OPTIMIZING BOTTOM-UP QUERY EVALUATION FOR DEDUCTIVE DATABASES

By

SUNDARARAJARAO SUDARSHAN

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN – MADISON

1992

Abstract

Deductive databases extend the power of traditional database query languages such as SQL by allowing recursive definitions of predicates. Bottom-up query evaluation is an important query evaluation mechanism for deductive databases and logic programs. In recent years, deductive databases have been extended by allowing facts to contain complex terms that can possibly include variables, and by allowing the use of aggregate operations on sets of answers. This thesis addresses optimization issues related to these extensions.

In the first part of the thesis we compare bottom-up and Prolog query evaluation. We show that using existing techniques, bottom-up evaluation performs no more "actions" than (a model of) Prolog for a restricted class of programs, but this does not hold for all programs. We develop rewrite-based optimization techniques that help us extend the above results to all logic programs. We then develop novel techniques for evaluating these rewritten programs. We compare bottom-up query evaluation (using our rewrite optimizations along with our evaluation optimization) with Prolog query evaluation, and show the following. Suppose we are given a program; if (our model of) Prolog evaluation of a query takes time t on a database, bottom-up query evaluation on the database, without subsumption checking, takes time $O(t \cdot \log\log t)$. For a restricted class of programs, bottom-up query evaluation on the database, with subsumption checking, takes time at worst $O(t)$. (In both cases, the time taken by bottom-up evaluation also depends on the size of the program, which we assume to be small). On the other hand, for many programs, Prolog is arbitrarily slower than bottom-up evaluation. Our optimization techniques are of importance in evaluating programs that generate facts containing variables.

In the second part of the thesis, we develop optimizations related to the use of aggregate operations such as *min* or *max*. We show how to view several such operations as "selections", and how to propagate these selections into programs. We demonstrate the power and utility of the optimization techniques, using programs for problems such as computing shortest paths and critical paths.

Contents

Chapter 1

Introduction

Deductive databases extend the power of traditional databases by allowing derived relations (views) to be defined recursively using logic programs. In the area of query evaluation for logic programs, the de facto standard is Prolog, which is a top-down evaluation strategy. Bottom-up query evaluation has advantages over Prolog with respect to completeness[1] and IO costs [BR86, Ull89b]. (In our model, bottom-up query evaluation consists of Magic rewriting of the program and query [BMSU86, BR87b, Ram88] (see Section 2.2.1) followed by fixpoint evaluation of the rewritten program [BR87a, Ban85] (see Section 2.2.3).)

In recent years, deductive databases have been extended by allowing facts to contain complex terms that can possibly include variables, and by allowing the use of aggregate operations on sets of answers. This thesis addresses optimization issues related to these extensions.

This thesis has two main parts. In the first part, we consider the question of time-complexity of bottom-up evaluation vs. Prolog evaluation for logic programs that can generate complex terms that may contain variables. Motivated by this comparison, we develop several rewrite-based optimization techniques for bottom-up evaluation. We summarize the contributions of this part of the thesis in Section 1.2. In the second part of the thesis, we present optimization techniques for an extension of logic programs that allows the use of aggregate operations on sets of facts. We summarize the contributions of this part of the thesis in Section 1.3.

1.1 Memoing vs. Non-Memoing Query Evaluation Techniques

A *memoing* evaluation technique for definite clause programs is one that stores subgoals and answers that are generated during the evaluation. Bottom-up query evaluation is an example of a memoing evaluation technique. The term *top-down evaluation* is used for evaluation techniques based on SLD resolution and its variants (e.g. SLDNF, SLD-AL, OLDT, etc. — see, e.g., [Llo87, War92]).[2] There are a number of memoing top-down evaluation techniques such as the Query-Subquery (QSQ) approach and its extensions

[1]Completeness of evaluation implies that given any answer to the query, there is a finite point of time at which evaluation generates the answer; there may be an infinite number of answers, and evaluation may not terminate. Bottom-up evaluation is complete for (finite) definite clause programs with a finite database.

[2]This definition is not very precise. Bottom-up evaluation using Magic rewriting can be viewed as a compiled form of OLDT resolution, although there are some differences. However, the terms bottom-up and top-down have been used historically to refer to these two categories of evaluation techniques.

[Vie86, Vie88], and Extension Tables [Die87]. SLD-AL resolution, and OLDT resolution are theoretical models of top-down evaluation techniques that perform memoing of facts.

The de facto standard for evaluating queries on logic programs is Prolog, and Prolog does not perform memoing as part of the built-in evaluation mechanism. (However, *ad hoc* use of memoing is common in programs written in Prolog.)

Natural questions that arise are (1) "how do memoing evaluation techniques compare with non-memoing evaluation techniques?", and (2) "how do bottom-up evaluation techniques compare with top-down evaluation techniques?", To make comparison (1) precise, we have to talk of a specific memoing evaluation technique and a specific non-memoing evaluation technique. To make (2) precise, we have to talk of a specific top-down evaluation technique, and a specific model for bottom-up evaluation.

Initial comparisons of bottom-up evaluation and memoing top-down evaluation techniques were based on the number of distinct facts derived by the different techniques. Thus, Ramakrishnan [Ram88, Ram90] presents a class of evaluations and shows that within this class bottom-up fixpoint evaluation of a program rewritten using Magic Templates computes an optimal number of facts. Seki [Sek89] presents a direct comparison between the set of facts computed using Alexander Templates rewriting, and using SLD-AL resolution. Bry [Bry90] shows that several top-down and bottom-up evaluation techniques can be viewed as specializations of a technique called the Backward Fixpoint Procedure; all these techniques essentially compute the same set of facts and generate the same subgoals. These results ignore the number of times facts are generated, ignore the actual time cost of evaluation, and ignore optimizations such as tail-recursion optimization (Section 2.3) that are routinely performed by Prolog systems.

There is a considerable amount of similarity between memoing top-down evaluation techniques and bottom-up evaluation; we do not explore the differences in this thesis. We concentrate instead on the differences between memoing and non-memoing evaluation techniques. We use bottom-up evaluation as the canonical memoing evaluation technique in this thesis. We also use Prolog as the canonical non-memoing evaluation technique.

Bottom-up query evaluation using Magic rewriting (as also several of the memoing top-down evaluation techniques mentioned above) has three significant advantages over non-memoing techniques such as Prolog: (1) Bottom-up evaluation using Magic Templates rewriting is complete for definite clause programs, and the declarative least Herbrand model semantics is always enumerated for definite clause programs. (2) Redundant derivations are avoided through memoing, leading to significant improvements in time complexity for programs in which goals or facts can be derived in many ways. (3) As a consequence of (1), no operational guarantees need be made, thereby making possible a number of semantic optimizations. The reader is referred to [RSS92c] for a brief survey of several such semantic optimization techniques.

On the other hand, some operations may be cheaper if facts are not memoed. Therefore, it is important to perform a comparison of bottom-up and non-memoing top-down evaluation techniques in terms of the cost of evaluation.

Ullman ([Ull89a, Ull89b]) has compared bottom-up evaluation with top-down evaluation for the class of range-restricted Datalog programs (programs that generate only ground facts (i.e., facts that do not contain variables) and do not use function symbols). His results show that bottom-up evaluation using Magic Sets along with rectification (MSR) rewriting (BU-MSR evaluation for short) has time complexity (i.e., ignoring constant costs) less than or equal to Queue-based Rule Goal Tree (QRGT) evaluation (a top-down query evaluation strategy).

There are several limitations to Ullman's result. First, the comparison is only for range-restricted Datalog. Some of the assumptions made in the comparison do not hold if *non-ground facts* (i.e., facts that contain variables) are generated. Second, Ullman's comparison ignores optimizations that are routinely performed on Prolog programs such as tail-recursion optimization (Section 2.3). Third, the comparison is with respect to a particular top-down evaluation technique, and does not extend to Prolog, which is the de facto standard for evaluating logic programs. Fourth, the comparisons assume that all answers are required, and do not provide insight for the case that only one answer is required (although there is no change in the worst case comparison).

In this thesis, we address the first three problems above. We compare bottom-up evaluation with Prolog evaluation, in terms of time complexity of query evaluation, for the class of all definite clause programs (which can possibly generate non-ground facts).

The fourth problem, namely the case that only one answer is required, is harder. In particular, the depth-first search strategy used by Prolog has advantages over the breadth-first search strategy used by bottom-up evaluation with Magic rewriting in some contexts where not all answers to a query are desired. There have been some attempts to provide the benefits of depth-first search in the context of bottom-up evaluation; we mention these, and discuss open problems in Chapter 7.

1.2 Bottom-Up vs. Prolog

An important question in the area of logic programming and deductive databases is "How does bottom-up query evaluation compare with Prolog query evaluation in terms of time complexity?".

There are programs for which bottom-up query evaluation is considerably faster than Prolog. As an example, consider the *path* program with a query, shown below. We assume we are given a finite set of facts for *edge*, although we do not show them below. On this program Prolog loops for ever if the *edge* relation has a cycle, whereas bottom-up evaluation terminates, generating all answers.

$path(X,Y) :- edge(X,Y).$
$path(X,Y) :- edge(X,Z), path(Z,Y).$
Query: ?-$path(X,Y)$.

However, there are programs for which the time complexity of Prolog evaluation is considerably less than that of bottom-up evaluation using current techniques. In the first part of the thesis, we discuss the reasons for the inefficiency, and present optimization techniques that help us show that bottom-up evaluation can be made almost as fast as Prolog evaluation, in the sense of time complexity, over all programs. (The work in this thesis, like that of Ullman [Ull89a], ignores IO costs, and assumes that all answers are generated.)

For the purpose of comparison, we use a model of Prolog evaluation that we call Prolog* evaluation; we believe that this model reflects current Prolog implementations fairly accurately.[3]

[3]Our model of Prolog evaluation incorporates tail-recursion optimization, but assumes that intelligent backtracking (see, e.g., [CD85]) is not used. It also assumes that all answers are computed.

A close look shows several problems in making bottom-up evaluation comparable to or better than Prolog for all programs. Magic Templates rewriting [Ram88] and Alexander Templates [Sek89] are the main bottom-up evaluation techniques that deal with general logic programs. (Ullman's MSR rewriting does not deal with general logic programs, which can generate facts containing complex terms built from function symbols, constants and variables.) Let us denote bottom-up query evaluation using Magic Templates (resp. Alexander Templates) as BU-MT evaluation (resp. BU-AT evaluation). There are three problems with both the above evaluation techniques when non-ground facts are generated.

1. Both techniques can make considerably more inferences than Prolog, even for Datalog programs, even ignoring the effect of optimization such as tail-recursion optimization. The basic problem was noted by Codish, Dams and Yardeni [CDY90], but is not widely recognized. Consider the following program

   $R1: q :- p(a), p(X), r(X).$
   $R2: p(X).$
   Query: ?-q.

   On this program, the only subgoal generated for the predicate r by Prolog evaluation is $?r(X)$.[4] Bottom-up evaluation using Magic Templates rewriting generates an answer fact $p(a)$, and uses this with literal $p(X)$ to generate a query $?r(a)$. A fact $p(X)$ is generated later, and $p(a)$ is found to be subsumed, but the query $?r(a)$ is generated before the subsumption is detected. (The query $?r(X)$ is generated after the answer $p(X)$ is generated.) Thus bottom-up evaluation can generate subgoals (and generate corresponding answers) that Prolog evaluation avoids.

   We extend this observation in Example 3.1.2, and illustrate how it can lead to BU-MT query evaluation performing asymptotically worse than Prolog. We formalize the problem through the definition of mgu-subgoals and mgu-answers (Section 3.1); the problem is that Magic Templates rewriting can generate answers (and subgoals) that are not mgu-answers (resp. mgu-subgoals).

   Our contribution in this respect is as follows:

   - We refine Magic Templates rewriting to avoid the problems noted by Codish et al.; we call this refinement MGU Magic Templates (MGU MT) rewriting. Bottom-up query evaluation using MGU MT rewriting generates only mgu-subgoals and mgu-answers. This refinement is described in Section 3.3.

2. Prolog performs tail-recursion optimization, which we describe in Section 2.3. Even for safe Datalog programs, tail-recursion optimization can reduce the number of inferences made by Prolog evaluation to much less than the inferences made by Ullman's BU-MSR evaluation technique, or by BU-MT evaluation. Example 2.3.1 illustrates this problem. Ross [Ros91] presents a variant of Magic Templates to incorporate tail-recursion optimization (we call Ross' rewriting MTTR rewriting).

   MTTR rewriting suffers from the same problems with subsumed answers (described above) as do Magic Templates and Alexander Templates rewriting. Our contributions in this context are as follows:

[4]To keep the example simple, we do not have any rules defining r, and hence the subgoal fails.

- We use the ideas behind MGU MT rewriting to refine Ross' MTTR rewriting; we call this refinement MGU MTTR rewriting. MGU MTTR rewriting is described in Section 3.4. This refinement is important since it enables us to account for tail-recursion optimization while also dealing with the problem of using non-mgu-answers in derivations.
- We show (in Section 4.3) that bottom-up evaluation using MGU MTTR rewriting performs no more "actions" than a small constant number of times the number of "actions" performed by Prolog* evaluation of the query on the program. In many cases bottom-up evaluation performs far fewer actions than the number of actions performed by Prolog* evaluation.

3. The cost per inference in bottom-up evaluation can be more than for Prolog evaluation. For instance, queries on the well-known predicate *append* run on Prolog in time linear in the size of the lists in the query. If the query contains lists with variables, an unoptimized query evaluation using MT rewriting (or any of its variants mentioned above such as MGU MT rewriting or MGU MTTR rewriting) takes time quadratic in the size of the lists (although the number of inferences does not change). The basic reason is that bottom-up evaluation of the Magic rewritten program performs some unifications that Prolog evaluation does not perform, when answers are returned for a query. We call such unification *answer-return* unifications. Unification is in general linear in the size of the terms to be unified, and can be costly for large non-ground terms.[5] This is discussed in Example 5.1.1.

   It is important that bottom-up evaluation of programs that generate non-ground facts be done efficiently. Non-ground data-structures such as difference lists (Example 5.1.1) are important in some applications, and support some operations (such as list append) more efficiently than ground data-structures in the context of Prolog. Many applications that benefit from bottom-up evaluation would also benefit from the use of non-ground data-structures, if bottom-up evaluation using non-ground facts can be done efficiently. Example 5.9.2 shows a shortest-path program that keeps track of the actual path that is computed, and benefits from using a difference list representation. Chart parsing of Definite Clause Grammars is another area where non-ground data-structures and bottom-up evaluation are both useful.

   Our main contribution in this area is as follows:

   - We present (in Chapter 5) a version of bottom-up evaluation that incorporates several optimizations that are applicable to programs that have been rewritten using MGU MTTR (or MGU MT) rewriting. These optimizations are able to reduce the cost of answer-return unifications performed by bottom-up evaluation to nearly a constant per unification. These optimizations are important since we are also able to show the following important result:

     Suppose we are given a logic program and a query. If the time taken by Prolog* to evaluate the query [6] on a given database is t, then evaluating the query using MGU MTTR rewriting and the above mentioned optimizations, on the given database, takes time $O(t \cdot \log\log t)$, provided that we do not check for subsumption.[7] (The size of the program is assumed to be fixed, and is not taken

[5]Unification can be done in constant time for ground terms in certain cases, by using a technique called hash-consing [Got74, SG76]. This requires that all facts generated by the program be ground, and is not applicable to non-ground terms.

[6]Where evaluating the query is interpreted as generating *all* answers to the query.

[7]Recall that Ullman's result, while more limited in several respects, did account for the cost of subsumption checking in bottom-up evaluation. We discuss the issue of subsumption checking later in this section.

into account in the time complexity measure.) Subsumption-checking has a cost, but may also have significant benefits if subgoals are repeated; it can be done where desired. The above result provides an upper bound on how much worse bottom-up evaluation can be compared to Prolog* evaluation. For the other direction, there are programs where Prolog* evaluation is arbitrarily worse than bottom-up evaluation with subsumption-checking.

Equally importantly, our optimization techniques allow efficient evaluation of programs that generate non-ground facts, and must be evaluated with memoing (for instance, the program in Example 5.9.2).

We have also developed an efficient evaluation technique for a restricted class of programs [SR92b]. Using this evaluation technique, we have shown that for a class of programs that properly contains safe Datalog, the time complexity of optimized bottom-up query evaluation with MTTR rewriting is never more than that of Prolog* evaluation, even taking the cost of subsumption checking into account.[8] This result extends those of Ullman [Ull89a] since it handles a larger class of programs. We discuss these results briefly in Section 5.10.

What these results show is that we can optimize bottom-up evaluation so that its time complexity is at worst marginally greater than that of Prolog* evaluation, and at best much better.

There are a few points that must be kept in mind when interpreting these results. First, the results leave open the question of constants. We expect that for purely in-memory implementations, the constant costs will favor Prolog for programs that do not perform duplicate computation, and are not set-oriented. For data-intensive programs as well as programs that repeat computations (such as programs for dynamic programming problems), bottom-up evaluation is likely to beat Prolog evaluation. However, such questions can only be settled by actual optimized implementations. Second, the results assume that all answers to the query are required; the case that only some answers are required is not addressed. (See Chapter 7 for a brief discussion of this case). Third, these results do not incorporate space complexity. However, independent of the time and space costs of evaluation, bottom-up evaluation, even without subsumption-checking, is complete, unlike Prolog — a desirable property in many circumstances.

1.3 Optimizations Related to Aggregate Operations

Database query languages such as SQL provide aggregation operations, that let one compute aggregate values over sets of answers. For example, SQL provides the *group-by* construct that can be used along with a variety of aggregate operations. The use of aggregation with recursive queries has been considered by several researchers (e.g., [BNR+87, MPR90]).

In Chapter 6 we develop an optimization technique for bottom-up evaluation, using a notion of relevance of facts to some aggregate operations such as *min* and *max*. Our notion of relevance can be seen as an extension of the notion of relevance used in optimizations such as Magic sets rewriting [BMSU86, BR87b, Ram88]. One can think of the aggregate operations as providing a form of "selection" on generated facts; we refer to such selections as "aggregate selections".

[8] As before, we assume that all answers are generated by Prolog* evaluation.

The optimization technique consists of two parts — a rewriting technique that "pushes" aggregate selections into rules in the program, and an evaluation technique that makes use of aggregate selections when evaluating the rewritten program. The combined technique is able to detect many facts as irrelevant, and avoids using them to make derivations. As an example of the power of our techniques, we consider a naive program to find shortest paths. The program first computes all paths, and then selects shortest paths. The rewriting technique deduces that for any pair of nodes, any path between them that is not shortest is irrelevant for computing shortest paths. Thus the "optimality principle" is deduced automatically. The evaluation technique when applied to this rewritten program is essentially an extension of Dijkstra's algorithm.

The evaluation techniques developed in this section of the thesis are orthogonal to the optimization techniques developed in the first part of the thesis. We present an example (Example 5.9.2) where both kinds of optimizations are very useful.

