Well-Founded Ordered Search (Extended Abstract)

Peter J. Stuckey[1] and S. Sudarshan[2]

[1] Department of Computer Science, University of Melbourne
Parkville 3052, Australia
pjs@cs.mu.oz.au
[2] AT&T Bell Labs., 600 Mountain Ave.
Murray Hill, NJ 07974, U.S.A.
sudarsha@research.att.com

Abstract. 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.

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. The well-founded semantics [19] is the leading candidate among the three-valued semantics that have been proposed. The well-founded semantics is non-trivial to compute; in particular, it is non-trivial to make the computation 'goal-directed', that is, given a query on a program, 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 [18], were not goal-directed. Other techniques, such as that of Ross [14], were goal-directed, but (as with Prolog) can repeat computation of subgoals multiple times and, worse, were non-effective (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 [1, 12, 17]. Several attempts have been made at extending some of these for computing the well-founded semantics. These past attempts have the problem that either computation is not completely goal-directed [9, 7, 8, 11] since some facts that are

irrelevant to the computation may be generated, or compute only relevant facts, but may compute some of them multiple times [5]. We present more details on related work in Section 4. But, in particular, for the important special case of modularly stratified programs [15], these techniques are less efficient than special purpose techniques such as Ordered Search [13].

Although Ordered Search is more efficient than the general purpose evaluation techniques proposed in the past, as described in [13] 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 (non-floundering) 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 [6, 4] 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 4.

1.1 Background

Due to space limitations we omit background material from this extended abstract. We assume familiarity with logic programming terminology (see [10]) and the issues involved in the bottom-up evaluation of logic programs. In particular, we assume the reader is familiar with Magic Templates rewriting [12] (however, we give an example of its use), and with Semi-Naive bottom-up evaluation [1].

For the purposes of this paper, a *program* is a set of definite clauses with possible use of negation in the rule body. We assume familiarity with the modularly stratified semantics [15], the well-founded semantics [19] and the alternating fixpoint evaluation method for computing the well-founded semantics [18, 7]. We assume that the programs we evaluate are *non-floundering*, i.e., any subgoal set up on a negative literal is ground. We say that a subgoal $?p(\overline{s})$ *depends on* a subgoal $?q(\overline{t})$ if there is an instantiation of a rule with $p(\overline{s})$ as the head, $q(\overline{t})$ as a body literal, and every literal preceding $q(\overline{t})$ is either true or undefined in the well-founded model of the program.

2 Ordered Search

We now describe the Ordered Search evaluation method [13], which is applicable to modularly stratified programs. The Ordered Search evaluation algorithm [13] has two phases. The first rewrites the program at compile time. The second evaluates the rewritten program.

2.1 Modified Magic Templates Rewriting

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

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

the modified Magic Templates rewriting [13] of the rule generates the following rules:

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

The first rule is basically the original rule, but with two modifications. First, 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 in Magic Templates rewriting to avoid generating irrelevant facts. Second, a literal $done(q(Y))$ has been added to the rule to guard the $\neg q(Y)$ literal . A fact $done(q(a))$ is created when Ordered Search decides that all answers to the query $?q(a)$ have been generated. 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 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.

2.2 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 refer the reader to [13] for details. The algorithm makes inferences from the rewritten rules, using the incremental evaluation idea of Semi-Naive evaluation. But unlike normal Semi-Naive evaluation it orders the use of generated subgoals in a manner somewhat like Prolog, but with 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 $ContextNodes$. Each $ContextNode$ has an associated set of query facts and each query fact is associated with a unique $ContextNode$. 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} and query $?q(\overline{a})$.

Output: Answers to $?q(\overline{a})$.

1. Insert a seed query fact $query(q(\overline{a}))$.
2. Repeat
   Repeat
   Evaluate the rules of the program using Semi-Naive evaluation.
   However, instead of inserting newly generated facts $query(p(\overline{b}))$
   into the $query$ relation,
   2(a) insert them in $Context$ and
   2(b) perform duplicate elimination as described later.
   /* Consequently they are hidden from the evaluation. */
   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 to the evaluation, as usual in Semi-Naive evaluation. When a fact in $Context$ is made available to the evaluation, it is said to be "marked" in the $Context$. A $ContextNode$ is said to be "marked" if any fact associated with the $ContextNode$ is marked.

