SPACE OPTIMIZATION IN DEDUCTIVE DATABASES

by

Divesh Srivastava
S. Sudarshan
Raghu Ramakrishnan
Jeffrey F. Naughton

Computer Sciences Technical Report #1063

December 1991

Space Optimization in Deductive Databases*

Divesh Srivastava S. Sudarshan Raghu Ramakrishnan†
Jeffrey F. Naughton‡
{divesh,sudarsha,raghu,naughton}@cs.wisc.edu
Computer Sciences Department,
University of Wisconsin-Madison, WI 53706, U.S.A.

Abstract

In the bottom-up evaluation of a logic program, all generated facts are usually assumed to be stored until the end of the evaluation. Considerable gains can be achieved by instead discarding facts that are no longer required: the space needed to evaluate the program is reduced, I/O costs may be reduced, and the costs of maintaining and accessing indices, eliminating duplicates etc. are reduced. Thus, discarding facts early could achieve time as well as space improvements.

Given an evaluation method that is sound, complete and does not repeat derivation steps, we consider how facts can be discarded during the evaluation without compromising these properties. Our first contribution is to show that such a space optimization technique has three distinct components. Informally, we must make all derivations that we can with each fact, detect all duplicate derivations of facts and try to order the computation so as to minimize the "life-span" of each fact.

This separation enables us to use different methods for each of the components for different parts of the program. We present several methods for ensuring each of these components. We also briefly describe how to obtain a complete space optimization technique by making a choice of techniques for each component and combining them. Our results apply to a significantly larger class of programs than those considered in [NR90].

1 Introduction

Bottom-up evaluation of a logic program proceeds by repeatedly applying rules to generate facts until no new facts can be produced. Bottom-up evaluation has been shown to have several advantages over top-down evaluation in the area of deductive databases (see, for example, [Ull89]). However, a disadvantage of bottom-up evaluation is that all generated facts are usually assumed to be stored until the end of the evaluation. Since the number of facts generated can be extremely large in the case of many programs, reducing the space requirements of a program by discarding facts during the evaluation may be very important. The following example (from [NR90]) illustrates this point.

Example 1.1 The program computes the length of the longest common subsequence (LCS) of two strings a and b, and is representative of more general sequence analysis programs. The algorithm is from

*A preliminary version of this paper appeared in Proc. ACM SIGMOD '91.

†The work of the first three authors 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 this author supported by NSF grant IRI-8909795. A shorter version of this paper appeared in the proceedings of the 1991 ACM SIGMOD International Conference on the Management of Data.

Hirschberg [Hir75]; we use the representation that if letter j of string a (resp. b) is α, then the database contains the fact $a(j, \alpha)$ (resp. $b(j, \alpha)$).

$R1 : lcs(m, N, 0).$
$R2 : lcs(M, n, 0).$
$R3 : lcs(M, N, X) \leftarrow M < m, N < n, a(M, C), b(N, C), lcs(M + 1, N + 1, X - 1).$
$R4 : lcs(M, N, X) \leftarrow M < m, N < n, a(M, C), b(N, D), C \neq D, lcs(M + 1, N, X1), lcs(M, N + 1, X2), X = \max(X1, X2).$
Query: ?-$lcs(0, 0, X)$.

If the strings are of length m and n, then evaluating the program using the top-down Prolog evaluation strategy gives a running time that is $\Omega(\binom{m+n}{n})$. Using the Magic Sets rewriting strategy followed by bottom-up evaluation produces a running time that is $O(mn)$, but is also space $\Omega(mn)$. Suppose m and n are 10^6, a value that we are likely to see in applications such as DNA sequencing. Then the number of facts generated is around 10^{12}, which is clearly impractical to store.

Sliding Window Tabulation, as described in [NR90], evaluates this program in $O(m + n)$ space and $O(mn)$ time, by discarding facts in the course of the evaluation. Storing 10^6 facts is certainly feasible, as opposed to 10^{12}. Thus this improvement in the space complexity is essential if the program is to be run on large data sets. □

In addition to improving the space requirements, discarding facts that are no longer needed can have other advantages. I/O costs may be reduced, even eliminated, if the program can be evaluated in main memory; the costs of maintaining and accessing indices, eliminating duplicates etc. are also reduced. Thus, discarding facts early could achieve time as well as space improvements. We refer to techniques that discard facts during the course of the evaluation of a logic program as *space optimization techniques*[1].

While Sliding Window Tabulation is effective on the LCS example, the applicability of Sliding Window Tabulation as presented in [NR90] is fairly limited. For instance, suppose we extend the LCS program so that instead of being base predicates, a and b are defined by additional rules in the program—this will be the case if the program preprocesses "rough" base data before searching for common subsequences. Sliding Window Tabulation cannot be used on this extension of the LCS program. Similarly, if the above program is embedded in a larger program that uses the length of the longest common subsequence to perform further analysis, such as find the region of a given DNA sequence that best matches the given test sequence, Sliding Window Tabulation is again inapplicable.

One of the main contributions of this paper is to show that space optimization techniques have three components:

1. Ensuring that each fact is used in every possible derivation even though facts may be discarded before the end of the evaluation.
2. Ensuring that multiple derivations of a fact are detected, in order to avoid repeated inferences.
3. Synchronizing the evaluation to ensure that derivations of facts are "close" to all their uses, and discarding facts soon after their uses.

[1] In this paper, we do not consider other space saving techniques, such as allowing facts to share parts of their structure with other facts.

We describe these components in more detail in Section 3. This decomposition provides a framework in which to reason about space optimization techniques. It also gives us the flexibility of choosing different techniques for each component, and synthesizing new space optimization techniques.

In this paper, we also describe several new techniques for each of these three components, and briefly discuss how to combine these techniques to generate space optimization techniques. We also discuss how to automatically combine these techniques to get a space optimization technique for the full program. Sliding Window Tabulation turns out to be just one particular way of combining techniques for each of the components. Thus our techniques significantly extend the class of programs optimized in [NR90]. In particular, we can deal with some programs (rewritten using Magic Sets) in which a predicate p and the corresponding magic predicate *magic_p* are mutually recursive whereas the techniques of [NR90] do not handle such programs.

2 Definitions

In this paper, we consider Horn clause logic programs[2], and assume the usual definitions including those of terms, atoms and rules (clauses). We assume standard notation for dealing with logic programs (see [Ull88]). We assume familiarity with semi-naive evaluation, and in some of the sections of the paper we also assume some familiarity with the Magic Sets transformation. We refer the reader to [Ull89, NR89] for an introduction.

A *program* is treated as a set of rules and EDB facts. While analyzing the program, we do not need to know the specific EDB facts, but we often make use of information such as functional dependencies on EDB relations. A *program fact* is used to mean any fact that is used or derived by a program.

Assumption 2.1 *In this paper we assume that all program facts are ground.* □

A sufficient condition that guarantees this is *range-restrictedness* of the program, i.e., for every rule in the program, every variable that appears in the head of the rule also appears in the body of the rule.

Definition 2.1 Evaluation : Consider a program P.

- A *state* in a program execution is a tuple $< \mathcal{F}, \mathcal{H} >$, where $\mathcal{F}$ is a set of facts, and $\mathcal{H}$ denotes a hidden component of the state. (Initially $\mathcal{F}$ is the set of given facts in the program, and $\mathcal{H}$ is empty.)
- A *state transition* changes $< \mathcal{F}_1, \mathcal{H}_1 >$ to $< \mathcal{F}_1 \cup \mathcal{F}, \mathcal{H}_2 >$, where $\mathcal{F}$ is a set of facts such that each fact can be derived from $< \mathcal{F}_1, \mathcal{H}_1 >$ using a rule in P. (Note that there could be several state transitions out of a given state.)
- A *final state* is a state such that no new facts can be generated.

An *evaluation* of P according to strategy $\mathcal{M}$ is a progression from the initial state to a final state, through a sequence of state transitions according to $\mathcal{M}$; each of the states in this progression is referred to as a *point in the evaluation* of P.

The hidden component contains information about an execution that is irrelevant for our purposes. □

[2]This can be extended to include programs with stratified negation and aggregation.

A *derivation step* at state $< \mathcal{F}_i, \mathcal{H}_i >$ consists of a rule R along with a substitution σ on its variables, such that each positive body literal is present in the set $\mathcal{F}_i$[3]. Every fact p' in $\mathcal{F}_{i+1} - \mathcal{F}_i$ has one or more derivation steps such that p' is the head of $R[\sigma]$.

Definition 2.2 Semi-Naive Evaluation : An evaluation is said to be a *semi-naive evaluation* iff at each state $< \mathcal{F}_i, \mathcal{H}_i >$ every derivation step D made at state i is such that D was not used earlier.

Such an evaluation is said to have the *semi-naive property.* □

Definition 2.3 Basic Semi-Naive Evaluation : We use the term Basic Semi-Naive evaluation to refer to the semi-naive evaluation technique presented in [Ban85] (possibly with improvements suggested by [BR87]). This evaluation technique proceeds in iterations; in each iteration it makes all new derivations that can be made by using facts derived up to the previous iteration. □

Definition 2.4 Locally Semi-Naive Evaluation : An evaluation is said to be a *locally semi-naive evaluation* iff at each state $< \mathcal{F}_i, \mathcal{H}_i >$: (1) no derivation step is repeated at the ith point in the evaluation, and (2) every derivation step D made at the ith point in the evaluation is such that either: (a) D was not used earlier, or (b) There is a $j \leq i$ such that D uses at least one fact that is present in $\mathcal{F}_j - \mathcal{F}_{j-1}$, and there is no other use of D at any point k in the evaluation, $j \leq k \leq i$. □

Suppose we modify a semi-naive evaluation by discarding facts during the evaluation. If a fact p is discarded and then derived again at a later point in the evaluation, we may repeat some derivation that uses the fact, and hence the modified evaluation would not be a semi-naive evaluation. However, it would be a locally semi-naive evaluation. A locally semi-naive evaluation has the property of not repeating inferences if no facts are discarded. By definition every semi-naive evaluation is also a locally semi-naive evaluation. Several of the evaluations that we consider in this paper are locally semi-naive, but not semi-naive.

In the following definition we weaken the notion of "base" predicates. In some cases, predicates that are defined by rules (hence "derived", under the usual sense) can be treated as base predicates for a number of optimization and evaluation techniques. The following definition identifies a sufficient condition for treating predicates as base in any of the techniques described in this paper. Since this definition is strictly weaker than traditional definitions, the use of the traditional definition does not affect the correctness of any of the results in this paper.

Definition 2.5 Base Set : Given an evaluation and a rule R, a set of predicates $q_1, \ldots, q_n$ is said to be a *base set with respect to* R if the following condition holds every time R is applied in the evaluation: the set of facts for $q_i, 1 \leq i \leq n$ that are available to the application of R has the property that even if every q_i fact that could possibly be derived is actually made available, the result of the rule application would be unchanged. □

Note that a rule may have more than one base set; however, the union of base sets is not necessarily a base set. Hence, among all the base sets, one such set is chosen for each rule, and is referred to as the base set for the rule; each predicate in it is a *base predicate with respect to the rule*. The other predicates in the rule are *derived with respect to the rule*.

A predicate p_2 is said to be *derived with respect to another predicate* p_1 if, either (1) there is a rule R such that p_1 is the head predicate of R and p_2 is derived with respect to R, or (2) there is a predicate p_3 such that p_3 is derived with respect to p_1 by (1) and p_2 is (recursively) derived with respect to p_3. Note that the relation $p1$ derived-with-respect-to $p2$ is not necessarily symmetric.

[3] When negation is allowed, we also require that each negative literal cannot be derived.

3 Ensuring Soundness, Completeness and Non-Redundancy

In general, discarding a fact could result in the non-derivation of other facts that should have been derived and thereby, in the presence of negation, derivation of facts that should not have been derived. This could compromise completeness, and in the presence of negation, also soundness. The following condition ensures that facts are used in all possible derivations and is used to ensure soundness and completeness:

Condition U : A fact $p(\overline{a})$ satisfies Condition U with respect to an evaluation, at a point $e1$ in the evaluation, iff

1. Every derivation using it has been made at or before $e1$, or
2. If $p(\overline{a})$ is discarded at $e1$, it will be recomputed at some later point $e2$ in the evaluation, and any derivation that could have been made using $p(\overline{a})$ after $e1$ (had $p(\overline{a})$ not been discarded) will be made after the fact has been recomputed at $e2$. Also, if the program has negation, any derivation that would have been prevented by the presence of $p(\overline{a})$ is not made between $e1$ and $e2$.

□

The restriction of Condition U to a predicate occurrence in the body of a rule is defined in a straightforward manner, by considering only uses of a fact in a particular predicate occurrence in the body of that rule.

The following proposition is straightforward.

Proposition 3.1 *Condition U is satisfied by a fact $p(\overline{a})$ at a point $e1$ in an evaluation iff, for each body occurrence of p in every rule, the restriction of Condition U to that predicate occurrence is satisfied by $p(\overline{a})$.* □

In this paper, we assume that an evaluation is a locally semi-naive evaluation. Such an evaluation as the desirable property of not repeating inferences if no facts are discarded. However, if a fact is discarded and subsequently rederived, we may not detect this duplicate derivation and thus may repeat some derivations that use this fact. This could compromise the semi-naive property. Further, not detecting duplicate derivations of a fact could compromise termination if cyclic derivations are possible. If every fact that is discarded satisfies the following condition before it is discarded, then multiple derivations of facts will be detected:

Condition D : A fact $p(\overline{a})$ satisfies Condition D with respect to an evaluation, at a point $e1$ in the evaluation, iff

1. It is not derived again at or after the point $e1$, or
2. If $p(\overline{a})$ is discarded at $e1$, then for any later point $e2$ in the evaluation where $p(\overline{a})$ is derived, no inference using $p(\overline{a})$ made at or before $e1$ is repeated after $e2$.

