Rule Ordering in Bottom-Up Fixpoint Evaluation of Logic Programs

Raghu Ramakrishnan* Divesh Srivastava
S. Sudarshan†

Computer Sciences Department, University of Wisconsin-Madison, WI 53706, U.S.A.

Abstract

Logic programs can be evaluated bottom-up by repeatedly applying all rules, in "iterations", until the fixpoint is reached. However, it is often desirable — and in some cases, e.g. programs with stratified negation, even necessary to guarantee the semantics — to apply the rules in some order.

An important property of a fixpoint evaluation algorithm is that it does not repeat inferences; we say that such algorithms have the *semi-naive property*. The semi-naive algorithms in the literature do not address the issue of how to apply rules in a specified order while retaining the semi-naive property. We present two algorithms; one (GSN) is capable of dealing with a wide range of rule orderings but with a little more overhead than the usual semi-naive algorithm (which we call BSN). The other (OSN, and a variant, PSN) handles a smaller class of rule orderings, but with no overheads beyond those in BSN. This smaller class is sufficiently powerful to enforce the ordering required to implement stratified programs.

We demonstrate that rule orderings can offer another important benefit: by choosing a good ordering, we can reduce the number of rule applications (and thus joins). We present a theoretical analysis of rule orderings. In particular, we identify a class of orderings, called cycle-preserving orderings, that minimize the number of rule applications (for all possible instances of the base relations) with respect to a class of orderings called fair orderings. We also show that while non-fair orderings may do a little better on some data sets, they can do much worse on others; this suggests that it is advisable to consider only fair orderings in the absence of additional information that could guide the choice of a non-fair ordering.

We conclude by presenting performance results that bear out our theoretical analyses.

*The work of R. Ramakrishnan was supported in part by a David and Lucile Packard Foundation Fellowship in Science and Engineering, an IBM Faculty Development Award and NSF grant IRI-8804319.

†The work of D. Srivastava and S. Sudarshan was supported by NSF grant IRI-8804319. The authors' e-mail addresses are {raghu,divesh,sudarsha}@cs.wisc.edu.

1 Introduction

There are essentially two components to fixpoint algorithms that preserve the semi-naive property. The first is a rewriting of the program that defines "differential" versions of predicates, in order to distinguish facts that have been newly generated (and not yet used in inferences). The second component is a technique to apply the rewritten rules and update these differentials, ensuring that all derivations are made exactly once. Semi-naive algorithms have been proposed by several researchers (e.g., [B85, BR87]). These algorithms evaluate the fixpoint in an iterative fashion, with every rule applied once in each iteration. In these algorithms, facts generated in an iteration can be used to generate other facts only in subsequent iterations.

We present two fixpoint evaluation algorithms, General Semi-Naive (GSN) and Ordered Semi-Naive (OSN); the latter has a simple variant called Predicate-Wise Semi-Naive (PSN). These algorithms can use any of the semi-naive rewriting techniques proposed earlier (e.g., [B85, BR87]) with minor modifications. GSN applies a rule to produce new facts, and then immediately makes these facts available to subsequent applications of other rules (possibly in the same iteration). PSN makes facts generated for a predicate p available after all rules defining p have been applied.

Rule orderings are significant for three distinct reasons. First, they are sometimes required in order to compute the answers correctly. For example, in stratified programs, lower strata must be evaluated first, and this simple ordering can become much more complex once we rewrite the program using Magic Sets (which is an important technique used widely to avoid irrelevant inferences). Second, rule ordering can result in increased efficiency. It is recognized that evaluating a program clique-by-clique offers significant advantages. Again, this implies a rule ordering, especially if we wish to use the clique structure of the original program rather than the Magic rewritten program. An important

contribution of this paper is to demonstrate that rule orderings can also improve efficiency by reducing the number of rule applications. In effect, since the number of inferences remains constant — all semi-naive algorithms are optimal in this respect — this means that the processing becomes more set-oriented, with each rule application generating more tuples. Finally, rule orderings have been proposed to prune redundant derivations and to allow the user to specify a desired semantics [H87, H88]).

We use control expressions, in the form of regular expressions over rules, to specify orderings over the application of rules.

We begin by examining the use of rule ordering and presenting new semi-naive algorithms for evaluating the fixpoint. The GSN algorithm deals with a large set of control expressions; we examine its use both in clique-by-clique evaluation and in evaluating a single clique (or *strongly connected component, SCC*). We describe OSN and its role in clique-by-clique evaluation. OSN can also be used to order rules within a clique by slightly refining our treatment of control expressions; for clarity, we instead present a simple variant, called PSN, that does this. Compared to traditional semi-naive algorithms for evaluating a single clique, PSN has an important advantage — it can considerably reduce the number of rule applications, while requiring no additional overheads. GSN has the potential for reducing the number of rule applications even further, but at the cost of some overheads.

In the second part of our paper, we study rule orderings in detail, and establish a close connection between cycles in rule graphs and orderings that minimize the number of iterations and rule applications. We define what it means for a rule ordering to preserve a simple cycle, and prove that a rule ordering that preserves all simple cycles in the rule graph (if such an ordering exists) is optimal within a certain class of rule orderings, in minimizing the number of iterations, and hence the number of rule applications.

In the third part of our paper, we present a summary of a performance study that underscores the importance of utilizing facts early, and choosing a good rule ordering, in reducing the number of iterations, rule applications and joins.

1.1 Related Work

Lu [L87] and Kabler et al. [KIC89] considered how facts could be (partially) utilized in the same iteration that they were generated, in the context of transitive closure algorithms, and noted that this reduced the number of disk I/Os. Kuittinen et al. [KNSS89] proposed a fixpoint evaluation algorithm for logic programs based on the immediate utilization of facts. The results presented in our paper and in [KNSS89] were obtained independently. (However, we have drawn upon and extended their performance evaluation.) Their algorithm also reduces the number of iterations, and is dependent upon a choice of rule orderings. They do not analyze the effect of rule orderings theoretically. Although their technique avoids repeating most derivations, it does not have the semi-naive property since it is possible for some derivations to be repeated. This affects performance. Further, it makes the algorithm inapplicable when the semi-naive property is required for certain further optimizations. (For example, if a program has the *duplicate-freedom* property, a fixpoint algorithm with the semi-naive property can be modified easily to eliminate run-time checks for duplicates [MR89].) To the best of our knowledge, no one has considered the efficient implementation of general rule orderings.

2 Motivation

We examine two uses of rule orderings, for ensuring a desired semantics and for achieving efficiency. We do not explore the use of control expressions as a user-specified form of control that refines the logic of the program. (The separation of logic and control as a programming paradigm has been widely advocated in the logic programming literature, and our techniques provide the basis for exploring this paradigm using bottom-up evaluation.)

2.1 Rule Ordering For Semantics

Stratified programs have been identified as a class of logic programs that can be efficiently evaluated using a bottom-up fixpoint evaluation mechanism. In a program with stratified negation, predicates are partitioned into layers, or strata. Such a program can be evaluated stratum by stratum. The fixpoint of all rules defining predicates in lower strata must be evaluated before the rules defining the predicates in a stratum are applied. This ordering on rule applications becomes considerably more complex when the Magic Sets technique is used to restrict the computation: the rewritten version of a stratified program may not be stratified, and is equivalent to the original program only when the rules in it are applied in a particular order (which reflects the structure of the Magic Sets transformation as well as the rule ordering implicit in stratification). Beeri et al. [BRSS89] present an algorithm to identify this order and to specify it through *control expressions*, in the form of regular expressions over rules.

Several evaluation strategies (e.g., the LDL system, and the technique of Balbin et al.[BPRM] for evaluating stratified programs) for logic programs use a built-in control strategy to implicitly order rules. Using such a default control strategy results in the control aspect becoming closely linked with the logic of the program. An important advantage of making the control strategy explicit (say, in the form of control expressions over rules) is that it separates the control aspect of logic programs from its declarative semantics, and

exposes the possibility of using other equivalent, possibly more efficient, rule orderings.

2.2 Rule Ordering For Efficiency

Besides implementing desired semantics, ordering of rules can also be used to reduce the cost of bottom-up evaluation of logic programs.

The number of inferences has been widely used as a cost metric in the evaluation of logic programs. However, any evaluation technique that has the semi-naive property makes each inference that can be made, exactly once, and hence all the techniques we study are equivalent under this criterion. Algorithms having the semi-naive property differ in how inferences are distributed across iterations. However the cost measure of the number of inferences hides many implementation costs, and we desire more accurate cost criteria that refine the criterion of the number of inferences.

