RULE ORDERING IN BOTTOM-UP FIXPOINT EVALUATION OF LOGIC PROGRAMS

by

Raghu Ramakrishnan
Divesh Srivastava
S. Sudarshan

Computer Sciences Technical Report #1059

December 1991

Rule Ordering in Bottom-Up Fixpoint Evaluation of Logic Programs

Raghu Ramakrishnan Divesh Srivastava S. Sudarshan

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.

A desirable property of a fixpoint evaluation algorithm is that it does not repeat inferences. We say that such an algorithm has the non-repetition property, and the "semi-naive" algorithms in the literature have this property. However, these algorithms do not address the issue of how to apply rules in a specified order while retaining the non-repetition property. We present two algorithms that address this issue. One of them (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 (PSN) handles a smaller class of rule orderings, but with no overheads beyond those in BSN.

We also demonstrate that 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 analysis.

Index Terms: Bottom-up Evaluation, Control Expression, Cyclic Ordering, Deductive Database, Query Evaluation, Logic Programming, Rule Ordering, Semi-naive Evaluation.

1 Introduction

Bottom-up evaluation of logic programs is an important issue in deductive database applications. Bottom-up evaluation proceeds by starting with the facts in the database and repeatedly applying all the rules of the program, in iterations, to compute new facts. The evaluation terminates once we have reached a fixpoint, i.e. when no new facts are computed by any of the rule applications. One of the main concerns is the duplication of inferences, which could considerably decrease the efficiency of bottom-up evaluation. Algorithms which do not repeat inferences are considered desirable and said to have the *non-repetition*

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 are with the Computer Sciences Department, University of Wisconsin-Madison, WI 53706, U.S.A. The authors' e-mail addresses are {raghu,divesh,sudarsha}@cs.wisc.edu.

A preliminary version of this paper appeared in the Proceedings of the Sixteenth International Conference on Very Large Data Bases, Brisbane, Australia, 1990.

property. There are essentially two components to fixpoint algorithms that have the non-repetition property.

1. 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) from older facts.
2. The second component is a technique to apply the rewritten rules to generate facts, and update these differentials, ensuring that all derivations are made exactly once.

Semi-naive algorithms have been proposed by several researchers (e.g., [3, 2]), and these algorithms have the non-repetition property. 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. Unfortunately, these algorithms do not address the issue of how to apply rules in a specified order while retaining the non-repetition property. Rule orderings are significant for several reasons.

- First, they are sometimes required to compute the answers correctly.

  For example, in programs with stratified negation, lower strata must be evaluated before higher strata, and this simple ordering can become much more complex once we rewrite the program using Magic Sets [24, 4, 8, 23] (which is an important technique used widely to avoid inferences that are not "relevant" to the query).

- Second, rule ordering can result in increased efficiency.

  For example, in an SCC-by-SCC evaluation of a program, rules in lower SCCs do not need to be considered while applying rules in higher SCCs; this can improve the efficiency of evaluation. This is a simple form of rule ordering which has been implemented using existing semi-naive algorithms.

  An important contribution of this paper is to demonstrate that ordering rules *within* an SCC can also improve efficiency by further reducing the number of rule applications. Although the orderings we consider for this purpose do not affect the number of inferences made, 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 [13, 12, 14].

In this paper, we use regular expressions over rules to specify orderings of the application of rules; we call these *control expressions*.

In the first part of our paper, we present two fixpoint algorithms that address the issue of how to apply rules in a specified order while retaining the non-repetition property. One of them, General Semi-Naive (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). The GSN algorithm can deal with a large set of control expressions, and is described in Section 4. The other algorithm we present, Predicate Semi-Naive (PSN), can utilize facts produced for a predicate p in the same iteration they have been derived in, although not always in the immediately following rule application; this is described in Section 5. It handles a more restricted set of control expressions compared to GSN, but is cheaper than GSN. Indeed, it has no additional overheads compared to traditional semi-naive algorithms.

Each of the algorithms we present is a technique to apply (semi-naive rewritten) rules and update the differentials. These algorithms can use any of the semi-naive rewriting techniques proposed earlier

(e.g., [3, 2]) with minor modifications. The algorithms we describe in this paper are independent of program rewriting optimization techniques such as Magic Sets that also seek to reduce the number of derivations. We can apply such techniques (and other techniques such as those described in [13, 12] that generate control expressions on rule applications) on a given program, and then evaluate the resultant program using our algorithms.

In the second part of our paper, in Section 6, we study rule orderings in detail, and establish a close connection between cycles in rule graphs (which are a variant of rule/goal graphs defined in [5, 28]) 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 show 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 and joins.

In the third part of our paper, in Section 7, we present 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. The performance metrics used in this section include number of joins, number of rule applications, number of iterations, join costs (without overheads) and duplicate elimination costs. We also discuss related work, which uses total elapsed time on a disk-based deductive database system as the metric, to show the benefits of using PSN over BSN. The terminology used in the rest of this paper is introduced in Section 2, and related work is presented in Section 3.

2 Background

2.1 Definitions

The language considered in this paper is that of Horn logic. Such a language has a countably infinite set of variables and countable sets of function and predicate symbols, these sets being pairwise disjoint. We adopt the Prolog convention of denoting variables by strings of characters starting with an upper case letter (e.g. $X, Y1$); function and predicate symbols are strings of characters starting with a lower case letter. It is assumed, without loss of generality, that with each function symbol f and each predicate symbol p is associated a unique natural number n, referred to as the *arity* of the symbol; f and p are then said to be n-ary symbols. A 0-ary function symbol is referred to as a *constant*.

A *term* in such a language is a variable, a constant, or a compound term $f(t_1, \ldots, t_n)$ where f is an n-ary function symbol and the t_i are terms. A *tuple* of terms is sometimes denoted simply by the use of an overbar, e.g. $\bar{t}$. A *literal* (or *predicate occurrence*) in such a language is of the form $p(t_1, \ldots, t_n)$ (a positive literal) or $\neg p(t_1, \ldots, t_n)$ (a negative literal), where p is an n-ary predicate symbol and the t_i are terms. Predicates in such a language are also referred to as *relations*. However, the two are interpreted differently; a relation is interpreted by a set of tuples and a predicate by a **true/false** function.

A substitution is a mapping from the set of variables of the language under consideration to the set of terms. Substitutions are denoted by lower case Greek letters θ, σ, φ, etc. A substitution σ is *more general* than a substitution θ if there is a substitution φ such that $\theta = \sigma[\varphi]$. Two terms t_1 and t_2 are said to be *unifiable* if there is a substitution σ such that $t_1[\sigma] = t_2[\sigma]$; σ is said to be a *unifier* of t_1 and t_2. Note that if two terms have a unifier, they have a most general unifier (*mgu*) that is unique up to renaming of variables.

A *clause* is the disjunction of a finite number of literals, and is said to be a Horn clause if it has

at most one positive literal. A Horn clause with exactly one positive literal is referred to as a *definite clause.* The positive literal in a definite clause is its *head,* and the remaining literals, if any, constitute its *body.* Following the syntax of Prolog, definite clauses (or *rules*) are written as:

$$p :- q_1, \ldots, q_n.$$

This is read declaratively as q_1 and ... and q_n implies p. A predicate definition consists of a set of definite clauses, whose heads all have the same predicate symbol; a goal is a set of negative literals. We consider a logic program to be a pair $< P, Q >$ where P is a set of predicate definitions and Q is the input, consisting of a query, or goal, and a (possibly empty) set of facts for "database predicates" appearing in the program.[1]

We follow the convention in deductive database literature of separating the set of rules with non-empty bodies from the set of facts, or unit clauses, which appears in P. The set of facts is called the *database.* Predicates appearing in the heads of rules with non-empty bodies are referred to as *derived* predicates, and predicates appearing in database facts are referred to as *base* predicates. The program can be normalized to make the set of base predicates disjoint from the set of derived predicates, and we assume that this has been done. The motivation for separating the program from the database is that optimizations are applied only to the program, and not to the database. This is important in the database context since the set of facts can be very large. However, the distinction is artificial, and we may choose to consider (a subset of) facts to be rules if we wish.

The meaning of a logic program is given by its least Herbrand model [31]. From [31], this is equivalent to the least fixpoint semantics. A derived 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 least Herbrand model of P.

We use the notion of derivation trees and derivation steps in several proofs.

Definition 2.1 Derivation tree : Given a program P with database D, derivation trees in $< P, D >$ are defined as follows:

- Every fact h in D is a derivation tree for itself, consisting of a single node with label h.
- Let R be a rule: p: $-q_1, \ldots, q_n$ in P, let $d_i, 1 \leq i \leq n$, be facts with derivation trees T_i, and let θ be the *mgu* of $(q_1, \ldots, q_n)$ and $(d_1, \ldots, d_n)$. Then the following is a derivation tree for $p[\theta]$: the root is a node labeled with $p[\theta]$ and R, and each $T_i, 1 \leq i \leq n$, is a child of the root.

A *derivation step* for fact $p[\theta]$ consists of a rule $R : p$: $-q_1, \ldots, q_n$, and facts $d_i, 1 \leq i \leq n$, with derivation trees, such that θ is the *mgu* of $(q_1, \ldots, q_n)$ and $(d_1, \ldots, d_n)$. Thus, a derivation step consists of a non-leaf node and all its children in a derivation tree. □

Note that the substitution θ is not applied to the children of $p[\theta]$ in the second part of the above definition. Thus, a derivation tree records which set of (previously generated) facts is used to generate a new fact using a rule, rather than the set of substitution instances of these facts that instantiated the rule. Derivation trees are important due to the following well-known property (see for example [21])

Proposition 2.1 . *For every fact t in the least Herbrand model, there is a derivation tree with root r, and a substitution σ such that $t = r[\sigma]$, and for every fact r' that is the root of a derivation tree, each fact t' that is a ground instance of r' is in the least Herbrand model.* □

[1] Our definitions and results can be extended to handle features such as negation, set grouping and aggregation [7].

Definition 2.2 Derivation height : The *height* of a derivation tree is defined to be the number of nodes in the longest path in the tree (which is always from the root to a leaf).

The *derivation height* of a fact is defined as the minimum among the heights of its derivation trees. □

Definition 2.3 Rule application : The *application* of a rule R, using a given set of facts D, denoted by $R(D)$, produces the set of *all* facts that can be derived in a single derivation step, using R and only the facts in D. □

We define an *evaluation* as a sequence of rule applications.

Definition 2.4 Non-repetition property : An evaluation is said to have the *non-repetition property* if no derivation step is repeated in the evaluation. □

Note that this definition intentionally does not distinguish between top-down and bottom-up algorithms.

A fact is said to have been *seen* by a rule R if the fact was available to an application of R. The *independent* application of a set of rules $\{R_1, \ldots, R_n\}$ on a set of facts D is defined as $R_1(D) \cup \ldots \cup R_n(D)$; i.e. each rule is applied once but the facts produced using a particular rule application are not seen by any of the other rules in the set. The *closure* of a set of rules $\mathcal{R} = \{R_1, \ldots, R_n\}$ using a given set of facts D refers to the derivation of all facts that can be computed using the given facts, and any number of applications of the rules, i.e.

$$closure(\mathcal{R}, D) = D \cup F(D) \cup F^2(D) \cup F^3(D) \cup \ldots$$

where $F(D) = R_1(D) \cup \ldots \cup R_n(D)$.

We also use some terminology from graph theory. A directed graph $G = (V, E)$ (where V is the vertex set and E is the edge set) is said to be *strongly connected* if every vertex in V is reachable (using the directed edges in E) from every other vertex in V. Given a directed graph G, a subgraph $G1$ of G is said to be a *strongly connected component* (SCC) of G if $G1$ is a maximal subgraph in G that is strongly connected. Note that the SCCs of a directed graph G partition the vertices of G. Define the *reduction* of G wrt the SCCs as the graph $G' = (V', E')$ obtained with vertex set V' being the set of SCCs $\{S_1, \ldots, S_k\}$ of G. An edge $(S_i, S_j), i \neq j, \in E'$ if there is an edge $(V_k, V_l) \in E$, where V_k is a vertex in S_i and V_l is a vertex in S_j. The reduction G' of G reflects the SCC structure of G, and is acyclic. An SCC S_1 is said to be *lower* than SCC S_2 in G if there is a (non-trivial) path from S_1 to S_2 in G'.

Definition 2.5 Rule (predicate) graph : Given a program P with rules $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$, we define the *rule graph* of P as the directed graph $G = (\mathcal{R}, E)$, where $(R_i, R_j) \in E$ iff the head of R_i unifies with a predicate occurrence in the body of R_j. We refer to SCCs in the rule graph as *Rule-SCCs*.

Given a program P with predicates $Pred = \{p_1, p_2, \ldots, p_n\}$, we define the *predicate graph* of P 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 SCCs in the predicate graph as *Pred-SCCs*. □

Where the distinction is not relevant, we refer to Rule-SCCs and Pred-SCCs as SCCs.

Given a program P, Pred-SCCs $S_1, \ldots, S_m$ are said to be in *topological order*, if whenever S_i contains a predicate used in S_j, then $i \leq j$.

Iteration	*Derivations made*
1	$\{\mathbf{R1 : anc(1,2), R1 : anc(2,3), R1 : anc(4,5)}\}$
2	$\{\mathbf{R2 : anc(1,3), R3 : anc(1,3)}, R1 : anc(1,2), R1 : anc(2,3), R1 : anc(4,5)\}$
3	$\{R2 : anc(1,3), R3 : anc(1,3), R1 : anc(1,2), R1 : anc(2,3), R1 : anc(4,5)\}$

Table 1: Derivations in a Naive Evaluation of P

2.2 Basic Semi-Naive Evaluation

Given a program P with an input database D, the *Naive evaluation* of $< P, D >$ proceeds in iterations. In each iteration, each of the rules in the program is independently applied on the set of available facts, and the set of available facts is updated at the end of the iteration. The Naive evaluation terminates when no new facts can be computed for any of the derived predicates of the program. The database D constitutes the initial set of available facts. This evaluation strategy can be refined by evaluating one strongly connected component (SCC) of the rule graph at a time, in a topological ordering of the Rule-SCCs. The Naive evaluation of a program on a database is extremely inefficient since it repeats a lot of derivations; each derivation step made in an application of a rule is repeated in every subsequent application of the same rule. Example 2.1 below illustrates this.

