance is made false by information in W (i.e., there is a negative literal $\neg s(\bar{e})$ s.t. $s(\bar{e}) \in T(W)$, or there is a positive literal $s(\bar{e})$ s.t. $s(\bar{e}) \in F(W)$), or*
- (*b*) *the query facts for every literal in the body are in W and $un(p(\bar{b})) \in W$, or*
- (*c*) *there is a positive literal $r(\bar{c})$ in the rule such that $query(r(\bar{c}))$ is an instance of a fact in the last ContextNode and $un(r(\bar{c})) \notin W$.*

Furthermore, $un(p(\bar{b})) \in W$ only if there is a rule for p that is in category (b) above.

PROOF. Consider a ground instance of a rule in P of the form

$$p(\bar{b}) \leftarrow B, [\neg]r(\bar{c}), B'.$$

Case 1: For all literals $[\neg]s(\bar{e})$ in the rule, $un([\neg]s(\bar{e})) \in W$. Now $MagUnd(P)$ contains a rewritten version of the rule using which (a fact that subsumes) $un(p(\bar{b}))$ is derived, and also has query rules such that for each literal $[\neg]s(\bar{e})$, a query fact (that subsumes) $query(s(\bar{e}))$ is generated. Condition (b) is then satisfied.

Case 2: There is a literal $[\neg]s(\bar{e})$ in the rule such that $un([\neg]s(\bar{e})) \notin W$. Let $[\neg]r(\bar{c})$ be the first such literal in the left-to-right order. We consider two subcases: (a) the literal is positive, and (b) the literal is negative. In subcase 2(a), $MagUnd(P)$ contains rules defining *query* such that $query(r(\bar{c}))$ is generated using the *un* facts from earlier literals, and the *query* fact for the head of the rule. Now, if $query(r(\bar{c}))$ is in the last *ContextNode*, Condition (c) is satisfied. Else, the query must have been solved already, since any query fact that $query(p(\bar{b}))$ depends on cannot be in an earlier *ContextNode*. Then the WF-OS algorithm must have inserted a $done(r(\bar{c}))$

fact. Since $un(r(\bar{c}))$ is not present, by Invariant 2, the literal is not satisfied, and hence Condition (a) is met. This completes subcase 2(a).

In subcase 2(b), the first such literal is a negative literal, $\neg r(\bar{c})$. A fact $query^{\neg}(r(\bar{c}))$ must then have been generated. If the fact is in the last context node, a fact $un(\neg r(\bar{c}))$ must also have been inserted in Procedure Add_Undefined. The fact cannot be present, since we are in Case 2 of this proof. But removing the fact could only happen if $done(r(\bar{c}))$ is added and $r(\bar{c})$ is present. But by Invariant 1, the literal is not satisfied, and Condition (a) is met. This completes subcase 2(b), and the proof of the first part of the lemma statement.

We note that in case 2, $un(p(\bar{b}))$ could not have been derived from the instance of the rewritten form of the rule. This proves the last part of the statement of the lemma, and completes the proof of the lemma. □

Lemma 5.2. *Suppose the invariants are satisfied before the start of* Local_Alternation. *Every time execution reaches the first line in* Local_Alternation *for every atom* $q(\bar{a}) \in T[P] \cup U[P]$, *if* $query(q(\bar{a}))$ *is an instance of a fact in the last ContextNode, then* $un(q(\bar{a})) \in W$, *where* W *is the set of ground instances of facts present at that time.*

PROOF. Let M be the set of query facts in the last *ContextNode* together with all complete query facts, i.e., $query(q(\bar{a})) \in W$ such that $done(q(\bar{a})) \in W$. We show $q(\bar{a}) \in (T[P] \cup I[P])/M = (T_P(T[P]) \uparrow \omega)/M$ implies $un(q(\bar{a})) \in W$ by induction.