One of the advantages of bottom-up evaluation of logic programs is the increased degree of set-at-a-time computation. Given that the total number of inferences made by two different evaluation techniques is identical, the technique that performs more set-at-a-time computation is expected to perform better. If more inferences are made using the set of tuples in a page fetched from disk, the number of I/O operations can be expected to reduce considerably. This can be seen from the results of Lu [L87], who considered how facts could be (partially) utilized in the same iteration that they were generated (thus reducing the total number of iterations and hence increasing the set-at-a-time computation in each iteration), in the context of transitive closure algorithms, and noted that this reduced the number of disk I/Os. Thus, the greater the number of inferences made in a single rule application, the lesser is the number of I/O operations expected (given that the total number of inferences made does not change).

The fixpoint evaluation techniques we describe (GSN and PSN) use facts computed by a rule application in the application of other rules within the same iteration, while preserving the semi-naive property. Our theoretical analysis and performance results show that these techniques can greatly reduce the number of rule applications, iterations, and thus the number of joins needed to reach the fixpoint in a sequential computation. Since the number of inferences made by each of the evaluation strategies is the same while the number of joins is reduced, the degree of set-orientedness in the processing of the program is increased by these techniques and hence, the I/O costs can be expected to reduce.

Associated with each join are several fixed overheads, e.g., possible accessing from (and storing to) secondary storage the relations involved in the join, and with each iteration there are other overheads, e.g., updating the various predicate extensions. The reduction in the number of rule applications, iterations and joins can be expected to reduce the costs due to the various fixed overheads associated with each iteration and with each join. The reduction in cost due to ordering of rules is orthogonal to other techniques of reducing the overall cost, such as merging different semi-naive versions of rules, efficient join and indexing strategies, and duplicate elimination techniques—none of these is made inapplicable by ordering rules.

3 Background

3.1 Definitions

Consider the computation of a fact $p(\overline{c})$ using a rule R[1]. A *derivation step* for $p(\overline{c})$ consists of the rule R and a fact for each body predicate occurrence, such that $p(\overline{c})$ can be derived using the rule and only the given fact for each body predicate occurrence. A *derivation tree* for a fact $p(\overline{c})$ is defined as below. The leaf nodes of the tree are labeled with base facts. Each internal node n is labeled with a derived fact and a rule such that there is a unique child of n corresponding to each predicate occurrence in the body of the rule, and the rule labeling n along with the facts labeling the children nodes forms a derivation step for the fact that labels n. A predicate p in a program P is said to be *safe* if, given any finite extension for each of the base predicates, p has a finite extension in the minimal model for P.

The *application* of a rule R, using a given set of facts, produces the set of all facts that can be derived using R and only the facts given. The *independent* application of a set of rules means that each rule is applied once but the facts produced using a particular rule application are not available to the application of any of the other rules in the set. A fact is said to have been *seen* by a rule if the fact was available to an application of this rule. The *closure* of a set of rules using a given set of facts refers to the derivation of all facts that can be computed using the given facts, and any number of applications of the rules. We define an evaluation to have the *semi-naive property* if no derivation step is repeated in the evaluation.

Given a program with rules $R = \{R_1, R_2, \ldots, R_n\}$, we define the *rule graph* for the program as the directed graph $G = (R, E)$, where $(R_i, R_j) \in E$ iff the head of R_i unifies with a predicate occurrence in the body of R_j. The strongly connected components of the rule graph are referred to as SCCs. Given a program with predicates $Pred = \{p_1, p_2, \ldots, p_n\}$, we define the *predicate graph* for the program as the directed graph $G = (Pred, E)$, where $(p_i, p_j) \in E$ iff p_i occurs in the body of a rule defining p_j. We refer to strongly connected components in the predicate graph as *Pred-SCCs*.

[1] We only consider positive predicates in the rule body. These definitions can be extended to handle negation and set grouping.

3.2 Basic Semi-Naive Evaluation

Given a program P, the *Basic Semi-Naive* (BSN) evaluation [B85, BR87] of the program proceeds an SCC at a time in a topological order of the SCCs. For each SCC S we apply the following technique. First we create a semi-naive rewritten version of each rule in S as follows. Given a rule of the form $p \leftarrow p_1, \ldots, p_n, q_1, \ldots, q_m$, where $p_1, \ldots, p_n$ are mutually recursive to p and $q_1, \ldots q_m$ are not, the set of semi-naive rewritten rules obtained from this rule has a rule of the form, $\delta p^{new} \leftarrow term, q_1, \ldots, q_m$ for each $term$ in the expansion of $(p_1^{old} + \delta p_1^{old}) \ldots (p_n^{old} + \delta p_n^{old}) - (p_1^{old} \ldots p_n^{old})$. Alternatively, a rewriting based on the technique suggested by Balbin and Ramamohanarao [BR87] can be used.

In evaluating S, an iteration consists of the application of each of the semi-naive rewritten versions of each rule in S, followed by updating the extensions of the semi-naive relations for each p_i as follows,

procedure SN_Update(p_i)
(1) $p_i^{old} := p_i^{old} + \delta p_i^{old}$
(2) $\delta p_i^{old} := \delta p_i^{new} - p_i^{old}$
(3) $\delta p_i^{new} := \phi$
end SN_Update

The evaluation of S proceeds by iterating until no new facts are computed for any of the predicates defined in S. (Note that the operator " - " involves subsumption checks if non-ground facts are generated.)

At every stage of the evaluation, the set of relations p_i^{old}, for all i, has the property that every derivation that uses only these facts has been made.

4 Control Expressions and Semi-Naive Evaluation

4.1 Control Expressions

Helm [H87, H88] introduces the notion of control on the bottom-up evaluation of logic programs using control expressions, and also looks at control as a way of increasing the efficiency of evaluation by eliminating some redundant derivations. Beeri et al. [BRSS89] use control expressions to evaluate the magic rewritten versions of stratified logic programs. While the applications of control expressions have been considered, not much attention has been given to efficient implementation of control expressions.

Let $R_1, \ldots, R_n$ denote the rules of a program. We now give the grammar for our control expressions[2], and describe the semantics of these control expressions.

$$
\begin{aligned}
S &\rightarrow T \\
T &\rightarrow F \mid F + T \mid F \oplus T \mid F \cdot T \\
F &\rightarrow R_i \mid (T) \mid F^* \mid F^\circ
\end{aligned}
$$

The semantics of a control expression is defined as a monotone mapping $\mathcal{D} \rightarrow \mathcal{D}$, where $\mathcal{D}$ is the set of all database states. (A database state is a set of facts for the base and derived predicates.) The initial database state consists of the set of all given facts. (1) When the control expression is a single rule R_i, the resultant database is obtained by applying the rule using the input database and adding the newly generated facts to the input database. (2) The control expression $\alpha \cdot \beta$ maps D to D'', if α maps D to D' and β maps D' to D''. (3) The control expression $\alpha \oplus \beta$ nondeterministically maps D to either D' or D'', if α maps D to D' and β maps D to D''. (4) The control expression α° maps D_0 to D_i, for an arbitrary choice of $i \geq 0$, where α maps D_j to D_{j+1}, $0 \leq j < i$. (5) The control expression α^* maps D_0 to D_i, such that α maps D_j to D_{j+1}, for all $j \geq 0$, and $D_i = D_{i+1}$. This resultant database is uniquely defined due to monotonicity of the mapping. (6) The control expression $\alpha + \beta$ maps D to $D' \cup D''$, if α maps D to D' and β maps D to D''. A fact is present in $D' \cup D''$ if and only if it is present in either of D' or D''.

4.2 Nested-SCC Evaluation of Programs

In this section we examine the evaluation of logic programs using a special form of control expressions, and the advantages of using such control expressions.

The traditional semi-naive evaluation of logic programs proceeds an SCC at a time in topological order. Every predicate defined in the SCC is viewed as a derived predicate for the purposes of semi-naive rewriting. Semi-naive rewriting (as formulated by Balbin and Ramamohanarao [BR87]) generates a large number of semi-naive versions of a single original rule ($n - 1$ semi-naive versions, where n is the number of predicates in the body of the rule that are recursive with the head predicate of the rule—older semi-naive rewritings generate even more).

The Magic Sets rewriting technique is an important technique for evaluation queries on logic programs. Consider the following example.