Example 2.1 Consider the following program $< P, Q >$:

$F1 : par(1,2).$
$F2 : par(2,3).$
$F3 : par(4,5).$
$R1 : anc(X,Y) :- par(X,Y).$
$R2 : anc(X,Y) :- par(X,Z), anc(Z,Y).$
$R3 : anc(X,Y) :- anc(X,Z), anc(Z,Y).$
Query: ?-$anc(1,X)$.

A naive bottom-up evaluation ([3]) of $< P, Q >$ would result in the derivations of *anc* facts as shown in Table 1. (The rule used to derive a fact is also indicated, and new derivations are shown in bold-face.) Only $anc(1,2)$ and $anc(1,3)$ are answers to the query. Note that each derivation made in an iteration of naive evaluation is repeated in subsequent iterations. □

Recall that an evaluation is said to have the non-repetition property if no derivation step is repeated in the evaluation. A number of researchers including [3, 6, 2] independently proposed an evaluation technique, which we call *Basic Semi-Naive*, or BSN, that has the non-repetition property. Given a program P, the *Basic Semi-Naive* (BSN) evaluation of P proceeds a Pred-SCC at a time in a topological ordering of the SCCs. There are essentially two components to the BSN evaluation of a Pred-SCC.

1. The first is a rewriting of the Pred-SCC S that defines "differential" versions of predicates, in order to distinguish facts that have been newly generated (and not yet used in inferences) from older facts.

   For each predicate p defined in S, we have four predicates $p, p^{old}, \delta p^{old}$ and δp^{new}. For each rule in S of the form:

   $$R : p :- q_1, \ldots, q_m.$$

   where $q_1, \ldots q_m$ are all non-recursive to p, the following semi-naive rewritten rule is obtained ([2]) from R:

   $$R_1 : \delta p^{new} :- q_1, \ldots, q_m.$$

Iteration	*Derivations made*
1	$\{\mathbf{R1} : \mathbf{anc(1,2)}, \mathbf{R1} : \mathbf{anc(2,3)}, \mathbf{R1} : \mathbf{anc(4,5)}\}$
2	$\{\mathbf{R2} : \mathbf{anc(1,3)}, \mathbf{R3} : \mathbf{anc(1,3)}\}$
3	$\{\}$

Table 2: Derivations in a BSN Evaluation of P

We call such rules *non-recursive* semi-naive rules.

For each rule in S of the form:

$$R : p :- p_1, \ldots, p_n, q_1, \ldots, q_m.$$

where $p_1, \ldots, p_n$ $(n > 0)$ are mutually recursive to p and $q_1, \ldots q_m$ are not, the following n semi-naive rewritten rules are obtained ([2]) from R:

$$\begin{aligned}
R_1 &: \delta p^{new} :- \delta p_1^{old}, p_2, \ldots, p_n, q_1, \ldots, q_m. \\
R_2 &: \delta p^{new} :- p_1^{old}, \delta p_2^{old}, p_3, \ldots, p_n, q_1, \ldots, q_m. \\
&\vdots \\
R_n &: \delta p^{new} :- p_1^{old}, p_2^{old}, \ldots, p_{n-1}^{old}, \delta p_n^{old}, q_1, \ldots, q_m.
\end{aligned}$$

We call such rules *recursive* semi-naive rules.

2. The second component is a technique to apply the rewritten rules and update these differentials, ensuring that all derivations are made exactly once.

   In evaluating S, the first iteration consists of applying each of the semi-naive (both non-recursive and recursive) rewritten rules in S. Subsequent iterations consist of applying only the recursive semi-naive rules. The evaluation of S proceeds by iterating until no new facts are computed for any of the predicates defined in S. After applying the semi-naive rules in an iteration, the extensions of the semi-naive relations for each p_i are updated using Procedure SN_Update below. (Note that the operators "$-$" and "$\cup$" involve subsumption checks if non-ground facts are generated.)

```
procedure SN_Update(p_i)
{
    p_i^old := p_i^old ∪ δp_i^old.
    δp_i^old := δp_i^new − p_i^old.
    p_i := p_i^old ∪ δp_i^old.
    δp_i^new := φ.
}
end SN_Update
```

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. This can be seen from the nature of the δ terms in rule bodies, and the order of updates of the various semi-naive relations.

Example 2.2 Consider again the program P of Example 2.1. Using BSN, the sequence of derivations made is shown in Table 2. Note that each derivation made in the naive evaluation is also made in the BSN evaluation. However, the BSN evaluation does not repeat any derivations. But if a fact ($anc(1,3)$ in this case) is derived by two *different* derivations, each of these derivations is made in the BSN evaluation.

Such "redundant" derivations can be avoided by recognizing that every fact derived using rule $r2$ is also derived using $r3$ and vice versa. Consequently, a scheme that prunes redundant derivations ([12], for instance) could recognize this and never apply rule $R3$; the derivation of $R3 : anc(1,3)$ is not made in such a case. Given a program that has redundant derivations, its evaluation using BSN does *not* eliminate such derivations.

Rewriting the program P of Example 2.1 using Magic Sets, and evaluating the rewritten program also results in avoiding the derivation of $anc(4,5)$, since this is "irrelevant" to computing the answers to the query. Refer to [8, 23] for more details. □

2.3 Control Expressions

Helm [13, 12] introduced the notion of control on the bottom-up evaluation of logic programs using control expressions, and also looked at control as a way of increasing the efficiency of evaluation by eliminating some redundant derivations. Control expressions have also been proposed to let the user specify a desired semantics in the presence of negation ([14]), as well as evaluate the Magic Sets transformation of stratified logic programs ([9]). While 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. The syntax of our control expressions is given by the following grammar.

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

where S is the start symbol of the grammar.

A control expression α is a (non-deterministic) 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 for the base predicates. The following equations recursively define the semantics of control expressions.

1. If $\alpha = R_i$, $\alpha(D) = D \cup R_i(D)$.
2. If $\alpha = \alpha_1 \cdot \alpha_2$, $\alpha(D) = \alpha_2(\alpha_1(D))$.
3. If $\alpha = \alpha_1 \oplus \alpha_2$, non-deterministically choose α_1 or α_2 and call it β_1 and call the other β_2. if $\beta_1(D) \neq D$, $\alpha(D) = \beta_1(D)$, else $\alpha(D) = \beta_2(D)$.
4. $\alpha = \alpha_1^\bullet$, $\alpha(D) = \alpha_1^i(D)$, for some arbitrary choice of $i \geq 0$, where $\alpha^0(D) = D$ and $\alpha^{j+1}(D) = \alpha(\alpha^j(D)), j \geq 0$.
5. If $\alpha = \alpha_1^*$, $\alpha(D) = \alpha_1^i(D), i > 0$, such that $\alpha_1^{i+1}(D) = \alpha_1^i(D)$, where $\alpha^0(D) = D$ and $\alpha^{j+1}(D) = \alpha(\alpha^j(D)), j \geq 0$.
6. If $\alpha = \alpha_1 + \alpha_2$, $\alpha(D) = \alpha_1(D) \cup \alpha_2(D)$.

The semantics of the control operators that we consider are different from those considered by Helm and we do not discuss the evaluation of Helm's control operators in this paper.

The control expressions we use form a superset of those used by Imielinski and Naqvi [14]. It follows from [14] that our control expressions can be used to specify the inflationary semantics for negation ([17]). Our control expressions also form a superset of those used by Beeri et al. [9].

3 Related Work

The idea of semi-naive evaluation (also referred to as differential evaluation) dates back several years and has been independently rediscovered many times. The concept appears in Fong and Ullman [10] and in Paige and Schwartz [22]. In the area of deductive database systems the idea appears independently in Bayer [6], Bancilhon [3] and Balbin and Ramamohanarao [2]. Apt [1] sets up fixpoint computation in a formal setting, and systematically derives the Basic Semi-Naive algorithm.

Gonzalez-Rubio, Rohmer and Bradier [11] present a technique for parallelizing semi-naive evaluation. Their technique has some interesting connections to one of our techniques (GSN). We defer a discussion of the connections until Section 4.5 so that we can refer to details of GSN evaluation in the discussion.

Schmidt et al. [26] present a scheme for ordering facts in a semi-naive evaluation. In some contexts it is beneficial to prioritize the use of facts. For instance, when evaluating Quantitative logic programs [27], it is beneficial to use facts with higher certainities first, and delay the use of other facts until there is no unused fact with higher certainity. The idea behind semi-naive evaluation is to partition facts in each relation p into those that have been "seen" by all rules (the p^{old} relations) and those that have not been used in some rule applications yet (i.e, the δp relations). The Sloppy Delta iteration technique of Schmidt et al., instead partitions each relation p into three parts. There is a p^{old} as before, the δp relation holds a set of facts that has not yet been used in some rule application, but will be used in rule applications during the current iteration, and there is a third set of facts that are "hidden" and cannot be used until they are made visible (by some meta-level control). When a hidden p fact is made visible, it is moved into the δp relation as if it was newly derived. This technique is orthogonal to the ordering of rules, and can be used in conjunction with the techniques described in this paper.

Transitive closure is an important special case of recursion in deductive databases. Ioannidis [15] and Valduriez and Boral [29] present algorithms for computing the transitive closure that perform significantly fewer joins and take fewer iterations than the traditional iterative algorithm, without performing redundant computation. Valduriez and Khoshafian [30] present a technique for parallelizing a differential computation of the transitive closure without performing redundant computation. Lu [20] and Kabler et al. [16] 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 accesses. However, all these papers make use of special properties of transitive closure. For example, the "Smart" algorithm of [15, 29] does not permit selections in the rules that define the transitive closure. It is not clear whether it is possible to or how to generalize these algorithms beyond transitive closure.

Schmidt [25] presents an evaluation technique that allows for some ordering of rules; however, this technique lacks the non-repetition property, and repeats derivations. This can adversely affect performance. Further, it makes the algorithm inapplicable when the non-repetition property is used to perform further optimizations. (For example, if a program has the *duplicate-freedom* property, a fixpoint algorithm with the non-repetition property can be modified easily to eliminate run-time checks for duplicates [21].)

Kuittinen et al. [18] proposed a fixpoint evaluation algorithm for logic programs based on the imme-

diate utilization of facts. Their algorithm also reduces the number of iterations, and is dependent upon a choice of rule orderings. They present a performance study of the effects of rule orderings, although they do not analyze the effect of rule orderings theoretically. The results presented in our paper and in [18] were obtained independently. (However, we have drawn upon and extended their performance evaluation.) Although the technique of Kuittinen et al. avoids repeating most derivations, it does not have the non-repetition property since it is possible for some derivations to be repeated. Further, their technique handles only a subset of the rule orderings that GSN can handle.

4 General Semi-Naive Evaluation

We now present a technique, *General Semi-Naive*, or GSN, evaluation, that makes facts computed by an application of a rule R available to *all* other rule applications immediately after the application of R, while maintaining the non-repetition property. We first look at how to apply a single rule using the GSN technique. Then, in Section 4.3, we look at how to evaluate general control expressions using this technique. Finally, in Section 4.4, we look at a specific form of control expression to evaluate a program Pred-SCC by Pred-SCC, in a topological ordering of the Pred-SCCs.

4.1 GSN for a Rule

Consider any rule R_i in the program:

$$R_i : p_h(\overline{t}) :- p_1(\overline{t_1}), \ldots, p_n(\overline{t_n}).$$

where $p_h, p_1, \ldots, p_n$ are not necessarily distinct. Similar to BSN evaluation, the GSN evaluation of a rule consists of two components.

1. The first is the semi-naive rewriting that defines the "differential" versions of predicates.

   Associated with each rule R_i of the program, and each (non-recursive as well as recursive) predicate p_j that occurs in the body of R_i, we maintain a relation p^{old}_{j,R_i}. (Even if a predicate occurs more than once in the body of R_i, only one copy of the relation needs to be maintained per rule.) The set of relations p^{old}_{j,R_i}, $1 \leq j \leq n$, has the property that every derivation that can be made by an application of R_i using only these facts has already been made. (In this respect, it is similar to the p^{old} relations maintained by BSN.)

   Associated with each predicate p_j, we also maintain its complete extension; only one copy of p_j needs to be maintained independent of the number of rules p_j occurs in. We also have "temporary" relations δp_j^{new} and δp_j^{old} associated with each derived predicate p_j that appears in the program. These δ relations are used to keep track of "new" facts; facts which have not been seen by a rule application.

   The semi-naive rewriting for GSN is obtained as follows. Each predicate p_j in the body of R_i is treated as being recursive to the head. The semi-naive rewriting described for BSN (in Section 2.2) is modified by replacing each predicate occurrence p_j^{old} in the body of the semi-naive rewritten versions of rule R_i by p^{old}_{j,R_i}.