We now intuitively describe some of the operations performed in Step 2 of the above algorithm

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

1. If $done(q(\overline{a}))$ is present do not insert $query(q(\overline{a}))$ in $Context$ (since it has been fully evaluated already).
2. Else, $query(p(\overline{b}))$ must be in the $Context$ and must be marked since it is visible and has just been used to derive $query(q(\overline{a}))$. Insert $query(q(\overline{a}))$ in a new unmarked $ContextNode$ immediately before the next marked $ContextNode$ following the marked node containing $query(p(\overline{b}))$. (If there is no such marked $ContextNode$, $query(q(\overline{a}))$ is inserted as the last $ContextNode$ in the $Context$.)

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

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

(ii) If the unmarked copy of $query(q(\overline{a}))$ occurs after the marked copy, $query(q(\overline{a}))$ depends on itself. We have thus detected a cyclic dependency between the set of all marked facts in $Context$ in between the two

occurrences of $query(q(\overline{a}))$. Ordered Search deletes the unmarked copy of $query(q(\overline{a}))$ and collapses the above set of marked facts into the node of the marked copy of $query(q(\overline{a}))$ in $Context$.

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 insertion step 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 subgoals 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.

3. Making Facts Visible This step makes 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 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 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. Once all subgoals in the last $ContextNode$ are determined to have been fully evaluated, the node is removed from $Context$, and for each subgoal $query(p(\overline{a}))$ in the node, a fact $done(p(\overline{a}))$ is created.

A major difference between Ordered Search and Well-Founded Ordered Search, which we describe in Section 3, is in the above step.

3 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.

3.1 The Undef Magic Templates Rewriting

We now give the intuition behind the *Undef Magic Rewriting*, our extension of Magic Templates rewriting [12] 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 [7]. For example, with a rule $p :- \neg p, q$, and no rule defining q, the truth value of q

is needed in order to determine that p is false; a subgoal $?q$ must be generated to find the truth status of q, at a point when the truth status of $\neg p$ 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 $un(p(\overline{a}))$ and $un(\neg q(\overline{a}))$. These facts respectively indicate that $p(\overline{a})$ is possibly true (i.e., has not been shown to be false), and $q(\overline{a})$ 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. We say a fact $p(\overline{a})$ is *possibly undefined* if a fact $un(p(\overline{a}))$ is present.

We consider again the rule used to describe Ordered Search:

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

Undef Magic rewriting of this rule generates the following rules:

$$\begin{array}{lll}
query(r(X,Y)) & \text{:-} & query(p(X)). \\
query^{\neg}(q(Y)) & \text{:-} & query(p(X)), un(r(X,Y)). \\
query(s(Y)) & \text{:-} & query(p(X)), un(r(X,Y)), un(\neg q(Y)). \\
un(p(X)) & \text{:-} & query(p(X)), un(r(X,Y)), un(\neg q(Y)), un(s(Y)). \\
p(X) & \text{:-} & query(p(X)), r(X,Y), done(q(Y)), \neg un(q(Y)), s(Y).
\end{array}$$

Further, for every predicate $p(\overline{X})$ (including base predicates) we generate rules

$$\begin{array}{lll}
un(p(\overline{X})) & \text{:-} & p(\overline{X}). \\
un(\neg p(\overline{X})) & \text{:-} & done(p(\overline{X})), \neg p(\overline{X}).
\end{array}$$

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 $un(\ldots)$ 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 $query(\ldots)$ as seperate 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 $un(\ldots)$ 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 presented in the full version of the paper. The rewriting of a program P, denoted $MagUnd(P)$, is the union of the rewriting of each of its rules.

An inspection of the above rules indicates that a fact of the form $un(p(\overline{a}))$ can be generated using the rules only if there is already a fact $p(\overline{a})$. 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.