Example 4.1 Consider the following program P, and query $?p1(a, X)$.

$$
\begin{aligned}
1 : p1(X, Y) &\leftarrow e1(X, Y). \\
2 : p1(X, Y) &\leftarrow p1(X, Z), e2(Z, Y), p(Z, Y), \\
&\qquad \neg p(X, Y). \\
3 : p(X, Y) &\leftarrow e3(X, Y). \\
4 : p(X, Y) &\leftarrow e4(X, Z), p(Z, Y).
\end{aligned}
$$

Using a left to right sip for each rule, the Magic Sets algo-

[2] The semantics of the control operators that we consider are different from those those considered by Helm and we do not discuss the efficient semi-naive implementation of Helm's control operators in this paper.

rithm rewrites P[3] to obtain P^{mg}:

$1 : m_p1(a).$
$2 : p1(X,Y) \leftarrow m_p1(X), e1(X,Y).$
$3 : p1(X,Y) \leftarrow m_p1(X), p1(X,Z), e2(Z,Y), p(Z,Y), \neg p(X,Y).$
$4 : m_p(Z,Y) \leftarrow m_p1(X), p1(X,Z), e2(Z,Y).$
$5 : m_p(X,Y) \leftarrow m_p1(X), p1(X,Z), e2(Z,Y), p(Z,Y).$
$6 : p(X,Y) \leftarrow m_p(X,Y), e3(X,Y).$
$7 : p(X,Y) \leftarrow m_p(X,Y), e4(X,Z), p(Z,Y).$
$8 : m_p(Z,Y) \leftarrow m_p(X,Y), e4(X,Z).$

□

Note that while the original program P had two Pred-SCCs, in the rewritten program P^{mg}, these collapse into one Pred-SCC. The number of recursive predicates in the body per rule in P^{mg} is much greater than the number per rule in P, and this would result in a much larger number of semi-naive versions of rules. We now present some definitions and see how we can reduce the number of semi-naive versions of rules in P^{mg}.

Consider a rule R in P^{mg}. If R defines a non-magic predicate, let $stratum_R$ denote the Pred-SCC of P to which the head of R belongs. If R defines a magic predicate, let $stratum_R$ denote the Pred-SCC of P to which the head of the rule from which R was derived belongs. Let B_R denote the set of body predicate occurrences of R that belong to Pred-SCCs of P that are lower than $stratum_R$. Let the remaining predicate occurrences in R be denoted as B'_R.

Definition 4.1 An evaluation of P^{mg} is said to have the *Nested-SCC evaluation* property if: for every rule R in P^{mg}, and for each predicate occurrence p_i in the set B_R, before an application of R the entire set of facts for p_i that match the set of all available facts for the predicate occurrences in the tail of the sips arcs entering p_i, is available. □

An evaluation that has the Nested-SCC evaluation property is called a *Nested-SCC* evaluation.

Proposition 4.1 In an evaluation of P^{mg}, for each rule R, each predicate occurrence in B_R can be considered as a "base" predicate for the purpose of semi-naive rewriting of R, if the evaluation has the Nested-SCC evaluation property. □

We can ensure the Nested-SCC evaluation property for a magic rewritten program using control expressions to order rule applications. Consider Example 4.1. The evaluation of P^{mg} using the control expression CE_2 below has the Nested-SCC evaluation property.

$$CE_2 = 1 \cdot (4 \cdot (6 + 7 + 8)^* \cdot 5 \cdot (6 + 7 + 8)^* \cdot (2 + 3))^*$$

If CE_2 is used to evaluate P^{mg}, then the occurrences of p and $\neg p$ in the bodies of rules 3 and 5 can be considered as "base" predicates.

Note that CE_2 is based on the Pred-SCC structure of the original program P. An evaluation of P^{mg} using CE_2 closely parallels a top-down evaluation of the program, with the modification that whenever a subgoal corresponding to a predicate defined in a lower Pred-SCC of the original program P is generated (by applying the appropriate magic rule), the entire set of answers matching that subgoal is immediately computed. We discuss how to automatically generate such control expressions in the full version of this paper.

[3]We suppress the adornment for clarity.

4.3 Stratified Programs and Control Expressions

A program that does not involve negation or aggregation can be evaluated SCC by SCC in a topological order of the SCC, with each SCC evaluated using the BSN algorithm (or the PSN or GSN algorithms for an SCC). It can also be evaluated using a Nested-SCC evaluation strategy, as described in the above section. However, if the original program has negation or aggregation, an SCC of the rewritten program cannot always be evaluated using BSN. Consider Example 4.1. In this example, the rewritten program P^{mg} is not stratified (although P itself is stratified), and hence it cannot be evaluated using BSN.

The magic rewritten version of a stratified program can, however, always be evaluated using a Nested-SCC evaluation strategy, to compute the same answer set as the original program on the given query. Thus in Example 4.1, P^{mg} can be evaluated using CE_2.

Beeri et al. [BRSS89] showed that it is possible to order the rule applications of the rewritten program using control expressions (such as CE_1 below[4]) such that an evaluation of the rewritten program using these control expressions computes the same answer set as the original program.

$$CE_1 = (1 \oplus 2 \oplus 5 \cdot (6 \oplus 7 \oplus 8)^* \cdot 3 \oplus 4 \oplus (6 \oplus 7 \oplus 8)^\circ)^*$$

This control expression can be evaluated using the GSN evaluation technique which is described in Section 5.3.

It should be noted that CE_2 imposes some ordering on the application of rules in P^{mg} that is not essential. For instance, it closes the (lower) stratum after the application of rule 4, whereas CE_1 does not; this could result in less set-orientedness in comparison with CE_1. However, in either case, an ordering of rules is required for *correctness*.

5 General Semi-Naive Evaluation

We now present a technique, *General Semi-Naive*, or GSN, evaluation, that makes facts computed by a rule available

[4]CE_1 is a simplification of the control expression automatically generated by the algorithm of Beeri et al. [BRSS89] for this program.

immediately, while maintaining the semi-naive property. We first look at how to apply a single rule using the GSN technique. We then look at how to evaluate control expressions using this technique in Section 5.3, and in Section 5.4 look at a specific form of control expression to evaluate a single SCC in an SCC by SCC evaluation.

5.1 GSN for a Rule

Associated with each rule R_i of the original program, and each predicate p_j that occurs in the body of R_i, we maintain the relation p^{old}_{j,R_i}. The set of relations p^{old}_{j,R_i}, for all j, has the property that every derivation that can be made by an application of R_i using only these facts has already been made. Associated with each predicate p_j, we also maintain its complete extension. We also have temporary relations δp^{new}_j and δp^{old}_j associated with each predicate p_j. The semi-naive rewriting described in Section 3.2 is modified by replacing each predicate occurrence p^{old}_j in the semi-naive rewritten versions of each rule R_i, by p^{old}_{j,R_i}.

procedure GSN_Rule(R)
  Let p_h be the predicate defined by rule R.
  (1) For every predicate p_j in the body of R
    $\delta p^{old}_j := p_j - p^{old}_{j,R}$.
  (2) $\delta p^{new}_h := \phi$.
  (3) Apply each semi-naive rewritten version of R
    (modified as described earlier) independently.
  (4) For every predicate p_j in the body of R
    $p^{old}_{j,R} := p_j$.
  (5) $p_h := p_h + \delta p^{new}_h$.
end GSN_Rule

The application of a rule R in a GSN evaluation consists of a call to the procedure GSN_Rule with argument R.

5.2 Efficient Implementation of GSN

The description of GSN above suggests that each rule has to separately maintain an extension for each predicate that occurs in its body and would thus appear to be inefficient in terms of the storage used. However, it has a simple and efficient implementation in which the extension of each predicate is maintained as an ordered set of tuples. The new facts produced by a rule application are appended to the extension of the predicate[5]. The extension of each p^{old}_{j,R_i} is replaced by a pointer into the extension of p_j, such that the facts in the extension preceding the pointer have already been seen by this rule and facts occurring after the pointer have not been seen yet. We replace the extension of each δp^{old}_j by another pointer into the extension of p_j such that the set of facts between the pointers for p^{old}_{j,R_i} and δp^{old}_j constitutes the extension for δp^{old}_j. The set of facts beyond the pointer for δp^{old}_j constitutes the extension of δp^{new}_j. Thus separate extensions of a predicate do not have to be maintained for each rule, and the need for set difference to compute the δp^{old}_j relations is also eliminated.