2. The second is a technique to apply the rewritten rules and update these differentials in the order specified for rule applications, ensuring that all derivations are made exactly once. Procedure GSN_Rule below describes the application of a rule R_i on a set of facts D.

procedure GSN_Rule(R_i, D)
{
/* All relations mentioned below are part of D. */
Let p_h be the predicate defined by rule R_i.
(1) For every predicate p_j in the body of R_i
$\delta p_j^{old} := p_j - p_{j,R_i}^{old}$.
(2) $\delta p_h^{new} := \phi$.
(3) Apply each semi-naive rewritten version of R_i
(modified as described earlier) independently.
(4) For every predicate p_j in the body of R_i, $p_{j,R_i}^{old} := p_j$.
(5) $p_h := p_h \cup \delta p_h^{new}$.
(6) return δp_h^{new}.
}
end GSN_Rule

Example 4.1 Consider rule $R3$ from P in Example 2.1.

$$R3 : anc(X,Y) :- anc(X,Z), anc(Z,Y).$$

The semi-naive rewriting of this rule for GSN evaluation is given by:

$$R3' : \delta anc^{new}(X,Y) :- \delta anc^{old}(X,Z), anc(Z,Y).$$
$$R3'' : \delta anc^{new}(X,Y) :- anc_{R_3}^{old}(X,Z), \delta anc^{old}(Z,Y).$$

Prior to the application of these rules, δanc^{old} is initialized to $anc - anc_{R_3}^{old}$, i.e. the tuples that have not been seen as yet by an application of R_3. The relation δanc^{new} is initialized to ϕ. Application of rules $R3'$ and $R3''$ (potentially) results in adding facts to the δanc^{new} relation. After the application of this rule, each fact in δanc^{old} also has been seen by R_3. Consequently, the $anc_{R_3}^{old}$ relation is updated to reflect this. The newly generated facts are also added to the anc relation.

Example 4.2 describes the order in which rules are applied (and relations updated) in the GSN evaluation of program P. □

4.2 Efficient Implementation of GSN

The description of General Semi-Naive 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 implementation in which each relation is maintained as a list of tuples. The new facts produced by a rule application are appended to the extension of the predicate. (In the case of generalized tuples, if a new fact subsumes an existing fact then the existing fact may have to be deleted.)

In step (4) of GSN_Rule(R_i, D), every fact in a relation p_j becomes part of p_{j,R_i}^{old}. Since the tuples in each relation p_j are ordered by "time" of insertion, a pointer to the end of the extension of p_j at step (4) has the following property: every p_j fact preceding the pointer is part of p_{j,R_i}^{old}, and every p_j fact occurring after the pointer has not been seen by rule R_i yet. Hence, the extension of each p_{j,R_i}^{old} is replaced by a pointer into the extension of p_j. Similarly, we replace the extension of each δp_j^{old} by another pointer into the extension of p_j such that the set of facts between the pointers for p_{j,R_i}^{old} and

δp_j^{old} constitutes the extension for δp_j^{old}. The set of facts beyond the pointer for δp_j^{old} constitutes the extension of δp_j^{new}. 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_j^{old} relations is also eliminated.

Indices are important for efficiently accessing a relation during evaluation. The pointers into the extent of a relation p_j partition it based on the order of insertion of tuples. Each of the relations p_{j,R_i}^{old}, $\delta p_{j,R_i}^{old}$ and δp_j^{new} is the union of a contiguous set of these partitions. To index any of these relations, we index in turn each partition that is contained in it. Since there is at most one pointer per rule into p_j, the number of partitions is bounded by the number of rules in the program. There are two operations on pointers into the relation. The first is the creation of a new pointer to the end of a relation, and the second is the deletion of a pointer into a relation. In the first case a new partition is created (and is initially empty), and in the second case two existing partitions get merged. This involves moving tuples from the indices of one partition to the indices of the other. However, each tuple is moved in this fashion only a constant number of times, and hence this will not increase the time complexity of evaluation. Thus, this indexing technique can be expected to provide efficient access to tuples. (Alternatively, we could index each of the relations p_{j,R_i}^{old}, $\delta p_{j,R_i}^{old}$ and δp_j^{new} separately. This could result in each tuple being indexed multiple times.)

4.3 Implementing Control Expressions Using GSN

The semantics of control expressions was described (using recursive equations) in Section 2.3. This description also suggests a straightforward way of evaluating control expressions, and Procedure Simple_CE(α, D) below describes this. In this evaluation, every fact produced by a rule application on a given database is immediately added to the database, and made available to subsequent rule applications. Unfortunately, such an evaluation does not have the non-repetition property, and can be extremely inefficient.

```
procedure Simple_CE(α, D)
{
    /* We need to evaluate control expression α on database D. */
    case
        (1) α = R_i:   /* the exit case */
                return D ∪ R_i(D).
        (2) α = α_1 · α_2:
                return Simple_CE(α_2, Simple_CE(α_1, D)).
        (3) α = α_1 ⊕ α_2:
                Non-deterministically choose α_1 or α_2 and call it β_1 and call the other β_2.
                if Simple_CE(β_1, D) ≠ D, return Simple_CE(β_1, D)
                else return Simple_CE(β_2, D).
        (4) α = α_1^•:
                Choose n ≥ 0 non-deterministically.
                for i = 1 to n do
                    Let D denote the result of evaluating Simple_CE(α_1, D).
                end for
                return D.
        (5) α = α_1^*:
                repeat
```

```
                Let D' denote D at this stage.
                Let D denote the result of evaluating Simple_CE(α1, D).
            until (D = D')
            return D.
        (6) α = α1 + α2:
            return Simple_CE(α1, D) ∪ Simple_CE(α2, D).
    end case
}
end Simple_CE
```

Note that Procedure Simple_CE is non-deterministic in that the final database state may not be uniquely determined if the control expression contains $\oplus$ or $^\bullet$.

Since a control expression is a non-deterministic mapping, we use {Semantics(α, D)} to denote the set of all possible database states that are results of $\alpha(D)$. Similarly, we use {Simple_CE(α, D)} to denote the set of all possible results of the procedure Simple_CE(α, D).

Theorem 4.1 *For all databases D, and control expressions α,*

$$\{\text{Simple\_CE}(\alpha, D)\} = \{\text{Semantics}(\alpha, D)\}$$

Proof: The proof follows in a straightforward manner from the direct correspondence between each case in the semantics of control expressions and Procedure Simple_CE. □

Consider the restricted set of control expressions generated by the grammar described in Section 2.3 without the production that uses the "+" operator. Procedure GSN_CE (α, D) below describes how we can evaluate such a control expression while preserving the non-repetition property.

```
procedure GSN_CE(α, D)
{
    α is a control expression and D is an initial database.
    /* D contains the p_j as well as all p_{j,R_i}^{old} relations. */
    case
        (1) α = R_i:
            return D ∪ GSN_Rule(R_i, D).
        ⋮
            /* Cases (2)-(5) remain unchanged from Simple_CE(α, D). */
    end case
}
end GSN_CE
```

The updates to the various differential relations (corresponding to the predicates occurring in the head and body of rule R_i) performed by GSN_Rule maintain the set of facts that have been used by the rule R_i in previous applications of R_i. This ensures that in subsequent applications of R_i, none of the previous derivations is repeated. Intuitively, each rule in the program individually ensures that its applications have the non-repetition property; we prove this claim formally in Theorem 4.2. First we need the following proposition and its corollary.

Proposition 4.1 *Suppose a control expression α does not use the operator "+". Given a database D, any evaluation of* Simple_CE(α, D) *performs a sequence of rule applications; each rule application in the sequence takes as input the set of all facts derived up to the previous rule application in the sequence.*

Proof: We prove this by induction on the size of the control expression. For the base case, when the control expression consists of a single rule, this is trivial. For the induction step, assume that this is true for all control expressions of size less than n, and consider a control expression α of size n. The evaluation of α can be according to one of cases (2), (3), (4) or (5) of Procedure Simple_CE. In each of the cases, it is easily seen from the induction hypothesis that the claim is true of α. This proves the claim. □

As a corollary to the above proposition, we have the following:

Corollary 4.1 *Suppose a control expression α does not use the operator "+". Given a database D, any evaluation of* GSN_CE(α, D) *performs a sequence of calls to* GSN_Rule*; each call takes as input the set of all facts derived up to the previous call in the sequence.* □

Since GSN_CE is a non-deterministic procedure, we use {GSN_CE(α, D)} to denote the set of all possible results of GSN_CE(α, D).

Theorem 4.2 *For all databases D, and control expressions α that have no occurrence of the "+" operator,*

$$\{\text{GSN\_CE}(\alpha, D)\} = \{\text{Simple\_CE}(\alpha, D)\}$$

Further, evaluations of GSN_CE *have the non-repetition property.*

Proof: By Proposition 4.1 and Corollary 4.1, Simple_CE (resp. GSN_CE) evaluations of control expressions that do not contain "+" perform a sequence of rule applications (resp. calls to GSN_Rule).

We first prove three properties of a sequence of calls to GSN_Rule. Consider a particular call to GSN_Rule(R_i, D) in this sequence of calls.
P1. Before step (3) of GSN_Rule, for each predicate p_j occurring in the body of R_i, the set of facts $\delta p_j^{old} = p_j - p_{j,R_i}^{old}$ has 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_{j,R_i}^{old}, for all j, has already been made.
P3. After step (3) (but before step (5) where p_h is updated) of GSN_Rule, every derivation that could be made using R_i and only the facts in the set of relations p_j, for all j, has been made.

We prove these by induction on the number of times GSN_Rule$(R_i, _)$ is called. Before the first call to GSN_Rule$(R_i, _)$, p_{j,R_i}^{old} is empty for all p_j in the body of R_i. The call to GSN_Rule$(R_i, _)$ computes all facts that can be derived from the given set of facts for the body predicates. Hence, **P1** and **P2** are true before step (3) and **P3** is true after step (3), and the basis holds.

For the induction step, assume that **P1**, **P2**, and **P3** hold up to the kth call to GSN_Rule$(R_i, _)$, and consider the $k+1$th call to GSN_Rule$(R_i, _)$. Steps (4) and (5) of GSN_Rule in the kth call to GSN_Rule$(R_i, _)$, and step (1) in the $k+1$th call to GSN_Rule$(R_i, _)$ ensures **P1**. Since **P3** holds in the kth call to GSN_Rule$(R_i, _)$, step (4) guarantees that **P2** holds in the $k+1$th call to GSN_Rule$(R_i, _)$. Since **P2** holds in the $k+1$th call to GSN_Rule$(R_i, _)$, step (1) and the correctness of semi-naive rewriting ensures that **P3** holds in the $k+1$th call to GSN_Rule$(R_i, _)$. This completes the proof of **P1**, **P2**, and **P3**.

The GSN evaluation of a control expression without any occurrence of the "+" operator proceeds sequentially and does not have any parallel branches; databases never need to be merged and, hence, for

each predicate p and each rule R that uses p, the relations p_R^{old} and δp^{old} are well defined. Property **P1** then ensures that no derivations using any rule R are repeated and hence evaluations of GSN_CE (for control expressions without the "+" operator) have the non-repetition property.

Claim 1: Consider a particular evaluation of Simple_CE(α, D) (where the choices, if any, due to occurrences of the $\oplus$ or $\bullet$ operators, have been made). Then, there is an equivalent evaluation of GSN_CE(α, D).

Proof of Claim 1: By Proposition 4.1, the evaluation of Simple_CE(α, D) performs a sequence of rule applications. Now replace each rule application $R_i(D)$ in this sequence by a call to GSN_Rule(R_i, D). By property **P3** above, after the call every derivation that could be made using R_i and only the facts in D has been made. Therefore, at this point, every fact that would be derived by $R_i(D)$ has already been derived. Hence, at each point in the modified evaluation, the set of facts that has been derived is exactly the same as that derived by the evaluation of Simple_CE(α, D).

We claim that the modified evaluation is a possible evaluation of GSN_CE(α, D). After each call to GSN_Rule the set of available facts is the same as the set of facts available after the corresponding rule application in the evaluation of Simple_CE. Hence, whatever choice was made by Simple_CE at any point in the evaluation can also be made by an evaluation of GSN_CE. Hence, the resultant evaluation is a valid evaluation of GSN_CE(α, D).

Claim 2: Consider a particular evaluation of GSN_CE(α, D) (where the choices, if any, due to occurrences of the $\oplus$ or $\bullet$ operators, have been made). Then, there is an equivalent evaluation of Simple_CE(α, D).

Proof of Claim 2: The proof proceeds exactly the same as the proof of Claim 1, and we omit the details.

This ends the proof of the theorem. □

Evaluating control expressions that contain the "+" operator using the GSN evaluation technique presents some difficulties. Recall that the evaluation of the "+" operator involved merging the two databases obtained by evaluating the operands of the "+" operator independently on the input database. Merging the p_{j,R_i}^{old} relations produced by the two operands of the "+" cannot in general be done consistently. In the special case that the two operands of a "+" operator do not have any rule in common, we can merge the databases consistently, by merely taking the union of the two. This handles the case of control expressions of the form $(R_1 + R_2 + \ldots + R_m)^*$, which simulate Basic Semi-Naive evaluation, as well as control expressions in the class $PCE(P)$ (described in Section 5.1), which can be evaluated using Predicate Semi-Naive evaluation.

4.4 GSN for a Program

Consider a program P. With each Pred-SCC S in P, we can associate all the (non-recursive and recursive) rules defining the predicates in S. The evaluation of a S consists of repeated application of the rules associated with S, and the program P can be evaluated Pred-SCC by Pred-SCC in a topological ordering of the Pred-SCCs of P. Procedure GSN_Prog below describes such a GSN evaluation of a program. The GSN evaluation of a single Pred-SCC is as described by GSN_SCC.

```
procedure GSN_Prog(P)
{
    Let S_1, ..., S_m be a topological ordering of the Pred-SCCs of P.
```

```
        /* S_0 is assumed to contain all the base predicates. */
    for j = 1 to m do    GSN_SCC(S_j)
}
end GSN_Prog

procedure GSN_SCC(S_i)
{
    Let the ordering of the non-recursive rules for the predicates in S_i be E_i^1, ..., E_i^m.
    Let the ordering of the recursive rules for the predicates in S_i be R_i^1, ..., R_i^n.
    if no predicate in S_i has any recursive rule defining it, then
        Evaluate the following control expression:
        (E_i^1 . ... . E_i^m) using GSN_CE.
    else   /* at least one of the predicates in S_i has a recursive rule defining it */
        Evaluate the control expression
        (E_i^1 . ... . E_i^m) . (R_i^1 . ... . R_i^n)* using GSN_CE.
}
end GSN_SCC
```

In the evaluation of a Pred-SCC S of program P, the non-recursive rules are applied once, followed by the repeated application of the recursive rules of S. The Pred-SCC structure of P ensures that each non-recursive rule E in S only has body predicates that are in "lower" Pred-SCCs of P. Consequently, the facts produced by the application of any of the non-recursive rules cannot be used by any of the other non-recursive rules in S; thus, the order of application of the non-recursive rules in GSN_SCC is irrelevant.