3.2 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 2.2, except that (a) the Undef Magic rewriting is used instead of Magic rewriting, (b) Steps 2(b) and 3 of the evaluation algorithm are modified as follows:

2(b). Duplicate elimination Unmarked copies of $query(q(\overline{a}))$ and $query^{\neg}(q(\overline{a}))$ are treated as distinct objects, and only the latest unmarked copy of each is retained. If there is a marked copy and an unmarked copy of $query^{[\neg]}(q(\overline{a}))$ (with or without tag '$\neg$') in $Context$, there are two possibilities:

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

3. Making Facts Visible

(i) While the last node in $Context$ has an unmarked query fact,
Choose an unmarked fact from the last node
If no marked (tagged or untagged) copy of the fact appears earlier in context
then found = 1; break
else found = 0; perform duplicate elimination (Step 2(b)(ii))
If (found == 1), mark the chosen fact and make it available to the evaluation by inserting it (sans 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 $query^{[\neg]}(p(\overline{a}))$ in the node, a fact $done(p(\overline{a}))$ is created.
Otherwise execute Procedure Add_Undefined. If no new facts are added by Add_Undefined, execute Procedure Local_Alternation.

The intuition behind the above is that if 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 $un(\neg q(\overline{a}))$.

Procedure Add_Undefined

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

In case some new $un(\ldots)$ facts is 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 has been 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 (as we prove in the full version of the paper). 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 [18, 7] 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 subprogram containing relevant facts,[3] we can reduce the time cost of computation considerably.

Procedure Local_Alternation

1. Repeat
2. For every query fact $query^{\neg}(q(\overline{a}))$ in the last $ContextNode$,
3. If $un(q(\overline{a}))$ is not present /* $q(\overline{a})$ is definitely false */
4. Add $done(q(\overline{a}))$ to the set of facts.
5. If $q(\overline{a})$ is present /* $q(\overline{a})$ is true */
6. Add $done(q(\overline{a}))$ to the set of facts.
7. Remove $un(\neg q(\overline{a}))$
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 */

[3] 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.

11. For every query fact $query(q(\overline{a}))$ that is in the last $ContextNode$,
and $done(q(\overline{a}))$ is not present
12. Remove all facts $un(q(\overline{a}, ...))$.
13. /* Note: Facts $un(\neg q(\overline{a}))$ are not removed at this step. */
14. Fire all rules that define un-predicates in the current 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 $query(q(\overline{a}))$ in the node,
20. Add a fact $done(q(\overline{a}))$ to the set of facts.

Example 1. Consider the following simple two-rule program.

p :- $\neg q, r$ $\qquad\qquad$ q :- $\neg p$

WF-OS evaluation starting with $query(p)$ would generate $query^{\neg}(q)$, which is then marked, and in turn generates $query^{\neg}(p)$. The facts appear in consecutive nodes in $Context$. Step 2(b)(ii) collapses all the above facts into the node of $query(p)$, and the node gets marked NEGLOOP. No more facts can be generated, and facts $un(\neg p)$ and $un(\neg q)$ are introduced by Add_Undefined.. A fact $query(r)$ then gets derived and put in a new context node, and gets marked, but generates no more subgoals/answers. The node is removed from $Context$. An iteration of Local_Alternation then derives q but not p, which means p is false. In the next iteration q is derived again. Alternating fixpoint then terminates with p false and q true. After the node containing $\{query^{\neg}(q), query^{\neg}(p), query(p)\}$ is removed, $Context$ is empty, and evaluation terminates. Thus r and p are false, and q is true in the well-founded model of the program. □

Theorem 1. *Let $T[P], F[P]$ and $U[P]$ denote the true, false and undefined facts in the well-founded semantics of program P. Given any non-floundering program P, and a terminating query $?q(\bar{t})$,* WF-OS *evaluation is sound and partially complete w.r.t. the well-founded semantics 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.

3.3 Extensions

We presented a simple version of WF-OS for ease of exposition. Straightforward improvements include not using *un* predicates for base predicates, and not generating *un* facts when it is clear that they are not needed (e.g. for programs without negation). Although our description of Undef Magic rewriting assumed a left-to-right evaluation order, WF-OS can be extended to handle arbitrary evaluation orders. The idea of common-subexpression elimination used in Supplementary Magic Templates rewriting [2, 12] can also be used to derive a "Supplementary" version of the Undef Magic Templates rewriting.