□

The restriction of Condition D to a rule is defined in a straightforward manner, by considering only derivations of a fact by that rule. A proposition similar to 3.1 holds for Condition D with "body predicate occurrence of p" replaced by "rule defining p."

For an evaluation of a program to be complete, all non-redundant derivations that can be made using the program must in fact be made by the evaluation. In the case when the number of derivations of a fact do matter, as when the multiset semantics of Maher and Ramakrishnan [MR90] is used, no derivation is redundant, although in other cases some derivations may be redundant. We say that an evaluation is *derivation-complete* if all derivations that can be made using the program are in fact made by the evaluation.

The following propositions summarize how ensuring Conditions U and D when discarding facts in an evaluation preserves soundness, derivation-completeness and the semi-naive property of the evaluation.

Proposition 3.2 *Consider an evaluation E of a program such that E is sound, and derivation-complete, and no facts are discarded during the evaluation. Suppose we modify E by discarding facts one at a time during the evaluation. Then the modified evaluation is sound and derivation-complete iff Condition U is satisfied by each fact whenever it is discarded.* □

Proposition 3.3 *Consider an evaluation E of a program such that E is sound, derivation-complete, has the semi-naive property and no facts are discarded during the evaluation. Suppose we modify E by discarding facts one at a time during the evaluation. Then the modified evaluation has the semi-naive property iff Condition D is satisfied by each fact whenever it is discarded.* □

From the above two propositions, the following result follows.

Theorem 3.1 *Consider an evaluation E of a program such that E is sound, derivation-complete, has the semi-naive property and no facts are discarded during the evaluation. Suppose we modify E by discarding facts one at a time during the evaluation. Then the modified evaluation is sound, derivation-complete and has the semi-naive property iff Conditions U and D are satisfied by each fact whenever it is discarded.* □

In the rest of this paper, we assume that facts are discarded one at a time, and Conditions U and D reflect this assumption. The discarding of one fact at a point in the evaluation could affect whether or not another fact is rederived at a later point in the evaluation. In general, after discarding a fact $p1$, Conditions U and D may need to be retested for other facts, say $p2$. In the special case of a fact $p2$ such that all derivations using the fact have been made, discarding $p1$ at a point $e1$ does not affect the satisfaction of Condition U by $p2$ at $e1$. Also, if $p1$ satisfies Condition D when it is discarded, the satisfaction of D by $p2$ is not affected by the discarding of $p1$. Conditions U and D can be easily generalized to handle discarding a set of facts at a point in an evaluation and all the results in this paper correspondingly generalize.

Theorem 3.2 *Given a program P and an EDB D, and an arbitrary point $e1$ in an evaluation of program P on EDB D, it is undecidable whether a given fact satisfies Conditions U and/or D at $e1$.*

Proof: Consider an arbitrary logic program L that defines p. Add the fact p_1 and the rule $p_2 \leftarrow p, p_1$ to L to get L1. (Neither p_1 nor p_2 should occur in L.) The fact p_1 can be used to compute p_2 iff $?p$ is satisfiable in L. Since satisfiability is not decidable for logic programs it is undecidable if p_1 will be used again after any point e_1 in the evaluation. Since there is no other derivation of p_1, Condition U is undecidable.

To show undecidability of Condition D, add the fact p and the rule $R : p_1 \leftarrow p$ to the logic program L to get L2. Consider a point e_1 in a semi-naive evaluation of L2 after p_1 has been derived using the given fact p and rule R. Since it is undecidable whether $?p$ is satisfiable in L, if p is discarded at e_1 it is undecidable whether the derivation of p_1 (using R) is repeated after e_1. □

Consequently, it is undecidable whether discarding a fact during an evaluation will compromise the soundness, completeness or semi-naive property of the evaluation. Hence, we must look for sufficient conditions for ensuring D and U for program facts. Even the stronger conditions that only test the first parts of Conditions U and D are undecidable. Our sufficient conditions are often based on the first parts of Conditions D and U.

3.1 An Overview of Our Approach

We now present a brief overview of our techniques. In subsequent sections we look in detail at some of the techniques that we outline here. Consider facts of the form $p(\overline{a})$ in a program P. Consider an evaluation of P and let ψ be a schedule for discarding p-facts in this evaluation. We justify ψ by establishing that Conditions D and U hold for each p fact before it is discarded. At compile time we analyze the program, and decide on the applicability of each technique. We then add extra tests and auxiliary computations (that we describe along with each technique) to the compiled version of the program. In general these tests are performed at run time to decide when a fact satisfies Conditions D and U. These operations are quite efficient—see Section 9 for more details. Facts are discarded at run-time as soon as the tests determine that they satisfy both Conditions D and U.

Ensuring Condition D : Condition D can be checked on a per-rule basis, and different techniques can be used for different rules in a given program. Applicable techniques include the following:

1. Providing a bound on the total number of derivations of a fact.
   If a program is duplicate free ([MR90]), we know that once a fact is derived it will not be derived again. We look at this technique (and some extensions) for ensuring Condition D in Section 4.1.
2. Using monotonicity constraints.
   Monotonicity constraints ensure some monotone ordering on the derivation of facts. We look at this idea in Section 5.1.

Ensuring Condition U : Condition U can be checked on a per-body-literal basis, and different techniques can be used for different literals in a given program. Applicable techniques include the following:

1. Providing a bound on the total number of uses of a fact.
   Suppose a rule in linear, i.e. there is only one predicate in the body of the rule which is derived wrt the rule. Once a fact for the derived predicate is used (with all the facts for the base predicates), we know that no new derivations can be made using that fact in that rule. We look at this and more general ways of ensuring Condition U in Section 4.2.
2. Using monotonicity constraints.
   In Section 5.2 we consider using monotonicity constraints to satisfy Condition U.

If none of these approaches for ensuring D or U succeeds, we always have the option of not discarding any p-facts. We can still optimize the rest of the program, unlike the technique described in [NR90].

Synchronization : If (all) derivations of facts are "close" to all their uses, facts can be discarded soon after being derived. In Section 6 we consider techniques that can be used to order an evaluation so as to maximize this property, and we call them *synchronization* techniques. These include:

- Delaying first use of facts.
  The idea is to partition the set of derived facts into a set of "active" facts used in derivations and a set of "hidden" facts whose use is delayed (until they become "active"). The goal is to balance the derivation of new facts against the identification of facts that can be discarded so that the number of facts that are stored at any one point in the evaluation is minimized. (See Section 6.1.)
- Nested-SCC synchronization.
  This technique can be understood as identifying "subgoals" that are to be evaluated by a "subprogram" on each call. The idea is to generate facts for the subprogram as and when they are needed by the main program. We describe this in detail in Section 6.2.
- Interleaved-SCC synchronization.
  The acyclic graph of SCCs suggests a natural producer-consumer relationship. By interleaving the evaluation of producers and consumers, it is sometimes possible to ensure that facts are generated in a producer as and when they are needed by the consumers. This technique is described in detail in Section 6.3.

Combining Techniques : The various techniques for synchronization and for ensuring Conditions D and U are applicable to parts of a program (such as rules, predicate occurrences, etc). These need to be combined to get a space optimization technique for the full program. These issues are discussed in detail in Section 8.

4 Bounds on Derivations and Uses of Facts

4.1 Duplicate Freedom and Condition D

The simplest way to ensure that Condition D is satisfied for a fact $p(\overline{a})$ at a point in a locally semi-naive evaluation of a program is based on the following condition on the predicate p:

Condition DF1 : (1) No fact for p is derived by more than one rule, (2) there is at most one derivation for each p fact by any rule, and (3) no derivation for any p fact is repeated.[4] □

The techniques of [MR90] can be used to test the first two parts of this condition—part (1) can be tested by determining that no two rule heads unify and part (2) by checking that the head of the rule functionally determines the body. If facts are discarded in a locally semi-naive evaluation, part (3) is ensured if for all predicates q such that q is derived with respect to p, we make sure Condition D is satisfied by each q fact before it is discarded. Although the third part appears to be cyclic, it isn't really so; all we have to do is decide to discard facts only if they satisfy Condition D, and this part is satisfied for all the predicates. The problem of how to test the first two parts is all that is left.

Proposition 4.1 *If a predicate p in a locally semi-naive evaluation satisfies Condition DF1, Condition D is satisfied by each p fact after it is derived.* □

The essential idea is that no fact for p is derived more than once in this evaluation (i.e., p is duplicate free). We can weaken Condition DF1 in several ways. If part (1) does not hold, we can still ensure

[4] This would be true if the evaluation is semi-naive. However if derivation of p facts is done in a locally semi-naive but not semi-naive, fashion, this may not be true (see Section 7.2). The choice of locally semi-naive/semi-naive evaluation is done before checking this condition.

Condition D using a run-time check to determine that a fact has been derived once by every rule that could possibly derive it.

DF1 can also be weakened by modifying the requirement that "there is at most one derivation for each fact by any rule" to the requirement that "if there is more than one derivation for any fact by a rule, then the facts for the derived predicate occurrences in the corresponding rule instances are the same." Thus, multiple derivations would be allowed within a rule application (which is the join of the "current" extents of the body literals). To test this weaker requirement, we can check whether the head of a rule functionally determines the body occurrences of predicates that are derived (with respect to this rule); the head need not functionally determine the base predicate occurrences. Provided part (3) in DF1 is satisfied, once a rule has been applied to derive a fact, we know that no further derivations of that fact will be made using that rule. To summarize:

Condition DF2 : Parts (1) and (3) as in DF1, and (2) if there is more than one derivation for any fact by a rule, then the facts for the derived predicate occurrences in the corresponding rule instances are the same. □

A proposition similar to Proposition 4.1 also holds in the case of DF2. Again, if part (1) does not hold, we can ensure Condition D using run-time checks.

4.2 Bounds on Uses and Condition U

If we can determine a bound on the number of uses of p facts in a body predicate occurrence p' of p, once a p fact has been used in that many derivations in p', we know that it can no longer be used in this occurrence. The following condition seeks to capture this intuition. We use functional dependencies over a relation composed of all instances of a rule in order to state the condition.

Condition Bounds_U : Consider an evaluation of a program P, and a rule

$$R : p2() \leftarrow p(\bar{t}), b(), p1().$$

where $b()$ denotes the join of all the base predicate occurrences (other than $p(\bar{t})$, if it is base) in the body of the rule, and $p1()$ denotes the join of all derived predicate occurrences (other than $p(\bar{t})$, if it is derived) in the body of R. Suppose that no derivation steps for any $p2$ facts are repeated.[5] Then the predicate occurrence $p(\bar{t})$ in R satisfies Condition Bounds_U if it satisfies either of:

BU1 : $p(\bar{t})$ functionally determines $p1()$ in R, or

BU2 : $p(\bar{t})$ functionally determines the head $p2()$ of R.

□

Note that in the case when $p(\bar{t})$ is the only predicate occurrence that is derived with respect to the rule (i.e. R is a "linear" rule), Bounds_U is trivially satisfied. Once a fact for the derived predicate is used (with all the facts for the base predicates), we know that no new derivations can be made using that fact in that rule.

[5] See Condition DF1.

Proposition 4.2 *Consider an evaluation of a program, and let a body predicate occurrence $p(\bar{t})$ in R satisfy Condition BU1. A fact $p(\bar{a})$ will no longer be used in the occurrence $p(\bar{t})$ if: (1) $p(\bar{a})$ does not match any facts for $b()$, or (2) $p(\bar{a})$ matches base facts, and it has been used in the occurrence $p(\bar{t})$ in a successful rule application.* □

Suppose some derivations using R and $p(\bar{a})$ are repeated. If we discard the fact $p(\bar{a})$ after one successful derivation, we would prevent repetitions of that derivation and hence not satisfy Condition U. There are indeed cases where such repeated derivations are essential for completeness of the evaluation—see, for instance, the Nested-SCC Discarding technique in Section 7.2.

Condition BU1 can be generalized by requiring that $p(\bar{t})$ along with $b()$ functionally determine $p1()$ in R. Let $\#(p(\bar{a}), p(\bar{t}), R)$ denote the cardinality of the set of facts $b(\overline{a_i})$ that can join with $p(\bar{a})$ used in the occurrence $p(\bar{t})$ in rule R. Then the fact $p(\bar{a})$ can no longer be used in the occurrence $p(\bar{t})$ of R, if it has been successfully used in $\#(p(\bar{a}), p(\bar{t}), R)$ derivations in the predicate occurrence $p(\bar{t})$ in R. Note that if any derivations using R are repeated, the count of uses of $p(\bar{a})$ in this occurrence may be wrong.

If the occurrence $p(\bar{t})$ in R satisfies Condition BU2, and a fact $p(\bar{a})$ matches at least one fact for $b()$ and has been used in this occurrence in a successful derivation, then no new facts can be generated by any subsequent derivations using the same fact for this p occurrence. Condition U is not satisfied, since it is possible that there are more derivations of the same head fact using the given p fact in this rule. However, if we are not interested in the number of derivations, we may effectively consider Condition U to be satisfied, without compromising soundness or completeness (although we do sacrifice derivation-completeness). Condition BU2 can also be generalized in a fashion similar to BU1.

Note that the use of functional dependencies is conservative; if $p \rightarrow q$, for example, we know that there is at most one q fact that can join with a given p fact, but there may in fact be no such q fact as the example below shows. The condition can therefore be refined by using a notion of dependencies that requires the existence of exactly one such q fact for each p fact, but we do not pursue this here.

Example 4.1 The program P_{Ack} is:

$R1 : ack(0, Q, 2 * Q).$
$R2 : ack(P, 0, 0) \leftarrow P > 0.$
$R3 : ack(P, 1, 2) \leftarrow P > 0.$
$R4 : ack(P, Q, N) \leftarrow P > 0, Q > 1, ack(P, Q - 1, N1), ack(P - 1, N1, N).$

Here, the FDs $ack : \{1, 2\} \rightarrow 3$ and $\{1, 3\} \rightarrow 2$ hold. As a consequence, each body occurrence of *ack* functionally determines the other occurrence of *ack* in rule (4). Also, each occurrence of *ack* in the body of rule (4) functionally determines the head of that rule. Both Conditions BU1 and BU2 are therefore applicable; since *ack* can also be shown to be duplicate-free, we can discard each *ack* fact after two uses. However, there are several *ack* facts that can be used only once, and based on these conditions they are never discarded. As a result, no asymptotic savings in space is achieved. □

The following example illustrates the use of bounds on derivations and uses of facts in space optimization.