The base case is trivial. Take $q(\bar{a}) \in (T_P(T[P]) \uparrow h + 1)/M$; then there exists a ground instance of a rule in P of the form

$$q(\bar{a}) :- p_1(\bar{b}_1), \ldots, p_m(\bar{b}_m), \neg r_1(\bar{c}_1), \ldots, \neg r_k(\bar{c}_k).$$

where $p_i(\bar{b}_i) \in T_P(T[P]) \uparrow h \subseteq T[P] \cup U[P]$ and $r_j(\bar{c}_j) \notin T[P]$. By Lemma 5.1, the rule instance must fall in category (a), (b), or (c). Clearly, it cannot fall in category (a) since this implies either $p_i(\bar{b}_i) \in F[W]$ and hence, by Invariant 2, $p_i(\bar{b}_i) \in F[P]$, or $r_j(\bar{c}_j) \in T[W]$ and, by Invariant 1, $r_j(\bar{c}_j) \in T[P]$. Suppose the rule falls in category (c); then one of the positive literals $p_i(\bar{b}_i)$ is such that $query(p_i(\bar{b}_i)) \in M$ and $un(p_i(\bar{b}_i)) \notin W$, but by induction, since $p_i(\bar{b}_i) \in (T_P(T[P]) \uparrow h)/M$, it must be that $un(p_i(\bar{b}_i)) \in W$. Thus the rule falls in category (b) and hence $un(q(\bar{a})) \in W$. □

Lemma 5.3. *Suppose the invariants hold at the time of a call to* Local_Alternation. *During the repeat loop of procedure* Local_Alternation, *the invariants are maintained and no new query or un facts are generated* (*hence the Context does not change throughout the procedure*).

PROOF. For Lines 3–4, by Lemma 5.2, since $un(q(a)) \notin W$, it must be that $q(a) \in F[P]$. Hence, adding $done(q(a))$ to W maintains Invariant 2. For Lines 5–7, because $q(\bar{a})$ is generated by a ground instance of a rule where $\neg p(\bar{b})$ is replaced by $done(p(\bar{b}))$, $\neg un(p(\bar{b}))$, then by Invariants 1 and 2, each of the literals in the rule is true in W_P^* and hence $q(\bar{a}) \in T[P]$, maintaining Invariant 1. The maintenance of Invariant 3 follows trivially since we are at a fixpoint, and there are rules introduced by Undef Magic rewriting that must have derived the necessary facts. The maintenance of Invariant 4 follows from the second part of this lemma, proved below.

The rules for query facts and *un* facts are positive, and depend only on the predicates *query* and *un*, except for the rules of the form $un(p(\bar{a}))\text{:-}\,p(\bar{a})$ and $un(\neg\, p(\bar{a}))\text{:-}\,done(p(\bar{a}), \neg\, p(\bar{a})$. No *un* fact not present at the call to Local_ Alternation can be created during the repeat loop unless at least one new *un* fact arises from a rule like these above, since the execution simply removes $un(\neg\, p(\bar{a}))$ facts. Suppose $q(\bar{a})$ enters W. Then it follows by the above that $q(\bar{a}) \in T[P]$ and hence, by Lemma 5.2, $un(q(\bar{a})) \in W$ at the call to Local_Alternation. Suppose $done(q(\bar{a}))$ enters W, then $un(\neg\, q(\bar{a}))$ was in W, and could not have been removed. Hence the sequence of *un* facts generated is decreasing and no new query facts are computed. It follows from the structure of Ordered Search that *Context* does not change in this period. □

Lemma 5.4. Suppose the invariants are satisfied before a call to Local_Alternation. *Let M be the query facts in the last ContextNode, and let N be the union of M together with all completed query facts, i.e., where $query(q(\bar{a}))$ and $done(q(\bar{a}))$ are both present.*

Let W_i be the ground instances of the set of facts present at the $(i+1)$th time evaluation reaches the first line of Local_Alternation *during the call to* Local_Alternation. *Let $T_0 = T[W_0]/(N-M)$, $U_0 = (HB_P - F[W_0])/(N-M)$, and $F_0 = F[W_0]$. Let $T_1 = T_P(HB_P - F_0)\uparrow \omega(T_0)$ and $U_1 = T_P(T_0)\uparrow \omega(U_0)$, and let $T_{i+1} = T_P(U_i)\uparrow \omega(T_0)$, $i > 0$, and let $U_{i+1} = T_P(T_i)\uparrow \omega(U_0)$, $i > 0$.*

For $n \geq 0$, $q(\bar{a}) \in W_n/N$ iff $q(\bar{a}) \in T_{n+1}/N$, and $un(q(\bar{a})) \in W_n/N$ iff $q(\bar{a}) \in U_{n+1}/N$.