1.4 Organization of the Thesis

This thesis is organized as follows. In Chapter 2, we present background material. In Chapters 3, 4, and 5 we develop our main result comparing optimized bottom-up evaluation and Prolog evaluation. In Chapter 3, we present rewriting refinements to Magic Templates rewriting and to MTTR rewriting, to avoid problems due to subsumed answers. In Chapter 4 we present a model of Prolog* evaluation and a model of bottom-up fixpoint evaluation. We then present our results comparing Prolog* evaluation with a fixpoint evaluation of the MGU MTTR rewritten program, at the level of number of inferences. In Chapter 5 we consider the cost of derivations, and present an optimized fixpoint evaluation technique, which we call Opt-NG-SN evaluation. In Section 5.7 we present our results comparing the time cost of query evaluation using Prolog* evaluation with the time cost of Opt-NG-SN evaluation of the MGU MTTR rewritten program. In Chapter 6 we describe our rewriting and evaluation techniques for programs that use aggregate operations.

Chapter 2

Background Material

2.1 Notation and Preliminary Definitions

The language used in this thesis is that of Horn logic (see, e.g., [Llo87]). In this section we present some basic definitions for the convenience of the reader.

2.1.1 First Order Languages

A first-order language has a countably infinite set of variables and countable sets of function and predicate symbols, these sets being mutually disjoint. It is assumed, without loss of generality, that with each function symbol[1] f and each predicate symbol p, is associated a unique natural number n, referred to as the *arity* of the symbol; f and p are then said to be n-ary symbols. A 0-ary function symbol is referred to as a constant. A *term* in a first order language is a variable, a constant, or a compound term $f(t_1, \ldots, t_n)$ where f is an n-ary function symbol and the t_i are terms. A tuple of terms is sometimes denoted simply by the use of an overbar, e.g., $\bar{t}$. Compound terms are also referred to as *structured terms.*

If p is a predicate symbol with arity n, and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is an *atom*, and $\neg p(t_1, \ldots, t_n)$ is the *negation of an atom.* A *literal* is an atom or the negation of an atom. A *positive literal* is an atom, and a *negative literal* is a negation of an atom.

A *simple expression* is either a term or an atom. An *expression* is either a simple expression, a literal, or a disjunction of literals. An expression is said to be *ground* if it contains no variables, and *non-ground* otherwise. A *substitution* θ is a finite set of the form $\{v_1/t_1, \ldots, v_n/t_n\}$, where each v_i is a variable, each t_i is a term distinct from v_i, and the variables $v_1, \ldots, v_n$ are distinct. Each element v_i/t_i is called a *binding* for v_i. θ is called a *ground substitution* if all the t_i are ground. Substitutions are denoted by lower case Greek letters θ, σ, ϕ, etc.

Let $\theta = \{v_1/t_1, \ldots, v_n/t_n\}$ be a substitution, and E an expression. Then $E[\theta]$, the *instance* of E by θ, is the expression obtained from E by simultaneously replacing each occurrence of the variable v_i by the term t_i $(i = 1, \ldots, n)$. If $S = \{E_1, \ldots, E_n\}$ is a finite set of expressions, and θ a substitution, then $S[\theta]$ denotes the set $\{E_1[\theta], \ldots, E_n[\theta]\}$. We sometimes omit the [ ], and write $E[\theta]$ as $E\theta$.

[1]Function symbols are also referred to as *uninterpreted function symbols.*

Let $\theta = \{u_1/s_1, \ldots, u_m/s_m\}$ and $\sigma = \{v_1/t_1, \ldots, v_n/t_n\}$ be substitutions. Then the *composition* $\theta[\sigma]$ of θ and σ is the substitution obtained from the set

$$\{u_1/s_1[\sigma], \ldots, u_m/s_m[\sigma], v_1/t_1, \ldots, v_n/t_n\}$$

by deleting any bindings $u_i/s_i[\sigma]$ for which $u_i = s_i[\sigma]$ and deleting any binding v_j/t_j for which $v_j \in \{u_1, \ldots, u_m\}$.

For example, the composition of substitutions $\{x/a, y/f(Z), t/x\}$ and $\{x/c, r/d\}$ is the substitution $\{x/a, y/f(Z), t/c, r/d\}$.

Let E and F be expressions. We say that E and F are *variants* if there exist substitutions θ and σ such that $E = F[\theta]$ and $F = E[\sigma]$. We also say that E is a variant of F, or F is a variant of E. Let E be an expression, and V be the set of all variables occurring in E. A *renaming substitution* for E is a substitution $\{x_1/y_1, \ldots, x_n/y_n\}$ such that $\{x_1, \ldots, x_n\} \subseteq V$, the y_i are all distinct variables, and

$$(V \backslash \{x_1, \ldots, x_n\}) \cap \{y_1, \ldots, y_n\} = \phi$$

Expressions E and F are variants iff there is a renaming substitution θ for F, such that $E = F[\theta]$.

For example, $\{x/y, y/x, z/w\}$, where x, y, z and w are variables, is a renaming substitution for an expression that does not contain the variable w. But if an expression does contain w, z and w are mapped to w by this substitution. We cannot distinguish between them after applying the substitution, and hence there cannot be an inverse substitution as required in the definition of variants.

A substitution σ is *more general* than a substitution θ if there is a substitution φ such that $\theta = \sigma[\varphi]$. Two simple expressions t_1 and t_2 are said to be *unifiable* if there is a substitution σ such that $t_1[\sigma] = t_2[\sigma]$. σ is said to be a *unifier* of t_1 and t_2. A unifier θ of simple expressions t_1 and t_2 is said to be *a most general unifier* of t_1 and t_2 if, for each unifier σ of t_1 and t_2, there exists a substitution γ such that $\sigma = \theta[\gamma]$. If two simple expressions have a unifier, they have a most general unifier that is unique up to renaming of variables. Given two simple expressions t_1 and t_2, $MGU(t_1, t_2)$ denotes the set of most general unifiers of $t1$ and $t2$; all the elements of this set are equivalent up to renaming. We let $mgu(t1, t2)$ denote an arbitrary element of $MGU(t_1, t_2)$.

For example, given terms $f(x, y)$ and $f(a, g(z))$, where x, y, z are variables, the substitution $\{x/a, y/g(b)\}$ is a unifier, while $\{x/a, y/g(z)\}$ is a most general unifier.

2.1.2 Definite Clause Programs

A *clause* is a formula of the form

$$\forall X_1, \ldots, \forall X_s(L_1 \vee \ldots \vee L_m)$$

where $L_1, \ldots, L_m$ are literals, and $X_1, \ldots, X_s$ are all the variables occurring in $L_1 \vee \ldots \vee L_m$. A *Horn clause* is a clause with at most one positive literal. A Horn clause with exactly one positive literal is referred to as a *definite clause*. Following the syntax of Edinburgh Prolog, definite clauses (usually referred to as *rules*) are written as

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

where p is the positive literal and $\neg q_1, \ldots, \neg q_n$ are the negative literals in the definite clause. Let the variables in the rule be denoted by $\overline{X}$. Then the rule is read declaratively as $\forall \overline{X}(q_1 \wedge q_2 \wedge \ldots \wedge q_n \rightarrow p)$. The positive literal in a definite clause is its *head*, and the remaining literals, if any, constitute its *body*.[2] The notation

$$R : p\text{:} - q_1, \ldots, q_n.$$

denotes a rule with a name R. We use the name to refer to the rule. A *fact* is a rule with empty body.

A predicate definition consists of a set of definite clauses, whose heads all have the same predicate symbol. A *definite clause program* is a finite set of definite clauses. A *goal* is a negative literal, and is usually written as $?p(t_1, \ldots, t_n)$.[3] We also refer to a goal as a *subgoal* or a *query*.

We use the convention that names of variables begin with upper case letters, while names of non-variable (i.e., function and predicate) symbols begin with lower case letters. We use the following special notation for lists. The *empty list* is a constant symbol [ ]. A *list* is either an empty list, or $cons(h, t)$ where h and t are terms. We use the special notation $[h|t]$ to denote $cons(h, t)$. We refer to h as the *head* of the list and t as the *tail* of the list. We use the notation $[h_1, h_2, \ldots, h_n|t]$ to denote the list $[h_1|[h_2| \ldots [h_{n-1}|[h_n|t]] \ldots]$. Further, $[h_1, h_2, \ldots, h_n]$ denotes $[h_1, h_2, \ldots, h_n|[\ ]]$.

2.1.3 Models of Programs

A *universe* is a set of elements. In order to give a semantics for a definite clause program, we have to first choose a universe for the program. Given a first order language L, the *Herbrand universe* U_L of L is the set of all ground terms in the language. (In case L has no constants, we add some constant, say, a, to form ground terms.) The *Herbrand base* B_L for L is the set of all ground atoms in the language.

An *interpretation* I of a definite clause program maps each function symbol of arity n in (the language of) the program to a total function of arity n on the universe, and each predicate symbol of arity n in (the language of) the program to a set of n-tuples from the universe. Thus each constant, which is a 0-ary function symbol, is mapped to an element in the universe. Such a mapping can be uniquely extended to a mapping from ground terms to elements of the universe. A *model* $\mathcal{M}$ of a definite clause program is an interpretation that is closed under rule implication, i.e., if

$$h(\overline{t})\text{:} - b1(\overline{t_1}), b2(\overline{t_2}), \ldots, bn(\overline{t_n}).$$

is a rule in the program, and θ is a ground substitution such that

$$(\forall i, 1 \leq i \leq n, \mathcal{M}(\overline{t_i}[\theta]) \in \mathcal{M}(bi))$$

then $\mathcal{M}(\overline{t}[\theta]) \in \mathcal{M}(h)$.

A *Herbrand interpretation* of a definite clause program is an interpretation of the program that satisfies the following properties. Let L be the language of the program.

1. The universe of the interpretation is the Herbrand universe U_L.
2. Constants in L are mapped to themselves in U_L.

[2]We assume that no literal is repeated in the body of a definite clause.

[3]This definition is more restricted than that of Lloyd [Llo87], which considers a goal to be a disjunction of negative literals. In this thesis, we only consider the case where all goals have a single literal, for ease of exposition.

3. If f is an n-ary function symbol in L, then the mapping from $(U_L)^n$ into U_L defined by

   $$(t_1, \ldots, t_n) \to f(t_1, \ldots, t_n)$$

   is assigned to f.

When we use a Herbrand interpretation, we do not need to distinguish between a term and its mapping under the interpretation.[4] A *Herbrand model* is a Herbrand interpretation that is a model. The least Herbrand model semantics of a logic program is given by its least Herbrand model; for definite clause programs, such a model always exists (see Lloyd [Llo87]).

Since the Herbrand model semantics of a program is a model, it supports the declarative reading of clauses as "if body is true, then head is true".

Example 2.1.1 Consider the following program, whose language has a constant a and a 1-ary function symbol f.

$p(X) :- q(X).$
$q(f(a)).$

The Herbrand universe of this program is $\{a, f(a), f(f(a)), f(f(f(a))), \ldots\}$. The least Herbrand model of the program is

$$\{q(f(a)), p(f(a))\}$$

The following is a Herband model that is not a least Herbrand model:

$$\{q(f(a)), q(a), p(f(a)), p(a)\}$$

□

An alternative way of defining the semantics of a program is by means of a 'least fixpoint', defined as below. (See Lloyd [Llo87] for more details.)

Let P be a definite clause program. The mapping $T_P : 2^{B_P} \to 2^{B_P}$ is defined as follows. Let I be a Herbrand interpretation, and let $P = \{R1, \ldots, Rp\}$. We define

$$T_{Ri}(I) = \{A \in B_P : A \leftarrow A_1, \ldots, A_n \text{ is a ground instance of } Ri,$$
$$\text{and } \{A_1, \ldots, A_n\} \subseteq I\}$$

and

$$T_P(I) = \cup_{i=1}^{p} T_{Ri}(I)$$

For definite clause programs, T_P is monotonically increasing. The least fixpoint semantics of P is defined as the least fixpoint[5] of the function $T_P(I)$. For definite clause programs, the least fixpoint always exists, and the least Herbrand model of P is equivalent to the least fixpoint of T_P [vEK76].

[4] In a Herbrand interpretation, function symbols can be viewed as "record constructors".
[5] That is, the least set that is mapped to itself by the function.

The least fixpoint of T_P can be computed as follows. Define $T_P^0(I) = \phi$, and define $T_P^{i+1}(I)$ as $T_P(T_P^i(I))$, and

$$T_P^\omega(I) = \cup_{i<\omega}(T_P^i(I))$$

Then the least fixpoint of T_P is equivalent to $T_P^\omega(I)$ [Llo87]. We do not necessarily have to compute the infinite set of values $T_P^i(I)$ for all i. If $T_P^{j+1}(I) = T_P^j(I)$ for some j, then $T_P^\omega(I) = T_P^j(I)$.

In this thesis, unless otherwise specified, we assume that the universe for a definite clause program is its Herbrand universe, and the semantics of the program is the least fixpoint semantics. We shall refer to this semantics as *the* semantics of the definite clause program.

2.1.4 Databases and Programs

A definite clause program consists of a finite set of definite clauses. In the context of databases, a large number of these clauses are likely to be facts. We follow the convention in deductive database literature of separating the *program* P from the *database* D. The database consists of a set of facts, while the program contains rules. The motivation is that the rewriting algorithms to be discussed are applied only to the program, and not to the database. This is important in the database context since the set of facts can be very large. However, the distinction is artificial, and we may choose to consider (a subset of) facts to be rules if we wish. In most cases, we refer only to the program; the database is used implicitly.

We assume that the predicates defined in the database (referred to variously as *database predicates*, *base predicates* or *Extensional DB (EDB) predicates*) are distinct from the predicates defined in the program (referred to as *derived predicates*), which can be ensured as follows: Rename all predicates in the database with new names, and for each n-ary predicate p_i renamed to r_i, add a rule

$$p_i(X_1, \ldots, X_n) :- r_i(X_1, \ldots, X_n).$$

to the program. A *base literal* is a literal whose predicate is base, and a *derived literal* is a literal whose predicate is derived.

We use the notation $\langle P, Q \rangle$ to denote a program P with a query Q; we call $\langle P, Q \rangle$ a *program-query pair.*

A *relation* is a finite set of facts. A relation is said to be ground if all facts in it are ground; otherwise the relation is said to be non-ground. We assume knowledge of the basic relational operators such as *select* (σ), *project* (π) and *join* ($\bowtie$). See [Ull88] for definitions of these operators.

By virtue of having its variables universally quantified, a non-ground fact represents the set of its ground instances in the Herbrand base. Given a fact f, let $gnd(f)$ denote the set of ground facts represented by f. A relation R with non-ground facts represents the relation containing the union of the ground facts represented by the facts in R. Given a relation R, let $gnd(R)$ denote the set of ground facts represented by R. Two facts $f1$ and $f2$ are *equivalent* if $gnd(f1) = gnd(f2)$. Two facts are equivalent iff they are variants of each other (in other words, they are equal up to renaming). Whenever we say that two facts are *equal*, unless otherwise specified we mean that they are equivalent.

A fact $f1$ *is subsumed by* a fact $f2$ if $gnd(f1) \subseteq gnd(f2)$. Given a fact f and a relation R, we say that f is *subsumed by* R if $gnd(f) \subseteq gnd(R)$.

Since variables in a fact are universally quantified, testing subsumption of a fact by another requires renaming of variables to avoid name clashes. A fact $f1$ subsumes a fact $f2$ iff there is a variant $f1'$ of $f1$

and a substitution θ such that $f1'[\theta] = f2$. If a fact f_1 subsumes a fact f_2, we say that f_1 is *more general* than f_2, or equivalently, f_2 is *more specific* than f_1.

Consider a program. Let R be a rule in the program, θ a substitution, and I an interpretation for the program. Then $R[\theta]$ is an *instantiation* of R. $R[\theta]$ is said to be a *successful instantiation* in interpretation I if for each literal $p_i(\overline{t_i})$ in the body of R, $p_i(\overline{t_i})[\theta]$ is subsumed by I.

A definite clause program is said to be a *Datalog* program if it does not use any function symbols other than constants, and the database facts do not use any function symbols other than constants. A rule is said to be *range-restricted* if every variable that appears in the head also appears in a literal in the body. (For the case of rules with empty body, this is equivalent to there being no variables in the rule.) A program is said to be *range-restricted* if all facts in the database are ground, and every rule in the program is range-restricted.[6]

2.2 The Bottom-Up Approach

The bottom-up approach to answering queries consists of a two-part process. First, the program-query pair is rewritten in a form so that the bottom-up fixpoint evaluation of the program will be more efficient; next, the fixpoint of the rewritten program is computed by bottom-up iteration. Section 2.2.1 describes the initial rewriting, while Section 2.2.3 investigates the computation of the fixpoint of the rewritten program. Both these steps can be refined further as discussed in later chapters.

2.2.1 The Magic Templates Rewriting Algorithm

Suppose we are given a query $?q(\overline{c})$ on a program that defines predicate q. An evaluation of the fixpoint of the program would generate all facts implied by the program, including many that are irrelevant to the query. Magic rewriting [BMSU86, BR87b, Ram88] addresses this problem.

We present below a simplified version of the Magic Templates rewriting algorithm [Ram88].[7] The idea is to compute an auxiliary predicate *query* that stores subgoals generated on derivated predicates in the program. A fact of the form $query(p(\overline{t}))$ denotes that $?p(\overline{t})$ is a subgoal generated on p. In the fact $query(p(\overline{t}))$, p is formally treated as a function symbol, rather than a predicate, since the language is first order. We thus have a predicate and a function symbol of the same name — they are distinguished based on where they occur in the rule.

The rules in the program are then modified by attaching a literal to the rule body that uses the *query* predicate to act as a filter that prevents the rule from generating irrelevant facts. Further, the rewriting generates rules that define how to generate a query fact for a body literal, given a query fact on the head literal.

Definition 2.2.1 The Magic Templates Algorithm
Let P be a program, and $?q(\overline{c})$ a query on the program. We construct a new program P^{mg}. Initially, P^{mg}

[6]The motivation for this definition is that the fixpoint evaluation of a range-restricted program generates only ground facts.

[7]As described in [BR87b, Ram88], the initial rewriting of a program and query is guided by a choice of *sideways information passing strategies*, or sips. For each rule, the associated sip determines the order in which the body literals are evaluated. The version we present is tailored to the case that sips correspond to left-to-right evaluation with all arguments considered "bound" (perhaps to a free variable), as in Prolog.

is empty.

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

□

We refer to the rules defining the *query* predicate as *query rules.* We sometimes refer to query rules as *magic rules*, and the query predicate as the *magic predicate*, when we need to be consistent with the terminology used in [BMSU86, BR87b, Ram88].

Example 2.2.1 Consider the following program. (In this program sg stands for "same generation".)

$$\begin{array}{lll} R1: \ sg(X,Y) & :- & flat(X,Y). \\ R2: \ sg(X,Y) & :- & up(X,U), sg(U,V), down(V,Y). \\ ?-sg(john,Z) & & \end{array}$$

The Magic Templates algorithm rewrites it as follows:

$$\begin{array}{llll} sg(X,Y) & :- & query(sg(X,Y)), flat(X,Y). & \text{[Mod. Rule R1]} \\ sg(X,Y) & :- & query(sg(X,Y)), up(X,U), & \\ & & sg(U,V), down(V,Y). & \text{[Mod. Rule R2]} \\ query(sg(U,V)) & :- & query(sg(X,Y)), up(X,U). & \text{[Query Rule]} \\ query(sg(john,Z)). & & & \text{[Seed Query]} \end{array}$$

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

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

Theorem 2.2.1 *[Ram88] P is equivalent to P^{mg} with respect to the set of answers to the query.*

Definition 2.2.2 We define the *Magic Templates Evaluation Method* as follows:

1. Rewrite the program and query ($\langle P, Q\rangle$) using the Magic Templates algorithm.
2. Evaluate the fixpoint of the rewritten program.