Example 4.2 Consider the program below.

$anc(X, Y) \leftarrow father(X, Y).$
$anc(X, Y) \leftarrow father(X, Z), anc(Z, Y).$

Suppose we know that the *father* relation is a tree of nodes. That is, we are given the following functional dependency: $father : 1 \rightarrow 2$, i.e. each node has only one father, and we also know that the *father* relation is acyclic. Suppose also that the user gives a query $? - anc(X, Y)$. We assume that if a fact is an answer to the query, it is printed out straight away to the users terminal.

We can deduce the following about the program:

- The program is duplicate free: The techniques of [MR90] may be used to deduce this. Informally, this is because the functional dependency shows that each fact can be deduced at most once by each rule, and the acyclicity of *father* along with the functional dependency shows that each fact can be deduced by at most one rule.
- Since the rule is linear, each derived fact for *anc* is used in at most one rule application. We can then discard *anc* facts once they have been used in one rule application.

We now look at the benefits due to discarding facts in this program. Let n be the number of facts in the relation *father*. The functional dependency on *anc* shows that for each node x, there is exactly one ancestor at each distance i. This means that at each point in the evaluation, at most n facts are computed. Thus at most $2 * n$ *anc* facts are stored at any point in the evaluation. If facts were not discarded, up to $O(n^2)$ facts may need to be stored, depending on the structure of the *father* relation. Note that monotonicity based techniques (such as Sliding Window Tabulation and extensions discussed in Section 5) are not applicable to this program, since there is no monotonicity inherent in the rules.

Even in the absence of the functional dependency and acyclicity requirements, facts for the *anc* program can be deduced to satisfy Conditions D and U. If there is a node a such that no fact $anc(\_, a)$ (for any value "_") is derived in a particular state of a locally semi-naive evaluation of this program, we know that no fact $anc(\_, a)$ will be either be derived again or used again in the evaluation. Therefore all such facts satisfy Condition D and U, and can be discarded. We expect the savings to be significant in practice on graphs that are not strongly connected. This idea can be extended to more general classes of programs, to derive another technique for ensuring Condition D. □

5 Monotonicity

In this section we look at how to use monotonicity to ensure Conditions D and U. Our results on the use of monotonicity extend the results of [NR90].

We make extensive use of the ϕ function defined in [NR90]. The function ϕ can take any predicate p as an argument and returns an arithmetic expression involving only the argument positions of p. Further, ϕ can also take any fact $p(\bar{a})$ as an argument, and returns an integer. Candidate functions for ϕ can be generated using the techniques discussed in [NR90]. We do not discuss this here, but assume that possible ϕ functions are made available. The function ϕ has a natural extension that also can take as argument an atom $p(\bar{t})$, and returns an arithmetic expression involving only the variables in $\bar{t}$. For instance, a ϕ that maps $fac(I, N)$ to I would map $fac(I + 1, N1)$ to $I + 1$. Such a ϕ also maps $fac(4, \_)$ to 4.

5.1 Monotonicity and Condition D

Definition 5.1 Locally Saturated : Suppose that no derivation for any p fact is repeated in an evaluation.[6] Then a set of facts S for derived body predicate occurrences of a rule R defining p is said to be *locally saturated with respect to* R if every derivation that can be made using (1) R, (2) all facts

[6] See footnote 4, Section 4.1.

for the base predicate occurrences of R, and (3) the given set of facts S, has already been made. A set of facts is said to be *locally saturated with respect to a set of rules* if it is locally saturated with respect to each of the rules. □

Since all derivations that could be made using just the set of locally saturated facts have been made, any new derivation requires at least one fact (for a derived predicate occurrence) that is not in the set of locally saturated facts.

In the case of a Basic Semi-Naive evaluation of an SCC (where the set of predicates derived with respect to p is just the set of predicates defined in the SCC of p), at any point in the $n+1$th iteration, the set of facts derived before the nth iteration is a set of locally saturated facts for p. If a different evaluation or synchronization technique is used, the sets of locally saturated facts may change, but the following results would not be affected. Thus we achieve a certain degree of independence from evaluation techniques in the following results.

Definition 5.2 Monotonicity : A rule is said to be *monotonically increasing with respect to a predicate occurrence* p' in its body if, for every instance of the rule (with $p(\bar{b})$ used in p'), $\phi(head) \geq \phi(p(\bar{b}))$, where *head* denotes the head of the instantiated rule. A rule is said to be *monotonically increasing* if it is monotonically increasing with respect to each body occurrence of a predicate that is derived with respect to the rule. □

The following is a sufficient algorithmic test for monotonicity. Consider a rule R:

$$R : p(\bar{t}) \leftarrow p_1(\overline{t_1}), \ldots, p_n(\overline{t_n}).$$

The rule is guaranteed to be monotonically increasing if for every derived literal $p_i(\overline{t_i})$, the arithmetic expression $\phi(p(\bar{t})) - \phi(p_i(\overline{t_i}))$ is always ≥ 0. This can be tested using symbolic manipulation on each expression $\phi(p(\bar{t})) - \phi(p_i(\overline{t_i}))$.

Condition Monotonicity_D : Consider an evaluation of a program P. Let p be a predicate defined in P, and S be the set of all predicates in P that are derived with respect to p (note that $p \in S$). Let $\mathcal{R}$ be the set of all the rules of P defining the predicates in S. The predicate p satisfies Condition Monotonicity_D iff every rule in $\mathcal{R}$ is monotonically increasing. □

Definition 5.3 Min-head-gap bounding function : For a predicate p as in Condition Monotonicity_D, a function γ mapping facts to integers is said to be a *min-head-gap bounding function for* p iff for each instance R' of any rule R defining p, if $p(\bar{a})$ is the head fact and $q(\bar{b})$ is a derived fact in the body of R', $(\phi(p(\bar{a})) - \phi(q(\bar{b}))) \geq \gamma(q(\bar{b}))$. □

Note that the constant function $\gamma = 0$ is always a min-head-gap bounding function—however, one might be able to get a "better" function for the purposes of the subsequent theorem.

We can algorithmically determine a min-head-gap bounding function as follows. Suppose for each rule R defining p and for each derived predicate $p_i(\overline{t_i})$ in the body of R, each expression $\phi(p(\bar{t})) - \phi(p_i(\overline{t_i}))$ not only is non-negative but also (after simplification) has as arguments only variables from $\overline{t_i}$. Then we can derive a min-head-gap bounding function for p by symbolic arithmetic manipulations on these functions. The simplification above could include replacement of variables using arithmetic equalities present in the body of the rule. For instance, if we have the rule

$$fac(X, X * N) \leftarrow X > 0, Y = X - 1, fac(Y, N).$$

we can replace Y by $X - 1$ to get the min-head-gap bounding function (the constant function 1) for fac.

Theorem 5.1 *Consider a semi-naive evaluation where predicate p satisfies Condition Monotonicity_D and γ is a min-head-gap bounding function for p. Let S and $\mathcal{R}$ be as in Condition Monotonicity_D. In this evaluation, let F be the set of all the facts that have been derived for predicates defined in S, and $F' \subseteq F$ be a set of facts such that F' is locally saturated with respect to the set of rules $\mathcal{R}$. Let*

$$m = \min\{\phi(f) + \gamma(f) \mid f \in F - F'\}$$

If a fact $p(\overline{a})$ is such that $\phi(p(\overline{a})) < m$, then $p(\overline{a})$ will not be derived again.

Proof: The set of facts F' is locally saturated with respect to $\mathcal{R}$. Hence, for any predicate $q \in S$ (derived with respect to p) any derivation of a q fact using some rule R ($\in \mathcal{R}$) must use at least one fact that is not in F'. Since the rules in $\mathcal{R}$ are monotonic, for any new fact $q(\overline{b})$, $\phi(q(\overline{b})) \geq \min\{\phi(f) \mid f \in F - F'\}$. Since γ is a min-head-gap bounding function for p, no p fact with a ϕ value less than $\min\{\phi(f) + \gamma(f) \mid f \in F - F'\}$ will be derived. □

An analogous theorem holds with monotonically decreasing rules in place of monotonically increasing rules in Condition Monotonicity_D. The theorem gives us a way of ensuring Condition D for facts when the conditions on monotonicity are satisfied.

In an iteration of Basic Semi-Naive evaluation of an SCC, the set of facts in the p relations (i.e. the facts derived two or more iterations before) constitutes F' (as mentioned earlier) and the set of facts in the δp relations (i.e. those derived in the previous iteration) and the facts derived during the current iteration constitutes $F - F'$.

Note that although the set of derived predicates as well as the set of locally saturated facts depends on the actual evaluation used, the theorem holds independent of the evaluation.

Example 5.1 Consider the following program that computes a list of factorials of all squares of integers less than some constant n.

$R1 : fac\_list(0, [1]).$
$R2 : fac\_list(N, [V \mid L]) \leftarrow N > 0, N < n, fac\_list(N-1, L), fac(N * N, V).$
$R3 : fac(0, 1).$
$R4 : fac(N, N * V) \quad \leftarrow N > 0, N < n * n, fac(N-1, V).$

Let the ϕ function map $fac\_list(N, \_)$ to N, and $fac(N, \_)$ also to N. We deduce that rule $R3$ and $R4$ are monotonically increasing. In rule $R2$, fac is a predicate from a lower SCC and hence is not derived wrt $R2$. Hence we deduce that $R1$ and $R2$ are monotonically increasing. Thus Condition Monotonicity_D is satisfied by predicates fac as well as fac_list. We also deduce min-head-gap bounding functions: the constant function 1 for fac as well as for fac_list.

From Theorem 5.1 we deduce that once a fac fact with index n is derived, no fac fact with index less than $n+1$ will ever be derived again. We deduce similar results for fac_list. □

5.2 Monotonicity and Condition U

In this section we discuss how to use monotonicity of rules to satisfy Condition U. We make use of the definitions and results in Section 5.1. Let ϕ be a function as described earlier in this section.

Definition 5.4 Body_gap : Let R be a rule and let p' and q' be predicate occurrences in its body. Let R' be an instance of R with facts $p(\overline{a_1})$ and $q(\overline{a_2})$ used in the occurrences p' and q' respectively. We then define $body\_gap(R', p', q') = \phi(p(\overline{a_1})) - \phi(q(\overline{a_2}))$. If R has at least one derived predicate occurrence

in its body, we define

$$body\_gap(R', q') = \max\{body\_gap(R', p', q') \mid p' \text{ is a derived predicate occurrence in } R\}$$

If R has no derived predicate occurrence in its body, $body\_gap(R', q') = \infty$. □

Note that if there is only one derived predicate occurrence q' in the body of a rule R, and R' is any instance of R, then $body\_gap(R', q') = 0$.

Monotonicity can be used to infer that a fact can no longer be used in a body predicate occurrence q' based on Condition Monotonicity_U and Theorem 5.2 below.

Condition Monotonicity_U : Consider an evaluation of a program P. Let R be a rule with a body predicate occurrence q'. Let $p'_1, \ldots, p'_n$ be the derived predicate occurrences in the body of the rule R. Let γ be a function that maps q facts to integers. The predicate occurrence q' in R satisfies Condition Monotonicity_U with function γ iff, for each instance R' (with $q(\overline{a})$ used in the occurrence q')

$$body\_gap(R', q') \leq \gamma(q(\overline{a})).$$

□

Intuitively the theorem states that if two facts are used in a rule to make a successful derivation, the indices of the facts are fairly "close" to each other. The function γ provides an upper bound on the gap.

Suppose for each derived predicate occurrence p'_i in the body of rule R, $\phi(p'_i) - \phi(q')$ (after simplification) involves only the variables in the literal q'. Then, by a process similar to the derivation of min-head-gap bounding functions in Section 5.1, we can derive a function γ as in Condition Monotonicity_U.

Theorem 5.2 *Consider a locally semi-naive evaluation of a program P. Let R be a rule in P and q' be a body predicate occurrence in R such that q' satisfies Condition Monotonicity_U with function γ. Suppose that no derivations using R are repeated.*[7] *Let m be an integer such that no fact for any p'_i, $1 \leq i \leq n$ with index (under the function ϕ) less than m will be derived again*[8]*. Suppose that the set of all facts $\{p_i(\overline{b}) \mid 1 \leq i \leq n$ and $\phi(p_i(\overline{b})) < m\}$ is locally saturated with respect to R.*

Then, a fact $q(\overline{a_1})$ can no longer be used in the predicate occurrence q' of R if $\phi(q(\overline{a_1})) + \gamma(q(\overline{a_1})) < m$.

Proof: Since the set of facts $\{p_i(\overline{b}) \mid 1 \leq i \leq n$ and $\phi(p_i(\overline{b})) < m\}$ is locally saturated with respect to R, and no p_i fact with ϕ value less than m will be derived again, any new derivation must use at least one derived fact with ϕ value of m or more. But by Condition Monotonicity_U if a fact $q(\overline{a_1})$ is used to make a derivation, all derived facts used with it are such that their ϕ values are less than or equal to $\phi(q(\overline{a_1})) + \gamma(q(\overline{a_1}))$. Hence a fact $q(\overline{a_1})$ cannot be used beyond this point in the evaluation if $\phi(q(\overline{a_1})) + \gamma(q(\overline{a_1})) < m$. □

Note that the theorem makes no mention of whether q is derived with respect to the head of the rule or not. An analogous theorem holds when the *body_gap* of the rule with respect to q' is bounded from below, and no fact for any p'_i with index (under ϕ) greater than some m will be derived again.