Proof. Throughout the proof, we restrict attention to facts that match the query facts M; the results easily follow for the remaining facts matching $(N-M)$ which are unchanged throughout Local_Alternation. Clearly, for each n, $q(\bar{a}) \in T_0 \leftrightarrow q(\bar{a}) \in W_n/(N-M)$ and $q(\bar{a}) \in U_0 \leftrightarrow un(q(\bar{a})) \in W_n/(N-M)$.

We examine the base case, i.e., the conditions for T_1 and U_1, first.

We show $q(\bar{a}) \in W_0/N$ implies $q(\bar{a}) \in T_1$ by induction on the order of facts generated in W_0. Now, $q(\bar{a}) \in W_0/N$ means there exists a ground instance of a rule in $MagUnd(P)$ of the form

$$q(\bar{a})\text{:-}\,query(q(\bar{a})), p_1(\bar{b}_1), \ldots, p_m(\bar{b}_m),$$
$$done(r_1(\bar{c}_1)), \neg\, un(r_1(\bar{c}_1)), \ldots, done(r_k(\bar{c}_k)), \neg\, un(r_k(\bar{c}_k)).$$

Each $p_i(\bar{b}_i)$ entered W_0 earlier, and by induction, $p_i(\bar{b}_i) \in T_1$. Now $done(r_j(\bar{c}_j)) \in W_0$ and $un(r_j(\bar{c}_j)) \notin W_0$; hence, $r_j(\bar{c}_j) \in F[W_0]$. Consider the ground instance of a rule in P of the form

$$q(\bar{a})\text{:-}\,p_1(\bar{b}_1), \ldots, p_m(\bar{b}_m), \neg\, r_1(\bar{c}_1), \ldots, \neg\, r_k(\bar{c}_k).$$

Then clearly $q(\bar{a}) \in T_1$ because $p_i(\bar{b}_i) \in T_1$ and $r_j(\bar{c}_j) \notin HB_P - F_0$.

We show $q(\bar{a}) \in (T_P(HB_P - F_0)\uparrow h(T_0))/N$ implies $q(\bar{a}) \in W_0$ by induction on h. The base case is trivial. Suppose $q(\bar{a}) \in (T_P(HB_P - F_0)\uparrow h + 1(T_0))/N$; then there exists a ground instance of a rule in P

$$q(\bar{a})\text{:-}\,p_1(\bar{b}_1), \ldots, p_m(\bar{b}_m), \neg\, r_1(\bar{c}_1), \ldots, \neg\, r_k(\bar{c}_k).$$

such that $p_i(\bar{b}_i) \in (T_P(HB_P - F_0) \uparrow h(T_0))$, and $r_j(\bar{c}_j) \notin HB_P - F_0$. Thus $r_j(\bar{c}_j) \in F_0$ and $done(r_j(\bar{c}_j)) \in W_0$ and $un(r_j(\bar{c}_j)) \notin W_0$. By the definition of N (and Invariant 4), $p_i(\bar{b}_i) \in (T_P(HB_P - F_0) \uparrow h(T_0))/N$, and by inductive assumption, $p_i(\bar{b}_i) \in W_0$.

Consider the ground instance of a rule in *MagUnd*(P) of the form

$$q(\bar{a}) :- query(q(\bar{a})), p_1(\bar{b}_1), \ldots, p_m(\bar{b}_m),$$
$$done(r_1(\bar{c}_1)), \neg\, un(r_1(\bar{c}_1)), \ldots, done(r_k(\bar{c}_k)), \neg\, un(r_k(\bar{c}_k)).$$

Clearly, $q(\bar{a}) \in W_0$ since it would be derived by this rule instance.