Theorem 4.3 *Procedure* GSN_Prog *has the non-repetition property, is sound and, if all the predicates defined in the program are safe, is complete wrt the least fixpoint semantics.*

Proof: Let $<$ be a topological ordering of Pred-SCCs such that $S_i < S_j$, if S_i contains a predicate used in a rule in S_j. The proof is by induction on this ordering $<$.

As the basis, consider S_0. It contains only base predicates and the theorem holds trivially. Suppose it holds for S_i. Consider evaluation of Pred-SCC S_{i+1}. Since each predicate defined in the program is safe, the set of facts in Pred-SCCs $S_0, \ldots, S_i$ are all enumerated before evaluation of S_{i+1} begins, by the induction hypothesis. From Theorem 4.2, the evaluation of S_{i+1} has the non-repetition property. Also, it is sound and complete wrt the control expression semantics of $(E_{i+1}^1 \cdot \ \ldots \ \cdot E_{i+1}^m) \cdot (R_{i+1}^1 \cdot \ \ldots \ \cdot R_{i+1}^n)^*$.

A simple induction shows that the set of facts generated with this control expression includes all facts for which a derivation tree exists. Since all lower Pred-SCCs have been fully evaluated, let us treat the set of facts for these (lower) Pred-SCCs to be base facts, of derivation height 0. Since each of the non-recursive rules is applied once in the control expression $(E_{i+1}^1 \cdot \ \ldots \ \cdot E_{i+1}^m)$, any fact with a derivation height of 1 is derived during the evaluation of this control expression. We show by induction on the derivation heights of facts in this Pred-SCC that each fact with a derivation height of $h, h > 0$ will be derived in or before the $h-1^{th}$ iteration of the control expression $(R_{i+1}^1 \cdot \ \ldots \ \cdot R_{i+1}^n)^*$. For the basis case, facts of derivation height 1 are derived using the control expression $(E_{i+1}^1 \cdot \ \ldots \ \cdot E_{i+1}^m)$, and hence are derived before any iteration of the control expression $(R_{i+1}^1 \cdot \ \ldots \ \cdot R_{i+1}^n)^*$. For the induction step, suppose the hypothesis holds for facts with derivation height h. Since every recursive rule in S_{i+1} is applied in iteration h, any fact with derivation height $h+1$ is derived in or before the h^{th} iteration of the control expression $(R_{i+1}^1 \cdot \ \ldots \ \cdot R_{i+1}^n)^*$. This completes the induction and the proof of the theorem. □

Iter/Rule	*Relation*	*Facts in relation at end of rule application (New facts in bold-face)*
$0/R1$	anc	$\{\mathbf{R1 : anc(1,2), R1 : anc(2,3), R1 : anc(4,5)}\}$
	par	$\{F1 : par(1,2), F2 : par(2,3), F3 : par(4,5)\}$
	par^{old}_{R1}	$\{\mathbf{F1 : par(1,2), F2 : par(2,3), F3 : par(4,5)}\}$
$1/R2$	anc	$\{R1 : anc(1,2), R1 : anc(2,3), R1 : anc(4,5), \mathbf{R2 : anc(1,3)}\}$
	anc^{old}_{R2}	$\{\mathbf{R1 : anc(1,2), R1 : anc(2,3), R1 : anc(4,5)}\}$
	par	$\{F1 : par(1,2), F2 : par(2,3), F3 : par(4,5)\}$
	par^{old}_{R2}	$\{\mathbf{F1 : par(1,2), F2 : par(2,3), F3 : par(4,5)}\}$
$1/R3$	anc	$\{R1 : anc(1,2), R1 : anc(2,3), R1 : anc(4,5), R2 : anc(1,3), \mathbf{R3 : anc(1,3)}\}$
	anc^{old}_{R3}	$\{\mathbf{R1 : anc(1,2), R1 : anc(2,3), R1 : anc(4,5), R2 : anc(1,3)}\}$
$2/R2$	anc	$\{R1 : anc(1,2), R1 : anc(2,3), R1 : anc(4,5), R2 : anc(1,3), R3 : anc(1,3)\}$
	anc^{old}_{R2}	$\{R1 : anc(1,2), R1 : anc(2,3), R1 : anc(4,5), \mathbf{R2 : anc(1,3)}, \mathbf{R3 : anc(1,3)}\}$
	par	$\{F1 : par(1,2), F2 : par(2,3), F3 : par(4,5)\}$
	par^{old}_{R2}	$\{F1 : par(1,2), F2 : par(2,3), F3 : par(4,5)\}$
$2/R3$	anc	$\{R1 : anc(1,2), R1 : anc(2,3), R1 : anc(4,5), R2 : anc(1,3), R3 : anc(1,3)\}$
	anc^{old}_{R3}	$\{R1 : anc(1,2), R1 : anc(2,3), R1 : anc(4,5), R2 : anc(1,3), \mathbf{R3 : anc(1,3)}\}$

Table 3: Derivations in a GSN Evaluation of P

We illustrate GSN_Prog using an example.

Example 4.2 Consider again program P of Example 2.1.

$F1 : par(1,2).$
$F2 : par(2,3).$
$F3 : par(4,5).$
$R1 : anc(X,Y) :- par(X,Y).$
$R2 : anc(X,Y) :- par(X,Z), anc(Z,Y).$
$R3 : anc(X,Y) :- anc(X,Z), anc(Z,Y).$

Facts $F1, F2$ and $F3$ are in S_0. Rules $R1, R2$ and $R3$ are all in Pred-SCC S_1. A Pred-SCC by Pred-SCC evaluation of P using GSN_Prog would proceed as follows.

S_1 contains only one non-recursive rule, $R1$. Assume the ordering of recursive rules in S_1 to be $R2, R3$. (Alternatively, $R3, R2$ could be the rule order assumed, and subsequent details worked through.) Consequently, the evaluation of S_1 requires evaluation of the control expression $(R1) \cdot (R2 \cdot R3)^*$. Table 3 describes the facts in the various relations during the GSN_Prog evaluation of P using this control expression.

Only the relevant relations that affect and are affected by the rule application are shown. Other relations remain unchanged from the previous step. Initially, par contains $\{F1 : par(1,2), F2 : par(2,3), F3 : par(4,5)\}$, and all other relations are empty. Iteration 0 refers to the evaluation of the control expression $(R1)$. Subsequent iterations refer to the evaluation of the control expression $(R2 \cdot R3)^*$.

Note that the *same* derivations are made in the GSN evaluation of P as in the BSN evaluation of P (Table 2). □

We discuss the issues involved in selecting an ordering of rules in Section 6.

4.5 Related Work

The parallel evaluation technique of Gonzalez-Rubio, Rohmer and Bradier [11] has some similarity to GSN evaluation. The idea behind their technique is to run the a copy of the BSN algorithm on each of a number of processors. When a fact is derived it is transmitted to some of the other processors (and may in some cases not be used locally). Their parallelization technique provides the mechanics of where to send/use facts, and is interesting in its own right, but it does not concern us in this paper. When a fact is received at a processor it is treated exactly as if it was derived locally during an iteration - it is saved in a δ relation and used in the next iteration.

In the algorithm of Gonzalez-Rubio et al., each processor runs a copy of the entire program. However, suppose we instead assigned one rule to each processor, and used the following scheme for transmitting facts: transmit a fact to any processor that could use it in the body of its rule. Then the relations p_j^{old} and δp_j^{old} in the BSN algorithm used in the processor with rule R_i would correspond to the relations p_{j,R_i}^{old} and δp_j^{old} in GSN. Their technique does not order the rules, and the rules would be applied repeatedly asynchronously. However we could conceivably run the processors one at a time in a desired order to achieve rule ordering. Thus, while the techniques of Gonzalez-Rubio et al. cannot handle rule orderings, it could provide a basis for deriving GSN evaluation. Our work was done independently of theirs.

5 Predicate Semi-Naive Evaluation

We now present a technique, *Predicate Semi-Naive*, or PSN, evaluation that has the non-repetition property and can utilize facts produced by a rule application in the same iteration they have been derived in, though not immediately.[2]. This technique maintains just one version of the extension of each predicate and incurs no additional overheads compared to BSN evaluation. We first look at how to evaluate a restricted set of control expressions for evaluating a program using this technique. Then, in Section 5.2, we look at a specific form of control expression to evaluate a program Pred-SCC by Pred-SCC in a topological ordering of the Pred-SCCs.

Similar to BSN (and GSN) evaluation, there are two components to PSN evaluation.

1. The first is the semi-naive rewriting that defines "differential" versions of predicates.
   Similar to BSN evaluation, we maintain three relations p^{old}, δp^{old} and δp^{new} for each predicate p defined in the program. The semi-naive rewriting is identical to the rewriting described for BSN evaluation.
2. The second component is a technique to apply the rewritten rules and update these differentials, ensuring that all derivations are made exactly once. Procedure PSN_CE describes this component for a restricted class of control expressions for a program.

5.1 Implementing Control Expressions Using PSN

Using PSN, we can evaluate a restricted class of control expressions while obtaining the non-repetition property. For a program P, this class of control expressions is given by $PCE(P)$.

[2] Although we describe just two evaluation strategies, a gradation is possible between GSN and PSN evaluation of 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 elaborate further on this.

The class $PCE(P)$:

Consider a program P. The class of expressions $PCE(P)$ is as follows. Let $S_1, \ldots, S_k$ be any topological ordering of the Pred-SCCs of P. Then, the expression $(PCE(S_1) \cdot \quad \ldots \quad \cdot PCE(S_k))$ is in the class $PCE(P)$.

The class $PCE(S)$:

Consider a Pred-SCC S. The class of expressions $PCE(S)$ is as follows. Let $p_1, \ldots, p_m$ be some (not necessarily all) of the predicates in S. Let $s_1, \ldots, s_n$ be any partitioning of the predicates $p_1, \ldots, p_m$. Let $E^1_{s_i}, \ldots, E^{a_i}_{s_i}$ be the non-recursive rules defining predicates in s_i, and let $R^1_{s_i}, \ldots, R^{b_i}_{s_i}$ be the recursive rules defining predicates in $s_i, 1 \leq i \leq n$. Let E_{s_i} stand for the control expression $E^1_{s_i} + \ldots + E^{a_i}_{s_i}$, and let R_{s_i} stand for the control expression $R^1_{s_i} + \ldots + R^{b_i}_{s_i}$. If none of $p_1, \ldots, p_m$ has any recursive rule defining it, then

$$(E_{s_1}) \cdot \quad \ldots \quad \cdot (E_{s_n})$$

is in $PCE(S)$. If at least one of $p_1, \ldots, p_m$ has a recursive rule defining it, then

$$(E_{s_1} + R_{s_1}) \cdot \quad \ldots \quad \cdot (E_{s_n} + R_{s_n}) \cdot (R_{s_1} \cdot \quad \ldots \quad \cdot R_{s_n})^*$$

is in $PCE(S)$.

Procedure PSN_CE(α, D) below describes the PSN evaluation of a restricted set of control expressions, without occurrences of the "$\oplus$" and the "*" operators. Note that the absence of these operators means that each control expression is a deterministic mapping.

```
procedure PSN_CE(α, D)
{
    /* We need to evaluate control expression α on database D. */
    case
        (1) α = R_1 + ... + R_n:      /* the exit case */
                for i = 1 to n do
                    Apply each semi-naive rewritten version of R_i on database D.
                    Let D denote the resulting database.
                end for
                for every predicate p_j defined by a rule in α, call SN_Update(p_j).
        (2) α = α_1 · α_2:
                Evaluate PSN_CE(α_1, D).
                Let D' denote the resulting database.
                Evaluate PSN_CE(α_2, D').
        (3) α = (α_1)*:
                repeat
                    Let D' denote D at this stage.
                    Evaluate PSN_CE(α_1, D).
                    Let D denote the resulting database.
                until (D = D')
    end case
}
end PSN_CE
```

The Simple_CE evaluation of a control expression in $PCE(S)$ first applies the rules in $(E_{s_1} + R_{s_1})$ independently. All facts computed are then added to the database. This is followed by applying the rules in $(E_{s_i} + R_{s_i})$ independently, adding facts computed to the database, for successive values of i until $i = n$. The Simple_CE evaluation then applies the rules in R_{s_j} independently, adding facts computed to the database, for $j = 1 \ldots n$, and repeats this until nothing new is derived.

The PSN_CE evaluation of a control expression in $PCE(S)$ is very similar. It first applies the (semi-naive versions of the) rules in $(E_{s_i} + R_{s_i})$, followed by an SN_Update of the predicates in the partition s_i, for successive values of i until $i = n$. The PSN_CE evaluation then applies the rules in R_{s_j} followed by an SN_Update of the predicates in the partition s_j for $j = 1 \ldots n$, and repeats this until nothing new is derived.

In the proof of subsequent results, we often use a numbering of the rule applications in the PSN_CE evaluation of a control expression in $PCE(S)$. We number rule applications using successive integers starting from 1 as follows. Each rule application in the control expression $(E_{s_1} + R_{s_1})$ is numbered 1; each rule application in the control expression $(E_{s_2} + R_{s_2})$ is numbered 2, and so on until each rule application in the control expression $(E_{s_n} + R_{s_n})$ is numbered n. Applications of (recursive) rules R_{s_1} in the first iteration of $(R_{s_1} \cdot \ \ldots \ \cdot R_{s_n})^*$ are numbered $n + 1$, and applications of rules in successive R_{s_i} are given successive integer numbers. Thus, applications of rules in R_{s_1} in the second iteration of $(R_{s_1} \cdot \ \ldots \ \cdot R_{s_n})^*$ are numbered $2 * n + 1$ and so on. The same numbering of rule applications can also be used for the Simple_CE evaluation of α.