□

Although the evaluation method and the rewriting algorithm both have the same name, the distinction should be clear from the context. The second step above is presented in more detail in Section 2.2.3. The rewriting has the important effect of mimicking Prolog in that (modulo optimizations such as tail recursion optimization and intelligent backtracking, and modulo some inefficiencies when non-ground facts are generated) only goals and facts generated by Prolog are generated.

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

2.2.2 Supplementary Magic Templates Rewriting

Some joins are repeated in the bodies of rules in the Magic Templates rewritten program. Supplementary Magic Templates rewriting is a version of Magic Templates rewriting that essentially identifies these common sub-expressions and stores them (with some optimizations that allow us to delete some columns from these intermediate, or supplementary, relations). We refer the reader to [BR87b] for details, but present below an example that gives some intuition.

Example 2.2.2 We continue with Example 2.2.1. The program generated by Magic Templates rewriting is as follows.

$$\begin{array}{lll}
sg(X,Y) & :- & query(sg(X,Y)), flat(X,Y). \\
sg(X,Y) & :- & query(sg(X,Y)), up(X,U), sg(U,V), down(V,Y). \\
query(sg(U,V)) & :- & query(sg(X,Y)), up(X,U). \\
query(sg(john,Z)). & &
\end{array}$$

Notice that the second and third rule above have a common prefix; this prefix is factored out to get the following rule set.

$$\begin{array}{lll}
sg(X,Y) & :- & query(sg(X,Y)), flat(X,Y). \\
sup_{1,1}(X,Y,U) & :- & query(sg(X,Y)), up(X,U). \\
query(sg(U,V)) & :- & sup_{1,1}(X,Y,U). \\
sg(X,Y) & :- & sup_{1,1}(X,Y,U), sg(U,V), down(V,Y). \\
query(sg(john,Z)). & &
\end{array}$$

The predicate $sup_{1,1}$ is referred to as a supplementary predicate. The two subscripts denote the number of the rule it is generated from and the position of the next literal in the rule (with numbering starting from 0). Such predicates can be thought of as intermediate predicates used for common-subexpression elimination. However, supplementary predicates actually have a deeper significance. In the above program, $sup_{1,1}(X,Y,U)$ stores bindings of the rule variables X, Y, U generated when a top-down evaluation of a query $?sg(X,Y)$ on the rule

$$sg(X,Y) :- up(X,U), sg(U,V), down(V,Y).$$

set up a subquery on $up(X, Y)$ and got back an answer $up(X, Y)$. Facts for supplementary predicates maintain, in some sense, variable bindings in a "context" of the evaluation of the rule. We generate query facts for derived literals in the rule by using the variable bindings in the supplementary facts, just as we would generate queries in a top-down evaluation coming left-to-right in the body of the rule.

In the above example we generated supplementary rules by factoring common subexpressions out of rules generated by Magic Templates rewriting. In describing variants of Magic rewriting, we find it easier to generate supplementary predicates in a more uniform manner. We store all variables in the rule as arguments of each supplementary predicate, and we introduce a supplementary predicate corresponding to each literal in the body of the rule. Thus the program is rewritten as follows.

$$\begin{array}{lll}
R1.1: \ sup_{1,0}(X,Y) & :- & query(sg(X,Y)). \\
R1.2: \ sg(X,Y) & :- & sup_{1,0}(X,Y), flat(X,Y). \\
R2.1: \ sup_{2,0}(X,Y,U,V) & :- & query(sg(X,Y)). \\
R2.2: \ sup_{2,1}(X,Y,U,V) & :- & sup_{2,0}(X,Y,U,V), up(X,U). \\
Q2.2: \ query(sg(U,V)) & :- & sup_{2,1}(X,Y,U,V). \\
R2.3: \ sup_{2,2}(X,Y,U,V) & :- & sup_{2,1}(X,Y,U,V), sg(U,V). \\
R2.4: \ sg(X,Y) & :- & sup_{2,2}(X,Y,U,V), down(V,Y). \\
Query: \ query(sg(john,Z)). & &
\end{array}$$

The first two rule are derived from $R1$ of the original program, and the next five rules are derived from $R2$. The last rule is the query fact.

Generating the nicer form of the rewritten program presented earlier from this form can be achieved by some simple transformations such as projecting out "unnecessary" variables from each supplementary predicate, and "unfolding"[8] literals that use supplementary predicates. We do not go into details here. □

2.2.3 Iterative Fixpoint Evaluation

A *derivation* in a fixpoint evaluation generates a fact, using a rule R and a fact for each body literal of the rule; there must be a substitution θ for the rule, such that

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

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

We describe an incremental version of fixpoint evaluation called Semi-Naive fixpoint evaluation. Semi-Naive evaluation avoids the repetition of derivations by performing in each iteration an incremental computation using facts generated in the previous iteration.

[8]For the case where there is only one rule R defining a predicate p, *unfolding a literal* $p(\overline{t})$ in a rule R' consists of replacing $p(\overline{t})$ by the body of $R[\theta]$ where θ is the mgu of $p(\overline{t})$ and the head of R (w.l.o.g, we assume that the variables in R and R' are distinct). For the general case, refer to [TS84, GS91].

Semi-Naive evaluation (*SN evaluation*) of definite clause programs was developed by several researchers [Ban85, Bay85, BR87a]. We look at a simplified form of SN evaluation. Without loss of generality, we assume that rules have at most two body literals; rules not in this form can be easily rewritten to be in this form. For each derived predicate q_i in the program we introduce four relations q_i, $q_i^{old}, \delta q_i^{old}$, and δq_i^{new}.[9]

We then rewrite each rule as follows.

Semi-Naive Rewriting(R):

1. If the rule is of the form: $R:\ p(\ldots)\!:-q_1(\ldots), q_2(\ldots)$
   where both q_1 and q_2 are derived predicates, we rewrite R as follows:

   $R':\ \delta p^{new}(\ldots):-\ \delta q_1^{old}(\ldots), q_2^{old}(\ldots).$
   $R'':\delta p^{new}(\ldots):-\ \delta q_2^{old}(\ldots), q_1(\ldots).$

2. If the rule is of the form: $R:\ p(\ldots)\!:-q_1(\ldots), b_2(\ldots)$
   where q_1 is a derived predicate and $b2$ is a base predicate, we rewrite it as follows:

   $R':\delta p^{new}(\ldots)\!:-\delta q_1^{old}(\ldots), b_2(\ldots).$

3. If the rule is of the form: $R:\ p(\ldots)\!:-q_1(\ldots)$
   where q_1 is a derived predicate, we rewrite it as:

   $R':\delta p^{new}(\ldots)\!:-\delta q_1^{old}(\ldots).$

4. If the rule is of the form: $R:\ p(\ldots)\!:-\ldots$
   where the body has no derived predicates, we rewrite it as follows:

   $R':\delta p^{new}(\ldots)\!:-\ldots.$

The above rewriting is called Semi-Naive rewriting [BR87a, Ban85]. Given a program P, let the program generated by Semi-Naive rewriting be denoted P^{SN}.

Semi-Naive evaluation is described in Algorithm SN_Iterate. Procedure *Apply*(R_i) performs the operations of making all derivations that can be performed using rule R_i and the facts in the current extents of the relations,[10] and inserting all derived facts into the relation for the head of R_i. We assume that *Apply* performs a left-to-right nested-loops join with indexing[11] on the rule. (This is important for some of our later theoretical results concerning time complexity, but not for correctness.)

In Semi-Naive iteration, the set of facts produced in iteration n is compared with the set of known facts to identify the new facts produced. Duplicates generated within the same iteration are eliminated implicitly, by the definition of sets.

[9]The distinction between the predicate and the relation should be clear from the context.
[10]In case non-ground facts are derived, it suffices to deduce a set of facts that subsumes the set of all facts that follow from rule R_i and current extents of the relations. This can be done by using most-general unifiers when unifying facts with the rule body.
[11]See, e.g., Ullman [Ull88] for a definition of nested-loops join with indexing.

```
Algorithm SN_Iterate($P^{SN}$)
    1. Foreach rule $R_i$ in $P^{SN}$ that has no derived literal in its body
            Apply( $R_i$).
    2. Repeat
        2.1 Foreach rule $R_i$ in $P^{SN}$ that has a derived literal in its body
                Apply($R_i$).
        2.2 Foreach derived predicate $q_i$ in $P^{SN}$
            a.   $q_i^{old} := q_i^{old} \cup \delta q_i^{old}$.
            b.   $\delta q_i^{old} := \delta q_i^{new} - q_i^{old}$.
            c.   $q_i := q_i^{old} \cup \delta q_i^{old}$.
            d.   $\delta q_i^{new} := \phi$.
    Until all relations $\delta q_i^{old}$ are empty.
```

We call the set of updates in Step 2.2 of the above algorithm as *Semi-Naive updates*. For each predicate q_i, δq_i^{old} denotes the set of q_i facts that were computed in the previous iteration but not in earlier iterations, and q_i^{old} denotes the set of q_i facts derived before the previous iteration. The relation q_i is the union of q_i^{old} and δq_i^{old}. We call the facts in relations of the form q_i^{old} as *old facts*, and facts in relations of the form δq_i^{old} as *new facts*.

The difference operation in Step 2.2.b ensures that δq_i^{old} and q_i^{old} are disjoint when Step 2.2.a is executed. Hence the union operation in Step 2.2.a does not need to check for duplicates; it can simply move facts from δq_i^{old} to q_i^{old}. We do not materialize q_i (Step 2.2.c), but treat it as an un-materialized union of the relations q_i^{old} and δq_i^{old}. To check if a fact is in q_i, we check if it is either in q_i^{old} or in δq_i^{old}.

The procedure *Apply* does not repeat derivations within a single execution of the procedure. Hence no derivations are repeated within an iteration of SN_Iterate. Due to Semi-Naive rewriting and the updates in Step 2.2 of SN_Iterate, in each iteration only derivations that use at least one new fact are carried out. Any derivation performed in an earlier iteration would have used only old facts, and hence no derivation is repeated in the evaluation. Further, any derivation that uses only old facts would have been made in an earlier iteration. Semi-Naive evaluation terminates when no new facts are generated. Thus the algorithm terminates if and only if the set of facts generated is finite.

We call literals of the form δp^{old} or δp^{new} as *δ literals*. With the Semi-Naive rewriting presented above, if a rewritten rule has a δ literal in the body, the first literal in the rewritten rule (and hence in the join order that we assume) is a δ literal.

Performing the join with non-ground facts involves details, such as renaming of variables, discussed in Section 4.2. We use subsumption checking instead of duplicate checking, if non-ground facts are generated, Thus when we add a fact to a relation, we need to check if the fact is subsumed by a fact in the relation, or if it subsumes facts in the relation, and delete facts that are subsumed. Similarly, the operators "$-$" and "$\cup$" used in SN_Iterate perform subsumption checks, rather than duplicate checks, if non-ground facts are generated.

Not-So-Naive (NSN) evaluation [MR89] is the same as Semi-Naive evaluation except for the following differences.

1. δq_i^{new} is a multi-set of facts rather than a set of facts
2. The step $\delta q_i^{old} := \delta q_i^{new} - q_i^{old}$ is replaced by the step $\delta q_i^{old} := \delta q_i^{new}$.
3. The $\cup$ operator does a multi-set union, i.e., it does not check for duplicates.

In the case of NSN evaluation, δq_i^{old} is the multi-set of q_i facts that were computed in the previous iteration, and q_i^{old} is the multi-set of q_i facts derived before the previous iteration.

We also use the terms *Semi-Naive evaluation without duplicate elimination* or *Semi-Naive evaluation without subsumption-checking*[12] to refer to NSN evaluation.

We have the following standard result on completeness of Semi-Naive and Not-So-Naive evaluation (see e.g. [MR89, RSS91]).

Theorem 2.2.2 (Completeness) Suppose a program P is evaluated using Semi-Naive or Not-So-Naive evaluation. If a fact is in the least fixpoint of P, then there is a finite i such that the fact is subsumed by facts derived before iteration i. $\Box$

A *derivation sequence* is a total ordering of derivations in a bottom-up fixpoint evaluation, such that the facts used in any derivation are either base facts, or are generated by earlier derivations. We often use such a total ordering of the derivations in a bottom-up fixpoint evaluation to prove properties of the evaluation.

2.2.4 Related Background Material

The Alexander method [RLK86] was proposed independently of the Magic Sets approach. It is essentially the *supplementary* variant of the Magic Templates method, described in [BR87b]. Seki has generalized the method to deal with non-ground facts and function symbols, and has called the generalized version Alexander Templates [Sek89].

The Magic Templates idea was developed in a series of papers ([BMSU86, BR87b, Ram88]). Several variants of the Magic Templates idea have also been proposed. For example, it is possible to compute supersets of the magic sets (in our notation, the set of facts for *query* is the magic set) without compromising soundness. Although this variant results in some irrelevant computation, it may be possible to compute supersets more efficiently than the magic sets themselves [SS88]. The technique can be extended to deal with SQL programs, including those containing features like group-by, aggregation and arithmetic conditions [MPR90, MFPR90b, MFPR90a]. A performance comparison presented in Mumick et al. [MFPR90a] shows that Magic Sets performs at least comparably to standard query evaluation techniques, and is often significantly better.

The Magic and Alexander methods are based on program transformations. Other methods use a combination of top-down and bottom-up control to propagate bindings. Pereira and Warren presented a memoing top-down evaluation procedure based on Earley deduction [PW83]. Vieille has proposed a method called

[12]It is possible to conceive of Semi-Naive evaluation with duplicate elimination, but without full subsumption-checking. However, we use the term Semi-Naive evaluation without subsumption-checking exclusively to refer to Semi-Naive evaluation without duplicate elimination.

QSQ [Vie86, Vie87, Vie88] that can be viewed as follows. Goals are generated with a top-down invocation of rules, as in Prolog. However, there are two important differences: 1) whenever possible, goals and facts are propagated set-at-a-time, and 2) all generated goals and facts are memoed. If a subgoal is found to have been generated earlier, it is not solved again, but answers derived for the first generation of the subgoal are used for the new subgoal. Dietrich has proposed a method called Extension Tables [Die87]. This method is very similar to QSQ, but performs computation tuple-at-a-time.

The reader is referred to [NR91, RSS92c] for a more detailed discussion of related work.

2.3 Magic Templates and Tail-Recursion

Consider a rule of the form: $R: \ p(\overline{t})\!: -q_1(\overline{t_1}), q_2(\overline{t_2}), \ldots, q_n(\overline{t_n})$. Suppose we had a subgoal $p(\overline{a})$, and in answering this subgoal in a top-down fashion, we had set up and solved subgoals $q_1(\overline{a_1}), \ldots, q_{n-1}(\overline{a_{n-1}})$, and have now set up a subgoal $q_n(\overline{a_n})$.

The subgoal $?q_n(\overline{a_n})$ will return zero or more successful answers. When each answer is returned, no more computation is done at rule R, but control merely passes back to the point where the subgoal $?p(\overline{a})$ was invoked. Prolog can therefore change the return address so that the call to $?q_n(\overline{a_n})$ returns directly to the query on R, bypassing R. This optimization is called *tail recursion optimization* (see for instance [MW88]).

In particular, when q_n is recursive with (possible even the same as) p, Prolog evaluation may return directly past a large number of invocations of R. By bypassing R, Prolog* in effect bypasses a step where a bottom-up evaluation using Magic Templates rewriting would have created a fact for the head predicate p of rule R.

The following example illustrates how Prolog evaluation of a query, using tail-recursion optimization, can be much faster than bottom-up query evaluation using Magic Templates rewriting.

Example 2.3.1 This example is from [Ros91]. Let P be the program

$R1 : p(X, Z) :- e(X, Y), p(Y, Z).$
$R2 : p(n, X) :- t(X).$

$e(1, 2).$
$\vdots$
$e(n-1, n).$

$t(1).$
$\vdots$
$t(m).$

Query: ?-$p(1, X)$.

Given the subgoal $?p(1, X)$ Prolog sets up subgoal $?e(1, X)$ and gets an answer that binds X to 2. Using this binding Prolog sets up a subgoal $?p(2, X)$, which in turn sets up subgoal $?p(3, X)$ and so on till the subgoal $?p(n, X)$ is set up. However, Prolog can deduce that there are no more answers to $e(1, X)$, and when an answer for $?p(2, X)$ is found, it can directly return (with bindings for variable X) to the subgoal $?p(1, X)$, bypassing the subgoal $?p(2, X)$. By applying this optimization repeatedly, when Prolog finds an answer for subgoal $?p(n, X)$, it returns directly (in unit time, with bindings for X) to the subgoal $?p(1, X)$, bypassing all intermediate subgoals. Applying this optimization again to the subgoal $?t(X)$ generated from $?p(n, X)$, when an answer is generated for $?t(X)$, evaluation can return directly to the subgoal $?p(1, X)$. Since there are m answers for $?t(X)$, Prolog backtracks to $?t(X)$ a total of m times, and evaluates the program in time $O(n + m)$.

Prolog "generates" only facts $p(1, j), 1 \leq j \leq m$ (here "generating" a fact is interpreted as the act of Prolog's control returning, with appropriate variable bindings, to the point where a subgoal was set up).

Bottom-up evaluation (using Magic Templates rewriting), on the other hand, works as follows. For each Prolog subgoal $?p(i, X)$, a fact $query(p(i, X))$ is generated. Facts $p(n, 1), \ldots, p(n, m)$ are generated using the modified rule $R2$. These facts are used with the modified rule $R1$ to generate facts $p(n-1, 1), \ldots, p(n-1, m)$, which in turn are used to generate more facts using the modified rule $R2$. Eventually, all facts $p(i, j), 1 \leq i \leq n, 1 \leq j \leq m$ are generated. Thus at least $m \cdot n$ facts are generated, and evaluation takes $o(m \cdot n)$ time. $\Box$

If bottom-up evaluation is to perform as well as Prolog* on this program, it too must bypass the step of computing a fact for the head of rule $R1$.[13] This is precisely the optimization achieved by the program rewriting technique of Ross [Ros91], which we describe in Section 2.3.2. We note that QoSaQ [Vie88], which is a set-oriented top-down evaluation technique that implements memoing, also incorporates a form of tail-recursion optimization.

2.3.1 Hilog Syntax

Before we describe Ross' rewriting technique, we briefly describe an extension of definite clause syntax that is used in the rewriting. The extended syntax is part of Hilog [CKW89]. We describe the extended syntax and its semantics informally. The extension to definite clause syntax allows rules such as the following:

$R1 : A{:}-query(p_j(X, Y), A), p_k(X, Y).$

The head of a definite clause rule must be an atom, whereas the the head of rule $R1$ is a variable — thus the syntax used is higher order.