We now show $un(q(\bar{a})) \in W_0/N$ implies $q(\bar{a}) \in U_1$ by induction on the order in which the *un* facts are generated in W_0. $un(q(\bar{a})) \in W_0/N$ means that either there exists a ground instance of a rule in *MagUnd*(P) of the form

$$un(q(\bar{a})) :- q(\bar{a}),$$

where $q(\bar{a}) \in W_0/N$, from which we have $q(\bar{a}) \in T_1 \subseteq U_1$, or there is a rule instance of the form

$$un(q(\bar{a})) :- query(q(\bar{a})), un(p_1(\bar{b}_1)), \ldots, un(p_m(\bar{b}_m)),$$
$$un(\neg\, r_1(\bar{c}_1)), \ldots, un(\neg\, r_k(\bar{c}_k)).$$

where $un(p_i(\bar{b}_i)) \in W_0/N$; hence, $un(p_i(\bar{b}_i)) \in U_1$ by induction, and $un(\neg\, r_j(\bar{c}_j)) \in W_0$. Either $done(r_j(\bar{c}_j)) \in W_0$ and $r_j(\bar{c}_j) \notin W_0$ Invariant 3, or $query(r_j(\bar{c}_j))$ is an instance of a query in M, and hence, in either case, $r_j(\bar{c}_j) \notin T_0 = T[W_0]/(N - M)$. Consider a ground instance of a rule in P of the form

$$q(\bar{a}) :- p_1(\bar{b}_1), \ldots, p_m(\bar{b}_m), \neg\, r_1(\bar{c}_1), \ldots, \neg\, r_k(\bar{c}_k).$$

Then, $q(\bar{a}) \in U_1$ since $p_i(\bar{b}_i) \in U_1$ by inductive assumption, and $r_j(\bar{c}_j)) \notin T_0$.

We show $q(\bar{a}) \in (T_P(T_0) \uparrow h(U_0))/N$ implies $un(q(\bar{a})) \in W_0$ by induction on h. The base case is trivial. Suppose $q(\bar{a}) \in T_P(T_0) \uparrow h + 1(U_0)$; then there exists a ground instance of a rule in P

$$q(\bar{a}) :- p_1(\bar{b}_1), \ldots, p_m(\bar{b}_m), \neg\, r_1(\bar{c}_1), \ldots, \neg\, r_k(\bar{c}_k).$$

such that $p_i(\bar{b}_i) \in T_P(T_0) \uparrow h(U_0)$ and $r_j(\bar{c}_j) \notin T_0$. Now we can show $p_i(\bar{b}_i) \in (T_P(T_0) \uparrow h(U_0))/N$, since $query(q(\bar{a})) \in N$, and we can show (by an inner level of induction) that each of $query(p_i(\bar{b}_i))$ would be generated from $query(q(\bar{a}))$. Hence, by inductive assumption, $un(p_i(\bar{b}_i)) \in W_0$. Similarly, each of $query^{\neg}(r_j(\bar{c}_j))$ is generated from $query(q(\bar{a}))$.

If $query^{\neg}(r_j(\bar{c}_j)) \notin M$, then by Invariants 4 and 2, $done(r_j(\bar{c}_j)) \in W_0$, $r_j(\bar{c}_j) \notin W_0$, and $un(\neg\, r_j(\bar{c}_j)) \in W_0$. If $query^{\neg}(r_j(\bar{c}_j)) \in M$, then $un(\neg\, r_j(\bar{c}_j)) \in W_0$ because it was added by Add_Undefined. Consider the ground instance of a rule in *MagUnd*(P) of the form

$$un(q(\bar{a})) :- query(q(\bar{a})), un(p_1(\bar{b}_1)), \ldots, un(p_m(\bar{b}_m)),$$
$$un(\neg\, r_1(\bar{c}_1)), \ldots, un(\neg\, r_k(\bar{c}_k)).$$

Clearly, $un(q(\bar{a})) \in W_0$ since it is derived by such a rule.

We have now completed the base case, and now examine the conditions for T_{n+1} and U_{n+1}.