Lemma 5.1 *Consider a Pred-SCC S, a database D, and a control expression α in the class $PCE(S)$. The resulting database after evaluating* PSN_CE(α, D) *is the same as* Simple_CE(α, D). *Further, this evaluation has the non-repetition property.*

Proof: A control expression α in the class $PCE(S)$ is either of the form: $(E_{s_1}) \cdot \ \ldots \ \cdot (E_{s_n})$ or of the form: $(E_{s_1} + R_{s_1}) \cdot \ \ldots \ \cdot (E_{s_n} + R_{s_n}) \cdot (R_{s_1} \cdot \ \ldots \ \cdot R_{s_n})^*$. Let $p_1, \ldots, p_m$ be all the predicates whose rules occur in α.

We first prove that the resulting database after evaluating PSN_CE(α, D) is the same as Simple_CE(α, D). We prove this using the following claim: a derivation of a fact in the evaluation of Simple_CE(α, D) is made for the first time in a rule application numbered i iff that derivation is made in a rule application numbered i of PSN_CE(α, D). We prove this claim by induction on the numbering of the rule applications (which is the same for PSN_CE(α, D) and Simple_CE(α, D)).

As a basis, the claim holds trivially prior to rule applications numbered 1, i.e. for base facts. Let us assume that the claim holds prior to rule applications numbered i. Now consider the rule applications numbered i, and the subsequent update operation (the operations performed prior to rule applications numbered $i + 1$) in the PSN_CE evaluation. In the PSN_CE evaluation, each rule application numbered i affects only some δp^{new} relation, and no δp^{new} relation occurs in the body of any semi-naive rewritten rule in S. Hence, the sequential evaluation of each of the (semi-naive rewritten) rule applications numbered i is equivalent to the independent application of these rules (without adding any facts to the database). Immediately following the rule applications numbered $i, i \geq 1$, only the differential relations of the predicates defined by the rules whose applications are numbered i will be updated using SN_Update.

We first show the "only if" part of the claim for the case when at least one of the predicates in one of the $s_i, 1 \leq i \leq n$ has a recursive rule defining it. Consider a fact $p(\overline{a})$ derived in the Simple_CE evaluation, and the associated derivation T that is made for the first time in rule application numbered i. Let the set of facts used in the derivation be denoted by SF, and let the rule used in the derivation be R. At least one of the facts in SF must have been derived in a rule application numbered $j, i - n \leq j < i$,

since otherwise there is an application of rule R numbered $i-n$ that would have performed derivation T in the Simple_CE evaluation (if $i > n$). Let one such fact be $p_{i1}(\overline{a})$. From rule applications numbered $j+1$ until rule applications numbered $j+n$, $p_{i1}(\overline{a})$ is present in the relation δp_{i1}^{old}, and all the other facts in SF are present either in relations of the form δp_{i2}^{old} or in relations of the form p_{i2}^{old}. From the form of the PSN_CE evaluation, the semi-naive versions of R are used in rule application i. By construction of the semi-naive rewritten rules, $p(\overline{a})$ will be derived (using derivation T) when the application of R numbered i is made. Showing the "only if" part of the claim is very similar when no predicate in any of the $s_i, 1 \leq i \leq n$ has a recursive rule defining it. This completes the "only if" part of the claim.

We next show the "if" part of the claim for the case when at least one of the predicates in one of the $s_i, 1 \leq i \leq n$ has a recursive rule defining it. Consider a fact $p(\overline{a})$ derived in the PSN_CE evaluation, and the associated derivation T which is made in rule application numbered i. Let the set of facts used in the derivation be denoted by SF, and let the rule used in the derivation be R. From the form of the PSN_CE evaluation, the semi-naive versions of R are used in rule application i. Hence, at least one of the facts, say $p_{i1}(\overline{a})$, in SF must be in a relation of the form δp_{i1}^{old}. (Other facts in SF are present either in relations of the form δp_{i2}^{old} or in relations of the form p_{i2}^{old}.) $p_{i1}(\overline{a})$ could not have been derived in a rule application numbered $j < i-n$, since then it would have been moved into the relation p_{i1}^{old} by an SN_Update prior to rule applications numbered i. Hence, $p_{i1}(\overline{a})$ was derived for the first time in a rule application numbered $j, i-n \leq j < i$ of PSN_CE. By the induction hypothesis, all facts in SF are available to rule applications numbered i in Simple_CE. Hence, derivation T is made by the Simple_CE evaluation in a rule application numbered i. This is the first time this derivation is made since, by the induction hypothesis, $p_{i1}(\overline{a})$ is not available to any application of rule R prior to this application of rule R in the Simple_CE evaluation. Showing the "if" part of the claim is very similar when no predicate in any of the $s_i, 1 \leq i \leq n$ has a recursive rule defining it. This finishes the induction step.

We next prove that the PSN_CE evaluation has the non-repetition property. First consider the case when at least one of the predicates in one of the $s_i, 1 \leq i \leq n$ has a recursive rule defining it. Consider a derivation step T, and let the set of facts used in the derivation be SF. Define the *application number* of a fact to be the least i such that the fact is derived by a rule application numbered i. Let the greatest application number among facts in SF be j_S. Thus, all facts in SF are derived in or before rule applications numbered j_S. Suppose derivation T is made in some rule application numbered i_T. We must have $i_T > j_S$, for before j_S not all facts needed for T were available.

A new fact generated by a rule application numbered j_S becomes part of some relation δp_i^{old} immediately after all rules applications numbered j_S are completed. After a complete iteration, i.e., after rule applications numbered $j_S + n$, facts in δp_i^{old} are moved to p_i^{old} due to SN_Update(p_i). Thus, after rule applications numbered $j_S + n$, all the facts in SF are in relations of the form p_k^{old}. SN_Update(p_k) ensures that once a fact is present in p_k^{old}, it will not appear in δp_k^{old} again. By the construction of semi-naive rewritten rules, derivation T cannot be made after this point. Hence we must have $i \leq j_S + n$. But between j_S and $j_S + n$, each rule R is applied at most once. Hence derivation T can be made at most once. Showing that the PSN_CE evaluation of a control expression in $PSN(S)$ has the non-repetition property when no predicate in any of the $s_i, 1 \leq i \leq n$ has a recursive rule defining it is very similar. This finishes the proof of the non-repetition property, and the proof of the lemma. □

Theorem 5.1 *Consider a program P, a database D, and a control expression α in the class $PCE(P)$. The resulting database after evaluating* PSN_CE(α, D) *is the same as* Simple_CE(α, D). *Further, evaluations of* PSN_CE *have the non-repetition property.*

Proof: Let $<$ be a topological ordering of Pred-SCCs in P such that $S_i < S_j$ if S_i contains a predicate used in a rule in S_j. Each control expression α in $PCE(P)$ is of the form: $PCE(S_1) \cdot \ \ldots \ \cdot PCE(S_k)$,

where the S_i are in topological order. The proof is by induction on this ordering $<$.

As a basis, consider S_0. It contains only base predicates and the theorem holds trivially. Let us suppose that it holds for S_i, i.e. the resulting database after evaluating PSN_CE(α_1, D) is the same as Simple_CE(α_1, D), where α_1 is in the class $PCE(S_1) \cdot \ \ldots \ \cdot PCE(S_i)$. Consider the PSN_CE evaluation of a control expression β in $PCE(S_{i+1})$ on the database given by Simple_CE(α_1, D).

Lemma 5.1 shows that the resulting database after evaluation of this control expression is the same as Simple_CE(β, Simple_CE(α_1, D)). Lemma 5.1 also shows that the PSN_CE evaluation of β has the non-repetition property. Since predicates in $S_0, \ldots, S_i$ are treated as base predicates in the semi-naive rewriting of β, the resulting database after the PSN_CE evaluation of $\alpha_2 = \alpha_1 \cdot \beta$ is the same as Simple_CE(α_2, D). This completes the proof of the equivalence of the PSN_CE evaluation to the Simple_CE evaluation. Since each rule in P is present in at most one Pred-SCC, and each Pred-SCC is evaluated only once, the PSN_CE evaluation of α has the non-repetition property. This completes the proof of the theorem. □

5.2 PSN for a Program

Consider a program P. With each Pred-SCC S in P, we can associate all the (non-recursive and recursive) rules defining the predicates in S. The PSN evaluation of S consists of repeated applications of the rules associated with S, and the program P can be evaluated Pred-SCC by Pred-SCC in a topological ordering of the Pred-SCCs of P. Procedure PSN_Prog below describes the PSN evaluation of a program P, using a control expression in the class $PCE(P)$ where in each Pred-SCC of P each predicate is in a partition by itself.

```
procedure PSN_Prog(P)
{
    Let S_1, ..., S_m be a topological ordering of the Pred-SCCs of P.
        /* S_0 is assumed to contain all the base predicates. */
    for j = 1 to m do    PSN_SCC(S_j)
}
end PSN_Prog

procedure PSN_SCC(S)
{
    Let the ordering of predicates in S be p_1, ..., p_k.
    Let E^1_{p_i}, ..., E^{a_i}_{p_i} be the non-recursive rules defining p_i, and
        let R^1_{p_i}, ..., R^{b_i}_{p_i} be the recursive rules defining p_i, 1 ≤ i ≤ k.
    Let E_{p_i} stand for the control expression E^1_{p_i} + ... + E^{a_i}_{p_i}, and
        let R_{p_i} stand for the control expression R^1_{p_i} + ... + R^{b_i}_{p_i}.
    if no predicate in S has any recursive rule defining it, then
        Evaluate the following control expression:
        (E_{p_1}) · ... · (E_{p_k}) using PSN_CE
    else   /* at least one of the predicates in S has a recursive rule defining it */
        Evaluate the following control expression:
        (E_{p_1} + R_{p_1}) · ... · (E_{p_k} + R_{p_k}) · (R_{p_1} · ... · R_{p_k})* using PSN_CE.
}
end PSN_SCC
```

Theorem 5.2 *Procedure* PSN_Prog *has the non-repetition property, is sound and, if all the predicates defined in the program are safe, is complete wrt the least fixpoint semantics.*

Proof: Let $<$ be a topological ordering of Pred-SCCs such that $S_i < S_j$, if S_i contains a predicate used in a rule in S_j. The proof is by induction on this ordering $<$.

As the basis, consider S_0. It contains only base predicates and the theorem holds trivially. Suppose it holds for S_i. Consider evaluation of Pred-SCC S_{i+1}. We prove the soundness of PSN evaluation of S_{i+1} using induction on the number of rule applications in the PSN evaluation. If the given set of facts is sound, then the set of facts produced by a rule application is also sound. The hypothesis holds trivially for the first rule application, and hence the proof of soundness follows.

Theorem 5.1 ensures that the PSN evaluation of S_{i+1} has the non-repetition property. We now prove that PSN evaluation of S_{i+1} is complete wrt the least fixpoint semantics.

Since all lower Pred-SCCs have been fully evaluated, let us treat the set of facts for these (lower) Pred-SCCs to be base facts, of derivation height 0. We prove completeness of the Simple_CE evaluation of the control expression (wrt the least fixpoint semantics) using induction on the derivation heights of facts. Theorem 5.1, which shows the equivalence of the Simple_CE evaluation to the PSN_CE evaluation, can then be used to show that the PSN_CE evaluation is complete wrt the least fixpoint semantics. As the basis case, facts with derivation heights of 1 are derived by non-recursive rules, and these are clearly derived in the course of Simple_CE evaluation. Assume inductively that facts with derivation heights up to h are derived using Simple_CE evaluation. Consider a fact $p(\overline{a})$ with derivation height $h+1$, and the associated derivation T. Let the set of facts used in the derivation be denoted by SF, and let the rule used in the derivation be R. All the facts in SF have derivation indices less than or equal to h. By the induction hypothesis these are derived during the Simple_CE evaluation.

Consider now the least i such that all the facts in SF are derived in rule applications with numbers less than or equal to i. Clearly at least one of the facts in SF must be derived in a rule application numbered i. From the form of the iterative loops in the Simple_CE evaluation of the control expression, R is used in some rule application $j1$ such that $i+1 \leq j1 \leq i+k$, and derivation T will be made when the application of R numbered $j1$ is made. From Theorem 5.1, derivation T for $p(\overline{a})$ will be made by the PSN_CE evaluation of the control expression. This finishes the proof of completeness, as well as the proof of the theorem. □

PSN can be generalized as follows:

1. Apply each semi-naive version of each recursive rule in a Pred-SCC exactly once in each iteration.
2. Perform the updates of the differential relations according to SN_Update for each predicate p exactly once at the same point (not necessarily immediately after applying the recursive rules for predicate p) in each iteration, and
3. Iterate till no new facts are computed in a complete iteration.

A result similar to Theorem 5.2 also holds for this generalization of PSN. The proof of the result follows from simple modifications of the proof of the above theorem.

Much as the ordering of rules could affect the performance of GSN, ordering of predicates could affect the performance of PSN. The results of Section 6 can be used to obtain good predicate orderings for the PSN evaluation of a program.

In a sequential evaluation of a program, 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. The performance results of Leask et al. [19] (described in Section 7.1) support this claim. PSN may not be able to utilize facts as early as GSN can, but the overheads associated with GSN could be higher, and the choice of which strategy to choose is not always obvious.

The following example illustrates the differences between BSN, PSN, and GSN evaluation of a program.