A disadvantage with this technique for implementing GSN is that it forces us to use a very specific storage organization for the extension of predicates. Further, some efficient organization strategies, such as hashing, could possibly be precluded by this technique, or may need to be modified.

5.3 Implementing Control Expressions Using GSN

Control expressions can be evaluated in a straightforward manner if a rule application produces every fact that can be derived using the rule and the given database, and we call this the *Naive evaluation* of control expressions. However, this does not preserve the semi-naive property.

Consider the restricted set of control expressions generated by the grammar described in Section 4.1 without the production that uses the "+" operator[6]. We modify the Naive evaluation of control expressions by replacing the application of a rule R by the execution of GSN_Rule(R). We call this technique the *Generalized Semi-Naive* (GSN) evaluation of control expressions. The updates performed in steps (1), (2), (4) and (5) in GSN_Rule maintain the set of facts that have been used by the rule in previous applications of this rule, and ensure that in subsequent applications of this rule, none of the previous derivations is repeated. Also, the execution of GSN_Rule(R) makes all derivations that can be made by the application of R.

Theorem 5.1 *The GSN evaluation of control expressions that do not have any occurrence of the "+" operator obeys the semantics of control expressions and has the semi-naive property.*

Proof: (Sketch) We prove this theorem by proving three properties of any GSN evaluation for the application of each rule R_i (using induction on the number of applications of R_i).

P1. Before step (3) of GSN_Rule, for each predicate p_j occurring in the body of R_i, the set of facts $\delta p^{old}_j = p_j - p^{old}_{j,R_i}$ have not been seen by R_i.

P2. Before step (3) of GSN_Rule, every derivation that could be made by an application of R_i using only the facts in the set of relations p^{old}_{j,R_i}, for all j, has already been made.

P3. After step (3) of GSN_Rule, every derivation that could be made using only the facts in the set of relations p_j, for all j, has been made. □

[5] In the case of generalized tuples, if a new fact subsumes an existing fact then the existing fact may need to be deleted.

[6] Control expressions with the "+" presents some problems that we don't discuss here — we can however handle some restricted cases.

5.4 GSN for an SCC

We now consider the evaluation of a single SCC S in an SCC by SCC evaluation of a stratified program. We assume that all lower SCCs of the program have been completely evaluated, and all the facts computed in those SCCs are available. The GSN evaluation of the SCC S is as described by GSN_SCC.

procedure GSN_SCC(S)
  Let the ordering of rules in S be $R_{i_1}, \ldots, R_{i_n}$.
  (1) Evaluate the control expression
    $(R_{i_1} \cdot \ldots \cdot R_{i_n})^*$ using GSN evaluation.
end GSN_SCC

Theorem 5.2 *GSN evaluation of an SCC is sound, complete and has the semi-naive property.* □

We discuss the issues involved in selecting an ordering of rules in Section 7.

6 Ordered Semi-Naive Evaluation

We now present a technique, Ordered Semi-Naive, or OSN, evaluation that maintains just one version of the extension of each predicate, and can handle a restricted class of control expressions while preserving the semi-naive property. OSN only handles a restricted class of control expressions, without the "$\oplus$" and the "^{o}" operators, and not all such control expressions are correctly evaluated. However, OSN is useful in some applications, and we mention a class of expressions for which this technique is applicable.

For each predicate p in the program P, we maintain the relations p^{old}, δp^{old} and δp^{new} as is done in the BSN evaluation of a program. The semi-naive rewriting of the rules of is done as in BSN.

The evaluation of a control expression by OSN is defined recursively.
(1) If the control expression is of the form R_i, each semi-naive rewritten version of R_i is applied independently. Note that the application of the semi-naive version of R_i affects only the δp^{new} relation where p is the head of rule R_i.
(2) If the control expression is of the form $\beta \cdot \gamma$, β is (recursively) evaluated, then for every predicate p_i defined by a rule in β, SN_Update(p_i) is applied, and finally γ is (recursively) evaluated.
(3) If the control expression is of the form β^*, β is evaluated, then for every predicate p_i defined by a rule in β, SN_Update(p_i) is applied, and if any new facts were produced, β^* is recursively evaluated.
(4) If the control expression is of the form $\beta + \gamma$, β is evaluated and then γ is evaluated.

Theorem 6.1 *Given a stratified program P and its magic rewritten version P^{mg}, there is an algorithm to generate a control expressions such that: (1) the evaluation of P^{mg} using the control expression has the Nested-SCC evaluation property, and (2) the OSN evaluation of the control expression is sound, has the semi-naive property, and if every predicate in P^{mg} is safe, is complete.* □

An algorithm to generate such control expressions is presented in the full version of this paper.

6.1 Ordered Evaluation of a Pred-SCC

In this section, we describe a technique to evaluate a single Pred-SCC of a program, which we call *Predicate-wise Semi-Naive*, or PSN, evaluation. Evaluation of a program proceeds Pred-SCC by Pred-SCC, in a topological order of the Pred-SCCs. PSN is a derivative of OSN in that with a slightly enhanced form of control expressions, it is possible to express the PSN evaluation strategy as a control expression that can be evaluated by (a correspondingly enhanced) OSN. The enhancement consists of separating the point where a rule is applied from the point where the facts it computes are made available to other rule applications. However, for clarity we present it as a separate technique.

The advantage of PSN is that it maintains just one version of the extension of each predicate while permitting the facts produced by a rule application to be used within the same iteration, though not immediately[7]. The rewriting used is the same as that used in BSN evaluation but the pattern of updates to the predicate extensions is different.

procedure PSN_Pred(p)
  Let $R_{i_1}, \ldots, R_{i_n}$ be the recursive rules defining p.
  (1) Apply each semi-naive rewritten version of
    each R_{i_j} independently.
  (2) SN_Update(p)
end PSN_Pred
procedure PSN_SCC(S)
  Let $R_{j_1}, \ldots, R_{j_m}$ be the exit rules of the
    Pred-SCC S.
  (1) Apply each semi-naive rewritten version of
    each R_{j_i} independently.
  Let the ordering of predicates in S be $p_{i_1}, \ldots, p_{i_k}$.
  (2) repeat
  (3) for $j = 1, k$ do PSN_Pred(p_{i_j}).
  (4) until no new facts are computed.
end PSN_SCC

The procedure SN_Update is defined in Section 3.2. Note that no updates to any p^{old} or δp^{old} are made after the application of the exit rules. Although the facts produced by a rule defining a predicate p_i are not immediately available to applications of other rules defining p_i, they are available immediately after the application of all the rules defining

[7] Although we describe just two evaluation strategies, a gradation is possible between GSN and PSN evaluation for SCCs, resulting in a range of evaluation strategies. Some set of predicates may be evaluated according to the strategy used by GSN, and other predicates evaluated according to PSN. We do not further elaborate on this, in this paper.

p_i, to subsequent rule applications (including the application, in the next iteration, of rules defining p_i). A new fact generated by any of the rules defining p_i becomes part of δp_i^{old} immediately after all the rules defining p_i have been applied. After a complete iteration, δp_i^{old} is added to p_i^{old}. Thus each rule sees each fact once and only once in δp^{old} and then onwards in p^{old}. This implies that PSN evaluation has the semi-naive property.

Theorem 6.2 *PSN evaluation of a Pred-SCC is sound, has the semi-naive property, and is complete if every predicate in the Pred-SCC is safe.* □

Much as the ordering of rules could affect the performance of GSN, ordering of predicates could affect the performance of PSN.

In a sequential evaluation, PSN evaluation is always preferable to BSN evaluation, since it can be implemented with the same overheads per iteration, but can do better in terms of the number of iterations. PSN may not be able to utilize facts as soon as GSN can, but the overheads associated with GSN could be higher, and the choice of which strategy to choose is not always obvious.

7 Rule Orderings that Minimize Rule Applications

Let q and r be base relations. Consider the following program:

$$
\begin{aligned}
&R_0 : p_k(X) \quad \leftarrow q(X).\\
&R_1 : p_1(f_1(X)) \leftarrow p_k(X).\\
&\ldots\\
&R_k : p_k(f_k(X)) \leftarrow p_{k-1}(X), r(X).
\end{aligned}
$$

In an iteration of a Basic Semi-Naive evaluation, all the rules of the SCC are applied independently. Rule R_k will be successfully applied for the first time only in the $(k+1)^{th}$ iteration. However, it would be possible to successfully apply rule R_k in the first iteration itself if the rules are applied in the order shown and the facts produced by each rule application are immediately made available to subsequent rule applications. In the above example, if the computation of a fact using this technique took n iterations, then computation of the same fact using the BSN evaluation strategy could take up to $O(k\,n)$ iterations.