We show $q(\bar{a}) \in W_n/N$ implies $q(\bar{a}) \in T_{n+1}$ by induction on the order of facts generated in W_n. $q(\bar{a}) \in W_n$ means there exists a ground instance of a rule in *MagUnd*(P) of the form

$$q(\bar{a}) :- query(q(\bar{a})), p_1(\bar{b}_1), \ldots, p_m(\bar{b}_m),$$
$$done(r_1(\bar{c}_1)), \neg un(r_1(\bar{c}_1)), \ldots, done(r_k(\bar{c}_k)), \neg un(r_k(\bar{c}_k)).$$

and $p_i(\bar{b}_i)$ enter W_n earlier; hence, by induction, $p_i(\bar{b}_i) \in T_{n+1}$. Now $done(r_j(\bar{c}_j)) \in W_n$ and $un(r_j(\bar{c}_j)) \notin W_n$. If $query(r_j(\bar{c}_j)) \notin M$, then $r_j(\bar{c}_j) \in U_0$ and also in U_n because these facts were never removed during Local_Alternation. Otherwise, at some W_l, $l \leq n$, we derived the fact $done(r_j(\bar{c}_j))$ either because (a) $un(r_j(\bar{c}_j)) \notin W_l$, thus $un(r_j(\bar{c}_j)) \notin W_n$ (since by Lemma 5.3 the *un* facts are decreasing) and hence $r_j(\bar{c}_j) \notin U_n$, or (b) $r_j(\bar{c}_j) \in W_l$, hence $r_j(\bar{c}_j) \in W_n$ and $un(r_j(\bar{c}_j)) \in W_n$, a contradiction. Hence $r_j(\bar{c}_j) \notin U_n$. Consider the ground instance of a rule in P of the form

$$q(\bar{a}) :- p_1(\bar{b}_1), \ldots, p_m(\bar{b}_m), \neg r_1(\bar{c}_1), \ldots, \neg r_k(\bar{c}_k).$$

Then clearly, $q(\bar{a}) \in T_{n+1}$ because $p_i(\bar{b}_i) \in T_{n+1}$ and $r_j(\bar{c}_j) \notin U_n$.

We show $q(\bar{a}) \in (T_P(U_n) \uparrow h(T_0))/N$ implies $q(\bar{a}) \in W_n$ by induction on h. The base case is trivial. Suppose $q(\bar{a}) \in (T_P(U_n) \uparrow h + 1(T_0))/N$; then there exists a ground instance of a rule in P

$$q(\bar{a}) :- p_1(\bar{b}_1), \ldots, p_m(\bar{b}_m), \neg r_1(\bar{c}_1), \ldots, \neg r_k(\bar{c}_k).$$

such that $p_1(\bar{b}_1) \in T_P(U_n) \uparrow h(T_0)$. By the definition of N, $p_i(\bar{b}_i) \in (T_P(U_n) \uparrow h(T_0))/N$, and hence, by inner inductive assumption, $p_i(\bar{b}_i) \in W_n$. Further, $r_j(\bar{c}_j) \notin U_n$ and thus, by the outer inductive assumption, $un(r_j(\bar{c}_j)) \notin W_{n-1}$. Hence we must have $done(r_j(\bar{c}_j)) \in W_n$ and $un(r_j(\bar{c}_j)) \notin W_n$.

Consider the ground instance of a rule in *MagUnd*(P) of the form

$$q(\bar{a}) :- query(q(\bar{a})), p_1(\bar{b}_1), \ldots, p_m(\bar{b}_m),$$
$$done(r_1(\bar{c}_1)), \neg un(r_1(\bar{c}_1)), \ldots, done(r_k(\bar{c}_k)), \neg un(r_k(\bar{c}_k)).$$

Clearly, $q(\bar{a}) \in W_n$.

We now show if $un(q(\bar{a})) \in W_n/N$ implies $q(\bar{a}) \in U_{n+1}$ by induction on the order in which the *un* facts are generated in W_n. $un(q(\bar{a})) \in W_n/N$ means either there exists a ground instance of a rule in *MagUnd*(P) of the form

$$un(q(\bar{a})) :- q(\bar{a})$$

where $q(\bar{a}) \in W_n/N$, from which we have $q(\bar{a}) \in T_{n+1} \subseteq U_{n+1}$, or there is a rule instance of the form

$$un(q(\bar{a})) :- query(q(\bar{a})), un(p_1(\bar{b}_1)), \ldots, un(p_m(\bar{b}_m)),$$
$$un(\neg r_1(\bar{c}_1)), \ldots, un(\neg r_k(\bar{c}_k)).$$

where $un(p_i(\bar{b}_i)) \in W_n$ and thus $p_i(\bar{b}_i) \in U_{n+1}$ by induction and $un(\neg r_j(\bar{c}_j)) \in W_n$. If $r_j(\bar{c}_j)$ does not match a query in M, then $done(r_j(\bar{c}_j)) \in W_0$ and $un(\neg r_j(\bar{c}_j)) \in W_0$ and hence $r_j(\bar{c}_j) \notin T_0$ and $r_j(\bar{c}_j) \notin T_n$. Otherwise, $un(\neg r_j(\bar{c}_j)) \in W_n$ implies $r_j(\bar{c}_j) \notin W_{n-1}$ and, by outer induction, $r_j(\bar{c}_j) \notin T_n$. Consider the ground instance of a rule in