A special case of the function γ is the constant function k (for some k). The above theorem generalizes the conditions of Sliding Window Tabulation ([NR90]), since only such constant functions could be used for γ in Sliding Window Tabulation. Example 5.2 shows the need for allowing general functions.

[7] See footnote 4, Section 4.1.

[8] Theorem 5.1 may be used to ensure this.

Example 5.2 We use the program from Example 5.1 again. Consider Rule $R4$:

$$R4 : fac(N, N * V) \leftarrow N > 0, N < n * n, fac(N - 1, V).$$

There is only one derived predicate in the body of this rule, hence a fac fact can be used at most once (Condition Bounds_U). Another way of looking at this is using monotonicity. A γ function on fac that bounds *body_gap* is the constant function 0. Hence if no fac fact with index less than n will be derived henceforth, fac facts with indices less than n will no longer be used in this rule. A similar result holds for uses of *fac_list* facts in rule $R2$ shown below:

$$R2 : fac\_list(N, [V \mid L]) \leftarrow N > 0, N < n, fac\_list(N - 1, L), fac(N * N, V).$$

The one predicate occurrence left is the occurrence of fac in rule $R2$. Now we derive a function γ on fac facts that satisfies Condition Monotonicity_U, using the technique described earlier: γ maps $fac(N * N, \_)$ to $N - 1 - N * N$, and hence $fac(M, \_)$ to $\sqrt{M} - M - 1$. Using this we deduce that if no *fac_list* facts with index less than n will be produced and there are no *fac_list* facts with index less than n in the differential δ relations, then fac facts $fac(M, \_)$ such that $M + \sqrt{M} - M - 1 < n$ will no longer be used. But from Example 5.1 we know how to find what *fac_list* facts will no longer be produced: if a fact $fac\_list(n, \_)$ has been produced in an iteration, no *fac_list* fact with index less than $n + 1$ will be produced hence.

Thus in a Basic Semi-Naive evaluation, one iteration after $fac\_list(n, \_)$ has been produced we know that any $fac(m, \_)$ fact with $\sqrt{m} - 1 < n$ can no longer be used in the occurrence of fac in rule $R2$. □

6 Synchronization

A synchronizing technique orders derivations in the evaluation of a program so that derivations of facts are "close" to their uses; this helps reduce the "life-spans" of facts. Intuitively, if each fact computed in an evaluation is stored for only a short while during the evaluation, the total space required for the overall evaluation is reduced. We begin with an example where synchronizing helps in improving the space utilization of a program evaluation. In the rest of this section we present three techniques for achieving synchronization.

Example 6.1 Consider the following program, where n is some constant.

$$\begin{array}{ll}
R1 : fac(0, 1). & \\
R2 : fac(N, N * X1) & \leftarrow N > 0, fac(N - 1, X1). \\
R3 : fac1(n, 1). & \\
R4 : fac1(N, N * X1) & \leftarrow N > 0, fac1(N - 1, X1). \\
R5 : fac2(n, 1). & \\
R6 : fac2(N + 1, Y * Y1 * Y2) & \leftarrow fac2(N, Y), fac(N, Y1), fac1(N, Y2). \\
\text{Query: } ?\text{-}fac2(m, X). &
\end{array}$$

Let us consider the case when $m \geq n$; for $m < n$, the answer set to the query is empty. If each fact in this evaluation is used as soon as it is derived (or in the following iteration as when Basic Semi-Naive evaluation is used), we would have to store $n + 6$ facts at any point in the evaluation (from the $n + 1$th iteration onwards, although less in previous iterations) based on satisfaction of Conditions U and D. However, if all uses of a $fac1(N, \_)$ fact are delayed till $fac(N, \_)$ has been derived, we need store only six facts at a point in the evaluation. Since n can be arbitrarily large, synchronizing the evaluation helps considerably in improving the space utilization of the program evaluation. □

6.1 Delaying First Use of Facts

An integral part of Sliding Window Tabulation is the idea of keeping all uses of a derived fact "close" together in the evaluation—this is done by delaying the first use of a (derived) fact. Each fact is assigned an integer index by a ϕ function. At each point in the evaluation, there is an active "window" of facts; a fact whose index is not in this window is not available for immediate use in rule applications—it is *hidden* and can be used only when its index falls in the current window. In this section, we generalize this idea of [NR90] and see how it helps in synchronization of evaluation.

Condition Hiding_Facts : An evaluation of an SCC satisfies Condition Hiding_Facts if:

1. All the rules of the SCC are monotonically increasing, and
2. Suppose a derived fact $p(\bar{b})$ is used in an occurrence p' of an instance R' of a rule in the SCC. Then there exists a function γ' mapping facts to integers s.t. $body\_gap(R', p') \leq \gamma'(p(\bar{b}))$.
3. There is a finite bound min_ϕ such that $\phi(p(\bar{b})) \geq min_\phi$ for all facts $p(\bar{b})$ for each predicate p defined in the SCC.

□

Proposition 6.1 *Consider an evaluation of an SCC that satisfies Condition Hiding_Facts. Let m_D be any integer. Let F be the set of $p(\bar{b})$ facts for which $\phi(p(\bar{b})) + \gamma'(p(\bar{b})) = m_D$. Facts $q(\bar{c})$ with $\phi(q(\bar{c})) \leq m_D$ must be made available to rule applications, for facts in F to be completely used. $q(\bar{c})$ facts with $\phi(q(\bar{c})) > m_D$ cannot be used along with any fact from F in any rule application.*

Proof: Since for rule instance R' (with $p(\bar{b})$ used in predicate occurrence p') $body\_gap(R', p') \leq \gamma'(p(\bar{b}))$, any derived fact used in R' must have a ϕ value less than or equal to $\phi(p(\bar{b})) + \gamma'(p(\bar{b}))$. Hence for any fact p in the set F, if a derived fact q is used in an instance of a rule in the SCC along with p, then $\phi(q) \leq m_D$. Facts with greater ϕ values cannot be used in a rule application with any fact from F. □

6.1.1 Evaluation With Hiding Facts

Proposition 6.1 provides a basis for the *hiding of facts* to reduce space utilization. Consider an SCC S that satisfies Condition Hiding_Facts. The value min_ϕ may be determined in one of several ways: it may be determined by analysis of the program (as in Example 6.1); or, if magic rewriting is used on the program and the magic predicates corresponding to the predicates in S are in a lower SCC, it may be determined based on an evaluation of the SCC containing the magic predicates[9].

At a point in the evaluation of S, let m_D be the greatest integer such that the set of all program facts with ϕ values $< m_D$ is locally saturated with respect to all the rules in the SCC. Initially, m_D is set to min_ϕ. Since the SCC S has monotone rules, the value of m_D can be determined at later points in the evaluation as discussed in Section 5.1. We modify the evaluation of S by always hiding derived facts with indices greater than m_D. The value m_D could increase each time facts are derived; it can be updated, for instance, at the end of each iteration in a Basic Semi-Naive evaluation of the SCC.

To see how delaying the first use of facts can improve space utilization, consider $q(\bar{c})$ facts with $\phi(q(\bar{c})) > m_D$. Any $p(\bar{d})$ fact that can be used in a rule application with such a $q(\bar{c})$ fact would have $\phi(p(\bar{d})) + \gamma'(p(\bar{d})) > m_D$. Since facts with a ϕ index of m_D can still be derived, such a $p(\bar{d})$ fact cannot be discarded at this point in the evaluation based on Theorem 5.2 to ensure Condition U. If these $q(\bar{c})$

[9] For some programs neither of these techniques may be applicable.

facts are used along with $p(\overline{d})$ facts in a rule application, new facts can be derived but none of the ($p(\overline{d})$ or $q(\overline{c})$) facts used to derive these new facts can be discarded. By hiding $q(\overline{c})$ facts with a ϕ value greater than m_D, derivations that use these facts are delayed until some of the $p(\overline{d})$ facts that can be used along with the $q(\overline{c})$ facts can be discarded; this can improve the space utilization of the program. Note that if the set of facts with a ϕ index of m_D are also hidden, the set of locally saturated facts would not change, the value of m_D would not increase and evaluation would not proceed any further.

As seen in Example 6.1, hiding facts in this fashion could greatly reduce the space utilized by a program.

Our contribution in this section is twofold. Firstly, we isolate the synchronization achieved by hiding facts in an evaluation from other components of space optimization techniques. Secondly, [NR90] had the restriction that the *body_gap* be bounded above by a constant. We generalize this to handle the *body_gap* being bounded by an arbitrary function of facts.

6.2 Nested-SCC Synchronization

Consider a program P, with two SCCs, S_1 and S_2. Let p be a predicate that is defined in S_2 and used in S_1. Let P^{mg} be the magic rewritten program ([Ram88]) obtained from P. P^{mg} can be partitioned into: $\mathcal{R}_1$, obtained by applying the magic transformation to the rules in S_1 treating predicates that are not in S_1 as base predicates; $\mathcal{R}_2$, obtained similarly from S_2; and $\mathcal{R}_{ext}$, which is the set of magic rules generated from body occurrences in S_1 of predicates defined in S_2. The Nested-SCC technique essentially views the rules in $\mathcal{R}_{ext}$ as generating subgoals, and solves them by obtaining the fixpoint of $\mathcal{R}_2$.

Nested-SCC synchronization should be used only if $\mathcal{R}_2$ is safely computable [KRS88]. The following algorithm assumes that P contains only S_1 and S_2, but can be easily extended to the general case, where P also contains SCCs other than S_1 and S_2.

Algorithm Nested-SCC_Synchronize $(\mathcal{R}_1, \mathcal{R}_2)$
  Let $R_1, \ldots, R_n$ be the rules in $\mathcal{R}_1$.
  Let $mR_{i,1}, \ldots, mR_{i,m_i}$ be the (magic) rules in $\mathcal{R}_{ext}$ derived from R_i,
    in some total ordering consistent with the sip order in R_i.
  Repeatedly apply the rules in $\mathcal{R}_1$, subject to the following restrictions, until a fixpoint is reached.
  (1) Before applying a rule R_j from $\mathcal{R}_1$, do for $k = 1 \ldots m_j$
  (2) Apply $mR_{j,k}$ and then compute the closure of the set of rules $\mathcal{R}_2$.
end Nested-SCC_Synchronize

As long as $\mathcal{R}_2$ is safely computable, the technique is guaranteed to be sound and complete.

Proposition 6.2 *If $\mathcal{R}_1$ and $\mathcal{R}_2$ are evaluated using Nested-SCC synchronization, each predicate defined in $\mathcal{R}_2$ is base with respect to every rule in $\mathcal{R}_1$.* □

Several of the techniques for satisfying Conditions D or U used the notion of predicates being base with respect to rules. By using Nested SCC synchronization we may enable the use of one of those techniques in a place where it may not otherwise be applicable. In Section 7.2 we see another way in which Nested SCC synchronization can help is in discarding facts.

6.3 Interleaved-SCC Synchronization

Interleaved-SCC synchronization is a form of synchronization that exploits SCC structure. The intuition behind the technique is as follows. Consider a predicate p defined in an SCC. A p-fact must be retained until Conditions U and D are satisfied by it in this ("producer") SCC; in addition, it must be retained until it has been used completely in all occurrences of p in other ("consumer") SCCs. If our evaluation proceeds SCC-by-SCC, the producer SCC evaluation must be completed before evaluation of the consumer SCCs can begin, and p-facts must therefore be retained at least until the end of the evaluation of the producer SCC. However, it is sometimes possible to use the p fact in all consumer SCCs soon after it is produced by interleaving the evaluation of SCCs, thereby making it possible to discard it sooner, while retaining all the advantages of an SCC by SCC semi-naive evaluation.

We present the technique by describing the interleaving of a producer SCC (defining a single predicate p) and one or more consumer SCCs for p. Any SCC (other than the producer) that contains occurrences of p must be treated as a consumer and the producer and all consumer SCCs must satisfy the following condition for the technique to be applicable[10].

Condition Interleaved-SCCs :

- The producer and each of its consumer SCCs must contain only monotonically increasing rules.
- In each consumer SCC S_j, for each rule R that contains a body predicate occurrence p' of p, either (1) for each occurrence q' of any derived predicate q in the body of R, there exists a function $\gamma_{p',q'}$ that maps q facts to integers such that for each instance R' of R (where say $q(\bar{b})$ is used in the occurrence q'), $body\_gap(R', p', q') \leq \gamma_{p',q'}(q(\bar{b}))$; or (2) there is a bound $max_{p'}$ such that for any fact $p(\bar{b})$ that can be used in the occurrence p', $\phi(p(\bar{b})) \leq max_{p'}$.

□

We now describe the *Interleaved-SCC* synchronization technique, which works on any subprogram that satisfies Condition Interleaved-SCCs. Consider a rule R in a consumer SCC S_j. Let p' and q' be occurrences in the body of R of predicates p and q; p defined in a producer SCC (of S_j) and q derived with respect to R. We define the following indices:

$$\begin{aligned} m(p', q') &= \max\{\{-\infty\} \cup \{\phi(q(\bar{b})) + \gamma_{p',q'}(q(\bar{b})) \mid q(\bar{b}) \text{ is an available fact}\}\} \\ M(p') &= \min\{m(p', q') \mid q' \text{ is a derived predicate occurrence in the body of } R\} \\ &= max_{p'} \text{ if there is no derived predicate occurrence in the body of } R \\ \psi(p, S_j) &= \max\{M(p') \mid p' \text{ is an occurrence of } p \text{ in the body of any rule in } S_j\} \end{aligned}$$

$m(p', q')$ is the index of the largest (under the ϕ function) p fact that can possibly be used in p' with an available q fact in q'. $M(p')$ is the index of the largest p fact that can be used in the occurrence p' (with the set of currently known facts in S_j). The index of the largest p fact that can be used with the set of currently known facts in S_j is given by $\psi(p, S_j)$ and this index is available to the SCC that defines p. Using these indices, Interleaved-SCC synchronization can be expressed as follows[11]:

[10] Although we consider only a single predicate defined in a producer SCC and require that all consumers of the predicate satisfy Condition Interleaved-SCCs, we can extend the condition as well as the synchronization technique to relax these restrictions.