In the above example, if the rules are applied in the opposite order, i.e. $R_k, R_{k-1}, \ldots, R_0$, the number of iterations taken is practically the same as BSN evaluation, even if facts are made available immediately. Thus we see the importance of a good ordering of rules.

In this section, we provide a theoretical analysis of how the number of rule applications (and iterations) in semi-naive fixpoint algorithms (that use the immediate update techniques described in Section 5) can be reduced through the use of rule ordering. Our results are significant in that they indicate how this number can be minimized, independent of the data in base relations, over a significant class of rule orderings (Section 7.2). We also present results which suggest that only this class of rule orderings should be considered in the absence of additional semantic information (Section 7.3).

The techniques described in this section deal with rule orderings, but can be extended, in a straight forward fashion, to deal with predicate orderings.

7.1 Class of Orderings Considered

Let the rules of the SCC S, whose closure we wish to compute, be $R_1, \ldots, R_n$. In Section 7.2 we consider orderings of the form $(R_{i_1} \cdot \ldots \cdot R_{i_n})$, where $i_1, \ldots, i_n$ is a permutation of $1, \ldots, n$. Such orderings are static, non-nested orderings in which no rule is applied more often than other rules. Such orderings are referred to as *fair* orderings since in the absence of any prior knowledge of the frequency with which different rules are used, or other semantic information, we have no basis for applying some rules more often than others.

In Section 7.3 we consider static orderings in which some rules can be applied more often than other rules. Such orderings are referred to as *non-fair* orderings. This class includes the class of nested orderings, such as those considered by Kuittinen et al. [KNSS89]. Non-fair orderings may perform somewhat better than fair orderings on some data sets, but, as we show in Section 7.3, such orderings may also perform considerably worse on other data sets. Hence, in the absence of any information about the kind of data sets, fair orderings are preferable.

7.2 Fair Orderings

Consider an SCC S, and let the rules in S be $R = \{R_1, \ldots, R_n\}$. Let $G = (R, E)$ be the rule graph for the given SCC. Let O be any fair ordering $(R_{i_1}, \ldots, R_{i_n})$ of the rules in R. Let C be any simple cycle[8] $R_{j_1}, \ldots, R_{j_m}$ in G. For a fair ordering O, O^n denotes the string formed by repeating O n times and is called an *order sequence*.

We say that a fair ordering O *preserves* a cycle C, if there is a cyclic permutation O_1 of O such that C forms a subsequence[9] of O_1. A fair ordering O on G is a *cycle preserving fair ordering* if for every simple cycle C in G, O preserves C. A fair ordering O that does not preserve a cycle C is said to break it. A cycle C is *broken by degree* $B(C, O) = i$, by a fair ordering O, if i is the least number such that for some cyclic permutation O_1 of O, C is a sub-

[8] R_{j_1} defines a predicate used in R_{j_2}, and so on, and R_{j_m} defines a predicate used in R_{j_1}. Though cycles have the same initial and final vertex, we omit the final vertex in our representation, for convenience.

[9] By subsequence, contiguity is not implied. For example, R_1, R_3, R_5 is a subsequence of $(R_1, R_2, R_3, R_4, R_5)$.

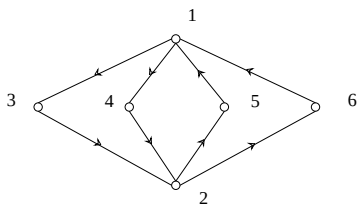

Figure 1: An Example Rule Graph

sequence of O_1^i. Thus a fair ordering that preserves a cycle can be said to break it by degree one.

We define a relation $\lhd$ on the class of fair orderings. Given two fair orderings O_1 and O_2 on a rule graph G, $O_1 \lhd O_2$ if for every simple cycle C in G, $B(C, O_1) \leq B(C, O_2)$. If we have $O_1 \lhd O_2$ and $O_2 \lhd O_1$, we say that the two orderings are *equivalent*. The use of this relation will be seen in Section 7.2.1, where we show in Theorem 7.2 that if $O_1 \lhd O_2$ and the two are not equivalent, then given any database, O_1 is better than O_2 based on the number of iterations needed to compute the closure of an SCC. We also show that if O_1 and O_2 are equivalent, the number of iterations needed by the each to compute the closure of an SCC differ by at most a data-independent constant. Thus we show that an ordering that preserves all cycles is optimal in the class of fair orderings, under this cost criterion. From the definition of the relation $\lhd$ we have, (1) Cyclic permutations of a fair ordering are equivalent under $\lhd$, and (2) Any two cycle preserving fair orderings are equivalent under $\lhd$.

Example 7.1 Consider the graph shown in Figure 1. The simple cycle $1, 3, 2, 6$ is preserved by the ordering $(2, 6, 4, 1, 3, 5)$ because the ordering has a cyclic permutation $(1, 3, 5, 2, 6, 4)$ which has the simple cycle as a subsequence. However, this ordering breaks the simple cycle $1, 4, 2, 5$ by degree 3. □

Lemma 7.1 *Consider a cycle $C = R_1 \ldots R_m$ and a fair ordering O. Let O_1 be the cyclic permutation of O that ends with R_m. Then C forms a subsequence of O_1^i for $i = B(C, O)$, but not for any smaller i.* □

The use of this lemma will be seen in subsequent sections.

For the class of fair orderings, we next show that cycle preserving fair orderings are optimal under the cost criterion of the number of iterations needed to compute the closure of an SCC, with an immediate update strategy. Since the number of rule applications is constant within an iteration, the optimality result carries over for the cost criterion of number of rule applications.

7.2.1 Optimality of Cycle Preserving Orderings

A *derivation path* for a fact is a path in a derivation tree for the fact, starting from a leaf node. We represent such a path concisely by listing the rules labeling the nodes in the derivation path in order, starting from the parent of the leaf. Note that two different paths may have the same representation, but that does not affect our analysis.

Let O denote a fair ordering of rules and T denote a particular derivation tree for $p(\overline{c})$. Consider the rule application sequence $\mathcal{O}' = O^j$, for arbitrarily large j. With each node in the derivation tree T, we associate a *derivation index*, which is an index into the sequence $\mathcal{O}'$, Leaf nodes (corresponding to base facts) are associated with the derivation index zero. The derivation index of each internal node n', labeled with a derived fact p' and a rule R', is the minimum possible k such that, $\mathcal{O}'[k] = R'$ and the derivation indices of the children nodes of n' are less than k. With each node in the derivation tree we associate an *iteration height* $\lceil k/|O| \rceil$ where k is the derivation index of that node. A derivation tree is said to be *computed* by O using n iterations if the iteration height of the root of the tree under O is n.

The iteration height of a node is defined syntactically but has the following semantic interpretation. If the iteration height of the root of a derivation tree T is n, then the corresponding fact $p(\overline{c})$ is computed in or before the nth iteration of the application of rules according to the ordering O. If the fact is computed in the nth iteration, there is a derivation tree with iteration height n for the fact. The *iteration count* of a fact, for a given fair ordering, is defined to be the minimum of the iteration heights under the given ordering, of derivation trees for this fact. This gives us the earliest iteration in which the fact is derived. This link between the semantic notion of the number of iterations needed to compute a fact, and the iteration heights of derivation trees for the fact enables us to argue about the computation of facts using purely syntactic criteria.

Given a derivation path s in T, the *iteration length* $L(s, O)$ of the path is the minimum n, such that the path forms a subsequence of O^n. The *minimum length order sequence* for s is defined to be $O^{L(s,O)}$. We next show the relationship between the notion of the iteration length of a path, and the iteration height of a tree. For a tree T, if T has no internal nodes, $L(T, O) = 0$. Otherwise $L(T, O) = \max\{L(s, O) \mid s$ is a path in $T\}$.

The following lemma permits us to argue about the number of iterations it takes to compute a derivation tree based on the iteration lengths of the derivation paths in the tree.

Lemma 7.2 *Given a derivation tree T for a fact $p(\overline{c})$, and a fair ordering O, the derivation tree can be computed by a bottom-up fixpoint evaluation using rule ordering O in $L(T, O)$ iterations. $L(T, O)$ is thus also the iteration height of T.* □

Every derivation path has a certain structure which is described in the next lemma, and which we use to prove the result in Theorem 7.1.