We require that in rules that use this extended syntax, the variable in the head of the rule must get bound to a term of the form $p_i(\overline{t})$ when the rule is successfully instantiated in bottom-up evaluation. The term $p_i(\overline{t})$ is interpreted as a literal when creating the head fact. For example, suppose we have facts $query(p_j(a, X), p_m(b, X)), p_k(a, c)$ and $p_k(a, d)$. Then $R1$ implies that the facts $p_m(b, c)$ and $p_m(b, d)$ are true. The semantics of rules using the extended syntax is first-order. The use of this higher-order syntax is not essential for our discussion, but it makes the presentation concise.

We can use Semi-Naive evaluation for programs using the above syntactic features, with very minor changes, which we now briefly discuss. The only change to Semi-Naive rewriting is to not transform the

[13]We assume that b and t are base predicates. The time complexity measure used in this thesis ignores the size of the program, based on the assumption that the number of rules is small. We use m and n in the time complexity measures for this program, and cannot assume the number of rules to be small if facts for b and t are treated as rules. We do not apply tail-recursion optimization to base literals. However, applying tail-recursion optimization to base literals provides no benefits, since the computation to solve a query on a base literal in the bottom-up context consists merely of looking up a table, and does not invoke a new rule.

heads of rules that use the above extended syntax. When the body of such an extended rule is satisfied, the head variable is instantiated to a term. This term is treated as a fact; suppose this fact is $p(\overline{a})$. The semi-naive version $\delta p^{new}(\overline{a})$ of this fact is then inferred.

2.3.2 MTTR Rewriting

Ross ([Ros91]) proposed a modification to Magic Templates ([Ram88]). We describe Ross' technique, which we call *Magic Templates with Tail Recursion* (MTTR) rewriting, in this section. The set of predicates to be treated as tail-recursive is a parameter to Ross' rewriting as described in [Ros91] — thus the tail-recursion optimization can be applied to a selected set of predicates. Unless otherwise specified, we assume that the optimization is used for all derived predicates, but not for base predicates. MTTR rewriting may perform worse than Magic Templates rewriting on some programs (see [Ros91] for an example). However, it is useful for the purposes of comparison with Prolog*, since MTTR rewriting can perform tail-recursion optimization whenever Prolog* does so.

Intuitively, the difference between MTTR rewriting and Supplementary Magic rewriting (Section 2.2.2) is as follows. Magic rewriting generates facts of the form $query(p(\overline{s}))$, that indicate that there is a query $?p(\overline{s})$. The rules in the program are modified to generate answers to such queries. With tail recursion optimization in Prolog, answers are not "generated" for a tail-recursive query; instead, answers are "generated" for some query that is an ancestor of the query. This effect is achieved in MTTR rewriting by generating facts of the form $query(p(\overline{s}), q(\overline{t}))$. Such a fact says that there is a query $?p(\overline{s})$; after instantiating a rule to solve this query, instead of generating answers for the query, answers should be directly generated for $?q(\overline{t})$, which is an ancestor of $?p(\overline{s})$. A solution to $?p(\overline{s})$ provides bindings for variables in $\overline{t}$; applying these bindings to $q(\overline{t})$ gives us answers for $q(\overline{t})$.

To handle the case of non-tail-recursive literals, any query fact generated due to such literals is of the form $query(p(\overline{s}), p(\overline{s}))$ (i.e., the first and second arguments of *query* are the same). Such a fact says that $?p(\overline{s})$ is a query, and answers must be generated for it.

We now present the rewriting; we give some intuition after presenting the rewriting.

MTTR Rewriting: Given program P and a query $?q(\overline{t})$ on P, we generate a program using the following rewrite rules. We call the resultant rewritten program P^T.

0. Generate the rule (actually a fact)

   $query(q(\overline{t}), q(\overline{t})).$

   Call this a *Type 0* rule.

Consider each rule R_j in the program P. Let rule R_j be of the form

$R_j : h(\overline{t}) :- p_1(\overline{t_1}), p_2(\overline{t_2}), \ldots, p_n(\overline{t_n}).$

Let $\overline{V}$ denote a tuple of all variables that appear in R_j.

1. Generate the rule

   $sup_{j,0}(\overline{V}, A) :- query(h(\overline{t}), A).$

Call such rules *Type 1* rules.

2. If the body of R_j is non-empty, generate the following rules and call them *Type 2* rules:

$$\begin{aligned} sup_{j,1}(\overline{V},A) &:- sup_{j,0}(\overline{V},A), p_1(\overline{t_1}). \\ &\vdots \\ sup_{j,n-1}(\overline{V},A) &:- sup_{j,n-2}(\overline{V},A), p_{n-1}(\overline{t_{n-1}}). \end{aligned}$$

3. If the body of R_j is empty, generate the rule

   A: $-sup_{j,0}(\overline{V},A)$.

   If the body of the rule is not-empty, and the last literal is base, or is not treated as tail-recursive, generate the rule

   A: $-sup_{j,n-1}(\overline{V},A), p_n(\overline{t_n})$.

   Call such rules *Type 3* rules.

4. If the body of R_j is non-empty, for each derived literal $p_i(\overline{t_i})$, $i \neq n$ in the body of R_j generate a rule

   $query(p_i(\overline{t_i}), p_i(\overline{t_i}))$: $-sup_{j,i-1}(\overline{V},A)$.

   If the body of the rule is non-empty, and $p_n(\overline{t_n})$ is a derived literal, but is not treated as tail-recursive, generate the rule:

   $query(p_n(\overline{t_n}), p_n(\overline{t_n}))$: $-sup_{j,n-1}(\overline{V},A)$.

   Call such rules *Type 4* rules.

5. If the body of R_j is non-empty and $p_n(\overline{t_n})$ is a derived literal and is treated as tail-recursive, generate the following rule:

   $query(p_n(\overline{t_n}), A)$: $-sup_{j,n-1}(\overline{V},A)$.

   Call such rules *Type 5* rules.

We say that P^T generates a subgoal $?p(\bar{t})$ if it derives a fact $query(p(\bar{t}),\ldots)$. Type 0 and Type 4 rules generate subgoals that must be explicitly solved; however, Type 5 rules provide tail recursion optimization—in effect they say "solve the last subgoal in rule R_j, but instead of generating answers for it, use the bindings to directly generate answers for the goal that invoked the rule". Type 1, 2 and 3 rules collectively perform the same function as rules in the original program, except that they are restricted to generate facts only if there is a corresponding subgoal; thus they avoid generating many irrelevant facts.

Example 2.3.2 [Ros91] The rewritten version of the program from Example 2.3.1 is as follows. We treat e and t as base predicates in this rewriting.

$$\begin{aligned} R1.1: sup_{1,0}(X,Y,Z,A) &:- query(p(X,Z),A). \\ R1.2: sup_{1,1}(X,Y,Z,A) &:- sup_{1,0}(X,Z,A), e(X,Y). \\ R1.3: query(p(Y,Z),A) &:- sup_{1,1}(X,Y,Z,A). \\ R2.1: sup_{2,0}(X,A) &:- query(p(n,X),A). \\ R2.2: A &:- sup_{2,0}(X,A), t(X). \end{aligned}$$

$e(1,2).$

$\quad\ldots$

$e(n-1,n).$

$t(1).$

$\quad\ldots$

$t(m).$

$query(p(1,X),p(1,X)).$

Rules $R1.1$ is a Type 1 rule generated from rule $R1$ of the original program, and $R1.2$ is a Type 2 rule generated from $R1$. Rule $R1.3$ is a Type 5 rule generated from $R1$. Rule $R2.1$ is a Type 1 rule generated from rule $R2$ of the original program, and rule $R2.2$ is a Type 3 rule generated from $R2$.

The query facts derived by the Semi-Naive evaluation of this program are of the form $query(p(i,Z),p(1,Z)), 1 \leq i \leq n$. Rule $R2.2$ derives facts $p(1,j), 1 \leq j \leq m$. Also, supplementary facts $sup_{1,0}(i,Y,Z,p(1,Z))$, $1 \leq i \leq n$, $sup_{1,1}(i,i+1,Z,p(1,Z))$, $1 \leq i < n$, and $sup_{2,0}(Z,p(1,Z))$ are derived. Finally answer facts $p(1,i), 1 \leq i \leq m$ are derived. Overall, Semi-Naive evaluation of the program derives $O(n+m)$ facts, which is the same as the number of inferences made by Prolog evaluation. On the other hand, evaluation of the Magic Templates rewriting of the program makes $O(n \cdot m)$ inferences, as described in Example 2.3.1. $\Box$

Semi-Naive evaluation of P^T may in some cases generate many more facts that Semi-Naive evaluation of the Magic Templates rewritten form of P [Ros91]. However, it has the advantage (for our purposes) that it is never more than a constant factor worse than (a model for) Prolog evaluation, in terms of the number of inferences made, provided that P is range-restricted (see Section 5.10). There are many program/query pairs for which the MTTR rewritten program makes far fewer inferences than Prolog; as an extreme example, there are program/query pairs for which Prolog does not terminate, but Semi-Naive evaluation of the MTTR rewritten program does terminate. In Section 3.4 we present a version of the rewriting that is never more than a constant factor worse than (our model of) Prolog evaluation in terms of the number of inferences made, for *all* programs.

The Hilog notation is not critical for MTTR rewriting — we can generate an equivalent rewritten program in definite clause syntax. The basic idea is that for any rule

$$R : A \colon\!- p_1(\overline{t_1}), \ldots, p_n(\overline{t_n}).$$

in the extended syntax, the variable A in the head can only get bound to terms built from one of a finite number of function symbols (corresponding to the predicates in the program). Hence, for each n-ary predicate p in the original program, we create an instantiated version $R[A/p(\overline{X_n})]$ of R, where $\overline{X_n}$ is an n-tuple of distinct variables that do not appear in R. We then replace R by the set of its instantiated versions. By performing this transformation for each rule of an MTTR rewritten program, we derive an equivalent definite clause program. Clearly, the Hilog notation is more concise.

Chapter 3

Magic Rewriting for Non-Range-Restricted Programs

In this chapter we describe extensions of Magic rewriting techniques for programs that generate non-ground facts. We begin the chapter by showing some pitfalls that bottom-up evaluation using Magic Templates (with or without Tail Recursion) can run into when non-ground facts are generated. The basic problem was noted by Codish, Dams and Yardeni [CDY90], but is not widely recognized. In Example 3.1.2 we extend their observation to show that bottom-up evaluation can make many more inferences than Prolog evaluation. We formalize the problem through the definition of mgu-subgoals and mgu-answers; Magic Templates rewriting can generate answers (and queries) that are not mgu-answers (resp. mgu-subgoals).

We refine Magic Templates rewriting (in Section 3.3) to avoid the problems noted by Codish et al.; we call this refinement MGU Magic rewriting. Bottom-up query evaluation using MGU Magic rewriting generates only mgu-subgoals and mgu-answers. MGU Magic rewriting generates programs that contain "meta-predicates"; in Section 3.2 we discuss the operational semantics of meta-predicates and of programs that use meta-predicates. We use the ideas behind the refinement of Magic Templates rewriting to also refine MTTR rewriting; we call this refinement MGU MTTR rewriting (Section 3.4).

MGU MTTR rewriting is important since it enables us to account for tail-recursion optimization while also dealing with the problems noted by Codish et al. We show in Chapter 4 (Section 4.3) that bottom-up evaluation using MGU MTTR rewriting performs no more "actions" than a small constant times the number of "actions" performed by Prolog* evaluation (a model of Prolog evaluation). (In many cases bottom-up evaluation performs far fewer actions than the number of actions performed by Prolog* evaluation.)

3.1 Problems With Subsumed Answers

Subsumption-checking bottom-up evaluation can make some derivations using subsumed facts that Prolog avoids even though it does not perform any subsumption checking. This observation was made by Codish, Dams and Yardeni [CDY90], using the following example.

Example 3.1.1 [CDY90]

$R1 : q :- p(a), p(X), r(X).$
$R2 : p(X).$
Query: ?-q.

On this program, the only goal generated for the predicate r by Prolog evaluation is $?r(X)$. (To keep the example simple, we do not have any rules defining r — this means that subgoals on r will fail, but this is irrelevant to the point we seek to make.) The Magic Templates rewriting of this program is as follows.

$$\begin{array}{lll}
R1' : & q & :- query(q), p(a), p(X), r(X). \\
M1.1 : & query(p(a)) & :- query(q). \\
M1.2 : & query(p(X)) & :- query(q), p(a). \\
M1.3 : & query(r(X)) & :- query(q), p(a), p(X). \\
R2' : & p(X) & :- query(p(X)). \\
Q : & query(q). &
\end{array}$$

Semi-Naive evaluation of the rewritten program generates the fact $query(p(a))$ first, followed by $p(a)$. The fact $p(a)$ is used for the literals $p(a)$ and $p(X)$ in rule $M1.3$, and a fact $query(r(a))$ is generated. Note that Prolog evaluation does not generate the subgoal $?r(a)$. The query generated from the literal $p(X)$ in this rule is $?p(X)$, which has a most general answer $p(X)$; thus a less general answer is being used for a query that has a more general answer. The answer $p(X)$ is generated later, and $p(a)$ is found to be subsumed (but $p(a)$ has already been used to derive $query(r(a))$). Rule $M1.3$ uses $p(X)$ to generate a query fact $query(r(X))$. $\square$

The above example illustrates the following problem. If a fact $p(X)$ is an answer to a subgoal $?p(X)$, then so is every fact of the form $p(a)$, for every a in the universe of discourse. Such facts may be generated in an evaluation, in response to more specific subgoals, and may be used unnecessarily for more general subgoals.

It is important to avoid using answers computed for less general subgoals as answers for more general subgoals since there are programs where doing so can result in a large loss in efficiency.[1] The following example illustrates an asymptotic slow-down.

Example 3.1.2 Consider the following program and query.

$$\begin{array}{ll}
R1 : q(X) & :- b(X), p(X). \\
R2 : q(X) & :- q2(1), q2(2), p(X), r(X). \\
R3 : p(X). & \\
R4 : r(X) & :- r2(X, n). \\
R5 : r2(X, Y) & :- Y > 0, r2(X, Y-1). \\
R6 : r2(X, 0). &
\end{array}$$

$q2(1).$
$q2(2).$

[1]Another motivation is that in the context of abstract interpretation (see e.g. [CDY90]), using answers computed for less general subgoals to solve more general subgoals can lead to answers that are overly conservative. However, using answers in such a fashion does not affect correctness.

$$\begin{array}{lll}
\text{Query}': & query(q(X)). & \\
MR1: & query(p(X)) & :-\ query(q(X)), b(X). \\
R1': & q(X) & :-\ query(q(X)), b(X), p(X). \\
MR2: & query(q2(1)) & :-\ query(q(X)). \\
MR2': & query(q2(2)) & :-\ query(q(X)), q2(1). \\
MR2'': & query(p(X)) & :-\ query(q(X)), q2(1), q2(2). \\
MR2''': & query(r(X)) & :-\ query(q(X)), q2(1), q2(2), p(X). \\
R2': & q(X) & :-\ query(q(X)), q2(1), q2(2), p(X), r(X). \\
R3': & p(X) & :-\ query(p(X)). \\
MR4': & query(r2(X,n)) & :-\ query(r(X)). \\
R4': & r(X) & :-\ query(r(X)), r2(X,n). \\
MR5': & query(r2(X,Y-1)) & :-\ query(r2(X,Y)), Y>0. \\
R5': & r2(X,Y) & :-\ query(r2(X,Y)), Y>0, r2(X,Y-1). \\
R6': & r2(X,0) & :-\ query(r2(X,0)). \\
 & q2(1) & :-\ query(q2(1)). \\
 & q2(2) & :-\ query(q2(2)). \\
 & b(1). & \\
 & \quad\vdots & \\
 & b(m). &
\end{array}$$

Figure 1: Magic Templates Rewritten Form of Program from Example 3.1.2

$$\begin{array}{l}
b(0). \\
b(1). \\
\quad\vdots \\
b(m).
\end{array}$$

Query: ?-$q(X)$.

If we used Prolog to run this query on this program, rule $R1$ would be used to set up a subquery ?$b(X)$, which returns m answers (one at a time). For each of these answers, a subquery ?$p(i)$ is set up, which succeeds right away, generating an answer $q(i)$. After trying all alternatives for rule $R1$, Prolog then tries $R2$, which generates goal ?$q2(1)$ which succeeds and ?$q2(2)$ which also succeeds. It then generates subgoal ?$p(X)$, which gets an answer $p(X)$. A subgoal $r(X)$ is set up, which is solved in $O(n)$ time by rules $R4, R5$ and $R6$. Rule $R2$ is deterministic, and hence there are no more answers, and Prolog solves this query in $O(m+n)$ time.

Consider now what happens if this query is run using Magic Templates rewriting, and Semi-Naive evaluation. The program obtained by Magic Templates rewriting of the above program is shown in Figure 1. We treat b as a base predicate since it has a large number of facts.

The set of facts computed in each iteration of a subsumption-checking Semi-Naive evaluation of the above program is shown in Table 1. (To keep the table compact, we use **Q** instead of *query*.)

It is clear from Table 1 that Semi-Naive evaluation of the Magic Templates rewritten program derives $O(m \cdot n)$ facts, and would take at least time $O(m \cdot n)$, even though it performs subsumption checking. □

Iteration	Facts Computed
0	$\mathbf{Q}(q(X))$
1	$\mathbf{Q}(q2(1)), \mathbf{Q}(p(0)), \mathbf{Q}(p(1)), \ldots, \mathbf{Q}(p(m))$
2	$q2(1), p(0), p(1), \ldots, p(m)$
3	$\mathbf{Q}(q2(2)), q(0), q(1), \ldots, q(m)$
4	$q2(2)$
5	$\mathbf{Q}(p(X)), \mathbf{Q}(r(0)), \mathbf{Q}(r(1)), \ldots, \mathbf{Q}(r(m))$
6	$p(X), \mathbf{Q}(r2(0,n)), \mathbf{Q}(r2(1,n)), \ldots, \mathbf{Q}(r2(m,n))$
7	$\mathbf{Q}(r2(X,n)), \mathbf{Q}(r2(0,n-1)), \mathbf{Q}(r2(1,n-1)), \ldots, \mathbf{Q}(r2(m,n-1))$
$\vdots$	
$n+6$	$\mathbf{Q}(r2(X,1)), \mathbf{Q}(r2(0,0)), \mathbf{Q}(r2(1,0)), \ldots, \mathbf{Q}(r2(m,0))$
$n+7$	$\mathbf{Q}(r2(X,0)), r2(0,0), r2(1,0), \ldots, r2(m,0)$
$n+8$	$r2(X,0), r2(0,1), r2(1,1), \ldots, r2(m,1)$
$\vdots$	
$2n+7$	$r2(X,n-1), r2(0,n), r2(1,n), \ldots, r2(m,n)$
$2n+8$	$r2(X,n), r(0), r(1), \ldots, r(m)$
$2n+9$	$r(X), q(0), q(1), \ldots, q(m)$
$2n+10$	$q(X)$

Table 1: Semi-Naive Evaluation of Program from Figure 1

Although we used Magic Templates rewriting in the above example, the problems we described would also occur with Supplementary Magic Templates rewriting. It is also not hard to modify the above example to show that the problems illustrated in the example also occur with Magic Templates with Tail Recursion rewriting.