[11] This definition assumes concurrent threads of execution, but it can be reformulated, with a loss of concurrency and some extra checks, as a sequential iteration.

Algorithm Interleaved-SCC_Producer (S)

(1) repeat
(2) Let $top = \min_j\{\psi(p, S_j) \mid S_j$ uses p and is waiting on $S\}$.
(3) Evaluate S till no facts $p(\bar{b})$ such that $\phi(p(\bar{b})) \leq top$ can be derived.
/* Tested using monotonicity; any technique may be used to evaluate S. */
(4) Release any SCCs S_j waiting on S such that $\psi(p, S_j) = top$.
(5) forever
end Interleaved-SCC_Producer

Algorithm Interleaved-SCC_Consumer (S_j)

(1) Evaluate S_j with the following restriction:
(2) Whenever new facts are made available for derived predicates in S_j do
(3) Update the indices m, M and ψ.
(4) Wait on producer SCCs of S_j.
end Interleaved-SCC_Consumer

Although the discussion so far assumed "monotonically increasing," if "increasing" is uniformly changed to "decreasing", the above results and algorithms hold with simple modifications.

Theorem 6.1 *If SCCs $S_0, S_1, \ldots, S_m$ are evaluated using Interleaved-SCC synchronization with S_0 as the producer and $S_1, \ldots, S_m$ as its consumers, each predicate defined in S_0 is base with respect to every rule in $S_1, \ldots, S_m$.*

Proof: Consider a single predicate p and a single consumer SCC S_j that uses p. In order to prove the theorem, we only need show that $\psi(p, S_j)$ is indeed the largest ϕ value of any p fact that can be used in a derivation with any of the current set of derived facts in S_j. It then follows from the algorithm that any p fact that could possibly be used is indeed made available, and hence p is base with respect to every rule in S_j.

We show that $\psi(p, S_j)$ works as claimed by starting with $m(p', q')$. By the *body_gap* requirement of Condition Interleaved-SCCs and the definition of $m(p', q')$, $m(p', q')$ is indeed the index of the largest (under the ϕ function) p fact that can possibly be used in p' with an available q fact in q'. For a given rule R, if p facts with index greater than some value n cannot be used in predicate occurrence p' with the available q facts for some predicate occurrence q', they cannot be used in a successful derivation with the available facts for the derived predicates. Hence in the definition of $M(p')$ we take the minimum over all derived body predicate occurrences q'; $M(p')$ is then the index of the largest p fact that can be used in the occurrence p' with the set of currently available facts in S_j. Since in the definition of $\psi(p, S_j)$ we take the maximum over all predicate occurrences, $\psi(p, S_j)$ works as claimed. □

Example 6.2 Consider again the program from Example 5.1, which we repeat below for easy reference.

$R1 : fac\_list(0, [1]).$
$R2 : fac\_list(N, [V \mid L]) \leftarrow N > 0, N < n, fac\_list(N-1, L), fac(N * N, V).$
$R3 : fac(0, 1).$
$R4 : fac(N, N * V) \leftarrow N > 0, N < n * n, fac(N-1, V).$

This program has two SCCs, the lower one containing the predicate fac and the upper one containing fac_list. Let us call the lower SCC which is a producer of fac as $S1$ and the higher SCC, which is a consumer of fac, as $S2$. There is only one rule $R2$ in $S2$ that uses the predicate fac. This rule has a derived predicate fac_list. We assume that we use Basic Semi-Naive evaluation for the consumer SCC.

We derive the function γ that maps $fac\_list(N-1,\_)$ to N^2-N+1, (and hence $fac\_list(N,\_)$ to N^2+N+1) to bound $body\_gap(R2, fac(N*N,V), fac\_list(N-1,L))$. SCCs $S1$ and $S2$ satisfy Condition Interleaved-SCCs with this function γ that bounds body-gap. We can then use Interleaved_SCC evaluation to evaluate this program.

After each Basic Semi-Naive iteration of the consumer SCC (in Procedure Interleaved_SCC_Consumer) new facts are produced. Using these facts we find the maximum value of $\phi(fac\_list(N,\_))+\gamma(fac\_list(N,\_))$. But this function simplifies to N^2+2N+1. Thus if $fac\_list(n,\_)$ has been produced, we need fac facts with indices up to n^2+2n+1. We then call Procedure Interleaved_SCC_Producer($S1$). SCC $S1$ then iterates, producing fac facts. Due to monotonicity of rules in $S1$, we know that when $fac(n^2+2n+1,\_)$ has been produced, all fac facts with indices $\leq n^2+2n+1$ have been produced. Hence Procedure Interleaved_SCC_Producer returns, and Procedure Interleaved_SCC_Consumer continues with its next iteration.

Suppose we use Interleaved-SCC synchronization on this program, along with monotonicity to test for Conditions D and U, and discard a fact once it satisfies Conditions D and U. The next question is, how much space is used? It is easy to see that in SCC $S2$, only two fac_list facts are retained at any point in the evaluation; each fac_list fact uses $O(n)$ space. As for SCC $S1$, we store at most facts with indices from $(n-1)^2$ to n^2, which means at most $2n-1$ facts are stored. Thus we use a total of $O(n)$ space using this space optimization technique. If we do not discard any facts during the evaluation, we would store $O(n^2)$ facts. By discarding facts during the evaluation, we have achieved an order of magnitude improvement in the space utilized in evaluating this program. □

6.4 Using Inverted Rules

In several cases (such as monotonically increasing SCCs that have been rewritten using the Magic Sets transformation), the conditions for Interleaved SCCs are almost met, except that the two SCCs are monotone in opposite directions. By using the notion of inverted rules, we can still use Interleaved-SCC evaluation in some cases. Consider the following example:

Example 6.3 The following Magic rewritten program P^{mg}_{Fac} is used to compute the nth Factorial number.

$R1: m\_fac(n).$
$R2: fac(0,1) \leftarrow m\_fac(0).$
$R3: m\_fac(N-1) \leftarrow m\_fac(N), N>0.$
$R4: fac(N, N*X1) \leftarrow m\_fac(N), N>0, fac(N-1, X1).$

There are two SCCs in this program—SCC $S1$ defining m_fac and SCC $S2$ defining fac. Unfortunately, the conditions for Interleaved-SCC synchronization are not satisfied, since the two SCCs are monotone in opposite directions. Each $m\_fac(i)$ fact is used in the computation of $m\_fac(i-1)$ (using rule R3) and in the computation of $fac(i)$ (using rule R4). Since none of the rules defining fac can be applied until all the m_fac facts have been computed, the two uses of an m_fac fact are considerably "separated" in the evaluation.

If an m_fac fact is not discarded until it has been used to compute the corresponding fac fact, we do not achieve any savings in the space complexity of this program; we still need to store $O(n)$ facts. □

Naughton and Ramakrishnan [NR90] introduced the notion of inverted magic rules. Suppose a set of rules is monotone. The set of *fringe facts* for these rules are those that do not generate any new facts. We can in some cases use the original set of rules in reverse—feed them the head facts and regenerate

the body facts. This is done using "inverted" rules created by swapping the head and one of the body literals in a rule. We now generalize the notion of inverted rules and see how inverted rules could be used to guarantee Condition U.

Definition 6.1 Inverted Rules : Consider a set of monotonically decreasing (or, increasing) rules $\mathcal{R}$ such that the evaluation of these rules, using a set of base facts B, computes a set D of derived facts. A set of monotonically increasing (resp. decreasing) rules $\mathcal{R}'$ is said to be the set of *inverted rules* with respect to $\mathcal{R}$, if there exists a set $D' \subset D$ (called "fringe" facts in [NR90]) such that $D' \cup$ the set of facts computed by the evaluation of $\mathcal{R}'$ using $B \cup D'$ is exactly equal to D.

A set of rules $\mathcal{R}$ is said to be *invertible* if there exists an $\mathcal{R}'$ that is the set of inverted rules with respect to $\mathcal{R}$. □

In general, a set of rules may not be (non-trivially) invertible. In some cases, the technique used by [NR90] (of interchanging the head and derived body predicates) can be used (with minor modifications) to generate inverted rules. This technique does not always succeed; the resultant rules may compute more facts than were computed by the original rules. However, computing a superset of the desired set of derived facts may still be acceptable in some cases and could achieve space improvements; see [NR90] for a further discussion.

If a given set of rules $\mathcal{R}$ is invertible, the inverted rules $\mathcal{R}'$ provide a mechanism to ensure that any (non-fringe) fact computed by an evaluation of $\mathcal{R}$ can be discarded without violating Condition U—these facts will be recomputed using $\mathcal{R}'$ (if $\mathcal{R}'$ is evaluated after $\mathcal{R}$); any derivation that uses these facts can be made once these facts are recomputed.

Condition Inverted_Rules : A set of SCCs $S_0, S_1, \ldots, S_k, S_{k+1}, \ldots, S_m$ satisfies this condition if:

- These SCCs satisfy the conditions for Interleaved-SCC synchronization with S_0 as the producer SCC and $S_1, \ldots, S_m$ as consumer SCCs, except that the rules in S_0 are monotonic in the opposite direction to the rules in (the non-empty set of SCCs) $S_{k+1}, \ldots, S_m$.
- The set $\mathcal{R}_0$ of rules in S_0 is invertible.
- The evaluation of SCCs $S_{k+1}, \ldots, S_m$ does not need to be done before or during the evaluation of SCCs $S_1, \ldots, S_k$.

□

Note that the last condition depends on the synchronization technique chosen for the other parts of the program. One way of making this choice is described in Section 8.2. If the above condition is satisfied for a set of SCCs, we can use the following variant of Interleaved-SCC synchronization:

Algorithm Inverted_Rules_Eval $(S_0, \{S_1, \ldots, S_m\})$
  Let the set of inverted rules obtained from $\mathcal{R}_0$ be $\mathcal{R}'_0$.
  (1) Evaluate S_0 and $S_1, \ldots, S_k$ using Interleaved-SCCs synchronization.
  (2) Evaluate the set of rules $\mathcal{R}'_0$ (with the fringe facts) and $S_{k+1}, \ldots, S_m$
    using Interleaved-SCCs synchronization with $\mathcal{R}'_0$ as the producer SCC.
end Inverted_Rules_Eval

Theorem 6.2 *Consider a set of SCCs in a program that satisfies Condition Inverted_Rules and is evaluated using Algorithm* Inverted_Rules_Eval. *Assume that in Step (2) of the algorithm, facts computed in*

$\mathcal{R}'_0$ satisfy U and D before they are discarded. If the restrictions of Conditions U and D to $S_0, S_1, \ldots, S_k$ are satisfied by any non-fringe fact computed in S_0, Conditions U and D are satisfied by that fact in the evaluation.

Proof: Suppose the restrictions of Conditions U and D to $S_0, S_1, \ldots, S_k$ are satisfied by a non-fringe fact $p(\overline{a})$. The fact will be computed again by the inverted rules (by Condition Inverted_Rules and Algorithm Inverted_Rules_Eval). Since the inverted rules and SCCs $S_{k+1}, \ldots, S_m$ are evaluated using Interleaved SCC Synchronization, p is base with respect to the rules in $S_{k+1}, \ldots, S_m$. Hence this fact will be computed again before the first time it could be used in one of $S_{k+1}, \ldots, S_m$. There are no other uses of the fact, so Condition U is satisfied by the fact. Further, Condition D is also satisfied since the fact is used only in $S_1, \ldots, S_k$ initially and only in $S_{k+1}, \ldots, S_m$ when it is derived again, and these two sets of SCCs are disjoint. □

Example 6.4 We continue with Example 6.3. Condition Inverted_Rules is satisfied by the pair of SCCs. The pair of SCCs are monotone in opposite directions, but satisfy Condition Interleaved-SCCs otherwise. A γ function which is the constant function 1 is used for rule $R4$. For rule $R2$ we have a bound 0 such that any m_fac fact used here has index ≤ 0. The rules in the producer SCC $S1$ can be inverted. There is only one inverted rule which is as follows[12]:

$$R' : m\_fac(N+1) \leftarrow m\_fac(N), N < n.$$

In the first phase of Algorithm Inverted_Rules_Eval we evaluate $S1$. Facts other than fringe facts can be discarded during the evaluation of $S1$ once they satisfy Conditions D and U with respect to $S1$ alone. We can use either monotonicity or bounds on number of uses of facts to satisfy Conditions D and U. In either case, we store only two facts during this evaluation. A fact $m\_fac(i)$ is discarded once it has been used to compute $m\_fac(i-1)$; since however $m\_fac(i)$ is needed in the computation of $fac(i)$, it is recomputed in the second phase of Algorithm Inverted_Rules_Eval using the inverted magic rule R'.

In the second phase we use Interleaved-SCC synchronization with the producer being rule R' and the consumer being $S2$. We skip the details here, but note that at most two m_fac facts and two fac facts are stored at any point in the evaluation. We have thus reduced the space complexity from $O(n)$ to $O(1)$. Sliding Window Tabulation is applicable on this program, and would also use $O(1)$ space.

Sliding Window Tabulation is not applicable, however, on the magic program obtained from the *fac_list* program (described in previous sections), whereas Inverted_Rules_Eval is applicable. □

7 Combining Techniques I: Examples

In this section, we give a few examples of combinations of synchronization techniques with techniques to ensure U and D to obtain space optimization techniques for subprograms.