Lemma 7.3 *For every derivation path s, there exists a sequence $s_0, \ldots, s_n$ of paths in the rule graph G, such that, (1) $s = s_n$. (2) s_0 is an acyclic path in G. (3) For each $i > 0$, s_i can be constructed from s_{i-1} as follows: Choose a rule*

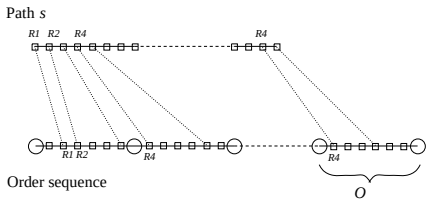

Figure 2: Relating Derivation Paths to Order Sequences

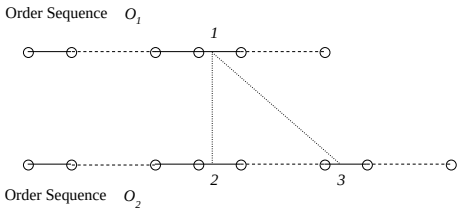

Figure 3: Order Sequences After Insertion of a Cycle in a Path

R_{j_k} in s_{i-1}, and a simple cycle $C_i = R_{j_1}, \ldots, R_{j_k}$ in G, and insert the cycle just after R_{j_k}. □

For any derivation path s, such a sequence $s_0, \ldots, s_n$ is called a *construction sequence* for s. Intuitively, we can create a construction sequence in reverse order by successively deleting simple cycles from a derivation path. A derivation path may not have a unique construction sequence.

Given a fair ordering O, we now relate the iteration length of a derivation path s with the length of the construction sequence for s and the degree by which the given ordering breaks each of the cycles inserted.

Theorem 7.1 *Consider any derivation path s and a construction sequence $s_0, \ldots, s_n$ for s as defined in Lemma 7.3. Let C_i be the cycle inserted in obtaining s_i from s_{i-1}. For every fair ordering O, the iteration length of s under O is given by $L(s, O) = L(s_0, O) + \sum_{i=1}^{n} B(C_i, O)$. Further, $L(s_0, O)$ is bounded by the length of the longest acyclic path in the rule graph of the SCC.*

Proof: We prove this by induction on on the length of the construction sequence for a derivation path s. As a basis, note that all construction sequences of length one have only one element s_0, and it follows that, $L(s, O) = L(s_0, O)$. Note that $L(s_0, O)$ is data-independent.

Assume inductively that for all paths s that have a construction sequence of length less than or equal to k, $L(s, O) = L(s_0, O) + \sum_{i=1}^{n} B(C_i, O)$, where there is a construction sequence $s_0, \ldots, s_n$ for s, $n < k$. Recall that $L(s, O)$ is the minimum number i such that s forms a subsequence of O^i. Corresponding to each rule occurrence in s, we have a rule occurrence in some repetition of O in O^i. Figure 2 depicts this pictorially.

Consider now a path that has no construction sequence of length less than or equal to k, but has a construction sequence of length $k+1$. Let the last cycle inserted in the sequence be $C_k = R_s, \ldots, R_l$. s_{k-1} has a construction sequence of length k, and therefore by the induction hypothesis, $L(s_{k-1}, O) = L(s_0, O) + \sum_{i=1}^{k-1} B(C_i, O)$. For the path s_{k-1} we have the corresponding (minimal length) order sequence, $\mathcal{O}_1 = O^{L(s_{k-1}, O)}$, such that s_{k-1} is a subsequence of $\mathcal{O}_1$. Similarly for the path s_k we have the corresponding (minimal length) order sequence $\mathcal{O}_2 = O^{L(s_k, O)}$. Figure 3 shows $\mathcal{O}_1$ and $\mathcal{O}_2$.

Let the point of insertion of C_k in s_{k-1} be after an occurrence of rule R_l. Let this occurrence of R_l be called R_l^1. R_l^1 corresponds to the the (unique) occurrence of R_l (labeled 1 in Figure 3) in some occurrence of O in $\mathcal{O}_1$. Similarly, for $\mathcal{O}_2$ R_l^1 corresponds to the occurrence of R_l at Point 2 in $\mathcal{O}_2$. Let the occurrence of R_l in the newly inserted cycle be labeled R_l^2. R_l^2 corresponds to the (unique) occurrence of R_l (labeled 3 in Figure 3) in some occurrence of O in $\mathcal{O}_2$.

We claim that the initial part of $\mathcal{O}_1$ up to (and including) 1 is the same as the initial part of $\mathcal{O}_2$ up to (and including) 2, and further, the part of $\mathcal{O}_1$ after 1 is the same as the part of $\mathcal{O}_2$ after 3. Once we have shown this, we show that the part of $\mathcal{O}_2$ between 2 and 3 has exactly $B(C_k, O)$ repetitions of O, which proves the result.

Point 1 and Point 3 both correspond to occurrences of rule R_l. Since there is only one occurrence of each rule within O, Point 1 is at the same position within a repetition of O in $\mathcal{O}_1$ as Point 3 is in a repetition of O in $\mathcal{O}_2$. The part of s_k after R_l^2 and the part of s_{k-1} after R_l^1 are the same, since s_k is derived from s_{k-1} by inserting a cycle at R_l^1, and this cycle ends at R_l^2. Call this part of s_k as s_{tail}.

Since $\mathcal{O}_1$ is a minimal length order sequence for the path s_{k-1}, the part of $\mathcal{O}_1$ after Point 1 must correspond to a minimal length order sequence for the path s_{tail}. Similarly the part of $\mathcal{O}_1$ after Point 3 must also correspond to a minimal length order sequence for s_{tail}. Further, Points 1 and 3 both correspond to occurrences of R_l. Hence, it follows that the part of $\mathcal{O}_2$ after Point 3 must be the same as the part of $\mathcal{O}_1$ after 1. By reasoning similar to the above, we can show that the part of $\mathcal{O}_1$ up to Point 1 is the same as the part of $\mathcal{O}_2$ up to Point 2.

Each rule occurs exactly once in each repetition of O, and Points 2 as well as 3 (in $\mathcal{O}_2$) correspond to occurrences of rule R_l. Thus the part of $\mathcal{O}_2$ between Points 2 and 3 (including 3, but not 2) corresponds to an integral number of repetitions of the cyclic permutation O_r of O that ends with rule R_l. Since both O_r and C_k end with R_l, and O_r is a cyclic permutation of O, it follows from Lemma 7.1 that C_k forms a subsequence of O_r^i for $i = B(C_k, O)$, but not for any smaller i. Thus the number of repetitions of O_r between 2 and 3 is $B(C_k, O)$, and the proof is complete. □

Note the interesting fact that the above theorem is true for any construction sequence. Since the actual iteration length of a path does not depend on the construction se-

quence chosen, this tells us that, in a certain sense, all construction sequences are equivalent. For any cycle preserving fair ordering s with a construction sequence $s_0, \ldots, s_n$, $L(s, O) = L(s_0, O) + n$.

Consider a rule graph G, and two fair orderings O_1 and O_2 such that $O_1 \lhd O_2$. We define $MaxR(O_1, O_2, G) = max\{B(C, O_2)/B(C, O_1) \mid C \text{ is a basic cycle in } G\}$. This serves as a bound on how much costlier, based on the number of iterations, O_2 can be compared to O_1.

Given any two fair orderings that are related by the $\lhd$ relation, we wish to compare the number of iterations taken to compute the closure of an SCC by the two orderings. To this end, we first compare the iteration lengths of derivation paths. This is used to compare the iteration heights of derivation trees for a fact. We then argue about the number of iterations taken to derive a fact by the two orderings, by comparing the iteration counts of the fact. [10] This leads finally to our main result, stated in Theorem 7.2, that relates the number of iterations taken to compute the closure of the SCC by the two fair orderings.