3.1.1 Mgu-Subgoals and Mgu-Answers

We now define mgu-subgoals and mgu-answers. The basic idea behind these definitions is to ensure that if a subgoal is generated from a literal in a rule, only answers to that subgoal or more general subgoals are used for that literal; answers to less general subgoals are not permitted to be used. The order of evaluation of literals in a top-down evaluation of a rule (a.k.a. sideways information passing strategies, or sips, in the context of Magic rewriting [BR87b, Ram88]) affects the subgoals that are generated from the rule. We assume a left-to-right order of evaluation (left-to-right sips) in the following definitions, although the definitions can be extended to the general case.

Recall that given two terms $t1$ and $t2$, $MGU(t1, t2)$ denotes the set of most general unifiers of $t1$ and $t2$, $mgu(t1, t2)$ denotes a an arbitrary element of this set.

Definition 3.1.1 (mgu-subgoals and mgu-answers)

Let P be a program with a given query $?query(\overline{u})$.

The given query $?query(\overline{u})$ is defined to be an mgu-subgoal.

Let R be any rule in the program, and $?q(\overline{s})$ an mgu-subgoal.

1. Suppose rule R is of the form $q(\overline{t})$ (i.e., its body is empty), and $\theta \in MGU(\overline{s}, \overline{t'})$, where $\overline{t'}$ is a renaming of $\overline{t}$ that shares no variables with $\overline{s}$. Then $q(\overline{s})[\theta]$ is an *mgu-answer* to the subgoal $?q(\overline{s})$.

2. Suppose R is of the form:

$$R : q(\overline{t}) \text{:} -p_1(\overline{t_1}), \ldots, p_n(\overline{t_n}).$$

such that $n \geq 1$ and for some $k, 1 \leq k \leq n$, and each i, $1 \leq i \leq k$ there are subgoals $?p_i(\overline{s_i})$, and answers $p(\overline{a_i})$ that satisfy all the following conditions: (W.l.o.g. assume that the $\overline{a_i}$'s share no variables with each other or with rule R.)

   (a) $p_i(\overline{a_i})$ is an mgu-answer to $?p_i(\overline{s_i})$.

   (b) Let $\theta_i = mgu(\langle q(\overline{t}), p_1(\overline{t_1}), \ldots, p_{i-1}(\overline{t_{i-1}})\rangle, \langle q(\overline{s}), p_1(\overline{a_1}), \ldots, p_{i-1}(\overline{a_{i-1}})\rangle)$
   Then $p_i(\overline{s_i}) = p_i(\overline{t_i})[\theta_i]$.

Then $?p_k(\overline{s_k})$ is an *mgu-subgoal* generated from $?q(\overline{s})$.

Further, if $k = n$, and

$$\theta \in MGU(\langle q(\overline{t}), p_1(\overline{t_1}), \ldots, p_n(\overline{t_n})\rangle, \langle q(\overline{s}), p_1(\overline{a_1}), \ldots, p_n(\overline{a_n})\rangle)$$

Then $q(\overline{s})[\theta]$ is an *mgu-answer* to subgoal $?q(\overline{s})$.

□

Note that the definition of mgu-subgoals and mgu-answers is cyclic. This causes no problems, since each answer generated by a program must have an acyclic derivation. The following example illustrates the use of this definition.

Example 3.1.3 Consider a modified version of the program from Example 3.1.1.[2]

$R1 : q(X) :- p(a), p(X), r(X).$
$R2 : p(X).$
$R3 : r(X).$
$R4 : q(1).$
Query: ?-$q(X)$.

Given a query $?q(X)$, by Part 1 of Definition 3.1.1 and rule $R4$, $q(1)$ is an mgu-answer to $?q(X)$. Given a query $?q(X)$, $?p(a)$ is an mgu-subgoal, by Part 2 of Definition 3.1.1 and rule $R1$. Using $R2$, $p(a)$ is an mgu-answer to $?p(a)$, by Part 1 of the definition. Now, $?p(X)$ is an mgu-subgoal, by Part 2 of the definition, using the prefix of $R1$ up to $p(X)$. Using $R2$, $p(X)$ is an mgu-answer to $?p(X)$. Next, $?r(X)$ is an mgu-subgoal, by Part 2 of the definition, and $r(X)$ is an mgu-answer to $?r(X)$, using rule $R3$. Finally, by Part 2 of the definition, $q(X)$ is an mgu-answer to $?q(X)$. Note that $?q(X)$ has two mgu-answers, one of which subsumes the other. □

[2]The modification to the program is in order to illustrate some aspects of the definition of mgu-answers that are not illustrated by the original program.

3.2 The goal_id Meta-Predicate

A *meta-predicate* is a predicate that does not have a logical semantics. A *meta-literal* is a literal that uses as predicate a meta-predicate. The rewritten programs that we generate using our refinements of Magic Templates rewriting uses a meta-predicate goal_id$(goal, id)$ that assigns identifiers to goals that are generated in the course of bottom-up evaluation. Before we start describing our refinements of Magic Templates rewriting, we need to define the semantics of programs that contain the meta-predicate goal_id.

Meta-predicates are different from ordinary predicates in two ways. First, given a normal predicate p, if a query $?p(X, 1)$ succeeds, each query $?p(a, 1)$ also succeeds, where a is an element of the universe of the program. However, if ?goal_id$(p(X), 1)$ succeeds, it does not follow that ?goal_id$(p(a), 1)$ succeeds, since $p(X)$ and $p(a)$ may be given different identifiers. Second, two occurrences of the same query on a meta-predicate can return different answers, as we illustrate after defining the goal_id meta-predicate.

We do not assign semantics to meta-predicates in the usual manner of assigning sets of facts to predicates. Instead, we assign semantics to meta-predicates operationally in terms of "answers" that are returned to queries on the meta-predicates.

Definition 3.2.1 **(goal_id)** The meta-predicate goal_id(g, n) is defined as follows. When it is *called* with a goal $g(\overline{t})$, it returns an integer identifier n for the goal, where the identifier satisfies the following conditions:

- If subsumption-checking is to be used, (a) all variants of a goal are given the same identifier, and (b) if two goals are not variants of each other, they are given distinct identifiers.
- If subsumption-checking is not to be used, the call returns an identifier that is distinct from those returned by any other calls to goal_id.
- 0 is not generated as the identifier of any goal.

□

For example, a call ?goal_id$(p(X), ID)$ may bind ID to 10. If subsumption-checking is used, all further calls ?goal_id$(p(X), ID)$ will bind ID to 10. However, a call ?goal_id$(p(a), ID)$ will bind ID to some value other than 10. If subsumption-checking is not used, even further calls ?goal_id$(p(X), ID)$ will bind ID to some value other than 10.

As defined above, goal_id does not perform full subsumption-checking on goals — if it did, and gave the same identifier to two goals, one of which subsumes the other, and we will not be able to use the identifier for the purpose of keeping track of mgu-answers to goals. In Section 3.3.4 we discuss how the evaluation technique can be extended in order to allow goal_id to perform some degree of subsumption-checking. It is straightforward to implement the meta predicate goal_id, and we do not discuss details.

It is not possible in general to use the traditional least model or least fix-point semantics for programs with meta-predicates. Instead we define the operational semantics to the programs generated by our rewriting techniques to be the result of Semi-Naive evaluation (either with or without subsumption-checking). With either semantics, the answers generated for the query on the original program are the same for the rewritten programs (as we show when proving the correctness of the rewriting algorithms).

Semi-Naive evaluation works in a straightforward manner with meta-predicates. Meta-predicates are treated in a fashion similar to base predicates. However, instead of indexing a relation for a base predicate and getting an answer, a query is set up on the meta-predicate, and solved.

3.3 MGU Magic Templates

In this section we present a version of Magic Templates rewriting; the Semi-Naive evaluation (using most general unifiers) of the rewritten program generates subgoals and answers only if they are mgu-subgoals or mgu-answers. We call this rewriting technique *MGU Magic Templates*.

For simplicity we describe the supplementary version of the rewriting.[3]

The idea behind MGU Magic Templates rewriting is to keep with each answer the goal for which it was generated as an mgu answer; this lets us avoid using answers to less general subgoals with rules instances (supplementary facts) that generated more general subgoals. If we stored the actual goal in the supplementary facts (without renaming variables in the goal), the process of unification during the generation of the answer would instantiate the goal. We store instead an identifier that tells us what the original goal is; this identifier is generated using the meta-predicate goal_id.

Intuitively, the main difference between Supplementary Magic Templates rewriting (Section 2.2.2) and MGU Magic Templates described below is that for each query fact, answer fact and supplementary fact, we have an extra argument that stores the identifier of a query. Facts of the form $answer(id, q(\overline{a_2}))$ are generated in the bottom-up evaluation of the MGU Magic Templates rewritten program. Intuitively, such a fact says that id is the identifier of a subgoal on q, and $q(\overline{a_2})$ is (at least as general as some) mgu-answer to the subgoal. Similarly, facts of the form $query(q(\overline{a_2}), id)$ are generated in the bottom-up evaluation of the MGU Magic Templates rewritten program. Intuitively, such a fact says that $q(\overline{a_2})$ is a subgoal, and id is the identifier of the subgoal.

Finally, there are facts of the form $sup_{i,j}(i, \overline{v}, i1)$. Intuitively, such a fact represents an instance of the prefix of a rule R_i up to the jth derived literal in R_i, such that R_i is being used to solve a subgoal with identifier i, and $i1$ is the identifier of a subgoal on the jth derived literal of the rule. The rules in the rewritten program (Type 2 rules below) are such that only answers for a query with identifier $i1$ can be used in a derivation with the supplementary fact $sup_{i,j}(i, \overline{v}, i1)$. Thus, the only answer facts that can be used with a supplementary fact are those that are mgu-answers to the query generated from the supplementary fact.

MGU Magic Templates Rewriting: Let P be a program, and $?q(\overline{t})$ a query on P. The following rewrite rules generate a rewritten program which we call P_Q^{MGU} from P and $?q(\overline{t})$.

Generate the rules:

$$\begin{array}{ll} Q_{R1} : initial\_query(q(\overline{t}), ID) & :- \mathsf{goal\_id}(q(\overline{t}), ID). \\ Q_{R2} : query(Q, ID) & :- initial\_query(Q, ID). \\ Q_{R3} : q(\overline{A}) & :- initial\_query(\_, ID), answer(ID, q(\overline{A})). \end{array}$$

[3]The rewriting assumes left-to-right sips (Section 2.2.1).

from the initial query $q(\overline{t})$, where $\overline{A}$ is a vector of distinct new free variables, of the same arity as q.

Call all the above rules *Type 0* rules.

Consider each rule R_j in the program P. Let rule R_j be of the form

$$R_j : h(\overline{t}) \colon -p_1(\overline{t_1}), p_2(\overline{t_2}), \ldots, p_n(\overline{t_n}).$$

Let $\overline{V}$ denote a tuple of all variables that appear in R_j.

1. If the body of R_j is empty generate the rule

   $$sup_{j,0}(HId, \overline{V}, 0) \colon -query(h(\overline{t}), HId).$$

   else generate the rules:

   $$\begin{array}{ll} sup1_{j,0}(HId, \overline{V}, p_1(\overline{t_1})) & :-\ query(h(\overline{t}), HId). \\ sup_{j,0}(HId, \overline{V}, I1) & :-\ sup1_{j,0}(HId, \overline{V}, G), \text{goal\_id}(G, I1). \end{array}$$

   Call the above rules Type 1 rules.

2. If the body of rule R_j is not empty, for each i, $1 \leq i \leq n-1$, generate the following rules:

   $$\begin{array}{ll} sup1_{j,i}(HId, \overline{V}, p_{i+1}(\overline{t_{i+1}})) & :-\ sup_{j,i-1}(HId, \overline{V}, I1), answer(I1, p_i(\overline{t_i})). \\ sup_{j,i}(HId, \overline{V}, I1) & :-\ sup1_{j,i}(HId, \overline{V}, G), \text{goal\_id}(G, I1). \end{array}$$

   Call these rules Type 2 rules.

3. If the body of R_j is empty generate the rule:

   $$answer(HId, h(\overline{t})) \colon -sup_{j,0}(HId, \overline{V}, \_).$$

   otherwise generate the rule:

   $$answer(HId, h(\overline{t})) \colon -sup_{j,n-1}(HId, \overline{V}, I1), answer(I1, p_n(\overline{t_n})).$$

   Call these rules Type 3 rules.

4. For each literal $p_i(\overline{t_i})$ in the body of R_j generate a rule

   $$query(p_i(\overline{t_i}), ID1) \colon -sup_{j,i-1}(HId, \overline{V}, ID1).$$

   Call such rules Type 4 rules.

For each base predicate b_i used in the program generate a rule:

$$answer(ID, b_i(\overline{X_i})) \colon -query(b_i(\overline{X_i}), ID), b_i(\overline{X_i}).$$

where $\overline{X_i}$ is a tuple of distinct variables, with arity equal to that of b_i. Call such rules Type 6 rules.

Note that there are no Type 5 rules above — the numbering is designed to be consistent with the numbering of rule types used in MTTR rewriting as described in Section 2.3.2.

Rule Q_{R1} generates an *initial_query* fact corresponding to the initial query on the program. This fact is used to generate a *query* fact using rule Q_{R2}. Rule Q_{R3} selects facts that are answers to the initial query. The structure of rule Q_{R3} ensures that the *id* field of any answer fact used in a successful instantiation of the

$$
\begin{array}{lll}
Q_{R1}: & \mathit{initial\_query}(anc(X,Y),ID) & :- \mathsf{goal\_id}(anc(X,Y),ID). \\
Q_{R2}: & query(Q,ID) & :- \mathit{initial\_query}(Q,ID). \\
Q_{R3}: & anc(X,Y) & :- \mathit{initial\_query}(\_,ID), answer(ID,anc(X,Y)). \\
S1.0: & sup1_{1,0}(HId,X,Y,parent(X,Y)) & :- query(anc(X,Y),HId). \\
S1.1: & sup_{1,0}(HId,X,Y,ID) & :- sup1_{1,0}(HId,X,Y,G), \mathsf{goal\_id}(G,ID). \\
M1.0: & query(parent(X,Y),ID) & :- sup_{1,0}(HId,X,Y,ID). \\
R1': & answer(HId,anc(X,Y)) & :- sup_{1,0}(HId,X,Y,ID), answer(ID,parent(X,Y)). \\
S2.0: & sup1_{2,0}(HId,X,Y,Z,parent(X,Z)) & :- query(anc(X,Y),HId). \\
S2.1: & sup_{2,0}(HId,X,Y,Z,ID) & :- sup1_{2,0}(HId,X,Y,Z,G), \mathsf{goal\_id}(G,ID). \\
M2.0: & query(parent(X,Z),ID) & :- sup_{2,0}(HId,X,Y,Z,ID). \\
S2.2: & sup1_{2,1}(HId,X,Y,Z,anc(Z,Y)) & :- sup_{2,0}(HId,X,Y,Z,ID), answer(ID,parent(X,Z)). \\
S2.3: & sup_{2,1}(HId,X,Y,Z,ID) & :- sup1_{2,1}(HId,X,Y,Z,G), \mathsf{goal\_id}(G,ID). \\
M2.1: & query(anc(Z,Y),ID) & :- sup_{2,1}(HId,X,Y,Z,ID). \\
R2': & answer(HId,anc(X,Y)) & :- sup_{2,1}(HId,X,Y,Z,ID), answer(ID,anc(Z,Y)) \\
B1: & answer(ID,parent(X1,X2)) & :- query(parent(X1,X2),ID), parent(X1,X2).
\end{array}
$$

Figure 2: MGU Magic Rewriting of Ancestor Program

rule must match the *id* field of the *initial_query* fact. Hence there is no need to actually unify the answer with the initial query.

We call the rewritten version of a program P with query Q as P_Q^{MGU}; we often refer to P_Q^{MGU} as P^{MGU} when the query Q is understood from the context, or is not relevant to the discussion.

The following is an example of MGU Magic Templates rewriting. We presented a simple version of the rewriting above in order to keep the proofs simple. If we use the simple version of the rewriting, there are a large number of rules in the rewritten program for each rule in the original program (although the number of rules is linear in the number of literals in the original rule). After presenting the example, we discuss how to improve the rewriting to reduce the number of rules generated.

Example 3.3.1 Suppose we had the program

$R_1 : anc(X,Y) :- parent(X,Y).$

$R_2 : anc(X,Y) :- parent(X,Z), anc(Z,Y).$

Here the only derived predicate is *anc*, and the only base predicate is *parent*. Given a query $?anc(X,Y)$, the rewritten program is as shown in Figure 2.

The first three rules above are generated from the query. The next four rules are generated from rule R_1. The first two generate a supplementary fact containing an identifier for the query on the first body literal. The third rule generates a query on the body literal $parent(X,Y)$. The fourth rule generates answers for the head from answers for the body literal. The rules generated from R_2 are similar to the above. The last rule in the program is a Type 6 rule, that generates answer facts for the base predicate *parent*. $\Box$

In an actual implementation, we would generate all the above rules except Q_{R1} above at compile time (when we do not have an actual query). At run time, we would generate a fact for *initial_query* from the actual query fact, and add it to the database.

3.3.1 Optimizations of MGU Magic Templates Rewriting

Several optimizations are possible on the above rewritten program. Henceforth we use these optimizations in the examples in this thesis , but to keep our proofs simple we use the original version of the rewriting in the proofs. We justify the correctness of the optimizations using simple arguments.

First, we can treat base predicates specially in the rewriting. We apply the following transformation to the rewritten program, for each base predicate b_i. First, each literal $answer(ID, b_i(\overline{t_i}))$ where b_i is a base predicate is replaced by the literal $b_i(\overline{t_i})$. Thus a rule

$$sup1_{j,i+1}(HId, \overline{V}, p_{i+1}(\overline{t_{i+1}})) \colon -sup_{j,i}(HId, \overline{V}, I1), answer(I1, b_i(\overline{t_i})).$$

is replaced by a rule

$$sup1_{j,i+1}(HId, \overline{V}, p_{i+1}(\overline{t_{i+1}})) \colon -sup_{j,i}(HId, \overline{V}, I1), b_i(\overline{t_i}).$$

Next consider rules of the following form:

$$sup1_{j,i-1}(HId, \overline{V}, b_i(\overline{t_i})) \colon -sup_{j,i-2}(HId, \overline{V}, I1), answer(I1, p_{i-1}(\overline{t_{i-1}})).$$

$$sup_{j,i-1}(HId, \overline{V}, I1) \colon -sup1_{j,i-1}(HId, \overline{V}, G), \mathsf{goal\_id}(G, I1).$$

We replace these rules by the rule

$$sup_{j,i-1}(HId, \overline{V}, 0) \colon -sup_{j,i-2}(HId, \overline{V}, I1), answer(I1, p_{i-1}(\overline{t_{i-1}})).$$

Note that any fact generated for $answer(I1, b_i(\overline{t_i}))$ must be generated from a fact for base predicate b_i. For any derivation made using a fact $answer(id, b_i(\ldots))$, there is an equivalent derivation in the modified program, using a fact for the base predicate b_i. Finally, we delete all rules that generate queries on b_i, and we delete the Type 6 rule that generates answer facts for b_i.