7.1 Sliding Window Tabulation

The Sliding Window Tabulation scheme of [NR90] is an example where the technique of adding inverted rules to a program is used in conjunction with delaying the first use of facts for synchronization and

[12]The inverted magic rule generated in [NR90] did not have the condition $N < n$ in it, but the algorithm ensured that evaluation did not proceed beyond n.

monotonicity of derivations and uses to ensure D and U. Sliding Window Tabulation works on programs that satisfy the following condition:

Condition Sliding_Window_Tabulation :

1. The magic program P^{mg} has exactly two SCCs—the lower SCC S_2 only containing the magic predicates (and rules defining them), and the higher SCC S_1 only containing the (derived) predicates (and the corresponding rules) of the original program.
2. The rules in S_1 are monotonic in the opposite direction to the rules in S_2.
3. The set of rules $\mathcal{R}_2$ in S_2 can be inverted to get $\mathcal{R}'_2$—the set of fringe facts being those magic facts derived using $\mathcal{R}_2$ that do not generate any new magic facts, and
4. In P^{mg}, the body_gap in each rule with respect to each of the (non-magic and corresponding magic) predicates is bounded by a constant.

□

If the rewritten magic program P^{mg} satisfies these conditions, the evaluation can be understood as follows:

Algorithm Sliding_Window_Tabulation_Eval (S_1, S_2)

Let the set of inverted magic rules obtained from the set of rules $\mathcal{R}_2$ in S_2 be $\mathcal{R}'_2$.

(1) Evaluate the rules $\mathcal{R}_2$ using monotonicity to satisfy Conditions D and the restriction of U to the uses of magic facts in S_2. The first use of facts is delayed by hiding facts based on the body_gap of the (magic) rules in S_2, and fringe facts are not discarded.

(2) $\mathcal{R}'_2$ and the set of rules in S_1 are evaluated using Interleaved-SCC Synchronization[13]. Monotonicity is used to satisfy Conditions D and U and the first use of facts is again delayed by hiding facts. The "lowest" fact defined in S_1 can be determined since $\mathcal{R}_2$ is evaluated before S_1.

end Sliding_Window_Tabulation_Eval

Using our generalized techniques for ensuring U, D and achieving synchronization based on monotonicity, the basic techniques of Sliding Window Tabulation can be extended in many ways beyond the class of programs described by Naughton and Ramakrishnan. One possible extension is based on using Algorithm Inverted_Rules_Eval for synchronization of multiple consumer SCCs for a single producer; another extension permits the body_gap of the rules to be bounded by some function of the facts, not just a constant.

7.2 The Nested-SCC Discarding Technique

The *Nested-SCC Discarding* technique combines Nested-SCC synchronization with a rather straightforward technique for ensuring U, described below, that may discard a fact even though it may not have been used to make all the derivations that it should have.

[13] Actually, we need to extend Interleaved Evaluation a little to handle the full generality of Sliding Window Tabulation. Sliding Window Tabulation can handle some exit rules for which no bound $max_{p'}$ (defined in Condition Interleaved-SCCs) exists. It treats these rules as though they were derived rules, and make only some magic facts available to them at a time. Although Interleaved SCC evaluation can be extended to handle such cases, we omit the tedious details of the extension.

Let $S_1, S_2, \mathcal{R}_1$ and $\mathcal{R}_2$ be defined for a program P and its corresponding magic program P^{mg} as in Section 6.2 and let $\mathcal{R}_2$ be safely computable. Algorithm Nested-SCC_Discard describes the evaluation when P consists of only two SCCs S_1 and S_2; its extension to more general cases is straightforward.

Algorithm Nested-SCC_Discard ($\mathcal{R}_1, \mathcal{R}_2$)

(1) Evaluate $\mathcal{R}_1$ and $\mathcal{R}_2$ using Nested-SCC Synchronization, with the following technique for discarding facts.

(2) While computing the closure of $\mathcal{R}_2$, discard facts computed in $\mathcal{R}_2$ (other than those that match the set of external subgoals) based on the restrictions of D and U to the rules of $\mathcal{R}_2$; the external subgoals can also be discarded after computing the closure.

(3) After applying a rule in $\mathcal{R}_1$, discard all answers to external subgoals generated by the rule.

end Nested-SCC_Discard

Example 8.1 shows the asymptotic improvements in space complexity that could be achieved via the use of this technique.

Theorem 7.1 *Facts discarded in Steps 2 and 3 of Algorithm* Nested-SCC_Discard *satisfy Condition U when they are discarded. However, they do not necessarily satisfy Condition D when they are discarded.* □

Derivations may be repeated when using the Nested-SCC Discarding technique, and hence the evaluation is not a semi-naive evaluation. However, it is a locally semi-naive evaluation (see Definition 2.4). The evaluation of the closure of $\mathcal{R}_2$ is entirely independent of the techniques used to evaluate $\mathcal{R}_1$. Computation within $\mathcal{R}_2$ can be synchronized using any applicable technique—the rules in $\mathcal{R}_2$ need not all be evaluated together. Although facts that are discarded may not satisfy Condition D overall, within a single call to $\mathcal{R}_2$ no derivations in $\mathcal{R}_2$ are repeated.

Since Nested-SCC_Discard does not satisfy Condition D in general, computation may be repeated by the evaluation. However, in some special cases no derivations are repeated.

Definition 7.1 Independence of External Subgoals : Consider a program P with an SCC S defining a predicate p_1, and let P^{mg} be the magic rewritten form of P. Assume we are given a set of external subgoals for predicates defined in S. A set of external subgoals is said to be *independent with respect to* p_1 if no p_1 fact is used in the derivation of answers to more than one external subgoal in the set. A set of external subgoals is said to be *independent with respect to* S if it is independent with respect to each of the predicates defined in S. □

In terms of derivation trees, independence of external subgoals s_1 and s_2 with respect to a predicate p means that no p fact is common to the derivation trees of answers to s_1 and s_2.

Theorem 7.2 *Consider a program as in Section 7.2 that is evaluated using* Nested-SCC_Discard. *Let p_1 be a predicate defined in S_2. Suppose that the set of external subgoals generated by $\mathcal{R}_{ext}$ (for the predicates defined in S_2) is independent with respect to p_1. Then, p_1 facts discarded in Steps 2 and 3 of* Nested-SCC_Discard *satisfy Condition D when they are discarded, provided that no external subgoal is repeated.* □

Several programs satisfy the conditions of Theorem 7.2. These including the Quicksort program to sort a list and P^{mg}_{Rev} to reverse a list (when the list elements are all distinct). However, determining whether these programs satisfy the conditions of Theorem 7.2 is based on checking non-trivial properties

of the program; this may be hard to determine automatically.

7.3 The Nested-SCC Cutset Technique

In this section, we combine the Nested-SCC synchronizing technique with another technique for satisfying Conditions U and D. Consider a program and let S_2 and $\mathcal{R}_2$ be as in Section 7.2. This technique differs from the Nested-SCC Discarding technique in that we retain a subset of the facts computed while evaluating the closure of $\mathcal{R}_2$ in response to an external subgoal, instead of retaining just the answers to the external subgoal. In the case when this subset of facts forms a "cutset," as defined below, we may improve the space requirements of the evaluation while satisfying Condition D even if the external subgoals are not independent with respect to the predicates in S_2. This space optimization technique is referred to as the *Nested-SCC Cutset* technique.

Definition 7.2 Fact Graph : The *fact graph* is a directed graph, with a node corresponding to each fact derived during the bottom-up evaluation of the program. There is an edge from $p_1(\overline{a_1})$ to $p_2(\overline{a_2})$ if there is a rule instance deriving $p_2(\overline{a_2})$ that has $p_1(\overline{a_1})$ in the body. □

Definition 7.3 Cutset : Given a directed graph $G = (V, E)$, a set of nodes $V_C \subseteq V$ is said to form a *cutset* between a set of nodes, $V_A \subseteq V$, and another set, $V_B \subseteq V$, in the graph, if every directed path from any node in V_A to any node in V_B and every directed path from any node in V_B to any node in V_A using the edges of E passes through a node in V_C. □

We say that a subgoal (magic fact) is *completed* at a point e in the evaluation if all derivations of all answers to the subgoal have been made before e.

Condition Nested-SCC_Cutset : Consider a program and let $S_2, \mathcal{R}_1, \mathcal{R}_2$ and $\mathcal{R}_{ext}$ be as in Section 7.2, where $\mathcal{R}_2$ is evaluated using Nested-SCC Synchronization with respect to $\mathcal{R}_1$. Consider an evaluation of the program such that no derivation using $\mathcal{R}_2$ is repeated. The evaluation of the program satisfies Condition Nested-SCC_Cutset if for any point $e1$ in the evaluation there are sets of facts A, B and C such that:

1. $B \cup C$ contains all subgoals that are completed at $e1$ and all the answers to the subgoals.
2. A contains all subgoals that are either generated at or before $e1$ but not completed at $e1$ or are generated externally after $e1$ and also contains the answers to all these subgoals.
3. C contains completed subgoals such that: (a) B forms a cutset between A and C in the fact graph for the program, and (b) all answers to subgoals in C are also present in C.

□

Note that A, B and C partition the facts that are computed by $\mathcal{R}_2$ at or before $e1$. In order to gain some space benefits, we also desire that C be non-empty, except possibly at the beginning of the evaluation of $\mathcal{R}_2$.

Theorem 7.3 *Consider an evaluation of a program that satisfies Condition Nested-SCC_Cutset and let $e1, A, B$ and C be as in the condition. Then:*

1. *No fact in C (resp. A) occurs in the body of any instantiated rule whose head is a fact in A (resp. C).*
2. *At point $e1$, facts in C satisfy Condition D and the restriction of Condition U to $\mathcal{R}_2$.*

Proof: The proof of part (1) of the theorem is trivial since B forms a cutset between C and A in the dependency graph for all p and m_p facts.

From part (1) of the theorem (that no A fact occurs in the body of any instantiated rule used to derive a C fact), we know that no A fact can be directly used to compute a C fact. Hence only B and C facts can be used to compute C facts. However we know that the subgoals in B and C are completed at $e1$ hence there are no new derivations of C facts. Since no derivation using $\mathcal{R}_2$ is repeated, it follows that that facts in C satisfy Condition D at $e1$.

Again from part (1) of the theorem we know that C facts cannot be used in the body of a rule ($\in \mathcal{R}_2$) that derives an A fact. Since the subgoals in B and C are completed, C facts cannot be used to make any new derivations of B or C facts using rules in $\mathcal{R}_2$. Since no derivations in $\mathcal{R}_2$ are repeated, facts in C satisfy the restriction of Condition U to $\mathcal{R}_2$ at $e1$. □

The above theorem gives us a technique for ensuring that facts satisfy Conditions D and U in the case when Nested-SCC synchronization is used, and Condition Nested-SCC_Cutset is satisfied.

7.3.1 Sufficient Conditions for the Cutset Property

In general, determining if a subprogram has the cutset property and can satisfy Condition Nested-SCC_Cutset is quite difficult. However, in some restricted cases, we can easily determine whether a given program has the cutset property, and if so determine what the cutsets are.

For a monotonically increasing rule R and an instance R' of R, we define $max\_head\_gap(R')$ as $\max\{\phi(head) - \phi(q') \mid q'$ is a fact in the body of R' for q derived with respect to $R\}$.

Condition Monotonic_Cutset : Consider a magic subprogram $\mathcal{R}$ that defines only predicates p and the corresponding magic predicate m_p. Let the only rules defining m_p be the magic rules obtained from body occurrences of p in the rules defining p. $\mathcal{R}$ satisfies Condition Cutset with function $\sigma_{\mathcal{R}}$ if:

1. m_p has only one argument (this is also an argument of p) and this argument takes only integer values,
2. the function ϕ maps p and m_p facts to the value of this integer argument,
3. all the rules defining p are monotonically increasing,
4. $\sigma_{\mathcal{R}}$ maps facts to integers such that for each instance R' of each rule R defining p in $\mathcal{R}$, $max\_head\_gap(R') \leq \sigma_{\mathcal{R}}(head)$,
5. the function $\phi(fact) - \sigma_{\mathcal{R}}(fact)$ is monotonically increasing with respect to ϕ, and
6. there exists a rule $m\_p(I-1) \leftarrow m\_p(I), I > a$ in $\mathcal{R}$, for some constant a such that $p(J, \_)$ facts are only defined for $J \geq a$.

□

Lemma 7.1 *Consider a subprogram $\mathcal{R}$ that satisfies Condition Monotonic_Cutset with function $\sigma_{\mathcal{R}}$. Then for any k,*

1. *The set of p and m_p facts with indices in the range $[k - \sigma_{\mathcal{R}}(p(k, \overline{a})) + 1, k]$ forms a cutset between facts for these predicates with indices $\leq k - \sigma_{\mathcal{R}}(p(k, \overline{a}))$, and facts for these predicates with indices $> k$ in the dependency graph for all p and m_p program facts.*

2. *Each of the p and m_p facts in this range are computed if the seed magic fact m_p(k) is added to* $\mathcal{R}$.

Proof: If an instantiated rule defining a p fact has a p or m_p fact with index $\leq k - \sigma_{\mathcal{R}}(p(k, \overline{a}))$ in its body, parts (4) and (5) of Condition Monotonic_Cutset ensure that the index of the head fact cannot be greater than k.

Since the rules defining p are monotonically increasing, $max\_head\_gap(R') \geq max\{body\_gap(R', p') \mid p'$ is a derived predicate occurrence in rule R defining $p\}$. Now, the magic rules (defining m_p) are derived from occurrences of p in the rules defining p. Hence, the absolute value of the max_head_gap for an instance of a magic rule is less than or equal to the max_head_gap for a corresponding instance of the rule from which this magic rule was derived. This shows that if an instantiated rule defining an m_p fact has a p or m_p fact greater than k in its body, the index of the head fact cannot be $\leq k - \sigma_{\mathcal{R}}(p(k, \overline{a}))$. This completes the proof of part (1) of the lemma.