Theorem 7.2 *Given an SCC S, any two fair orderings O_1 and O_2, such that $O_1 \lhd O_2$, and any set of base facts, let the number of iterations required to compute the closure of S by bottom-up fixpoint evaluations using rule orderings O_1 and O_2 be n_1 and n_2 respectively. n_1 and n_2 are related as $n_1 - k \leq n_2 \leq MaxR(O_1, O_2, G) \cdot n_1 + k$, where k is bounded by the length of the longest acyclic path in the rule graph for the SCC.*

Proof: (Sketch) Consider the fact that is computed last, when the closure of S is computed using ordering O_2, and call this fact $p(\overline{c})$. Clearly n_2 iterations are needed to compute this fact. We show that this fact will be computed using ordering O_1 in r_1 iterations, where $n_2 \leq MaxR(O_1, O_2, G) \cdot r_1 + k$. Further, $n_1 \geq r_1$, and hence we have $n_2 \leq MaxR(O_1, O_2, G) \cdot n_1 + k$.

Next consider the fact that is computed last when the closure of S is done using ordering O_1, and call this fact $p'(\overline{c'})$. This fact needs n_1 iterations to compute. We show that this fact is computed using ordering O_2 in r_2 iterations, with $n_1 - k \leq r_2$. Further $n_2 \geq r_2$, and hence we have $n_1 - k \leq n_2$. □

Corollary 7.1 *Given any two cycle preserving fair orderings, the number of iterations required to compute the closure of an SCC by bottom-up fixpoint evaluations using the two orderings differ by at most a (data-independent) constant. Also, the number of rule applications required by the two orderings differ by at most a (data-independent) constant.* □

7.3 Non-Fair Orderings

In this section we consider static orderings in which some rules may be applied more often than other rules. We divide this class into the class of flat orderings and the class of nested orderings. A *nested ordering* is an ordering O of the form $(O1)$, where $O1$ is generated by the grammar

$$O1 \rightarrow R_1 \mid \ldots \mid R_n \mid O1 \cdot O1 \mid (O1)^*$$

where $R_1, \ldots, R_n$ are the rules of an SCC S of the rule graph, such that each rule in the SCC occurs at least once in the ordering O. An example is the ordering $(R_1 \cdot R_2 \cdot (R_3 \cdot R_4)^* \cdot R_5)^*$. Note that a nested ordering can have more than one occurrence of any rule in the SCC.

A *flat ordering* is a nested ordering that has parentheses only at the outermost level. The *nesting level* of such an ordering is defined in the obvious manner, where a flat ordering is defined to have a nesting level of one. The following theorem summarizes our comparison of flat and nested orderings.

Theorem 7.3 Consider an SCC S, and any flat ordering O_f and any nested ordering O_n on S. Let $iter_{bsn}$ be number of iterations needed to compute the closure of S using BSN. If n_f and n_n denote the number of rule applications needed to compute the closure of S using O_f and O_n respectively, then $n_f/k \leq n_n \leq (iter_{bsn})^s \cdot k_1$, where k is the number of rule occurrences in O_f, k_1 is the number of rule occurrences in O_n, and s is the nesting level of O_n. □

Since an optimal fair ordering must take at least as many rule applications as an optimal rule ordering, the above theorem also directly bounds how much worse an arbitrary flat ordering can be compared to an optimal fair ordering.

Note that every fair ordering is also a flat ordering, and hence the above theorem applies when we compare fair orderings with nested orderings. The worst case performance of nested orderings is bounded, as shown above. In Section 8 we describe an example where the performance of a nested ordering is indeed as bad (to within a small constant factor) as the above upper bound allows (Program $P2$, data set $S64$). Thus, although a nested ordering can perform somewhat better (i.e. $n_f = cn_n$, $1 < c \leq k$) than fair orderings on some data sets, it is possible for it to perform much worse (when $n_n = (iter_{bsn})^s.k_1$) on other data sets. Note also that $iter_{bsn} \geq n_f/k$.

8 Summary of Performance Results

In this section we describe preliminary results of a performance study of the benefits of immediate availability of facts, and the benefits of ordering rules as described in Section 7. These results bear out the theoretical analysis presented in earlier sections. We then compare fair orderings with nested orderings, and show that for some data sets, nested orderings outperform fair orderings, whereas for other data sets they perform much worse. Our performance study draws upon and extends the work of Kuittinen et al. [KNSS89].

[10] This development, in a chain of lemmas, is omitted here.

Data Set	Basic	Pred-wise	Gen'l 1	Gen'l 2
$A10$	3579	1535	1023	2812
$B64$	146	73	68	80
$F10$	23	10	7	18

Table 1: Program $P1$: Number of Iterations

We consider two programs, referred to as $P1$[11] and $P2$ in this section. These programs and corresponding data sets are presented in Appendix A.

The semi-naive rewriting used is the version proposed in [BR87]. The above programs were hand-coded for each of the evaluation techniques, and measurements were made by running the resultant programs on the data sets described. There is a cycle preserving fair ordering for each SCC of $P1$ and $P2$, and the column "General 1" of the tables is for a Generalized Semi-Naive evaluation using such an ordering. The column "General 2" in the tables for $P1$ corresponds to a GSN evaluation of $P1$ using a fair ordering that breaks a cycle to degree six. The column "Nested" in the tables for $P2$ corresponds to the above nested ordering. Column "Basic" corresponds to a Basic Semi-Naive evaluation, and column "Pred-wise" is for a Predicate-wise Semi-Naive evaluation using a cycle preserving fair ordering based on the predicate graph.

For $P1$ we use the data sets $A10, B64$, and $F10$. Data set $A10$ results in no duplicate derivations with $P1$, but takes a large number of iterations. Data set $B64$ is large, takes a moderate number of iterations, and results in a moderate number of duplicates. Data set $F10$ results in a large number of duplicate derivations, but a fewer number of iterations.

For $P2$ we use two data sets, $C16$ and $S64$. $C16$ is designed such that the nested ordering we consider performs well, and $S64$ is designed such that the nested ordering performs very badly compared to the cycle preserving fair ordering.

Table 1 shows the number of iterations taken by each evaluation strategy on $P1$. It should be noted that the total number of rule applications (considered) is directly proportional to the number of iterations taken. PSN outperforms BSN on this measure. It improves over the performance of BSN by over 50% on all data sets considered. GSN with a cycle preserving fair ordering outperforms PSN by about 30% (on $A10$ and $F10$) and performs about 50% to 70% better than BSN. GSN with a bad fair ordering performs much worse than GSN with a good fair ordering (although it can never be worse than BSN), and this clearly brings out the benefits of good fair orderings.

If one of the relations in the join is empty, the result of the join is null, and we call such a join a *null join*. We may be able to detect this condition at run time without incurring much cost. Table 2 shows the number of joins used by each evaluation strategy on $P1$, divided into the number of null joins, and the number of non-null joins. The total number of joins taken by General 1 is better than PSN, which in turn is better than BSN. If we count only non-null joins, this is not strictly true. PSN always performs no worse than BSN, and on $B64$ performs about 50% fewer non-null joins. General 1 performs 15% to 45% better than BSN on this count. On one data set, $B64$, PSN is slightly better (less than 7%) than General 1. However, on the other two data sets, General 1 outperforms PSN by about 15% to 20%.

[11] $P1$ is the same program that was used in [KNSS89].

Data Set	Basic	Pred-wise	Gen'l 1	Nested
$C16$	282	221	207	179
$S64$	1717	588	583	2536

Table 3: Program $P2$: Number of Rule Applications

Table 3 shows the number of rule applications taken by each evaluation strategy on $P2$. It can be seen that for $C16$, Nested is better than the fair ordering indicated by General 1. However, for $S64$ Nested performs much worse than any of the other evaluation strategies. Both PSN and General 1 are about 20% to 65% better than BSN.

Table 4 shows the number of joins used by each evaluation strategy on $P2$. Again, the number of joins taken by Nested is better than the fair ordering indicated by General 1 for the data set $C16$. However, Nested performs very badly on the data set $S64$ compared to each of the other evaluation strategies. General 1 and PSN are about 20% to 30% better than BSN.

Our performance results underscore the theoretical results described in earlier sections. The benefits of immediate availability of facts is indicated by the fact that GSN is in general better than PSN, which, in turn, is in general better than BSN, under the cost criterion of number of rule applications and iterations. Further, our results clearly bring out the advantages of cycle preserving fair orderings. Our results also indicate that nested orderings may perform better than fair orderings on some data sets, but can perform much worse on others.

9 Conclusion

In this paper, we studied several aspects of rule ordering in the bottom-up evaluation of logic programs. Rule orderings are necessary for ensuring a desired semantics, such as the evaluation of the magic rewritten versions of stratified programs. Rule orderings were also shown to be useful for improving the total cost of sequential evaluation of logic programs. We presented three evaluation algorithms, GSN, OSN and PSN, that could be used for evaluating such rule orderings, while preserving the semi-naive property, and discussed cases where each was useful.