Example 5.1 Consider the following program P:

$F_1 : b_1(4,5).$
$F_2 : b_2(3,4).$
$F_3 : b_3(2,3).$
$F_4 : b_4(1,2).$
$R_1 : p(X,Y) :- b_1(X,Y).$
$R_2 : q(X,Y) :- b_2(X,Z), p(Z,Y).$
$R_3 : p(X,Y) :- b_3(X,Z), q(Z,Y).$
$R_4 : p(X,Y) :- b_4(X,Z), p(Z,Y).$
Query: ?-$p(1,X)$.

For the above program, BSN as well as GSN and PSN would evaluate the non-recursive rule, R_1 exactly once (in the beginning) to produce $p(4,5)$.

Evaluating the rules (R_2, R_3, R_4) repeatedly in this order, using GSN, would make the facts produced using an application of rule R_2 ($q(3,5)$ in the first iteration) immediately available to applications of R_3 and R_4 in the same iteration. Similarly, the facts produced using an application of R_3 ($p(2,5)$ in the first iteration) would be available to applications of R_4 (in the same iteration), and R_2 (in the subsequent iteration). Thus, the answer, $p(1,5)$, to the query would be produced at the end of the first iteration itself using GSN evaluation of P.

If the predicate ordering chosen for PSN is (q,p), the rule defining q (R_2) is applied first, and the fact ($q(3,5)$ in the first iteration) produced would be immediately available to applications of R_3 and R_4, in the same iteration. The application of R_2 is followed by the independent application of the rules defining p (R_3 and R_4). Consequently, the fact produced using R_3 ($p(2,5)$ in the first iteration) would be available to R_4 only in the next iteration. Thus, the answer, $p(1,5)$, to the query would be produced at the end of the second iteration using PSN evaluation of P.

If BSN evaluation is used instead, facts produced by a rule application would not be available to any other rule application in the same iteration. Consequently, $q(3,5)$ would be produced in the first iteration, $p(2,5)$ in the second iteration, and $p(1,5)$ (the answer to the query) would be produced only in the third iteration. □

6 Rule Orderings that Minimize Rule Applications

Example 6.1 Let q and r be base relations. Consider the following program P:

$$
\begin{array}{lll}
R_0: & p_k(X) & :- q(X). \\
R_1: & p_1(f_1(X)) & :- p_k(X). \\
R_2: & p_2(f_2(X)) & :- p_1(X). \\
 & \vdots & \\
R_{k-1}: & p_{k-1}(f_{k-1}(X)) & :- p_{k-2}(X). \\
R_k: & p_k(f_k(X)) & :- p_{k-1}(X), r(X).
\end{array}
$$

The non-recursive rule R_0 is the first rule to be applied in a BSN evaluation of P. In an iteration of a Basic Semi-Naive evaluation, all the recursive rules (i.e. $R_1, \ldots, R_k$) of the program are applied independently. Rule R_k will be successfully applied for the first time only in the k^{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, i.e. $(R_1, R_2, \ldots, R_k)$, and the facts produced by each rule application are immediately made available to subsequent rule applications. If the computation of a fact using this technique (and the given ordering of rules) took n iterations, then computation of the same fact using the BSN evaluation strategy could take up to $k * n$ iterations.

If, instead, the rules are applied in the opposite order, i.e. $(R_k, R_{k-1}, \ldots, R_1)$, the number of iterations taken is the same as BSN evaluation, even if facts produced by a rule application are made available immediately after the rule is applied. 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 fixpoint algorithms, that have the non-repetition property and apply a rule using GSN, 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 6.3). We also present results which suggest that only this class of rule orderings should be considered in the absence of additional semantic information (Section 6.4).

The techniques described in this section deal with rule orderings, but can be extended, in a straight forward fashion, to deal with predicate orderings, and can obtain good orderings for PSN evaluation.

6.1 Benefits of Rule Ordering

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 non-repetition 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 non-repetition 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 [20], 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 accesses. Thus, the greater the number of inferences made in a single rule application, the fewer are 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 produced by a rule application in the application of other rules within the same iteration, while preserving the non-repetition 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. 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 such as efficient join and indexing strategies, and duplicate elimination techniques—none of these is made inapplicable by ordering rules.

6.2 Class of Orderings Considered

Definition 6.1 Fair, static orderings : Let the rules of a Rule-SCC S be $R_1, \ldots, R_n$. A *fair, static ordering* is an ordering of the form $(R_{i_1}, \ldots, R_{i_n})$, where $i_1, \ldots, i_n$ is a permutation of $1, \ldots, n$. □

We consider such orderings in Section 6.3. Such an ordering is referred to as a *static* ordering since the same ordering is used in each iteration. In such static orderings 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 order to compute the closure of an SCC S, using a fair ordering, we apply the rules in the given order in each iteration, repeatedly, making the facts produced by a rule application available to all subsequent rule applications, till no more facts can be computed. Independent applications of rules are not considered, since making the facts produced by a rule application available to other rule applications may significantly reduce the number of rule applications needed to compute the fixpoint, and can never increase it in a sequential evaluation.

In Section 6.4 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. [18]. Non-fair orderings may perform somewhat better than fair orderings on some data sets, but, as we show in Section 6.4, 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.

We do not consider orderings where the next rule to be applied is chosen dynamically. Although

dynamic orderings may be better than static orderings in terms of the cost criteria described in Section 6.1, determining which rule to apply may be difficult and involve considerable overheads. While dynamic orderings may be worth investigating, we do not consider them in this paper.

6.3 Fair Orderings

Definition 6.2 Order sequence : Let O be any ordering $(R_{i_1}, \ldots, R_{i_n})$. An *order sequence* $S = O^m$ denotes the string formed by repeating O, m times.

We use the notation $S[k], 1 \leq k \leq m * n$ to denote the k^{th} rule in S. □

A sequence S_1 of length n is said to be a ***subsequence*** of a sequence S_2 of length $m \geq n$, if there exist n numbers $1 \leq k_1 < k_2 < \ldots < k_n \leq m$, such that $S_1[i] = S_2[k_i], 1 \leq i \leq n$. For example, (R_1, R_3, R_5) is a subsequence of $(R_1, R_2, R_3, R_4, R_5)$.

Definition 6.3 Cycle preserving fair orderings : Consider an SCC S, and let the rules in S be $\mathcal{R} = \{R_1, \ldots, R_n\}$. Let $G = (\mathcal{R}, E)$ 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 $R_{j_1}, \ldots, R_{j_m}$ in G.[3]

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 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 subsequence 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 6.3.1, where we show in Theorem 6.2 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 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:

- Cyclic permutations of a fair ordering are equivalent under $\lhd$, and
- Any two cycle preserving fair orderings are equivalent under $\lhd$.

Example 6.2 Consider the rule 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 6.1 *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.*

[3] Though cycles have the same initial and final vertex, we omit the final vertex in our representation, for convenience.

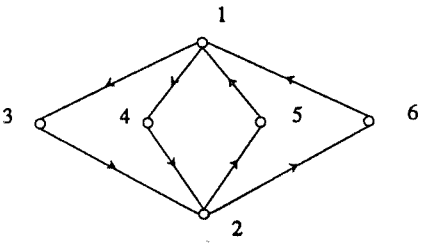

Figure 1: An Example Rule Graph

Proof: By definition of $B(C,O)$, for some cyclic permutation O_r of O, C is a subsequence of $O_r^{B(C,O)}$. Consider now R_m, the last rule in C. It must be mapped to the occurrence of R_m in the last repetition of O_r in $O_r^{B(C,O)}$. Now move the elements of O_r that follow R_m to the head of the $O_r^{B(C,O)}$. Clearly C is a subsequence of the result of this transposition, and the result is equal to $O_1^{B(C,O)}$. That C is not a subsequence of O_1^i for any $i < B(C,O)$ follows from the definition of $B(C,O)$. □

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 in a fair ordering, the optimality result carries over for the cost criterion of number of rule applications.

6.3.1 Optimality of Cycle Preserving Orderings

Definition 6.4 Derivation path : 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.

Definition 6.5 Derivation index : Let O denote a fair ordering of rules in the program and T denote a particular derivation tree for $p(\bar{c})$. 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. □

Definition 6.6 Iteration height : Let O denote a fair ordering of rules in the program and T denote a particular derivation tree for $p(\bar{c})$. With each node in the derivation tree we associate an *iteration height* $\lceil k/m \rceil$ where k is the derivation index of that node, and m is the length of O.

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 $p(\bar{c})$ 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.

Definition 6.7 Iteration count : 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.

Definition 6.8 Iteration length : Consider a derivation tree T for a fact, and a fair ordering O. 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, define $L(T, O) = 0$. Otherwise define $L(T, O) = \max\{L(s, O) \mid s \text{ is a path in } T\}$.

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 6.2 *Given a derivation tree T for a fact $p(\bar{c})$, 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.*

Proof: We prove by induction on the height of a tree that the iteration height of a node is the same as the maximum of the iteration lengths of paths from the leaves of the tree to that node. As a basis, note that the iteration height of a leaf node is zero.

For an internal node n, let $m(n)$ denote the maximum of the iteration lengths of derivation paths from the leaves to n and let $s(n)$ denote one such path with iteration length $m(n)$.

Consider any internal node n, and assume inductively that for every child n_i of n, the iteration height of n_i is the same as $m(n_i)$. Clearly the iteration length, $m(n)$ must be greater than or equal to every $m(n_i)$.

Let the rule corresponding to node n be R. Consider any child node n_j of n, that has iteration height $m(n_j) = m(n)$, if such a node exists. Assume that the rule, say R_j corresponding to node n_j, occurs after R in the ordering. But then we can use the path $s(n_j)$ with R appended to it, and this path would have an iteration length of $m(n) + 1$, which is not possible by maximality of the iteration length $m(n)$. Thus the iteration height of each child node of n is either less than $m(n)$, or it is equal to $m(n)$ and the rule that labels the child node occurs before R in the ordering. Hence, by the definition of iteration heights, the iteration height of n is $m(n)$. This completes the induction. □

Every derivation path has a certain structure which is described in the next lemma, and which we use to prove the result in Theorem 6.1.

Lemma 6.3 *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 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}.*

Proof: We prove by induction that such a path exists. As a basis, any path s of length one is acyclic, and thus has such a sequence $s_0 = s$.

Consider a path s of length k and assume inductively that for all paths of length less than k such a

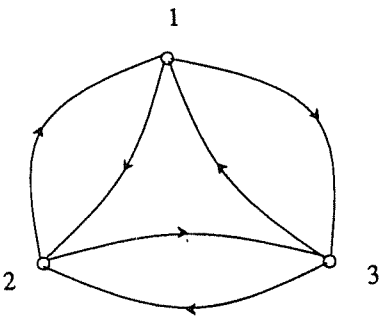

Figure 2: Path with Two Alternative Cycle Sequence Lengths

sequence exists. If in the path s there is no rule that occurs twice, s is acyclic, and we set $s_0 = s$, and we are done. If there is a rule that occurs twice in s, for each rule R_i consider every pair of occurrences of R_i. Choose a pair of occurrences (of any rule) that is separated by the least distance, in the path s, and let the rule chosen be R_l. Let the path between the two occurrences be $R_j, \ldots, R_p$. This path cannot contain two occurrences of any rule. Further, for each pair of consecutive rules R_{i_1}, R_{i_2} in the derivation path, there must be an edge (R_{i_1}, R_{i_2}) in the rule graph. Thus, $R_j, \ldots, R_p, R_l$ must be a simple cycle in the rule graph. Deleting this cycle from s we get a path of length less than k, and by the induction hypothesis there is a sequence $s_0, s_1, \ldots, s_i$ for this subpath. If we let $C_{i+1} = R_j, \ldots, R_p, R_l$, we get a sequence for s. □

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 as the following example shows.

Example 6.3 Consider the rule graph shown in Figure 1. The derivation path $1, 4, 2, 5, 1, 3, 2, 6$ has the following two construction sequences associated with it,

1. $(1, 3, 2, 6), (1, 4, 2, 5, 1, 3, 2, 6)$. This involves the insertion of the simple cycle $4, 2, 5, 1$ in the acyclic path $1, 3, 2, 6$ just after 1.
2. $(1, 4, 2, 6), (1, 4, 2, 5, 1, 3, 2, 6)$. This involves the insertion of the simple cycle $5, 1, 3, 2$ in the acyclic path $1, 4, 2, 6$ just after 2. □

Also, two such construction sequences may have different lengths as the following example shows.

Example 6.4 Consider the graph shown in Figure 2. The derivation path $1, 2, 3, 1, 3, 2$ has the following two construction sequences associated with it,

1. $(1, 3, 2), (1, 2, 3, 1, 3, 2)$.
2. $(1, 2), (1, 2, 3, 2), (1, 2, 3, 1, 3, 2)$. □

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 6.1 *Consider any derivation path s and a construction sequence $s_0, \ldots, s_n$ for s as defined in Lemma 6.3. Let C_i be the cycle inserted in obtaining s_i from s_{i-1}. For every fair ordering O, the*

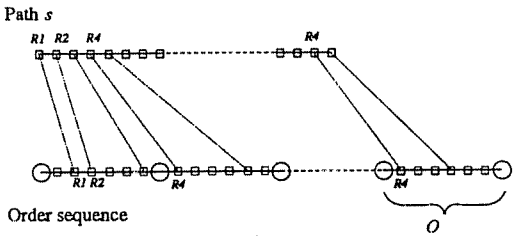

Figure 3: Relating Derivation Paths to Order Sequences

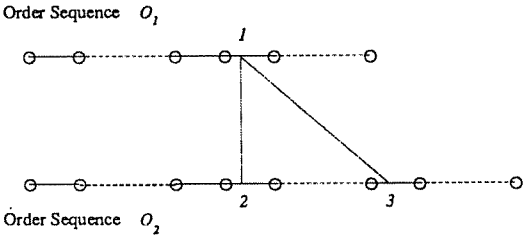

Figure 4: Order Sequences After Insertion of a Cycle in a Path

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 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 3 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 4 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 4) 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 4) 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 6.1 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 sequence chosen, this tells us that, in a certain sense, all construction sequences are equivalent. For any cycle preserving fair ordering O with a construction sequence $t_0, \ldots, t_n$, $L(t_n, O) = L(t_0, O) + n$.