Parts (1), (2), and (6) of Condition Monotonic_Cutset ensure that the m_p facts for each possible p fact in the range $[k - \sigma_{\mathcal{R}}(p(k, \overline{a})) + 1, k]$ are computed when the seed magic fact $m\_p(k)$ is added to $\mathcal{R}$. Since the rules defining p are monotonically increasing and the magic predicate in the bodies of rules defining p does not restrict the computation, each of the p facts in this range is computed during the bottom-up evaluation. This completes the proof of part (2) of the lemma. □

From Theorem 7.3 and Lemma 7.1, we get the following result:

Theorem 7.4 *Consider a program and let* $S_2, \mathcal{R}_1, \mathcal{R}_2$ *and* $\mathcal{R}_{ext}$ *be as in Section 7.2, where* $\mathcal{R}_2$ *is evaluated using Nested-SCC Synchronization with respect to* $\mathcal{R}_1$. *Further, let* $\mathcal{R}_2$ *satisfy Condition Monotonic_Cutset with function* $\sigma_{\mathcal{R}_2}$. *Let* e_1 *be any point in the evaluation of the program such that all* $m\_p(N)$ *facts generated using* $\mathcal{R}_{ext}$ *after* e_1 *are for* $N > k$; *also let the magic fact* $m\_p(k)$ *have been generated and the answers to it computed by evaluating* $\mathcal{R}_2$. *Then the set of p and m_p facts with indices* $\leq k - \sigma_{\mathcal{R}_2}(p(k, \overline{a}))$ *satisfy Condition D and the restriction of Condition U to* $\mathcal{R}_2$ *at* $e1$.

Proof: Since the magic fact $m\_p(k)$ has been generated and the answers to it computed, it follows from Lemma 7.1 that all p and m_p facts in the range $[k - \sigma_{\mathcal{R}}(p(k, \overline{a})) + 1, k]$ will have been generated by $e1$. These facts form a cutset between facts for these predicates with indices $\leq k - \sigma_{\mathcal{R}}(p(k, \overline{a}))$, and facts for these predicates with indices $> k$ in the dependency graph for all p and m_p program facts. Since the set of external $m\_p(N)$ facts generated after $e1$ are for $N > k$, p and m_p facts with indices $\leq k - \sigma_{\mathcal{R}_2}(p(k, \overline{a}))$ satisfy Conditions D and the restriction of Condition U to $\mathcal{R}_2$ at $e1$, according to Theorem 7.3. □

It can be seen that U_2 in Example 8.2 satisfies the above theorem with $\sigma = 1$; $fac(k, \_)$ and $m\_fac(k)$ form a cutset between all fac and m_fac facts with indices less than k and all such facts with indices greater than k.

8 Combining Techniques II: Framework

Recall that every space optimization technique has three components—ensuring Condition U for facts before they are discarded, ensuring Condition D for facts before they are discarded, and synchronization techniques to ensure that as new facts get computed, others become eligible for discarding. We now look at how these techniques (for synchronizing evaluation and for ensuring Conditions D and U for parts of a program) can be combined to obtain a space optimization technique for the full program. We first

discuss which techniques are orthogonal to each other (and hence, any applicable combination can be chosen) and which techniques are not (and hence, cannot be combined arbitrarily).

8.1 Orthogonality of Techniques

The first point to note is that synchronization techniques are *not* orthogonal to the various techniques for ensuring Conditions U and D—some techniques for ensuring U and D may be applicable only with certain synchronization techniques. For instance, Nested-SCC synchronization sets up subgoals when some facts are needed in a rule application; when the answers are computed (in a nested fashion) and used in the rule application, they automatically satisfy Condition U with respect to this predicate occurrence. This technique for satisfying Condition U, however, cannot be used with other synchronization techniques. Further, since synchronization techniques determine which predicates can be treated as base (with respect to a rule or predicate) in an evaluation, they could affect the applicability of techniques (to ensure U and D) that depend on which predicates are base and which derived. This suggests that techniques for ensuring U and D for a subprogram be chosen after choosing a synchronization technique (for that subprogram).

The second point to note is that, given a synchronization strategy for a subprogram, the techniques for ensuring U and D (for those subprogram facts) may be chosen independently. Note, however, that the applicability of techniques to ensure U for subprogram facts may depend on ensuring D for (possibly other) subprogram facts though it does not depend on which techniques are used for this purpose. Also, different techniques may be used to ensure U (or D) for different body predicate occurrences (resp. rules). If, however, several techniques for ensuring U, for instance, are applicable to each body predicate occurrence, the choice one makes in practice may depend on the relative "efficiency" of each of the techniques as well as the overheads incurred by the use of separate techniques. How to make this choice is a topic for future research.

In Section 8.2 we describe an algorithm that chooses synchronization techniques as well as techniques to ensure Conditions D and U, using some heuristics, to obtain a space optimization technique for the full program.

8.2 Automatically Combining Techniques

In the following discussion we assume that we are given a program-query pair $< P, Q >$, and wish to obtain a space optimization technique for the Magic rewritten form P^{mg} of the program. (We also assume that no rewriting is done on the program subsequent to Magic.) This assumption helps in presenting the algorithm concisely, but is not essential for the use of space optimization techniques on the program. (Also, we expect some variant of Magic rewriting to be used quite extensively in query optimization.) Algorithm Discard outlines the order in which the synchronization techniques and techniques to ensure U and D are chosen. The heuristics used to guide these choices are discussed subsequently.

For the purpose of this section, a program is treated as a set of *units*, $\{U_1, \ldots, U_n\}$, where each unit contains a set of predicates as well as a set of rules. The units are a partitioning of the predicates and rules of the program in that each predicate and each rule is contained in exactly one unit. Note that if a predicate p_i is contained in a unit U_j, not all the rules in the program defining p_i need be contained in U_j. We define the *unit graph* as follows: the units U_i form the nodes of the graph, and there is an edge from U_i to U_j iff some predicate contained in U_i is used in the body of a rule contained in U_j. The unit

graph is a directed graph, and its edges may be given labels. We require that all unit partitions used in this section are such that the unit graph is a DAG. An example of such a unit partition is one where each unit contains a maximal set of mutually recursive predicates, along with the set of rules defining the predicates.

The edges of this graph may be given labels from the following set: *Sequential, Nested, Interleaved, Inverted.* The edge labels specify how the program must be evaluated. If there is a Sequential edge from unit 1 to unit 2, then unit 1 must be evaluated before the evaluation of unit 2 is begun. The meaning of the other edges is similarly defined.

Algorithm Discard (P, P^{mg})

(1) Create_Nested_Units.
(2) Create_Nested_Edges.
(3) Create_Interleaved_Edges.
(4) Create_Sequential_Edges_and_Cleanup.
(5) Create_Nested_Sub-units.
(6) Decide_Hiding_Facts.
(7) Analyze_UD_Applicability.
(8) Choose_UD_Techniques.
(9) Evaluate_Program.

end Discard

Create_Nested_Units starts with the predicate-SCCs $S_1, \ldots, S_m$ of P^{mg} as the initial units $U_1, \ldots, U_m$. Each of these units U_i contains the predicates of the corresponding predicate-SCC S_i as well as every rule defining each of these predicates. If a unit U_i (of P^{mg}) contains predicates from two predicate-SCCs A_1 and A_2 of P and there is a path from A_1 to A_2 in the predicate-SCC graph of P, U_i is split into two units U_{i1} and U_{i2}. U_{i1} contains the predicates of A_1 as well as the corresponding magic predicates (these were also contained in U_i); U_{i2} contains the rest of the predicates in U_i. U_{i1} contains all the rules (except the external magic rules) from U_i defining the predicates in U_{i1}. The external magic rules defining magic predicates in U_{i1} (these were originally contained in U_i) are now put into the units containing the (modified) rules from which they were derived. The remaining rules in U_i are now put into U_{i2} and U_{i1} is labeled as a Nested-Unit. (Note that the unit graph is still acyclic.) This is repeated until no unit contains predicates from two or more SCCs of P.

Create_Nested_Edges creates edges $< U_i, U_j, Nested >$ if some predicate in U_i is used in a rule in $U_j, i \neq j$ and U_i is labeled as a Nested-Unit. (Thus, all edges out of a Nested-Unit are labeled as Nested.)

Create_Interleaved_Edges checks each unit U that is not a Nested-Unit. If Condition Interleaved-SCCs[14] is satisfied by U and all its consumers, create edges $< U, U_i, Interleaved >$ for each $U_i \neq U$ that has a rule using a predicate in U. If Condition Inverted_Rules is satisfied by U and all its consumers, create a unit U' that contains the inverted rules of U. (Note that U' does not contain any predicates.) Create edges $< U, U', Inverted >$, $< U, U_i, Interleaved >$ for each consumer U_i that is monotonic in the same direction as U and $< U', U_j, Interleaved >$ for each consumer U_j that is monotonic in the opposite direction to U.

Create_Sequential_Edges_and_Cleanup first creates edges $< U_i, U_j, Sequential >$ iff some predicate

[14] Although the various synchronization strategies in this paper were described in terms of SCCs, they are also applicable to units.

contained in U_i is used in a rule in U_j, and there is no other edge between U_i and U_j. After the creation of such Sequential edges, we could have the following situation: for a pair of units U_i, U_j one path of edges in the edge graph from U_i to U_j indicates that U_i must be evaluated before U_j and another path from U_i to U_j indicates that they must be evaluated in an interleaved fashion. This can happen if there is a Sequential or Inverted edge in the first path and there is an Interleaved edge in the second path and all other edges in the second path are Interleaved or Nested edges. In such situations, Interleaving is unlikely to be very helpful and, as a heuristic, we transform this situation by relabeling some of the Interleaved edges as Sequential edges. Similar situations can arise in other ways and the Cleanup procedure relabels edges to avoid such situations. We omit tedious details of the procedure.

Create_Nested_Sub-units analyzes Nested-Units (these contain both non-magic, say p, and the corresponding magic, m_p, predicates) U_i to obtain a synchronization technique that can be used to evaluate U_i during each "call." If within U_i, the rules are such that p and m_p are not mutually recursive to each other, then the rules for m_p need not be evaluated along with the rules for p. In such a case, two sub-units U_{i1} and U_{i2} are created within U_i; U_{i1} contains the magic predicates (and corresponding rules) and U_{i2} contains the other predicates (and corresponding rules) from U_i. These two sub-units can now be analyzed for the applicability of Condition Inverted_Rules (Condition Interleaved-SCCs will not be satisfied)—if this is satisfied, a sub-unit containing inverted rules is created and appropriate edges added between the sub-units (as is done by Create_Interleaved_Edges). If Inverted_Rules is not applicable a Sequential edge is created from U_{i1} to U_{i2}.

Decide_Hiding_Facts analyzes each unit (and sub-unit) U_i for the applicability of delaying the first use of facts during an evaluation of U_i, based on Condition Hiding_Facts.

Analyze_UD_Applicability checks the applicability of all the techniques for ensuring U for each body predicate occurrence and for ensuring D for each rule in the resultant program. Note that the various techniques for ensuring U and D are independent of each other.

Choose_UD_Techniques examines the set of applicable techniques for ensuring Conditions U and D and makes suitable (heuristic) choices based on their relative "efficiency" and the overheads incurred. For instance, if a Nested-Unit that has no incoming edges and no non-trivial technique for satisfying Condition D is applicable, we may decide not to satisfy Condition D and use Nested-SCC Discarding—the improvement in space requirements will hopefully outweigh the cost of recomputation. (Note that in this case, not satisfying Condition D for this unit does not adversely affect ensuring U for facts in other units.) For other Nested-Units if none of the non-trivial techniques for satisfying Condition D are applicable, we may decide to use the (trivial) technique of not discarding any facts for predicates in that unit to satisfy D.

Evaluate_Program takes the resultant set of units and evaluates the program based on the edge labels between units. (The Cleanup operation ensured that such an evaluation can be done consistently.) This results in a space optimization technique for the full program.

Theorem 8.1 *Evaluation of a safely evaluable program using Algorithm* Discard *is sound and complete.*

Proof: (Sketch) Were no facts to be discarded in the evaluation, the correctness of each of the synchronization techniques ensures the soundness and completeness of the evaluation using Algorithm Discard. We now show that discarding facts in such an evaluation does not affect the soundness or completeness; if Condition D were satisfied by each unit in the evaluation, the soundness and completeness would follow from Corollary 3.1 and the correctness of each technique used to ensure Conditions D and U. Now, the only technique that does not ensure D is the Nested-SCC Discarding technique; since this is used only for units that do not depend on other units, none of the techniques used to ensure U or D for

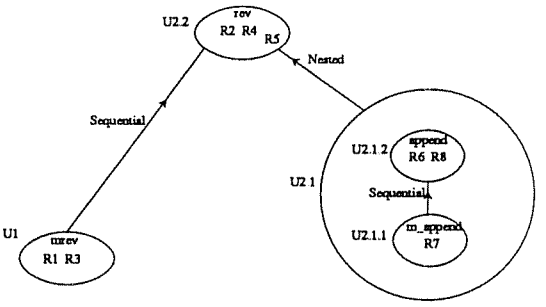

Figure 1: Unit Structure for P^{mg}_{Rev}

facts in other units are adversely affected. Since no derivations are repeated during a single call to these Nested-Units, techniques used to ensure U or D for facts in such units are also not adversely affected. □

We next describe how Algorithm Discard can be used to obtain space optimization techniques for some example programs.

8.3 Obtaining a Space Optimization Technique for Example Programs

We first describe how a space optimization technique can be obtained for P^{mg}_{Rev}, the magic program used to compute the reverse of a list. We next do the same for P^{mg}_{MFL}, the magic program used to compute a variation on the list of factorials.

Example 8.1 The following magic program P^{mg}_{Rev} can be used to compute the reverse of a list. The original program P_{Rev} can be obtained from rules R2, R4, R6 and R8 by deleting all occurrences of magic predicates from the rule bodies.