Procedure Local_Alternation is roughly equivalent to the magic sets based alternating fixpoint technique of [8] applied to a small part of the program. We can use the optimization of [8] suggested by [11], 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.

4 Related Work

The most closely related work is SLG resolution (Chen and Warren [6] and Chen, Swift and Warren [4]). 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 $un(\neg p(\bar{a}))$.

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. 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.

The second difference is that their technique does not use exact dependency information — a sequence of SCCs in the depends on relation may be merged and viewed as if it were a single SCC. As a result 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.

The third difference is that using the optimization of [8] proposed by [11] we can recognize that some queries are irrelevant and delete them in the course of the alternating fixpoint, as we noted in Section 3.3. 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 [8] and its optimization [11] 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 them 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! [3] and QSQR/SLS resolution [16] since both perform repeated computation even for programs without negation. Unlike XOLDTNF [5] our technique is able to share answers to subgoals effectively; XOLDTNF repeats computation even for modularly stratified programs. The technique of [9] 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).

Acknowledgements We would like to thank Divesh Srivastava and Weidong Chen for useful discussions.

References

1. I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *Journal of Logic Programming*, 4(3), September 1987.
2. Catriel Beeri and Raghu Ramakrishnan. On the power of Magic. In *Procs. of the ACM Symp. on Principles of Database Systems*, pages 269–283, Mar. 1987.
3. N. Bidoit and P. Legay. WELL! An evaluation procedure for all logic programs. In *Procs. of the International Conf. on Database Theory*, pages 335–348, Dec. 1990.
4. Weidong Chen, Terrance Swift and David S. Warren. Efficient Top-Down Computation of Queries under the Well-Founded Semantics Tech. Report 93-CSE-33, Southern Methodist University, Aug. 1993.
5. Weidong Chen and Davis S. Warren. A goal-oriented approach to computing the well founded semantics. In *Procs. of the Joint Int'l Conf. and Symp. on Logic Programming*, 589–606, 1992.
6. Weidong Chen and Davis S. Warren. Query Evaluation under the Well-Founded Semantics. In *Procs. of the ACM Symp. on Principles of Database Systems* 1993.
7. David Kemp, Divesh Srivastava, and Peter Stuckey. Magic sets and bottom-up evaluation of well-founded models. In *Procs. of the International Logic Programming Symposium*, 337–351, 1991.
8. David Kemp, Divesh Srivastava, and Peter Stuckey. Query restricted bottom-up evaluation of normal logic programs. In *Procs. of the Joint Int'l Conf. and Symp. on Logic Programming*, 288–302, 1992.
9. Leone, N. and Rullo, P. Safe computation of the well-founded semantics of DATALOG queries. *Information Systems* 17(1) (1992), 17–31.
10. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd ed., 1987.
11. Morishita, S. An alternating fixpoint tailored to magic programs. In *Procs. of the 1993 ACM Symp. on Principles of Database Systems*, 1993.
12. Raghu Ramakrishnan. Magic Templates: A spellbinding approach to logic programs. In *Procs. of the International Conf. on Logic Programming*, 140–159, 1988.
13. Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. Controlling the search in bottom-up evaluation. In *Joint Int'l Conf. and Symp. on Logic Programming 1992*, 273–287, 1992.
14. Kenneth A. Ross A procedural semantics for well-founded negation in logic programs. In *Procs. of the ACM Symp. on Principles of Database Systems* (1989).
15. Kenneth A. Ross. Modular Stratification and Magic Sets for DATALOG programs with negation. In *Procs. of the ACM Symp. on Principles of Database Systems*, 161–171, 1990.
16. Kenneth A. Ross. *The Semantics of Deductive Databases*. Ph.D. thesis, Department of Computer Science, Stanford University, Aug. 1991.
17. H. Tamaki and T. Sato. OLD resolution with tabulation. In *Procs. of the Third International Conference on Logic Programming* (LNCS 225), 84–98, 1986.
18. A. Van Gelder. The alternating fixpoint of logic programs with negation. In *Procs. of the ACM Symp. on Principles of Database Systems*, 1–10, 1989.
19. A. Van Gelder, K. Ross, and J. S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

This article was processed using the LaTeX macro package with LLNCS style