Second, although as described above, $\overline{V}$ is a tuple of all variables in the rule, and is used in each of the supplementary rules, it is possible to optimize the rewriting by storing in each $sup_{j,i}$ and $sup1_{j,i}$ literal only those variables that satisfy both the following conditions: (1) the variable appears either in the head of the rule, or in or after the $i+1$th literal of the rule, and (2) the variable appears either in the head of the rule,[4] or in or before the ith literal in the body of the rule. This optimization has no effect on the facts created for other predicates, since for each literal any variable that does not satisfy this condition is either not used anywhere (if the variable does not satisfy condition 1 above), or is guaranteed to be a free variable (if the variable does not satisfy condition 2 above). This optimization is the same as that described in [BR87b] for Supplementary Magic Sets rewriting.

3.3.2 Examples

Example 3.3.2 We consider the program in Example 3.3.1 again, and rewrite it using the optimizations outlined above, to illustrate the effect of the optimizations to MGU Magic rewriting. The rewritten program is shown in Figure 3.

[4]If adornment is used, the variable must appear in a bound argument of the head of the rule.

$$\begin{array}{lll}
Q_{R1}: & \mathit{initial\_query}(anc(X,Y),ID) & :-\mathsf{goal\_id}(anc(X,Y),ID). \\
Q_{R2}: & query(Q,ID) & :-\ \mathit{initial\_query}(Q,ID). \\
Q_{R3}: & anc(X,Y) & :-\ \mathit{initial\_query}(\_,ID), answer(ID,anc(X,Y)) \\
 & & \\
S1.0: & sup_{1,0}(HId,X,Y,0) & :-\ query(anc(X,Y),HId). \\
R1': & answer(HId,anc(X,Y)) & :-\ sup_{1,0}(HId,X,Y,ID), parent(X,Y) \\
 & & \\
S2.0: & sup_{2,0}(HId,X,Y,0) & :-\ query(anc(X,Y),HId). \\
S2.2: & sup1_{2,1}(HId,X,Y,Z,anc(Z,Y)) & :-sup_{2,0}(HId,X,Y,ID), parent(X,Z). \\
S2.3: & sup_{2,1}(HId,X,Y,Z,ID) & :-\ sup1_{2,1}(HId,X,Y,Z,G), \mathsf{goal\_id}(G,ID) \\
M2.1: & query(anc(Z,Y),ID) & :-\ sup_{2,1}(HId,X,Y,Z,ID). \\
R2': & answer(HId,anc(X,Y)) & :-\ sup_{2,1}(HId,X,Y,Z,ID), answer(ID,anc(Z,Y))
\end{array}$$

Figure 3: Optimized MGU Magic Rewriting of *ancestor* program

$$\begin{array}{lll}
Q_{R1}: & \mathit{initial\_query}(q,ID) & :-\ \mathsf{goal\_id}(q,ID). \\
Q_{R2}: & query(Q,ID) & :-\ \mathit{initial\_query}(Q,ID). \\
Q_{R3}: & q & :-\ \mathit{initial\_query}(\_,ID), answer(ID,q). \\
S1.0: & sup1_{1,0}(HId,p(a)) & :-\ query(q,ID). \\
S1.0': & sup_{1,0}(HId,ID1) & :-\ sup1_{1,0}(HId,G), \mathsf{goal\_id}(G,ID1). \\
M1.0: & query(p(a),ID1) & :-\ sup_{1,0}(HId,ID1). \\
S1.1: & sup1_{1,1}(HId,X,p(X)) & :-\ sup_{1,0}(HId,ID1), answer(ID1,p(a)). \\
S1.1': & sup_{1,1}(HId,X,ID1) & :-\ sup1_{1,1}(HId,X,G), \mathsf{goal\_id}(G,ID1). \\
M1.1: & query(p(X),ID1) & :-\ sup_{1,1}(HId,X,ID1). \\
S1.2: & sup1_{1,2}(HId,X,r(X)) & :-\ sup_{1,1}(HId,X,ID1), answer(ID1,p(X)). \\
S1.2': & sup_{1,2}(HId,X,ID1) & :-\ sup_{1,1}(HId,X,G), \mathsf{goal\_id}(G,ID1). \\
M1.2: & query(r(X),ID1) & :-\ sup_{1,2}(HId,X,ID1). \\
R1': & answer(HId,q) & :-\ sup_{1,2}(ID,X,ID1), answer(ID1,r(X)). \\
S2.0: & sup_{2,0}(HId,X) & :-\ query(p(X),HId). \\
R2': & answer(HId,p(X)) & :-\ sup_{2,0}(HId,X).
\end{array}$$

Figure 4: MGU Magic Templates Rewriting of Program from Example 3.3.3

The effect of the optimizations are as follows. Queries are no longer generated for *parent*, since *parent* is a base predicate, and the Type 6 rule that generates answer facts for *parent* has been removed. The number of supplementary rules has decreased since there is no need to compute goal-ids for base literals in the rule body. The number of variable bindings stored in the supplementary predicates $sup_{2,0}$ is less than before. □

Example 3.3.3 We use the following program from Example 3.1.1 to illustrate the differences between evaluation of the Magic Templates and the MGU Magic Templates rewritten programs.

$R1 : q :-\ p(a), p(X), r(X).$

$R2 : p(X).$

Query: ?-q.

The MGU Magic Templates rewritten version of the program, P^{MGU_MT}, is shown in Figure 4.

We assume that goal_id generates identifiers $1, 2, \ldots$ in sequence. The evaluation of this program first generates the following facts (in sequence): $initial\_query(q, 1)$, $query(q, 1)$, $sup1_{1,0}(1, p(a))$, $sup_{1,0}(1, 2)$, $query(p(a), 2)$. At this stage, rules $S2.0$ and $R2'$ generate the facts: $sup_{2,0}(2, a)$ and $answer(2, p(a))$. Now rules $S1.1, S1.1'$ and $M1.1'$ generate the facts: $sup1_{1,1}(1, X, p(X))$, $sup_{1,1}(1, X, 3)$, and $query(p(X), 3)$.

The evaluation of the Magic Templates rewriting of the program generates corresponding facts $query(p(a))$, $p(a)$, $query(p(X))$, and $p(X)$. Up to this stage, the evaluation of the MGU Magic rewritten program essentially parallels the evaluation of the supplementary Magic rewritten program:

The difference between the two versions of the rewriting is that the MGU Magic rewriting does not use $answer(2, p(a))$ in rule $S1.2$ (corresponding to the literal $p(X)$), since the supplementary fact $sup_{1,1}(1, X, 3)$ contains the goal-identifier 3. Rather, only $answer(3, p(X))$ is used in Rule $S1.2$. Following this derivation, facts $sup1_{1,2}(1, X, r(X)), sup_{1,2}(1, X, 4)$, and $query(r(X), 4)$ are generated. No query fact of the form $query(r(a), n)$ is generated. On the other hand, the Supplementary Magic rewritten program generates the fact $query(r(a))$ followed an iteration later by the facts $query(r(X))$, and $r(a)$, followed an iteration later by $r(X)$. MGU Magic rewriting has avoided generating a query (resp., an answer) that is not an mgu-subgoal (resp., an mgu-answer).

We do not go into details of the evaluation of the MGU Magic rewriting of Example 3.1.2, but note that (by a similar process as above) the subgoal $query(r(X))$ is generated, while the subgoals

$$query(r(0)), \ldots, query(r(m))$$

are not generated. The evaluation would derive $O(m + n)$ facts rather than the $O(m \cdot n)$ facts that the evaluation of the Magic rewritten program would generate. □

3.3.3 Correctness of MGU Magic Templates

We define the following property to make the statements of several of our lemmas and theorems concise.

Property 3.3.1 (MGU-Prop) Let P be any program, and Q a query on P. We say that an evaluation of P_Q^{MGU} has property MGU-Prop if

1. Every fact $answer(id, p(\overline{a}))$ generated in the evaluation is such that $p(\overline{a})$ is an mgu-answer to a subgoal on p that has identifier id.
2. Every fact $query(p(\overline{a}), id)$ generated in the evaluation is such that $?p(\overline{a})$ is an mgu-subgoal with identifier id. □

The following lemma provides some intuition behind the variable bindings that are stored in the supplementary facts.

Lemma 3.3.1 *Let P be any program, and Q a query on P. Consider a step in a derivation sequence for P_Q^{MGU} such that the evaluation prior to that step has property MGU-Prop.*

Suppose a supplementary fact $sup_{j,i}(id, \overline{v_i}, id_{i+1})$ is derived at this step. Let $sup_{j,i}$ be a supplementary predicate generated from a rule R_j of P,

$$R_j : p(\overline{t}) \colon\!- p_1(\overline{t_1}), p_2(\overline{t_2}), \ldots, p_n(\overline{t_n}).$$

such that the body of R_j is non-empty.

Then there are facts $answer(id_1, p_1(\overline{a_1})), \ldots, answer(id_{i-1}, p_{i-1}(\overline{a_{i-1}}))$, and a fact $query(p(\overline{s}), id)$, such that

1. *Each $id_m, 1 \leq m \leq i$, is the id of an mgu-subgoal generated from $?p(\overline{s})$, and*
2. *The substitution for variables of R_j specified by $\overline{v_i}$, is in*

$$MGU(\langle p(\overline{t}), p_1(\overline{t_1}), \ldots, p_{i-1}(\overline{t_{i-1}})\rangle, \langle p(\overline{s}), p_1(\overline{a_1}), \ldots, p_{i-1}(\overline{a_{i-1}})\rangle)$$

□

The proof of this lemma is presented in Appendix A.1.

Theorem 3.3.2 *Given any program P and query Q, the bottom-up evaluation of P_Q^{MGU} has property MGU-Prop.* □

The proof is by induction on derivation sequences for P^{MGU}, and the full proof is presented in Appendix A.1.

Theorem 3.3.3 *Given any program P and query Q, the bottom-up evaluation of P_Q^{MGU} is complete with respect to Q, i.e., if a fact p that is an answer to Q is present in the least model of P, then p is subsumed by a fact computed in the bottom-up evaluation of P_Q^{MGU}.* □

The proof of this theorem is presented in Appendix A.1. We sketch the idea below. The theorem is proved by proving the following more general result (p stands for any predicate, in the following): if a fact $query(p(\overline{b}), id)$ is available to the evaluation of P_Q^{MGU}, then for every fact $p(\overline{a})$ that unifies with $p(\overline{b})$, and is generated by a bottom-up evaluation of program P (the original program), evaluation of P_Q^{MGU} generates a fact $answer(id, p(\overline{c}))$ such that $p(\overline{c})$ subsumes $p(\overline{a})[mgu(\overline{a}, \overline{b})]$. The proof of the above result is by induction on derivation sequences for the original program P. Consider a step (i.e., a rule along with facts used in the derivation) in the derivation sequence. Suppose that the theorem holds for all facts used in the derivation step. An induction going left-to-right on the body of the rule shows that needed *query* facts are generated (and the outer induction shows that the corresponding *answer* facts for these queries are also generated). These facts are used to generate the required answer fact for the head of the rule.

3.3.4 Discussion

The sequence of results above show that given a program P and a query Q, the bottom-up evaluation of P_Q^{MGU} is sound, generates all answers to query Q on P, and further, the evaluation of P_Q^{MGU} generates only mgu-subgoals and mgu-answers. By not generating answers that are not mgu-answer, with programs such as those discussed in Section 3.1, bottom-up evaluation does not generate the subsumed answers that caused it to be less efficient than Prolog evaluation. Prolog evaluation may still be more efficient due to tail-recursion optimization, which is not performed by the MGU Magic rewriting. In the next section we describe how to incorporate the ideas from this section into Magic Templates with Tail Recursion rewriting.

The goal-ids that we generate are very similar to the lcid/lcont scheme used for indexing answers and goals in QSQR [Vie86, Vie88]. We used them primarily to avoid the use of answers to a query to directly answer a more general query. However, we can also use these goal-identifiers for the purpose of indexing answers and goals, as is done in QSQR. Suppose we have a supplementary fact (resp. answer fact) with a goal-id value id, and suppose that the fact unifies with the body literal of a (supplementary) rule. Then an answer fact (resp. supplementary fact) unifies with the other body literal of the instantiated rule if and only if it has the same goal-id value as the supplementary fact (resp. answer fact). The only if part is easy to see from the structure of the supplementary rule. The if part follows since the answer fact must be an answer to a query generated from the supplementary fact since it has the same goal-id.

The ids are ground values, so indexing on the id fields of relations can be done efficiently (in constant time using hash tables). This form of indexing is useful for linking supplementary facts with answer facts; any supplementary and answer facts fetched using the index are guaranteed to unify. We have implemented such an indexing scheme in the CORAL deductive database system [RSS92b].

Semi-Naive evaluation of an MGU Magic Templates rewritten program checks for variants of a goal, but does not perform full subsumption-checking on goals, due to the definition of the goal_id predicate, and since a goal id is stored with each goal. If there are two goals that are not equivalent up to renaming, both goals are stored. Not being able to do full subsumption-checking is a price we pay for keeping track of which answer is an mgu-answer to which goal. We can extend the definition of the meta-predicate goal_id to allow some subsumption checking on goals. If a new goal ng is subsumed by an old goal og, we give the same identifier to ng as we gave to og earlier. Let this identifier be id. It is critical that any query fact $query(ng, id)$ is eliminated by subsumption-checking before it is used, for otherwise we will generate answers for id that are not mgu-answers. It is possible to extend the rewriting, as well as the subsumption-checking in the evaluation algorithm to perform a greater degree of subsumption-checking. We do not go into details here.

3.4 MGU MTTR Rewriting

In this section we combine ideas from the MGU Magic Templates rewriting and Magic Templates with Tail Recursion rewriting to get a combined technique, which we call *MGU Magic Templates with Tail Recursion* rewriting, or *MGU MTTR* rewriting for short. In Chapter 4 we compare the semi-naive evaluation of the MGU MTTR rewriting with Prolog* evaluation (a model for Prolog evaluation), and prove that it makes no more inferences than Prolog* evaluation.

We describe MGU MTTR rewriting as an extension of MTTR rewriting that incorporates the ideas that we used in MGU Magic rewriting. The basic extension is to add goal-identifier fields to query, supplementary and answer predicates. As is the case with MGU Magic Templates rewriting, the goal-identifier field is used to ensure that an answer that is an mgu-answer for some query will not be used as an answer for a more general query. For simplicity, we assume that all predicates are tail-recursive when describing the rewriting, and later indicate how to relax this assumption. MGU MTTR rewriting is described below.

MGU MTTR Rewriting:

Let P be a program, and $?q(\bar{t})$ a query on P. The following rewrite rules generate a rewritten program which we call $P_Q^{MGU\_T}$ from P and $?q(\overline{t})$.

0. Generate the rules:

$$\begin{array}{ll} Q_{R1}: \ initial\_query(q(\overline{t}), ID, answer(ID, q(\overline{t}))){:}-\text{goal\_id}(q(\overline{t}), ID) & \\ Q_{R2}: query(Q, ID, Ans) : - \ initial\_query(Q, ID, Ans). & \\ Q_{R3}: q(\overline{A}) \quad : - \ initial\_query(\_, ID, \_), answer(ID, q(\overline{A})). & \end{array}$$

   from the initial query $q(\overline{t})$, where $\overline{A}$ is a vector of distinct new free variables, of the same arity as q.

   Call all the above rules *Type 0* rules.

Consider each rule R_j in the program P. Let rule R_j be of the form

$$R_j : h(\overline{t}){:}-p_1(\overline{t_1}), p_2(\overline{t_2}), \ldots, p_n(\overline{t_n}).$$

Let $\overline{V}$ denote a tuple of all variables that appear in R_j.

1. If the body of R_j is empty generate the rule

$$sup_{j,0}(HId, \overline{V}, 0, A){:}-query(h(\overline{t}), HId, A).$$

   else generate the rules:

$$\begin{array}{ll} sup1_{j,0}(HId, \overline{V}, p_1(\overline{t_1}), A) & : - \ query(h(\overline{t}), HId, A). \\ sup_{j,0}(HId, \overline{V}, I1, A) & : - \ sup1_{j,0}(HId, \overline{V}, G, A), \text{goal\_id}(G, I1). \end{array}$$

   Call these rules *Type 1* rules.

2. If the body of rule R_j is not empty, for each i, $1 \leq i \leq n-1$, generate the following rules.

$$\begin{array}{ll} sup1_{j,i}(HId, \overline{V}, p_{i+1}(\overline{t_{i+1}}), A) & : - \ sup_{j,i-1}(HId, \overline{V}, I1, A), \\ & \quad answer(I1, p_i(\overline{t_i})). \\ sup_{j,i}(HId, \overline{V}, I1, A) & : - \ sup1_{j,i}(HId, \overline{V}, G, A), \text{goal\_id}(G, I1). \end{array}$$

   Call these rules *Type 2* rules.

3. If the body of R_j is empty generate the rule:

$$A{:}-sup_{j,0}(HId, \overline{V}, \_, A).$$

   Call such rules *Type 3* rules.

4. For each literal $p_i(\overline{t_i})$ in the body of R_j other than the last literal, generate a rule

$$query(p_i(\overline{t_i}), ID1, answer(ID1, p_i(\overline{t_i}))){:}-sup_{j,i-1}(HId, \overline{V}, ID1).$$

   Call such rules *Type 4* rules.

5. If the body of R_j is non-empty, generate the following rule:

$$query(p_n(\overline{t_n}), ID1, A){:}-sup_{j,n-1}(HId, \overline{V}, ID1, A).$$

   Call such rules *Type 5* rules.

$$\begin{aligned}
Q_{R1}: &\ initial\_query(append([1,2,3],[4],X), ID, append([1,2,3],[4],X)) :- \\
&\qquad\qquad\qquad\qquad \mathsf{goal\_id}(append([1,2,3],[4],X), ID).\\
Q_{R2}: &\ query(Q, ID, Ans) :- initial\_query(Q, ID, Ans).\\
Q_{R3}: &\ append(X1, X2, X3) :- initial\_query(\_, ID, \_), answer(ID, append(X1, X2, X3)).
\end{aligned}$$

$$\begin{aligned}
S1.0: &\ sup_{1,0}(HId, X, 0, A) :- query(append([], X, X), HId, A).\\
R1': &\ A \qquad\qquad\qquad :- sup_{1,0}(HId, X, ID, A).
\end{aligned}$$

$$\begin{aligned}
S2.0': &\ sup1_{2,0}(HId, H, T, L, L1, append(T, L, L1), A) :- query(append([H|T], L, [H|L1]), HId, A).\\
S2.0: &\ sup_{2,0}(HId, H, T, L, L1, ID, A) :- sup1_{2,0}(HId, H, T, L, L1, G, A), \mathsf{goal\_id}(G, ID).\\
Q2.1: &\ query(append(T, L, L1), ID, A) :- sup_{2,0}(HId, H, T, L, L1, ID, A).
\end{aligned}$$

Figure 5: MGU MTTR Rewriting of the *append* Program

For each base predicate b_i used in the program generate a rule:

A: $-query(b_i(\overline{X_i}), ID, A), b_i(\overline{X_i})$.

where $\overline{X_i}$ is a tuple of distinct variables, with arity equal to that of b_i.