Consider a rule graph G, and two fair orderings O_1 and O_2 such that $O_1 \lhd O_2$. We define

$$MaxR(O_1, O_2, G) = max\{B(C, O_2)/B(C, O_1) \mid C \text{ is a simple cycle in } G\}.$$

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. This leads finally to our main result, stated in Theorem 6.2, that relates the number of iterations taken to compute the closure of the SCC by the two fair orderings.

Lemma 6.4 *Consider any two fair orderings O_1 and O_2 for a graph G, such that $O_1 \lhd O_2$. For any derivation path s, the iteration lengths of s under the two orderings are related as $L(s, O_1) - k \leq L(s, O_2) \leq MaxR(O_1, O_2, G) \cdot L(s, O_1) + k$, where k is bounded by the length of the longest acyclic path in G.*

Proof: Since $O_1 \lhd O_2$, for each cycle C, we have $B(C, O_1) \leq B(C, O_2)$. By definition of $MaxR(O_1, O_2, G)$, we have $B(C, O_2) \leq MaxR(O_1, O_2, G) \cdot B(C, O_1)$.

Consider any construction sequence $s_0, \ldots, s_n$ for s. By Theorem 6.1, the iteration lengths of s under O_1 and O_2 are given by $L(s, O_1) = L(s_0, O_1) + \sum_{i=1}^{n} B(C_i, O_1)$ and $L(s, O_2) = L(s_0, O_2) + \sum_{i=1}^{n} B(C_i, O_2)$.

For any fair ordering O and derivation path s, $L(s, O)$ is clearly bounded by the length of path s. Thus the difference between $L(s_0, O_1)$ and $L(s_0, O_2)$ is bounded by the length of the longest acyclic path in G, since s_0 is acyclic. The lemma follows from the above. □

Lemma 6.5 *Consider any two fair orderings O_1 and O_2 for a graph G, such that $O_1 \lhd O_2$. For any derivation tree T, the iteration heights of T under the two orderings are related as $L(T, O_1) - k \leq L(T, O_2) \leq MaxR(O_1, O_2, G) \cdot L(T, O_1) + k$, where k is bounded by the length of the longest acyclic path in G.*

Proof: Lemma 6.2 shows that the iteration height of T is given by the maximum of the iteration lengths of the paths in T. Consider the path in T that has the maximum iteration length under O_1, and call this path s. We have $L(T, O_1) = L(s, O_1)$. By Lemma 6.4, the iteration length of this path under O_2 is bounded from below as $L(s, O_1) - k \leq L(s, O_2)$. Clearly the maximum of the iteration lengths of paths in T under ordering O_2 must be greater than or equal to $L(s, O_2)$, and thus we have $L(T, O_1) - k \leq L(T, O_2)$.

For the other side of the inequality, consider the path s in T that has maximum iteration length under O_2. By Lemma 6.4, $L(s, O_2) \leq MaxR(O_1, O_2, G) \cdot L(s, O_1) + k$. Since $L(s, O_1) \leq L(T, O_1)$, and $L(s, O_2) = L(T, O_2)$, we have $L(T, O_2) \leq MaxR(O_1, O_2, G) \cdot L(T, O_1) + k$. □

Lemma 6.6 *Given any fact $p(\overline{c})$, and any two fair orderings, O_1 and O_2 such that $O_1 \lhd O_2$, let the iteration counts of $p(\overline{c})$, for 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: A fact $p(\overline{c})$ can have several derivation trees. Consider the derivation tree T for $p(\overline{c})$ that has the least iteration height under O_2. We have $L(T, O_2) = n_2$. By Lemma 6.5, $L(T, O_1) - k \leq L(T, O_2)$. The iteration count of $p(\overline{c})$ under ordering O_1 must be less than or equal to $L(T, O_1)$, and hence we have $n_1 - k \leq n_2$.

Consider now the derivation tree T' for $p(\overline{c})$, that has the least iteration height under O_1. By Lemma 6.5, we must have $L(T', O_2) \leq MaxR(O_1, O_2, G) \cdot L(T', O_1) + k$. The iteration count n_2 of $p(\overline{c})$ under O_2 must be less than or equal to $L(T', O_2)$, and $n_1 = L(T', O_1)$, and thus we have $n_2 \leq MaxR(O_1, O_2, G) \cdot n_1 + k$. □

Theorem 6.2 *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: 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. By Lemma 6.6 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. By Lemma 6.6, 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 6.1 *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.* □

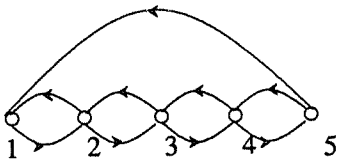

Figure 5: Graph on which ReversePopout May Not Work

6.3.2 Generating Cycle Preserving Fair Orderings

In the previous section, we have shown the optimality of cycle preserving fair orderings under the cost criterion of number of iterations needed to compute the closure of an SCC. However, there are SCCs that do not have any cycle preserving fair orderings. For instance, the SCC whose rule graph is a complete directed graph with three vertices is one such SCC. If an SCC has a simple cycle that has some subsequence x, y, z and another simple cycle in the same SCC has the subsequence z, y, x, then no fair ordering can preserve both these cycles. Thus this SCC has no cycle preserving fair ordering.

An SCC with n rules could have an exponential number of simple cycles. A naive algorithm would examine all $O(n!)$ orderings and all simple cycles to check if the ordering preserves every simple cycle. However, this would be very inefficient, and 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. A more general open problem is to find a fair ordering such that the maximum degree to which it breaks any cycle is minimized, in case there is no cycle preserving fair ordering. Finding what complexity classes these problems belong to is also an interesting open problem.

As a heuristic, we suggest the reverse of a depth-first search pop-out order of the rule graph. We call this ordering the *ReversePopout* ordering. 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.

Example 6.5 Consider the graph in Figure 1. One possible ReversePopout ordering is $(1, 4, 3, 2, 6, 5)$, and this is a cycle preserving fair ordering. On this example, every ReversePopout ordering is cycle preserving. □

Example 6.6 On the other hand, consider the graph in Figure 5. On this graph, if the node 1 is chosen as the starting node for the depth-first search, the ReversePopout ordering produced is the cycle preserving fair ordering $(1, 2, 3, 4, 5)$. However, if node 5 is chosen as the starting node, the fair ordering $(5, 4, 3, 2, 1)$ could be produced, which does not preserve the cycle $1, 2, 3, 4, 5$. □

6.4 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 non-flat nested orderings.

Definition 6.9 Nested ordering: 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.

Definition 6.10 Flat ordering: A *flat ordering* is a nested ordering that has parentheses only at the outermost level. □

The *nesting level* of nested orderings is defined in the obvious manner, where a flat ordering is defined to have a nesting level of one.

Proposition 6.1 *Consider an SCC S, and any flat ordering O of the rules in S. Let k_1 be the number of rule occurrences in O. Let n_{opt} be the minimum number of rule applications needed to compute the closure of S on a given database, using a sequential evaluation with immediate updates. If n_O denotes the number of rule applications needed to compute the closure of S on the same database, using the ordering O, then $n_O \leq n_{opt} \cdot k_1$.*

Proof: Consider the sequence of rule applications used to compute the closure of S using n_{opt} rule applications. Replacing every occurrence of each rule in this sequence by the entire flat ordering O, we get a sequence of n_{opt} occurrences of O that computes the closure of S. Since there are $n_{opt} \cdot k_1$ rule applications in this sequence, we are guaranteed that the closure of S using the flat ordering O will take at most $n_{opt} \cdot k_1$ rule applications. □

We now compare flat orderings with nested orderings.

Lemma 6.7 *Consider an SCC S, and let O be any nested ordering with k_1 rule occurrences and a nesting level of s. Let $iter_{bsn}$ be the number of iterations needed to compute the closure of S using BSN. If n_O denotes the number of rule applications needed to compute the closure of S using the ordering O, then $n_O \leq (iter_{bsn})^s \cdot k_1$.*

Proof: Consider the outermost level of nesting in O. On each iteration of this level, each rule in the SCC is applied at least once. Thus we can have at most $iter_{bsn}$ iterations of this level, before a fixpoint is reached. Consider any nested subexpression at the next level of nesting. This expression is encountered once for each iteration of the outermost level of nesting. The fixpoint of the entire program using BSN takes at most $iter_{bsn}$ iterations, hence there is no derivation path of length greater than $iter_{bsn}$. Thus, each time this subexpression is encountered, it can iterate at most $iter_{bsn}$ times. Proceeding inductively, we note that the innermost level of nesting iterates at most $(iter_{bsn})^s$ times. Thus no rule occurrence in O_n is applied more than $(iter_{bsn})^s$ times. Since there are k_1 rule occurrences in O_n, the result of the lemma is established. □

The following theorem summarizes our comparison of flat and nested orderings.

Theorem 6.3 *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.*

Proof: The number of rule applications using any flat ordering is bounded as $iter_{bsn} \leq n_f$, where $iter_{bsn}$ is the number of iterations needed to compute the closure of S using BSN. Clearly $n_{opt} \leq n_n$, and the theorem follows from Proposition 6.1 and Lemma 6.7. □

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 7 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. $n_f = c * n_n$, $1 < c \leq k$) 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 $iter_{bsn} \geq n_f/k$.

7 Performance Results

In this section we describe results of a performance study of the benefits of immediate availability of facts, and the benefits of ordering rules as described in Section 6. 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. We also consider several possible join strategies, and for each of these we compare the relative join costs (ignoring fixed overheads associated with each join) of the different evaluation techniques. Our performance study draws upon and extends the work of Kuittinen et al. [18]. We also discuss work by Leask et al. [19], which uses total elapsed time on a disk-based deductive database system as the metric, to show the benefits of using PSN over BSN.

We consider two programs, referred to as $P1$ and $P2$ in this section.[4] These programs and corresponding data sets are presented in Appendix A.

The semi-naive rewriting used is the version proposed in [2], and described in Section 2.2. 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 GSN 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 nested ordering described in Appendix A. Column "Basic" corresponds to a Basic Semi-Naive evaluation, and column "Pred" is for a Predicate 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 4 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

[4] $P1$ is the same program that was used in [18].

Data Set	Basic	Pred	General 1	General 2
$A10$	3579	1535	1023	2812
$B64$	146	73	68	80
$F10$	23	10	7	18

Table 4: Program $P1$: Number of Iterations

Data Set	Basic		Pred		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 5: Program $P1$: Number of Joins (Non-Null and Null)

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 5 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%.

The cost of a single join can be modeled in different ways, depending on the join strategy used. In Tables 6 and 10, we present the portion of the join cost that excludes the per-join overhead. To get the full join cost, we must add the overhead per join times the number of (non-null) joins. In these tables, the column "Hashed" is a measure of the join cost (excluding overheads) using the hash join technique. The column "Constant" corresponds to a nested loop join with indices that can be accessed in constant time (for instance, a hash index), the column "Logarithmic" corresponds to nested loops join with a logarithmic access time index (for instance, a B-tree index), and the column "Linear" corresponds to a nested loop join without indices (which is equivalent to an index with access time linear in the size of the relation). All these costs are normalized (with Basic = 1 for each technique and data set) order of

Data Set	Eval. Tech.	Non-Nl Joins	Hashed	Constant	Logarithmic	Linear
$A10$	Basic	6126	1	1	1	1
$A10$	Pred	6126	1.000	1.000	1.000	1.000
$A10$	General 1	5230	0.856	1.000	1.000	1.000
$B64$	Basic	1066	1	1	1	1
$B64$	Pred	549	0.525	1.000	0.999	1.000
$B64$	General 1	588	0.542	1.000	0.997	1.000
$F10$	Basic	73	1	1	1	1
$F10$	Pred	57	0.900	1.000	1.004	1.001
$F10$	General 1	45	0.822	1.003	1.013	1.025

Table 6: Program $P1$: Normalized Joins Costs (Without Overheads) Under Various Assumptions

Data Set	Eval. Tech.	Constant	Logarithmic	Linear
$A10$	Basic	1	1	1
$A10$	Pred	1.000	1.000	1.000
$A10$	General 1	1.000	1.000	1.000
$B64$	Basic	1	1	1
$B64$	Pred	1.000	1.023	1.007
$B64$	General 1	1.000	0.998	1.004
$F10$	Basic	1	1	1
$F10$	Pred	1.000	1.012	1.017
$F10$	General 1	1.009	1.024	1.049

Table 7: Program $P1$: Duplicate Elimination Costs

magnitude estimates derived directly from the sizes of the relations. The size of the result of the join is included in the join cost. Note the following important points:

- I/O costs and overhead costs for joins are not measured in these tables. The number of non-null joins may be used to estimate the total overhead and this should be added to the cost in the table. Since the overheads are implementation dependent and we currently lack an actual system to measure the overheads, we do not present the total cost. Some results comparing PSN with BSN using total elapsed time on a disk based system are presented in [19]. We discuss these results later in this section.
- The only comparisons that may be made from this table are the differences between the different evaluation strategies. These numbers should not be used to directly compare different join techniques (i.e. the numbers in a single row of the table should not be compared).

The performance study of Kuittinen et al. [18] only models the join cost in one way, the hashed join method.

The performance results in Table 6 indicate that for the hash join strategy the total join cost closely parallels the number of non-null joins. PSN and GSN do consistently better than BSN under this metric. The performance results also indicate that under other join cost assumptions (nested loops without indices, and with constant or logarithmic access time indices), the total join costs apart from the I/O costs and other overheads associated with each join, are practically the same for each of the fixpoint evaluation strategies we study (BSN, PSN, and GSN). In these cases the number of non-null joins becomes the significant factor.