We studied rule orderings theoretically, and showed that

Data Set	Basic		Pred-wise		General 1		General 2	
	Non-Nl	Null	Non-Nl	Null	Non-Nl	Null	Non-Nl	Null
$A10$	6126	14835	6126	4615	5230	2693	6126	11000
$B64$	1066	196	549	88	588	14	596	96
$F10$	73	78	57	17	45	10	73	53

Table 2: Program $P1$: Number of Joins (Non-Null and Null)

Data Set	Basic		Pred-wise		General 1		Nested	
	Non-Nl	Null	Non-Nl	Null	Non-Nl	Null	Non-Nl	Null
$C16$	459	203	372	162	364	139	304	30
$S64$	2002	1016	1386	389	1447	323	5230	446

Table 4: Program $P2$: Number of Joins (Non-Null and Null)

for the class of fair orderings, cycle preserving orderings were optimal and, in the absence of additional information, fair orderings are to be preferred to non-fair orderings.

An important open problem is to find an efficient algorithm that checks whether an SCC has a cycle preserving fair ordering and if so, produces it. As a heuristic, we suggest the reverse of a depth-first search pop-out order of the rule graph. This produces a fair ordering that often preserves cycles in an SCC. However, this procedure does not guarantee that all cycles are preserved, even if the SCC has a cycle preserving fair ordering, and in the worst case this procedure could break the cycles in the rule graph to a high degree.

We also presented a summary of some performance results to support our theoretical analyses.

References

[B85] F. Bancilhon, "Naive Evaluation of Recursively Defined Relations," TR DB-004-85, MCC, 1985.

[BR87] I. Balbin and K. Ramamohanarao, "A Generalization of the Differential Approach to Recursive Query Evaluation," *Journal of Logic Programming*, Vol. 4, No. 3, Sept. 87.

[BPRM] I. Balbin, G. S. Port, K. Ramamohanarao and K. Meenakshi, "Efficient Bottom-Up Computation of Queries on Stratified Databases," *Journal of Logic Programming*, (to appear).

[BRSS89] C. Beeri, R. Ramakrishnan, D. Srivastava and S. Sudarshan, "Magic Implementation of Stratified Programs", Manuscript, Sept. 89

[H87] R. Helm, "Inductive and Deductive Control of Logic Programs," *Proc. of the 4th International Conference on Logic Programming,* pp. 488-512, Melbourne, Australia, 1987.

[H88] R. Helm, "Detecting and Eliminating Redundant Derivations in Deductive Database Systems," TR RC 14244 (#63767), IBM Research Division, Yorktown Heights, NY 10598, May 1988.

[KIC89] R. Kabler, Y. Ioannidis, and M. Carey, "Performance Evaluation of Algorithms for Transitive Closure," Unpublished manuscript, Univ. of Wisconsin, Madison, December 1989.

[KNSS89] J. Kuittinen, O. Nurmi, S. Sippu and E. Soisalon-Soininen, "Efficient Implementation of Loops in Bottom-Up Evaluation of Logic Queries," Manuscript, June 1989.

[L87] H. Lu, "New Strategies for Computing the Transitive Closure of a Database Relation," Proc. of the 13th International VLDB Conference, Brighton, England, September 1987, pp. 267-274.

[MR89] M.J. Maher and R. Ramakrishnan, "Déjà Vu in Fixpoints of Logic Programs," To appear, NACLP, Cleveland, 1989.

$1 : msg(1).$
$2 : supm2(X, X1) \leftarrow msg(X), up(X, X1).$
$3 : supm3(X, X2) \leftarrow supm2(X, X1), sg(X1, X2).$
$4 : supm4(X, Y2) \leftarrow supm3(X, X2), flat(X2, Y2).$
$5 : sg(X, Y) \leftarrow msg(X), flat(X, Y).$
$6 : sg(X, Y) \leftarrow supm4(X, Y2), sg(Y2, Y1), down(Y2, Y).$
$7 : msg(X1) \leftarrow supm2(X, X1).$
$8 : msg(Y2) \leftarrow supm4(X, Y2).$
$9 : query(Y) \leftarrow sg(1, Y).$

Figure 4: Program $P1$

A Programs

In $P1$, rules 1 and 9 are in separate SCCs, and the other seven rules are in one SCC of the rule graph. The cycle

$1: \ anc(X,Y,1) \leftarrow manc(X), up(X,Y).$

$2: \ anc(X,Y,N) \leftarrow N > 1, manc(X), anc(X,Z,N-1), up(Z,Y).$

$3: \ desc(X,Y,1) \leftarrow mdesc(X,1), down(X,Y).$

$4: \ desc(X,Y,N) \leftarrow N > 1, mdesc(X,N), desc(X,Z,N-1), down(Z,Y).$

$5: \ sg(X,Y) \leftarrow msg(X), flat(X,Y).$

$6: \ sg(X,Y) \leftarrow msg(X), anc(X,X1,N), flat(X1,X2), sg(X2,Y2), flat(Y2,Y1), desc(Y1,Y,N).$

$7: \ manc(X) \leftarrow msg(X).$

$8: \ msg(X2) \leftarrow msg(X), anc(X,X1,N), flat(X1,X2).$

$9: \ mdesc(Y1,N) \leftarrow msg(X), anc(X,X1,N), flat(X1,X2), sg(X2,Y2), flat(Y2,Y1).$

$10: mdesc(X,N-1) \leftarrow mdesc(X,N), N > 1.$

$11: msg(1).$

$12: query(X) \leftarrow sg(1,X).$

Figure 5: Program $P2$

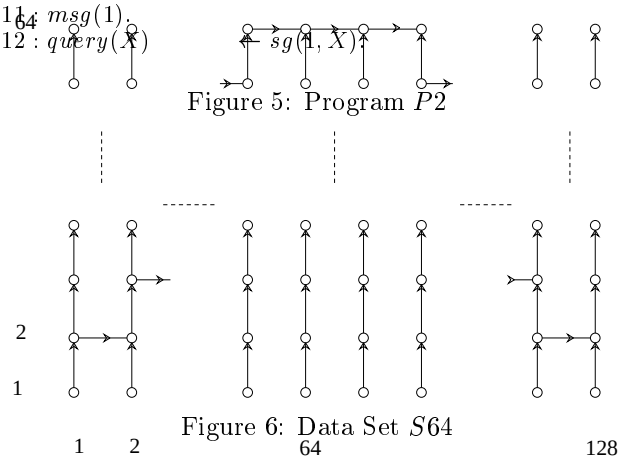

Figure 6: Data Set $S64$

preserving fair ordering for this program is $1 \cdot (2 \cdot 7 \cdot 5 \cdot 6 \cdot 3 \cdot 4 \cdot 8) \cdot 9$. The ordering $1 \cdot (2 \cdot 8 \cdot 4 \cdot 3 \cdot 6 \cdot 5 \cdot 7) \cdot 9$ breaks a cycle to a degree six, and is used in General 2.

There are two non-trivial SCCs in the rule graph of $P2$. The fair orderings chosen for each of the SCCs is cycle preserving. Combining the fair orderings for each SCC, the fair rule ordering used for $P2$ is $11 \cdot (7 \cdot 1 \cdot 2 \cdot 8) \cdot 5 \cdot (9 \cdot 10 \cdot 3 \cdot 4 \cdot 6) \cdot 12$. Combining the nested orderings for the SCCs, the nested ordering used for this program is $11 \cdot (7 \cdot 1 \cdot (2)^* \cdot 8)^* \cdot 5 \cdot (9 \cdot 10 \cdot 3 \cdot (4)^* \cdot 6)^* \cdot 12$.

Data sets $A10$ and $B64$ are the same as those used in [KNSS89]. A *grid* is defined to be a structure such that: (1) If i and j are nodes in the same column and i is below j, there is a fact $up(i,j)$ as well as a fact $down(j,i)$, and (2) If i and j are nodes in the same row and j is immediately to the right of i, there is a fact $flat(i,j)$. Data set Cn is a grid with n rows and 8 columns. Data set $F10$ represents a 10×10 grid, with the additional facts $up(i,j)$ and $down(j,i)$ for every pair of nodes i , j that are in the same column, and i is below (but not necessarily immediately below) j. Data set $S64$ is as shown in Figure 6.