Call such rules *Type 6* rules.

Rule Q_{R1} generates an *initial_query* fact corresponding to the initial query on the program. This fact is used to generate a *query* fact using rule Q_{R2}. Rule Q_{R3} selects answer facts that are answers to the initial query. Note that since the *id* field of the answer matches the *id* field of the initial query, there is no need to actually unify the answer and query arguments.

Example 3.4.1 The append program is defined as follows.

$R1 : append([], X, X)$.

$R2 : append([H|T], L, [H|L1]) :- append(T, L, L1)$.

Suppose the given query is $?append([1,2,3],[4],X)$. The rewritten program is shown in Figure 5. The first three rules in the rewritten program are Type 0 rules. Rule $S1.0$ is a Type 1 rule generated from rule $R1$, while $R1'$ is a Type 3 rule generated from rule $R1$. Rules $S2.0'$ and $S2.0$ are Type 1 rules generated from $R2$, and rule $Q2.1$ is a Type 5 rule generated from $R2$.

In Section 3.4.1 we discuss some optimizations that simplify the rewritten program, and in Example 3.4.2 we discuss the evaluation of the optimized rewritten program. □

3.4.1 Optimizations of MGU MTTR Rewriting

As in the case of MGU Magic rewriting, we can optimize MGU MTTR rewriting in several ways. For simplicity, our proofs are for the version of MGU MTTR rewriting without these optimizations.

We can choose to treat some literals as non-tail-recursive, even though they appear as the last literal in the rule. The changes to the rewriting are fairly straightforward. An alternative way of ensuring that the last literal in a rule is treated in a non-tail-recursive fashion is by introducing an extra literal *true*() at the end of the rule, and adding $true()$ as a rule (with an empty body). This extra literal does not significantly change the number of derivations made.

Some of the optimizations described in Section 3.3.1 are applicable to MGU MTTR rewriting. For instance, we can project out variables from supplementary literals as described in that section.

We can treat base predicates specially in the rewriting, by applying the following transformation to the rewritten program, for each base predicate b_i. First, each literal $answer(ID, b_i(\overline{t_i}))$ where b_i is a base predicate is replaced by the literal $b_i(\overline{t_i})$. Next rules of the form:

$$sup1_{j,i-1}(HId, \overline{V}, b_i(\overline{t_i}), A) \colon -sup_{j,i-2}(HId, \overline{V}, I1, A), answer(I1, p_{i-1}(\overline{t_{i-1}})).$$

$$sup_{j,i-1}(HId, \overline{V}, I1, A) \colon -sup1_{j,i-1}(HId, \overline{V}, G, A), \mathsf{goal\_id}(G, I1).$$

are replaced by the rule:

$$sup_{j,i-1}(HId, \overline{V}, 0, A) : -\ sup_{j,i-2}(HId, \overline{V}, I1, A), answer(I1, p_{i-1}(\overline{t_{i-1}})).$$

All Type 4 rules that generate queries on b_i are deleted. (Note that any fact $answer(id, p_i(\overline{t_i}))$ must be generated from a fact for base predicate p_i. For any derivation made using the *answer* fact, there is an equivalent derivation made using the new rule with the original fact for p_i.) If there is no Type 5 rule that generates a query on b_i, we delete the Type 6 rule that generates answer facts for b_i. We can ensure that there is no Type 5 rule that generates a query on any base literal b_i by treating all occurrences of base literals as non-tail-recursive.

The query rules that are removed are Type 4 rules. Such rules generate queries of the form

$$query(p_i(\overline{t}), ID, answer(ID, p_i(\overline{t})))$$

that result in facts of the form $answer(id, p_i(\ldots))$ being generated; such rules are not useful once the *answer* predicate is replaced by the base predicate p_i. This optimization is not applicable for Type 5 rules, since queries generated by such rules do not generate answers for the base predicate.

Projecting out extra variables from the supplementary predicates can be done as described in Section 3.3.1.

We can simplify the set of rules Q_{R1}, Q_{R2} and Q_{R3} by generating a query whose second argument is of the form $q(\overline{t})$ rather than $answer(ID, q(\overline{t}))$. Thus answers to the original query on the program get generated directly. The *initial_query* predicate and the rule Q_{R3} are used only for generating answers of the form $q(\overline{a})$ for the original query on the program, from facts of the form $answer(id, q(\overline{a}))$. We can therefore drop Q_{R3}. We merge the rules Q_{R1} and Q_{R2} to get the following rule:

$$Q'_{R1} : query(q(\overline{t}), ID, q(\overline{t})) \colon -\mathsf{goal\_id}(q(\overline{t}), ID).$$

3.4.2 An Example

We now consider an example of MGU MTTR rewriting with some of the optimizations described above.

Example 3.4.2 Consider the append program from Example 3.4.1.

$R1 : append([], X, X).$

$R2 : append([H|T], L, [H|L1]) : -\ append(T, L, L1).$

$$\begin{array}{ll} Q'_{R1}: & query(append([1,2,3],[4],X), ID, append([1,2,3],[4],X)) :- \\ & \quad \mathsf{goal\_id}(append([1,2,3],[4],X), ID). \\ & \\ S1.0: & sup_{1,0}(HId, X, 0, A) :- query(append([], X, X), HId, A). \\ R1': & A :- sup_{1,0}(HId, X, ID, A). \\ & \\ S2.0': & sup1_{2,0}(HId, H, T, L, L1, append(T, L, L1), A) :- query(append([H|T], L, [H|L1]), HId, A). \\ S2.0: & sup_{2,0}(HId, H, T, L, L1, ID, A) :- sup1_{2,0}(HId, H, T, L, L1, G, A), \mathsf{goal\_id}(G, ID). \\ Q2.1: & query(append(T, L, L1), ID, A) :- sup_{2,0}(HId, H, T, L, L1, ID, A). \end{array}$$

Figure 6: Optimized MGU MTTR Rewritten version of the *append* program

Suppose the given query is $?append([1,2,3],[4],X)$. The optimized MGU MTTR rewritten program is shown in Figure 6. The main difference between the optimized rewritten program and the MGU MTTR rewritten program generated in Example 3.4.1 is that the initial query rules Q_{R1}, Q_{R2} and Q_{R3} have been replaced by Q'_{R1}.

In the Semi-Naive evaluation of the above rewritten program, rule Q'_{R1} generates a fact

$$query(append([1,2,3],[4],X), 0, append([1,2,3],[4],X))$$

(which corresponds to the given query on the program). The last argument of this fact is the fact to be instantiated and generated as an answer to the query on the program.

Rules $S2.0'$ and $S2.0$ generate a supplementary fact containing variable bindings and the identifier for a query on the *append* literal in the body of rule $R2$. $Q2.1$ generates the actual query fact using this identifier. Thus after three iterations, a query fact

$$query(append([2,3],[4],X), 1, append([1,2,3],[4],[1|X]))$$

is generated. Three iterations later

$$query(append([3],[4],X), 2, append([1,2,3],[4],[1,2|X]))$$

is generated.

$$query(append([],[4],X), 3, append([1,2,3],[4],[1,2,3|X]))$$

is then generated. This fact is used with rules $S1.0$ and $R1'$; X gets bound to $[4]$, and a fact

$$append([1,2,3],[4],[1,2,3,4])$$

is generated. This completes the evaluation of the program.

In this particular example, the goal identifiers stored with the facts are not of particular use. They are useful if there is a supplementary rule that unifies supplementary facts with answer facts, as is the case for derived literals that are not the last literal in the body of a rule. □

3.4.3 Correctness of MGU MTTR Rewriting

For simplicity, we prove correctness with respect to the unoptimized version of MGU MTTR rewriting. We define the following property of the evaluation of P^{MGU_T}, and prove it as an intermediate step in proving correctness.

Property 3.4.1 (MGU_T-Prop) Let P be any program and Q a query on P. We say that an evaluation of $P_Q^{MGU\_T}$ has property MGU_T-Prop if

1. Every fact $answer(id, a)$ generated in the evaluation is such that a is an mgu-answer to a subgoal with identifier id.
2. Every fact $query(p(\overline{a}), id1, answer(id2, q(\overline{b})))$ generated in the evaluation is such that (a) $?p(\overline{a})$ is an mgu-subgoal, and $id1$ is the identifier of $?p(\overline{a})$, and (b) if $p(\overline{a'})$ is an mgu-answer to the subgoal $?p(\overline{a})$ (wlog assume that $\overline{a}$ and $\overline{b}$ share no variables with $\overline{a'}$), and $\theta = mgu(p(\overline{a}), p(\overline{a'}))$, then $q(\overline{b})[\theta]$ is an mgu-answer to the subgoal with identifier $id2$. □

Theorem 3.4.1 *Consider any program P with query Q. Then a bottom-up evaluation of P^{MGU_T} has property MGU_T-Prop.* □

The above theorem shows that bottom-up evaluation of P^{MGU_T} is sound, and generates only mgu-answers to mgu-subgoals. The proof is by induction on derivation sequences for P^{MGU_T}, and is presented in the Appendix A.2. The following theorem shows completeness of bottom-up evaluation of P^{MGU_T} with respect to the query on the program. The proof of the theorem may be found in Appendix A.2.

Theorem 3.4.2 *Given any program P and query Q, the bottom-up evaluation of $P_Q^{MGU\_T}$ is complete with respect to Q, i.e., if the bottom-up evaluation of P generates a fact p that is an answer to Q, then p is subsumed by a fact computed in the bottom-up evaluation of $P_Q^{MGU\_T}$.* □

To summarize this section, we have shown the following. Given a program P and a query Q, the bottom-up evaluation of $P_Q^{MGU\_T}$ is sound, generates all answers to query Q on P, and further, the evaluation of $P_Q^{MGU\_T}$ generates only mgu-subgoals and mgu-answers. Finally, we note that $P_Q^{MGU\_T}$ performs tail-recursion optimization in the same fashion as P^T.

Chapter 4

Bottom-up vs. Prolog* — A High Level Comparison

In this chapter, we first present a cost model, which we call Prolog*, of Prolog evaluation of a query, and a model of a Semi-Naive evaluation of a program. We then use these models to perform a high level comparison of Prolog* with bottom-up evaluation using MGU MTTR rewriting. The comparison is at the level of "actions", and applies to all definite clause programs. We also use these models in Chapter 5 to perform a more detailed comparison (at the level of time complexity) of Prolog* with bottom-up evaluation using MGU MTTR rewriting.

The chapter is organized as follows. We present our model of Prolog evaluation in Section 4.1. This model accounts for tail-recursion optimization. In Section 4.2 we present a model for Semi-Naive evaluation. This model helps reduce the actions in Semi-Naive evaluation of a program to a series of attempted derivations. In Section 4.3 we use this model to compare the Semi-Naive evaluation of the MGU MTTR rewritten program with Prolog* query evaluation.

In our models, we consider only definite clause programs, which do not have negated literals in the bodies of rules.[1]

4.1 A Model for Prolog Evaluation

In this section we present a cost model for Prolog evaluation of definite clause logic programs, in order to make precise the comparison of bottom-up evaluation and Prolog that we make in later sections. Since there is no such thing as a "standard implementation" of Prolog, we define what we mean by a Prolog evaluation. The formal model for Prolog computation is a depth-first exploration of the SLD tree for the query on the program (see e.g. [Llo87]). However, the SLD tree model leaves some important aspects of the evaluation unspecified. For instance, it does not specify if tail-recursion optimization is used or not.

We present the Prolog* cost model of Prolog query evaluation below. The main purpose of this cost model is to provide a lower bound on the cost of Prolog evaluation. Hence we take the liberty of ignoring

[1]We can extend this class to cover certain restricted forms of negation such as modularly stratified negation, by using extended bottom-up evaluation techniques such as Ordered Search [RSS92a]. Details are beyond the scope of this thesis.

details that are not critical for our cost analysis. Prolog* is intended to model Prolog evaluation with tail recursion optimization, but without using any other optimizations that affect the number of subgoals set up, or the number of answers generated.[2]

If a Prolog* evaluation is not complete, or does not terminate, bottom-up evaluation can certainly do no worse. Hence, we only consider Prolog* evaluations that terminate and are complete. This also has the benefit of simplifying our proofs considerably. The Prolog* model of subgoal evaluation ignores many details of control flow. In particular, the depth-first search strategy used by Prolog is not reflected in the model.

Each subgoal g set up in Prolog* evaluation has a "return-point" r that is either the subgoal itself, or an ancestor of the subgoal. The subgoal r is referred to as the *return-point subgoal* for g. The "return-point" indicates to which subgoal control must return when an answer is generated for the subgoal, and is used to implement tail-recursion optimization (Sections 2.3 and 4.1.1). (In case of failure to generate an answer, control does not return to the return-point subgoal; rather the backtracking mechanism decides which goal to retry. The details of control are irrelevant to our model, and we ignore them.) In the course of generating an answer to a return-point subgoal r, the subgoal is progressively instantiated. Thus when some subgoal g' is generated, such that the return-point of g' is r, the variables in r have been instantiated. Let the instantiated version of r at the point when g' is generated be r'. Then r' is said to be the *instantiated return-point subgoal* of g'.

As a base case, the return-point as well as the instantiated return-point subgoal of the initial query on the program are defined to be the initial query itself. If a subgoal g is generated from a literal other than the last literal in the rule, both its return-point and its instantiated return-point subgoal are set to g. If a subgoal g is generated from the last literal of a rule, its return-point is defined to be the return-point of the subgoal on the head of the rule. Let the return-point subgoal of g be r. The instantiated return-point subgoal of g is defined to be the instantiation of r at the point when g was generated.

Definition 4.1.1 (Prolog*) We define the *Prolog** cost model of query evaluation through the procedure "Prolog* Evaluation of a Subgoal" shown below. We assume that Prolog* evaluation proceeds till all answers are generated (i.e., Prolog* does not stop at the request of the user), and that Prolog* evaluation terminates and is sound.

The return-point as well as the instantiated return-point subgoal of the initial query on the program are defined to be the initial query.

Prolog* Evaluation of a Subgoal:

Suppose we have a subgoal $g = ?p(\bar{t})$, with instantiated return point subgoal gr, and a set of rules defining p. For each rule R defining p, Prolog* does the following:

Let rule R be of the following form:

$$R : p(\overline{t_0}) :- q_1(\overline{t_1}), q_2(\overline{t_2}), \ldots, q_n(\overline{t_n}).$$

1. Prolog* first attempts to unify the subgoal g with the head of R. If the unification fails, the attempt to solve g using R fails.

   If unification succeeds, let θ be the most general unifier of g and $p(\overline{t_0})$.

[2]For instance, we disallow Intelligent Backtracking (see eg. [CD85]).

2. If the body of the rule is empty, Prolog* evaluation returns an answer $gr[\theta]$ to the return-point of g.

   Otherwise, $next\_literal$ is set to the first literal in the body.

3. Prolog* evaluation generates a subgoal $next\_literal[\theta]$.

   The return-point and the instantiated return-point subgoal of the generated subgoal are set to the subgoal itself, if $next\_literal$ is not the last literal in the body of the rule, or if tail-recursion optimization is not used for the last literal.

   Otherwise, the return point of the subgoal is set to the return point of g, and the instantiated return point subgoal is set to $gr[\theta]$.

   (Conceptually, in the first case, θ is saved at this point, to be used when an answer is returned for the subgoal. In practice in Prolog, the use of depth-first search with backtracking implies that we do not have to physically save θ. We do not assign any cost to this conceptual "saving" of θ.)

4. Prolog* then computes answers to the subgoal.

   If $next\_literal$ is not the last literal in the body of the rule, or if tail-recursion optimization is not used for the literal, answers are returned for the generated subgoal.

5. For each answer a returned for a subgoal $l[\theta]$ on a literal l, Prolog* evaluation does the following:

   Conceptually, a must be unified with $l[\theta]$, where θ is the binding saved (conceptually) in Step 3. The unification is done implicitly by Prolog evaluation when generating the answer a. (We do not assign any cost to this conceptual unification.) Let θ' be the mgu of a and $l[\theta]$.

   If there are more literals in the rule body, $next\_literal$ is set to the next literal in the rule body, and θ is set to θ'. Computation proceeds as in Step 3.

   If there are no more literals in the body of the rule, the return-point subgoal has been solved. $gr[\theta]$ is returned as an answer to the return-point of g.

□

The above model is a simplified description of Prolog evaluation, and omits many details such as how control flow is directed. The details of control flow are important as far as the actual costs of Prolog evaluation are concerned. However, since our goal is to obtain a lower bound for the cost of Prolog evaluation, and the cost of implementing control flow is not counted in our model, we can ignore the details. We can get the depth-first search strategy used by Prolog by using the procedure describing subgoal evaluation as a coroutine. Control passes to the procedure whenever a (new) answer is required for the subgoal, and the procedure returns control to its calling point when an answer is generated, and also when no (more) answers can be generated.

The important point to note is that each of the steps described is assumed to take $\Omega(1)$ time, and the computation can be viewed as a sequence of such steps. We refer to each of these steps in a Prolog* evaluation as an *action* performed in the evaluation. We often view the step where an answer is returned to a return-point subgoal as two actions — the first action being the "generation of an answer" and the second action being the return of the answer to the point where the subgoal was generated. This view is valid, since we assign $\Omega(1)$ cost to each step, and the sum of their costs is still $\Omega(1)$.

The above model is used in Section 4.3 to show that Prolog* evaluation of a query performs at least as many actions as the number of attempted derivations made by bottom-up evaluation of the MGU MTTR rewriting of the program and the query.

To incorporate the time cost of evaluation into the comparison, we need to assign costs to each step. We assume that all the steps above, except the unification of a query with a rule head in Step 1, take unit time. (This assumption provides a lower bound on their cost.) Step 1 performs a unification, and may take more than $O(1)$ time; when comparing Prolog* evaluation with bottom-up evaluation, we show that for each unification action performed by bottom-up evaluation of an MGU MTTR rewritten program, Prolog* evaluation performs a corresponding unification.

4.1.1 Tail Recursion Optimization

Tail-recursion optimization and its benefits are described in Section 2.3. In many implementations of Prolog, tail-recursion optimization would be performed only if the last literal in a rule was recursive with the head, since this is case where it offers the maximum benefit. We assume for simplicity that it is done always. Further, in many implementations of Prolog, tail-recursion optimization is actually done only under more stringent conditions, when the space allocated for the call to R can be deallocated. Such an optimization, although often loosely associated with tail-recursion optimization, is better termed *last-call optimization* [MW88]. Last-call optimization helps reduce space utilization; however, we concentrate on time utilization in this thesis, and do not take the space savings into account in our model.

4.2 A Model for Semi-Naive and Not-So-Naive Evaluation

We now consider a model for the actions in Semi-Naive and Not-So-Naive bottom-up evaluation of an MGU MTTR rewritten program (or an MGU Magic Rewritten program). We assume that the body of each rule has at most two literals. Rules in MGU Magic and MGU MTTR rewritten programs are in this form. We use the term *evaluable predicate* to refer to a base predicate whose set of facts is not stored explicitly, but is computed using imperative code.

Let us denote the rewritten program as P^{MGU_T}. Step 1 of Algorithm SN_Iterate (described in Section 2.2.3) derives facts using rules that have no derived predicates in their bodies. There is only one such rule, Q_{R1}, and it performs only one derivation.