$R1 : m\_rev([\ldots]).$
$R2 : rev(nil, nil) \leftarrow m\_rev(nil).$
$R3 : m\_rev(T) \leftarrow m\_rev(H \mid T).$
$R4 : rev(H \mid T, T1) \leftarrow m\_rev(H \mid T), rev(T, T2), append(T2, H \mid nil, T1).$
$R5 : m\_append(T2, H \mid nil) \leftarrow m\_rev(H \mid T), rev(T, T2).$
$R6 : append(nil, L, L) \leftarrow m\_append(nil, L).$
$R7 : m\_append(T, L) \leftarrow m\_append(H \mid T, L).$
$R8 : append(H \mid T, L, H \mid L1) \leftarrow m\_append(H \mid T, L), append(T, L, L1).$

Choosing Synchronization techniques:

Figure 1 shows the unit structure (and edge labels) obtained using Algorithm Discard. We describe below how this labeled unit graph is obtained. There are two SCCs in P^{mg}_{Rev}, U_1 (containing the predicate *m_rev* and rules R1 and R3) and U_2 (containing the predicates *rev*, *append* and *m_append* and the other rules of the program). Since U_2 contains predicates (*rev* and *append*) from two SCCs of P_{Rev}, it is split into two units $U_{2.1}$ (containing *append*, *m_append* and rules R6, R7 and R8) and $U_{2.2}$ (containing *rev* and rules R2, R4 and R5). $U_{2.1}$ is labeled as a Nested-Unit, and an edge $< U_{2.1}, U_{2.2}, Nested >$ is created. (This results in *append* becoming base with respect to rule R4.) Since the rules in U_1 are not invertible (though the other requirements for Condition Inverted_Rules are satisfied for U_1 and $U_{2.2}$), an edge $< U_1, U_{2.2}, Sequential >$ is created. (This results in *m_rev* becoming base with respect to the rules in $U_{2.2}$.) Since $U_{2.1}$ is a Nested-Unit and *append* and *m_append* are not mutually recursive to each

other in $U_{2.1}$, two sub-units $U_{2.1.1}$ (containing *m_append* and rule R7) and $U_{2.1.2}$ (containing *append* and rules R6 and R8) are created within $U_{2.1}$. However, since the rule in $U_{2.1.1}$ is not invertible, an edge $< U_{2.1.1}, U_{2.1.2}, Sequential >$ is created. (This results in *m_append* becoming base with respect to the rules in $U_{2.1.2}$.)

Though Condition Hiding_Facts is satisfied by each of the units $U_1, U_{2.2}, U_{2.1.1}$ and $U_{2.1.2}$, every rule in each of these units is "linear." As a consequence, delaying first use of facts is not useful in any of the units in this program.

Choice of techniques for satisfying Condition D:

The rules in U_1 and $U_{2.2}$ satisfy Condition Monotonicity_D as well as DF1. Since $U_{2.1}$ is a Nested-Unit that has no incoming edges and the Nested-SCC Cutset technique is not applicable, we decide not to satisfy Condition D for $U_{2.1}$ and Nested-SCC Discarding is chosen for this unit. (Unless the list to be reversed has no duplicate-elements, Condition D will not be satisfied for *append* and *m_append* facts if Nested-SCC Discarding is used.) However, D can be satisfied during each evaluation of $U_{2.1}$ since the rules in $U_{2.1.1}$ and $U_{2.1.2}$ are monotonic as well as duplicate-free. Since using DF1 involves no overheads (whereas using monotonicity involves maintaining some indices), it is the strategy of choice.

Choice of techniques for satisfying Condition U:

Either of BU1, BU2 or monotonicity may be used for ensuring U for the occurrences of *m_rev* in the rules in U_1 and $U_{2.2}$. Either of BU1 or monotonicity may be used for ensuring U for the occurrences of *rev* in the rules in $U_{2.2}$. Either of BU1, BU2 or monotonicity may be used for ensuring U for occurrences of *m_append* in the rules in $U_{2.1.1}$ and $U_{2.1.2}$. Since Nested-SCC synchronization is used, U is automatically satisfied for *append* in the body of rule R4 (in $U_{2.2}$); BU1 or monotonicity may be used for the occurrence of *append* in the body of rule R8 (in $U_{2.1.2}$). Monotonicity (having fewer overheads than BU1) is the strategy of choice in this case.

Evaluation:

Given the choice of synchronization techniques and techniques to ensure Conditions U and D for the various parts of the program, P^{mg}_{Rev} can be evaluated as follows. First U_1 is evaluated; no facts are discarded during this evaluation. *m_rev* and *rev* facts are discarded during the evaluation of $U_{2.2}$. $U_{2.1}$ is evaluated using Nested-SCC Discarding with respect to $U_{2.2}$. During a call to $U_{2.1}$, first $U_{2.1.1}$ is evaluated; no facts for *m_append* are discarded. This is followed by the evaluation of $U_{2.1.2}$; during this evaluation *append* facts (not matching the external subgoal) and *m_append* facts are discarded. The *append* fact computed in response to an external subgoal (by computing the closure of $U_{2.1}$) is used in rule R4, and then discarded.

Improvements in Space Complexity:

Using this space optimization technique, we reduce the asymptotic number of facts stored from $O(n^2)$ (in case no discarding of facts is done) to $O(n)$. However, since each fact could have a list of length n; hence, the total space utilized is $O(n^2)$ (without discarding facts, the space complexity was $O(n^3)$). If facts can share parts of structures with other facts we can reduce the total space utilized to $O(n)$. Note that if the space optimization techniques described above were not used, sharing of parts of structures alone would result in a space complexity of $O(n^2)$.

Since Condition D is not satisfied in general by *append* and *m_append* facts (due to the use of Nested-SCC Discarding for $U_{2.1}$ with respect to $U_{2.2}$), some of these facts could be recomputed. However, it can be easily verified (manually) that the worst case time complexity of the program is not affected—in

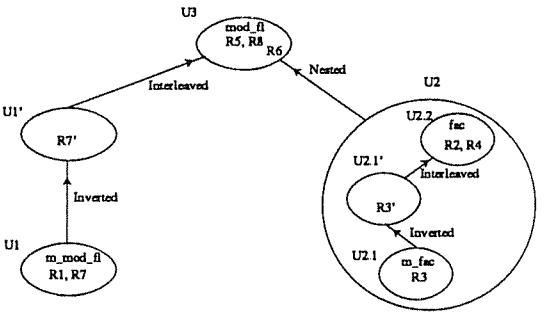

Figure 2: Unit Structure for P^{mg}_{MFL}

some cases the actual time needed to evaluate the program may increase (as when the list has duplicate elements), but in most cases the increased time needed to evaluate the program is not considerable. □

Note that using P^{mg}_{Rev} for computing the reverse of a list is given here purely for pedagogic purposes. There is a more efficient program to compute the reverse of a list that can be evaluated storing just a linear number of facts (without using any space optimization technique), which can be found in any standard text on logic programming.

Example 8.2 The following magic rewritten program P^{mg}_{MFL} produces a list of factorials, such that $mod\_fl(n, Y)$ is true for $Y = [fac(fac \ldots fac(3) \ldots), \ldots, fac(fac(3)), fac(3), 3]$, where the list contains $n+1$ elements. (The original program P_{MFL} can be reconstructed if desired by deleting the occurrences of magic predicates from the bodies of rules R2, R4, R5 and R8, and discarding other rules.)

$R1 : m\_mod\_fl(n).$
$R2 : fac(0, 1) \leftarrow m\_fac(0).$
$R3 : m\_fac(N-1) \leftarrow m\_fac(N), N > 0.$
$R4 : fac(N, N * X) \leftarrow m\_fac(N), N > 0, fac(N-1, X).$
$R5 : mod\_fl(0, [3]) \leftarrow m\_mod\_fl(0).$
$R6 : m\_fac(H) \leftarrow m\_mod\_fl(I), I > 0, mod\_fl(I-1, [H|X]).$
$R7 : m\_mod\_fl(I-1) \leftarrow m\_mod\_fl(I), I > 0.$
$R8 : mod\_fl(I, [N|H|X]) \leftarrow I > 0, mod\_fl(I-1, [H|X]), fac(H, N).$

The unit structure (and edge labels) obtained using Algorithm Discard is indicated in Figure 2. Each unit in the figure also indicates the predicates and rules it contains. The inverted rules corresponding to the rules in U_1 (rule R7') and the rules in $U_{2.1}$ (rule R3') are given below:

$R3' : m\_fac(N) \leftarrow m\_fac(N-1), N > 0.$
$R7' : m\_mod\_fl(I) \leftarrow m\_mod\_fl(I-1), I > 0, I \leq n.$

Every rule set in each of the units satisfies DF1; this is the strategy used for ensuring D, though monotonicity could also be used. The Nested-SCC Cutset technique is used to evaluate U_2 with respect to U_3, thus satisfying Condition D for U_2 as well—the cutset that is retained after computing the closure of U_2 (in response to an external subgoal $m\_fac(M)$ generated by R6) is the external subgoal $m\_fac(M)$ and the answer to that subgoal $fac(M, \_)$.

Monotonicity can be used to ensure U for each of the predicate occurrences in the program (except for the occurrence of fac in R8, where the use of Nested-SCC Synchronizing ensures U), though BU1 is also applicable.

The evaluation of P^{mg}_{MFL} proceeds as follows. U_1 is evaluated and all non-fringe magic facts are appropriately discarded. Then U'_1 and U_3 are evaluated using Interleaved-SCC synchronization; *m_mod_fl* and *mod_fl* facts are discarded using the techniques mentioned earlier for ensuring the satisfaction of Conditions D and U. The Nested-SCC Cutset technique is used to evaluate U_2 with respect to U_3. Each call to U_2 is evaluated as follows: $U_{2.1}$ is evaluated using the external *m_fac* magic fact as a seed; all non-fringe *m_fac* facts are discarded (except the external sub-goal) based on satisfying U and D to the rules in $U_{2.1}$. Then $U'_{2.1}$ and $U_{2.2}$ are evaluated using Interleaved-SCC synchronization in a similar fashion to the evaluation of U'_1 and U_3, discarding *m_fac* and *fac* facts suitably. The answer $fac(M, \_)$ to an external subgoal $m\_fac(M)$ is retained along with the corresponding magic fact even after it has been used in the occurrence of *fac* in R8 since this forms a cutset—all subsequent external subgoals $m\_fac(N)$ will have $N > M$ and the corresponding answers can be computed without using $m\_fac(N1)$ and $fac(N1, \_)$ facts for any $N1 < M$.

Using this space optimization technique, the evaluation can be done storing just a constant number of facts. □

9 Overheads

There are three aspects to the overheads involved with these techniques.

- Compile-time time overheads.
  Suppose we are given (a) dependency information about all predicates in the program, (b) duplicate-freedom information, (c) ϕ functions for all predicates in the program, and (d) γ functions for different predicates as necessary. Then the cost of testing various conditions is linear in the size of the input. We have indicated briefly how to derive some of the functions, and we expect our algorithms to be efficient in practice.

- Run-time time overheads.
  These overheads are minimal for tests based on bounds—in some cases there is no overhead for any tests, and at most, in other cases, a few simple counts need to be maintained for each fact, and updated when the fact is used. Tests based on monotonicity are a little more complicated. When a fact is derived we need to compute its ϕ value, and possibly its value under some of the γ functions. This computation is constant time per fact and quite efficient. The only important cost here is the cost of secondary indices on the ϕ value so that facts can be discarded when index m (from Theorem 5.1) reaches a certain value.

- Run-time space overheads.
  For bounds based techniques, there is no overhead in some cases, and a constant overhead of one to a few integers per stored fact in other cases. For monotonicity based techniques, we can choose to either store various function values for each stored fact, or recompute them on demand and thus avoid all space overheads. There is at most a constant space overhead per stored fact, even if we decide to store the function values. When the number of facts stored is reduced by an order of magnitude, a constant space overhead per stored fact is clearly negligible.

10 Conclusion

In this paper we have described how to reduce the space required during bottom-up evaluation of logic programs by discarding facts. We showed that any space optimization technique that discards facts during the evaluation has three basic components: (1) ensuring that all derivations are made, (2) ensuring that derivations are not repeated, and (3) synchronizing the derivation and use of facts. We presented some techniques for ensuring each of these three components, and showed how they can be combined to get a space optimization technique for the full logic program. Since Sliding Window Tabulation [NR90] can be shown to be just one way of combining techniques for each of these three components, our results subsume those in [NR90].

We presented a variety of techniques to ensure Conditions D and U. These are of course not exhaustive, and other useful techniques may be discovered, such as the one mentioned in Example 4.2.

Incorporating these optimizations into an actual deductive database runtime system, and enhancing the compiler/interpreter to test for and use these optimizations is being considered for CORAL, a deductive database system being developed at the University of Wisconsin, Madison.

Future work in this area includes finding more methods for ensuring each of the three components of an effective space optimization technique. For instance, the generate and test paradigm could benefit from a form of synchronization where facts are generated and tested in a synchronized fashion, and may be discarded once they have been tested. Work is also needed in determining which technique to use when more than one technique is applicable to a given part of the program.

References

[Ban85] Francois 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.

[BR87] I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *Journal of Logic Programming*, 4(3), September 1987.

[Hir75] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, June 1975.

[KRS88] Ravi Krishnamurthy, Raghu Ramakrishnan, and Oded Shmueli. A framework for testing safety and effective computability of extended datalog. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 154–163, Chicago, Illinois, May 1988.

[MR90] Michael J. Maher and Raghu Ramakrishnan. Dèjá vu in fixpoints of logic programs. In *Proceedings of the Symposium on Logic Programming*, Cleveland, Ohio, 1990.

[NR89] Jeffrey F. Naughton and Raghu Ramakrishnan. A unified approach to logic program evaluation. Technical Report 889, Computer Sciences Department, University of Wisconsin, Madison, November 1989.

[NR90] Jeffrey F. Naughton and Raghu Ramakrishnan. How to forget the past without repeating it. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, August 1990.

[Ram88] Raghu 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.

[RSS90] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. Rule ordering in bottom-up fixpoint evaluation of logic programs. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, August 1990.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, 1989.