P of the form

$$q(\bar{a}) :- p_1(\bar{b}_1), \ldots, p_m(\bar{b}_m), \neg r_1(\bar{c}_1), \ldots, \neg r_k(\bar{c}_k).$$

Then $q(\bar{a}) \in U_{n+1}$ since $p_i(\bar{b}_i) \in U_{n+1}$ and $r_j(\bar{c}_j) \notin T_n$.

We show $q(\bar{a}) \in (T_P(T_n) \uparrow h(U_0))/N$ implies $un(q(\bar{a})) \in W_n$ by induction on h. The base case is trivial. Suppose $q(\bar{a}) \in (T_P(T_n) \uparrow h + 1(U_0))/N$; then there exists a ground instance of a rule in P

$$q(\bar{a}) :- p_1(\bar{b}_1), \ldots, p_m(\bar{b}_m), \neg :r_1(\bar{c}_1), \ldots, \neg r_k(\bar{c}_k).$$

such that $p_i(\bar{b}_i) \in T_P(T_n) \uparrow h(U_0)$. We can again show by induction that each of $query(p_i(\bar{b}_i)) \in N$, and by inductive assumption, each $p_i(\bar{b}_i) \in W_n$, and hence also $un(p_i(\bar{b}_i)) \in W_n$. Further, $r_j(\bar{c}_j) \notin T_n$. If $query(r_j(\bar{c}_j)) \notin M$, then by Invariants 4 and 2, $done(r_j(\bar{c}_j)) \in W_0$, $r_j(\bar{c}_j) \notin W_0$, and $un(\neg r_j(\bar{c}_j)) \in W_0$, and hence also in W_n.

If $r_j(\bar{c}_j)$ corresponds to a query fact in M, then $un(\neg r_j(\bar{c}_j)) \in W_0$ because it was added by Add_Undefined and it is only removed if $r_j(\bar{c}_j) \in W_n$; but that would imply $r_j(\bar{c}_j) \in T_n$. Consider the ground instance of a rule in *MagUnd*(P) of the form

$$\begin{aligned} un(q(\bar{a})) :- {} & query(q(\bar{a})), un(p_1(\bar{b}_1)), \ldots, un(p_m(\bar{b}_m)), \\ & un(\neg r_1(\bar{c}_1)), \ldots, un(\neg r_k(\bar{c}_k)). \end{aligned}$$

Clearly, $un(q(\bar{a})) \in W_n$.

This completes the proof. □

We would like to thank Divesh Srivastava for useful discussions on some details of the well-founded ordered search algorithm. We would also like to thank Weidong Chen for discussions that helped us understand better the relationship between SLG resolution and our technique, and for pointing out an error in our original presentation of the WF-OS algorithm.

REFERENCES

1. Apt, K. R., Blair, H. A., and Walker, A., Towards a Theory of Declarative Knowledge, in: J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, Los Altos, CA, 1988, pp. 89–148.
2. Balbin, I. and Ramamohanarao, K., A Generalization of the Differential Approach to Recursive Query Evaluation, *Journal of Logic Programming* 4(3) (1987).
3. Bancilhon, F., Naive Evaluation of Recursively Defined Relations, in: Brodie and Mylopoulos (eds.), *On Knowledge Base Management Systems—Integrating Database and AI Systems*, Springer-Verlag, Berlin/New York, 1985.
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 ACM Symposium on Principles of Database Systems*, Cambridge, MA, Mar. 1986, pp. 1–15.
5. Baral, C. and Subrahmanian, V. S., Dualities between Alternate Semantics for Logic Programming and Nonmonotonic Reasoning, in: *Proceedings of the 1st International Workshop on Logic Programming and Non-Monotonic Reasoning*, MIT Press, Cambridge, MA, 1991, pp. 69–86.
6. Beeri, C. and Ramakrishnan, R., On the Power of Magic, in: *Proceedings of the ACM Symposium on Principles of Database Systems*, 1987, pp. 269–283.