The cost of checking for and eliminating duplicates could form a significant part of the total cost of evaluation. In our analysis of duplicate elimination costs, we measure these costs as a function of the sizes of the relations involved and as before do not measure I/O costs. As before, these values are normalized order of magnitude estimates and should not be used to compare different duplicate elimination techniques. In Tables 7 and 11, the column "Constant" refers to the cost of duplicate elimination assuming the availability of a constant access time index, the column "Logarithmic" assumes the availability of a logarithmic access time index and the column "Linear" assumes that no index is available.

The performance results in Table 7 indicate that the duplicate elimination costs for each of the evaluation strategies (BSN, PSN and GSN) is approximately the same, for each of the indexing strategies

Data Set	Basic	Pred	General 1	Nested
$C16$	282	221	207	179
$S64$	1717	588	583	2536

Table 8: Program $P2$: Number of Rule Applications

Data Set	Basic		Pred		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 9: Program $P2$: Number of Joins (Non-Null and Null)

used. Thus, the relative costs of the evaluation techniques should not depend on the duplicate elimination costs.

Table 8 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 9 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.

The performance results in Table 10 are similar to those in Table 6 for the BSN, PSN and GSN evaluation strategies. It should be noted that using the hashed join strategy, the nested evaluation ordering performs slightly better on one data set (namely, $C16$), but performs considerably worse on $S64$.

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.

Data Set	Eval. Tech.	Non-Nl Joins	Hashed	Constant	Logarithmic	Linear
$C16$	Basic	459	1	1	1	1
$C16$	Pred	372	0.899	1.119	1.110	1.063
$C16$	General 1	364	0.900	1.118	1.117	1.045
$C16$	Nested	304	0.582	0.867	0.852	0.857
$S64$	Basic	2002	1	1	1	1
$S64$	Pred	1386	0.631	0.977	1.002	0.977
$S64$	General 1	1447	0.764	0.977	1.001	0.977
$S64$	Nested	5230	4.827	0.973	0.940	0.977

Table 10: Program $P2$: Joins Costs Under Various Assumptions

Data Set	Eval. Tech.	Constant	Logarithmic	Linear
$C16$	Basic	1	1	1
$C16$	Pred	1.110	1.175	1.337
$C16$	General 1	1.095	1.163	1.329
$C16$	Nested	0.882	0.971	1.251
$S64$	Basic	1	1	1
$S64$	Pred	0.974	1.129	1.081
$S64$	General 1	0.974	1.116	1.075
$S64$	Nested	0.974	0.894	1.002

Table 11: Program $P2$: Duplicate Elimination Costs

7.1 Related Work

Leask et al. [19] investigate the parallelism that is available in answering queries bottom-up when the logic program is compiled into a relational language. While their main results are on parallelism, their results also shed light on the relative performance of BSN and PSN.

PSN can be implemented with the same overheads per iteration as BSN, but can do better in terms of the number of iterations taken. Our performance results also indicate that PSN is always preferable to BSN on this metric. However, our performance results do not compare the *total* cost of evaluation using the various strategies described; in particular, the I/O costs and the fixed overhead incurred in performing a join, a rule application, or an iteration is not included in the relative cost estimates. Consequently, our results do not quantify the relative advantage of PSN over BSN.

Leask et al. [19] use two programs—one which admits of a good fair ordering, and one which does not—to compare the various evaluation strategies. The join strategy used was sort-merge. The performance metric they used was total elapsed time; this includes the cost of accessing relations, as well as the various overheads incurred in a bottom-up evaluation, and is a realistic cost estimate for comparing the performance of the various strategies. On each of the two programs, PSN considerably outperforms BSN. For the program which admits of a good fair ordering, PSN reduces the total time taken by about a half, while on the other program, the improvement is about 15%.

The results of [19] also indicate that PSN is the evaluation strategy of choice, in the absence of semantic information about the data in the base relations, even when compared with the parallel evaluation strategies described in [19].

8 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. We presented two evaluation algorithms, GSN and PSN, that could be used for evaluating such rule orderings, while preserving the non-repetition property, and discussed cases where each was useful.

We studied rule orderings theoretically, and showed that for the class of fair orderings, cycle preserving orderings were optimal and, in the absence of additional information, fair orderings are to be preferred

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

Figure 6: Program $P1$

$1 : anc(X, Y, 1) :- manc(X), up(X, Y).$
$2 : anc(X, Y, N) :- N > 1, manc(X), anc(X, Z, N-1), up(Z, Y).$
$3 : desc(X, Y, 1) :- mdesc(X, 1), down(X, Y).$
$4 : desc(X, Y, N) :- N > 1, mdesc(X, N), desc(X, Z, N-1), down(Z, Y).$
$5 : sg(X, Y) :- msg(X), flat(X, Y).$
$6 : sg(X, Y) :- msg(X), anc(X, X1, N), flat(X1, X2), sg(X2, Y2), flat(Y2, Y1), desc(Y1, Y, N).$
$7 : manc(X) :- msg(X).$
$8 : msg(X2) :- msg(X), anc(X, X1, N), flat(X1, X2).$
$9 : mdesc(Y1, N) :- msg(X), anc(X, X1, N), flat(X1, X2), sg(X2, Y2), flat(Y2, Y1).$
$10 : mdesc(X, N-1) :- mdesc(X, N), N > 1.$
$11 : msg(1).$
$12 : query(X) :- sg(1, X).$

Figure 7: Program $P2$

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.

We also presented a performance study to support our theoretical analyses. Rule orderings were also shown to be useful for improving the total cost of sequential evaluation of logic programs.

Acknowledgements

We would like to thank the referees for their comments, which helped improve the presentation of the paper. We would also like to thank them for pointing out some related work.

A Programs

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

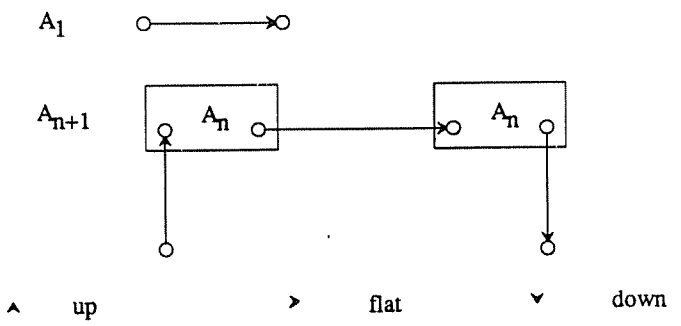

Figure 8: Data Set $A10$

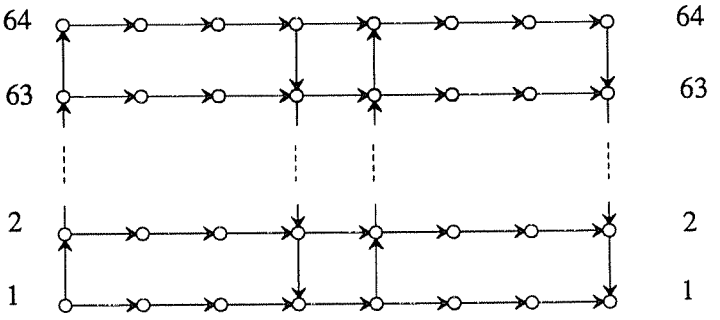

Figure 9: Data Set $B64$

Rule-SCCs is cycle preserving. Combining the fair orderings for each Rule-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 Rule-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 [18]. They are depicted in Figures 8 and 9 respectively.

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 $flat(i, j)$.

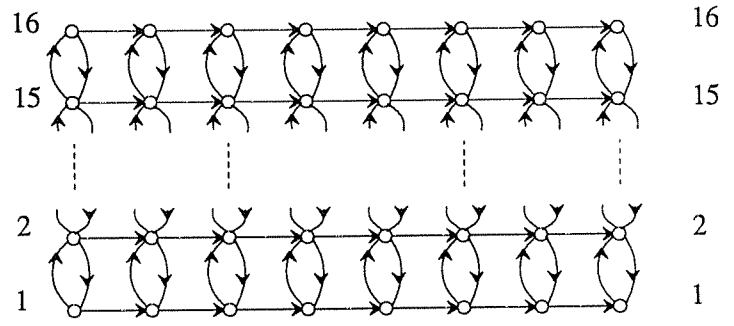

Figure 10: Data Set $C16$

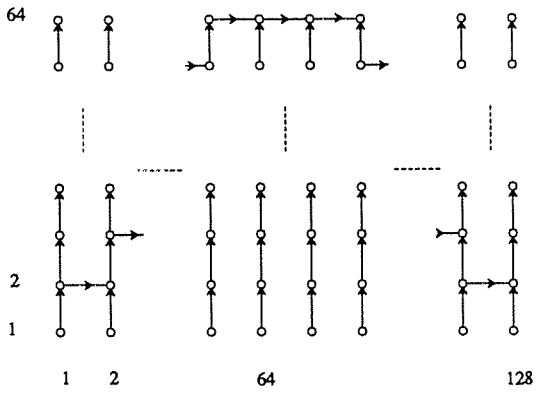

Figure 11: Data Set $S64$

Data set Cn is a grid with n rows and 8 columns. We depict $C16$ in Figure 10. 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 11.

References

[1] K. Apt. Efficient computation of least fixpoints. Technical Report 88-33, University of Texas at Austin, Austin, TX 78712, Sept. 1988.

[2] I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *Journal of Logic Programming*, 4(3), September 1987.

[3] F. Bancilhon. Naive evaluation of recursively defined relations. In Brodie and Mylopoulos, editors, *On Knowledge Base Management Systems — Integrating Database and AI Systems*. Springer-Verlag, 1985.

[4] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 1–15, Cambridge, Massachusetts, March 1986.

[5] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 16–52, Washington, D.C., May 1986.

[6] R. Bayer. Query evaluation and recursion in deductive database systems. Unpublished Memorandum, 1985.

[7] C. Beeri, S. Naqvi, R. Ramakrishnan, O. Shmueli, and S. Tsur. Sets and negation in a logic database language. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 21–37, San Diego, California, March 1987.

[8] C. Beeri and R. Ramakrishnan. On the power of magic. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 269–283, San Diego, California, March 1987.

[9] C. Beeri, R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Magic implementation of stratified programs. Manuscript, September 89.

[10] A. C. Fong and J. Ullman. Induction variables in very high-level languages. In *Proc. Third ACM Symposium on Principles of Programming Languages*, pages 104–112, 1976.

[11] R. Gonzalez-Rubio, J. Rohmer, and A. Bradier. An overview of DDC: Delta Driven Computer. In *Parallel Architectures and Languages Europe, Volume 1: Parallel Architectures*, Lecture Notes in Computer Science, No. 258, pages 414–433, 1987.

[12] A. R. Helm. Detecting and eliminating redundant derivations in deductive database systems. Technical Report RC 14244 (#63767), IBM Thomas Watson Research Center, December 1988.

[13] R. Helm. Inductive and deductive control of logic programs. In *Proceedings of the International Conference on Logic Programming*, pages 488–511, Melbourne, Australia, 1987.

[14] T. Imielinski and S. Naqvi. Explicit control of logic programs through rule algebra. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 103–116, 1988.

[15] Y. E. Ioannidis. On the computation of the transitive closure of relational operators. In *Proceedings of the Twelfth International Conference on Very Large Databases*, pages 403–411, 1986.

[16] R. Kabler, Y. Ioannidis, and M. Carey. Performance evaluation of algorithms for transitive closure. Unpublished manuscript, Univ. of Wisconsin, Madison, December 1989.

[17] P. Kolaitis and C. Papadimitriou. Why not negation by fixpoint? In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 231–239, 1988.

[18] J. Kuittinen, O. Nurmi, S. Sippu, and E. Soisalon-Soininen. Efficient implementation of loops in bottom-up evaluation of logic queries. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, Aug. 1990.

[19] T. S. Leask, K. Ramamohanarao, and P. J. Stuckey. Exploiting parallelism in bottom-up computation in Aditi. In *Proceedings of the Post ILPS'91 Workshop on Deductive Databases*, 1991.

[20] H. Lu. New strategies for computing the transitive closure of a database relation. In *Proceedings of the Thirteenth International Conference on Very Large Databases*, pages 267–274, Sept. 1987.

[21] M. J. Maher and R. Ramakrishnan. Dèjá vu in fixpoints of logic programs. In *Proceedings of the Symposium on Logic Programming*, Cleveland, Ohio, 1990.

[22] R. Paige and J. T. Schwatz. Reduction in strength of high level operations. In *Proc. Fourth ACM Symposium on Principles of Programming Languages*, pages 58–71, 1977.

[23] R. Ramakrishnan. Magic Templates: A spellbinding approach to logic programs. In *Proceedings of the International Conference on Logic Programming*, pages 140–159, Seattle, Washington, August 1988.

[24] J. Rohmer, R. Lescoeur, and J. M. Kerisit. The Alexander method — a technique for the processing of recursive axioms in deductive database queries. *New Generation Computing*, 4:522–528, 1986.

[25] H. Schmidt. *Meta-Level Control for Deductive Database Systems.* Lecture Notes in Computer Science, Number 479. Springer-Verlag, 1991.

[26] H. Schmidt, W. Kiessling, U. Güntzer, and R. Bayer. Compiling exploratory and goal-directed deduction into sloppy delta iteration. In *IEEE International Symposium on Logic Programming*, pages 234–243, 1987.

[27] E. Shapiro. Logic programs with uncertainties: A tool for implementing rule-based systems. In *Proceedings of IJCAI*, pages 529–532, 1983.

[28] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volumes I and II.* Computer Science Press, 1989.

[29] P. Valduriez and H. Boral. Evaluation of recursive queries using join indices. In *Proceedings of the First International Expert Database Systems Conference*, pages 197–208, 1986.

[30] P. Valduriez and S. Khoshafian. Transitive closure of transitively closed relations. In *Proceedings of the Second International Conference on Expert Database Systems*, pages 377–400, 1988.

[31] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, October 1976.