7. Bidoit, N. and Legay, P., WELL! An Evaluation Procedure for All Logic Programs, in: *Proceedings of the International Conference on Database Theory*, Dec. 1990, pp. 335–348.
8. Chen, W., Swift, T., and Warren, D. S., Efficient Top-Down Computation of Queries under the Well-Founded Semantics, Technical Report 93-CSE-33, Southern Methodist University, Aug. 1993.
9. Chen, W. and Warren, D. S., A Goal-Oriented Approach to Computing the Well Founded Semantics, in: *Proceedings of the Joint International Conference and Symposium on Logic Programming*, 1992, pp. 589–606.
10. Chen, W. and Warren, D. S., Query Evaluation under the Well-Founded Semantics, in: *Proceedings of the ACM Symposium on Principles of Database Systems*, 1993.
11. Kemp, D., Srivastava, D., and Stuckey, P., Magic Sets and Bottom-Up Evaluation of Well-Founded Models, in: *Proceedings of the International Logic Programming Symposium*, 1991, pp. 337–351.
12. Kemp, D., Srivastava, D., and Stuckey, P., Query Restricted Bottom-Up Evaluation of Normal Logic Programs, in: *Proceedings of the Joint International Conference and Symposium on Logic Programming*, 1992, pp. 288–302.
13. Leone, N. and Rullo, P., Safe Computation of the Well-Founded Semantics of DATALOG Queries, *Information Systems* 17(1):17–31 (1992).
14. Lloyd, J. W., *Foundations of Logic Programming*, 2nd edition, Springer-Verlag, Berlin/New York, 1987.
15. Morishita, S., An Alternating Fixpoint Tailored to Magic Programs, in: *Proceedings of the 1993 ACM Symposium on Principles of Database Systems*, 1993.
16. Przymusinski, T. C., On the Declarative Semantics of Stratified Deductive Databases, in: J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, Los Altos, CA, 1988, pp. 193–216.
17. Ramakrishnan, R., Magic Templates: A Spellbinding Approach to Logic Programs, in: *Proceedings of the International Conference on Logic Programming*, 1988, pp. 140–159.
18. Ramakrishnan, R., Srivastava, D., and Sudarshan, S., Controlling the Search in Bottom-Up Evaluation, in: *Joint International Conference and Symposium on Logic Programming*, 1992, pp. 273–287.
19. Ramakrishnan, R., Srivastava, D., and Sudarshan, S., Controlling the Search in Bottom-Up Evaluation. Full version of [18] submitted, 1993.
20. Ross, K. A., A Procedural Semantics for Well-Founded Negation in Logic Programs, in: *Proceedings of the ACM Symposium on Principles of Database Systems*, 1989.
21. Ross, K. A., Modular Stratification and Magic Sets for DATALOG Programs with Negation, in: *Proceedings of the ACM Symposium on Principles of Database Systems*, 1990, pp. 161–171.
22. Ross, K. A., The Semantics of Deductive Databases, Ph.D. Thesis, Department of Computer Science, Stanford University, Aug. 1991.
23. Swift, T., Efficient Evaluation of General Logic Programs, Ph.D. Thesis, State University of New York at Stony Brook, Dec. 1994.
24. Tamaki, H. and Sato, T., OLD Resolution with Tabulation, in: *Proceedings of the Third International Conference on Logic Programming* (*LNCS 225*), Springer-Verlag, Berlin, 1986, pp. 84–98.
25. Van Gelder, A., The Alternating Fixpoint of Logic Programs with Negation, in: *Proceedings of the ACM Symposium on Principles of Database Systems*, 1989, pp. 1–10.
26. Van Gelder, A., Ross, K., and Schlipf, J. S., Unfounded Sets and Well-Founded Semantics for General Logic Programs, *Journal of the ACM* 38(3):620–650 (1991).
27. Vieille, L., Recursive Query Processing: The Power of Logic, *Theoretical Computer Science* 1–53 (1989).