We model the actions performed in Step 2.1 of Algorithm SN_Iterate as a sequence of uses of derived facts to derive other facts, as shown below.

As noted in Section 2.2.3, we assume that a left-to-right nested-loops join with indexing is used to evaluate Semi-Naive rewritten rules. Consider a call Apply (R_s, I) in Step 2.1 of Algorithm SN_Iterate. Procedure Apply performs a nested-loops join. Procedure Make_Inferences$(R_s, p_i(\overline{a_i}))$, shown below, is a model of the actions in a single iteration of the outer loop of the nested-loops join. The model also incorporates subsumption-checking actions.

Make_Inferences$(R, p_i(\overline{a_i}))$.

1. Standardize apart $p_i(\overline{a_i})$ from R, i.e., make variable names in $p_i(\overline{a_i})$ distinct from those in R by renaming variables if required.[3]
2. Compute an mgu θ_1 of (the renamed version of) $p_i(\overline{a_i})$ with the first literal in the body of (the renamed version of) R.
3. If R has only one body literal, set θ_3 to θ_1.
   /* Else R has two body literals */
   Else perform the following actions:
   (a) Let the second literal of (the renamed version of) R be $p_j(\overline{t_j})$. Index the relation p_j, to fetch facts that unify with $p_j(\overline{t_j})[\theta_1]$ (in case the predicate is an evaluable predicate, a query is set up and evaluated instead).
   (b) When each fact is fetched, standardize it apart from the (renamed versions of) R and $p_i(\overline{a_i})$.
   (c) Compute an mgu θ_2 of the fetched fact and $p_j(\overline{t_j})[\theta_1]$. Let $\theta_3 = \theta_1[\theta_2]$.
4. Let the head literal of (the renamed version of) R be $p(\overline{t})$. Create a fact $p(\overline{t})[\theta_3]$ for each mgu θ_2 as above.
5. Check if $p(\overline{t})[\theta_3]$ is subsumed by previously generated facts for p. If it is not subsumed, discard all p facts that are subsumed by it, insert it into the p relation, and mark it as a newly derived fact. (In the case of Not-So-Naive evaluation, the subsumption check is omitted, and the fact is inserted into the relation even if it is subsumed.) The newly derived fact $p(\overline{t})[\theta_3]$ is not used for making inferences until the next iteration.

We split the computation described above into 'attempted derivation steps', which we define below. This lets us allocate the cost of evaluation to different attempted derivation steps. We split attempted derivation steps into two cases, depending on whether the derivation is successful or unsuccessful. In the case where successful derivations are made using a fact, we split the computation into 'successful derivation steps', one for successful derivation. In the case that no successful derivation is made using a fact, we have an 'unsuccessful derivation step'. We define these formally below.

Definition 4.2.1 (Derivation Steps) Consider a call Make_Inferences$(R, p_i(\overline{a_i}))$. The following actions are performed in the call.

Steps 1 and 2 of Make_Inferences attempt to unify the fact $p_i(\overline{a_i})$ with the literal $p_i(\overline{t_i})$.

1. If the unification in Step 2 of Make_Inferences fails, the actions performed by Steps 1 and 2 with the given fact $p_i(\overline{a_i})$ constitute an *unsuccessful derivation step.*
2. If the unification in Step 2 of Make_Inferences succeeds, and rule R has two body literals, the other literal in the body of R is indexed.
   (a) If no fact is fetched by the indexing, the actions performed by Steps 1, 2 and 3a with the given fact $p_i(\overline{a_i})$ constitute an *unsuccessful derivation step.*

[3]We can standardize apart R and $p_i(\overline{a_i})$ by renaming one of them. We do not specify which one.

(b) If facts are fetched by the indexing, for each fetched fact $p_j(\overline{a_j})$, Steps 3b, 3c, 4 and 5 are performed.

The actions performed in Steps 1, 2 and 3a with fact $p_i(\overline{a_i})$, and the actions in Steps 3b, 3c, 4 and 5 with facts $p_i(\overline{a_i})$ and $p_j(\overline{a_j})$ constitute a *successful derivation step.*

3. If the unification in Step 2 of Make_Inferences succeeds, and rule R has only one body literal, a head fact is created, and inserted into the appropriate relation. The actions performed in Steps 1, 2, 4 and 5 with the given fact $p_i(\overline{a_i})$ constitute a *successful derivation step.*

A successful derivation step in an SN evaluation can be identified[4] by the rule R used in the step, and the fact used for each body literal of R. Each successful derivation step has associated with it the *fact derived by* the derivation step.

An *unsuccessful derivation step* in the evaluation can be identified by the rule R used in the step, and the fact $p_i(\overline{a_i})$.

An *attempted derivation step* is either a successful or an unsuccessful derivation step.

In the case of NSN evaluation, we assume that facts are labeled with integers.[5] We extend the definitions above, to the case where the facts used in the derivation steps are labeled facts, and thereby define *labeled successful derivation steps*, *labeled unsuccessful derivation steps*, and *labeled attempted derivation steps.* We then identify[6] successful and unsuccessful derivation steps as above, but using labeled facts. □

For all Semi-Naive rewritten rules other than the rule Q_{R1}, the first literal in the rule is a literal of the form δp. Hence, any fact used for such a literal must be 'newly derived' (i.e., derived in the previous iteration), and all derivation steps (other than those involving rule Q_{R1}) use a 'newly derived' fact for the first literal.

The actions performed in Steps 1, 2 and 3a may be identified with several successful derivation steps, and may hence be double counted. When counting the cost of evaluation (in Section 5.7) we recognize this, and avoid double counting.

The definitions of derivation steps above are in terms of rules in the Semi-Naive rewritten version of P^{MGU_T}. We often talk of derivation steps using rules from P^{MGU_T} rather than from the Semi-Naive rewritten version of P^{MGU_T}. Whenever we do so, we specify which literal is used first in the derivation step, and this uniquely identifies which Semi-Naive rewritten version of the rule is used.

All the actions in making inferences, given a fact for the first literal in a Semi-Naive rewritten rule, have been allocated to attempted derivation steps as described above. We still have to account for two other kinds of actions in Semi-Naive evaluation: (a) checking if there is a fact for the first literal in a Semi-Naive rewritten rule — this is done once per rule in each iteration and (b) the Semi-Naive update steps, executed once in each iteration. We now consider how to map the costs of these actions to attempted derivation steps, so that we need not consider the costs of these actions in the rest of the thesis.

1. The cost of checking if a rule can be used in Step 2.1 of SN_Iterate in an iteration is $O(1)$. Since we assumed that the number of rules is a constant, we map this cost to the cost of the derivation of

[4]Uniquely, as we shall show.

[5]The integer labels are used to distinguish repeated occurrences of a fact, since subsumption-checking is not performed.

[6]Uniquely, as we shall show.

some (labeled) fact in some relation of the form δp^{old}. (There must be some such fact, else Semi-Naive evaluation would have terminated after the previous iteration.) Since any such fact is present in the δ relation for at most one iteration, the cost of the derivation does not increase by more than a constant, and the increase can be ignored.

2. Subsumption checking actions have been accounted for by the model above. Apart from subsumption-checking, Semi-Naive update actions move facts from one relation to another. This is done at most a constant number of times per fact, and we assume this can be done at unit cost per fact (which is reasonable, assuming hash-based indices are used). We map the cost of moving a fact between relations to the derivation step that derived the fact; this does not increase the cost of the derivation step by more than a constant, and can be ignored.

Thus the cost of Semi-Naive evaluation is at most a constant times the cost of attempted derivation steps (assuming that the size of the program is a constant). In the rest of this thesis, we shall treat the cost of attempted derivation steps as synonymous with the cost of Semi-Naive evaluation.

4.2.1 The Non-Repetition Property

Due to Semi-Naive rewriting, no attempted derivation step is repeated within an iteration. Every attempted derivation uses at least one fact from a δ relation. The Semi-Naive updates ensure that each fact is in a δ relation for precisely one iteration. Hence, Semi-Naive evaluation has the property that no attempted derivation step is repeated in the evaluation. We call this property of Semi-Naive evaluation the non-repetition property (see e.g., [MR89, RSS90]).

Not-So-Naive evaluation has a weaker non-repetition property. Each fact can have several occurrences, derived by different successful derivation steps. In the case of Not-So-Naive evaluation, we give an identifier to each successful derivation step, and label each occurrence of a fact with the identifier of the derivation step that derived it. There can thus be two or more occurrences of a fact, but each occurrence has a distinct label. Each labeled fact is in a δ relation for exactly one iteration. Not-So-Naive evaluation has the property that no labeled attempted derivation step is repeated in the evaluation. We formalize this property through the following theorem.

Theorem 4.2.1 (Non-Repetition) Consider a Semi-Naive (resp. Not-So-Naive) evaluation of a program. No attempted derivation step (resp. labeled attempted derivation step) is repeated in the evaluation. □

4.2.2 Discussion

We made the assumption above that the size of the program is a constant; we do not take the size of the program into account in our time complexity analysis, even though it may contribute to the cost of evaluation. There are essentially two places where this assumption is used. First, each iteration of Semi-Naive evaluation applies all the rules, but may find the δ relations empty for all but one rule. To keep the number of rule applications proportional to the number of attempted derivations, and independent of the number of rules in the program, we can devise a rule indexing scheme. We discuss the rule indexing scheme briefly in Section 5.9. Second, there may be many relations, but semi-naive updates may be required only for a few of them, if only a few facts are derived in each iteration. To avoid the cost of checking which relations need to be updated, we keep track of which δ relations are non-empty, and perform Semi-Naive updates only for these relations. Thus the cost of updates can be kept proportional to the number of facts derived, and independent of the number of predicates in the program.

4.3 Bottom-Up Evaluation vs. Prolog* — Number of Inferences

In this section we present a high-level comparison (based on the number of "actions" performed) of bottom-up evaluation using MGU MTTR rewriting with Prolog* evaluation. In Chapter 5 (Section 5.7), we extend this comparison by taking into account the cost of each action.

The comparison is performed essentially by mapping each attempted derivation in bottom-up evaluation to a corresponding action of Prolog* evaluation, and showing that not more than a constant number of attempted derivations map on to the same Prolog* action. We prove this by showing how to construct such a mapping, given a derivation sequence for the MGU MTTR rewritten program.

In order to specify the mapping, we assume that each attempted derivation step has a unique identifier, and we label facts derived by SN evaluation with the identifier of the derivation step that generated the fact. In a similar fashion, we label actions (such as generation of a query or answer) performed by Prolog* in order to distinguish between multiple occurrences of the action. Thus the mapping is in terms of labeled derivation steps; if a derivation step is repeated (as is the case if subsumption-checking is not performed), each repetition of the step uses facts with different labels, due to the non-repetition property of NSN evaluation.

The mapping is somewhat intricate, and we build it up inductively. We assume that we have a mapping with the required properties for an initial part of a derivation sequence, and show how to extend it in a manner such that the required properties are preserved. We present details of the mapping in Appendix B. The mapping is defined using a case analysis on the types of rules in the rewritten program. The mapping for Type 2 rules of the form

$$sup1_{j,i}(\ldots){:}-sup_{j,i-1}(\ldots), answer(ID, p(\ldots)).$$

is the critical part of the mapping. We show that for each successful derivation using such a rule with some facts $sup_{j,i-1}(hid, \overline{v}, id, ans)$ and $answer(id, p(\overline{a}))$, (1) Prolog* evaluation returns an answer $p(\overline{a})$ for a query on the ith literal of rule R_j, and (2) the bindings of rule variables in the Prolog* evaluation when the query was generated are the same as the bindings stored in $\overline{v}$, and (3) the (instantiated) return-point query in Prolog* evaluation, when the query on the i'th literal was generated, is equal to ans. We then have the following theorem, whose proof is presented in Appendix B.

Theorem 4.3.1 *Let P be a definite clause program, and Q be a query on the program. There are constants c_1 and c_2 (that may depend on the size of P) such that the following is satisfied.*

Let P^{MGU_T} be the MGU MTTR rewriting of $\langle P, Q\rangle$. Given any database, let the number of labeled attempted derivation steps performed by a Semi-Naive evaluation (with or without subsumption checking) of P^{MGU_T} be n, and let the number of actions performed by Prolog evaluation of query Q with the same database be m. Then $n < c_1 \cdot m + c_2$.* $\Box$

There may be many actions of Prolog* evaluation that are not in the image of any attempted derivation step of bottom-up evaluation. Thus, the theorem helps establish an upper bound on how much worse (in terms of number of actions) Semi-Naive evaluation using MGU MTTR rewriting can be compared to Prolog* evaluation. In contrast, no such bound exists for the opposite direction. For queries on the following simple program to detect reachability in a graph, Prolog* may not terminate if there are cycles in the *edge* relation, whereas Semi-Naive evaluation of P_{MGU_T} always terminates if the *edge* relation is finite.

$reachable(X,Y) :- edge(X,Z), reachable(Z,Y).$
$reachable(X,Y) :- edge(X,Y).$

Chapter 5

Evaluation of Non-Range-Restricted Programs

In this chapter we consider the efficient evaluation of programs that have been rewritten using MGU MTTR rewriting. In Section 5.1 we motivate the need for bottom-up evaluation of programs that generate non-ground facts; to the best of our knowledge, no efficient bottom-up evaluation schemes for such programs were known in the past. We develop a term representation using "persistent versioned" binding environments (Section 5.3). We then develop an evaluation technique that keeps extra information with facts, and uses this information to reduce the cost of some unifications (Section 5.4).

The evaluation technique we develop is quite efficient in terms of time complexity. We show in Section 5.7 that given a program and a query, if Prolog* evaluation of the query on a database takes time t, the bottom-up evaluation of the MGU MTTR rewritten program using our evaluation technique, on the given database, without subsumption checking, takes time $O(t \cdot \log\log t)$. (The size of the program is not taken into account in the time complexity measure.) If subsumption checking is used, in many cases bottom-up evaluation is much more efficient than Prolog query evaluation. We discuss some extensions of our techniques in Section 5.9. In Section 5.10 we summarize earlier results of ours on the evaluation of a restricted class of programs that have been rewritten using MTTR rewriting. We discuss related work in Section 5.11

5.1 Introduction

Programs that generate non-ground terms are of considerable importance. For instance, difference-lists[1] are used in Prolog to append lists in constant time; they are an instance of a more general technique for creating data structures top-down, filling in "holes" as computation proceeds. One such application is the parsing of Definite Clause Grammars — it is natural to create the outer structure of the parse tree, and fill in fields as computation proceeds. By doing so, attributes of one part of the parse tree are available for reference when another part of the tree is being constructed. Meta-interpreters, partial evaluators, abstract interpreters and other such programs operate on data structures that contain variables. Memoing of goals and answers is very

[1]Difference lists are a form of representation of lists. $dlist(L, X)$ is such that L is a list that contains variable X at the end, rather than "nil". See Example 5.1.1 for a more detailed description.

important for some of these programs [War92]. For instance, chart parsing of DCGs is naturally supported using memoing, and can be much more efficient than top-down parsing in some situations. It is therefore necessary to support efficient memoing evaluation of programs that generate non-ground facts.

In the absence of non-ground facts, bottom-up evaluation using MGU MTTR rewriting is as fast (up to a data-independent constant factor) as Prolog, as discussed in Sudarshan and Ramakrishnan [SR92b] (see Section 5.10). However, for logic programs that can generate non-ground facts there is a significant overhead per inference for unoptimized bottom-up evaluation techniques; the overhead is no longer within a constant factor of Prolog.

In this chapter we present an efficient evaluation mechanism for programs that have been rewritten using MGU MTTR rewriting. This evaluation mechanism is of particular use for programs that generate non-ground facts. We call this evaluation mechanism *Opt-NG-SN* evaluation (which stands for Optimized NonGround Semi-Naive evaluation). We use the term *Opt-NGBU* query evaluation for the query evaluation mechanism that first rewrites programs using MGU MTTR rewriting, and evaluates the rewritten program using Opt-NG-SN evaluation.

Using the Opt-NGBU evaluation mechanism, we show the following: modulo the cost of subsumption-checking, the overhead of actually memoing goals and facts and "looking up" the answers that correspond to a given goal is quite small ($O(\log\log m)$, where m is bounded by the cost of Prolog evaluation). (To be more precise, we show that given a program and a query, if Prolog evaluation of the query on a database takes time t, then Opt-NGBU query evaluation on the given database, without subsumption-checking, takes time $O(t \cdot \log\log t)$. The size of the program is not taken into account in the time complexity measure.) In essence, our results show how to memo non-ground facts "almost" as efficiently as ground facts. An important consequence is that memoing techniques can always perform "almost" as efficiently as Prolog (often, much better). This result assumes that the size of the program (but not the database) is a constant.

Checking whether a goal (similarly, an answer) is already memoed can be expensive. The cost of subsumption-checking must be balanced against the cost of recomputation. (See Section 5.8 for a more complete discussion.) Opt-NGBU evaluation can be faster if some redundant computation is avoided through subsumption-checks; in fact, the time complexity of evaluation may be significantly better. There are many programs where Prolog evaluation and Opt-NGBU evaluation without subsumption-checking run for ever, but Opt-NGBU evaluation with subsumption-checking terminates.

In the light of these results, the biggest difference between Prolog and Opt-NGBU evaluation without subsumption-checking is that between "*pipelining*" and "*materialization*" [CGK89]. The constants are higher for memoing since facts and goals are explicitly created and stored, but there are benefits in many cases due to avoided recomputation.

However, we note that even *without* subsumption-checking, Opt-NGBU evaluation has several benefits. Opt-NGBU evaluation is complete for definite clause programs — including programs with function symbols — even without subsumption-checking. Further, much of the repeated computation in iterative deepening (a technique used to make Prolog evaluation complete; see, e.g. [O'K90]) is avoided, even without subsumption-checking.

As an example of the power of the optimized evaluation technique, the bottom-up evaluation of append with non-ground lists of length n is performed in time $O(n)$ by our optimized techniques, as against $O(n^2)$ by unoptimized bottom-up query evaluation. Top-down evaluation techniques that perform memoization, e.g. QSQR extended to deal with non-ground terms, are also quadratic. (These times are under the assumption that occur-checks and subsumption-checking — the later being unnecessary for this program — are omitted.)

We present an example (Example 5.9.2) of the benefits to be had from the use of difference-lists (which are non-ground structures) in a program that is best evaluated bottom-up.

5.1.1 A Motivating Example

We now consider an example that motivates the results in this chapter, and illustrates the main issues involved.

Example 5.1.1 Difference lists (see e.g., [O'K90]) are a representation for lists. The following is an example of a difference list.

$$dlist([1|2|X], X)$$

It represents the list $[1, 2]$. Difference list append is defined by following rule:

$$dappend(dlist(X, Y), dlist(Y, V), dlist(X, V)).$$

A goal

$$?dappend(dlist([1|2|X], X), dlist([3|4|Z], Z), Ans)$$

would unify with the head of the above rule to give the structure

$$dappend(dlist([1|2|3|4|Z], [3|4|Z]), dlist([3|4|Z], Z), dlist([1|2|3|4|Z], Z)\ )$$

Thus Ans gets bound to

$$dlist([1|2|3|4|Z], Z)$$

which is in the required format. Answering a query on *dappend* takes constant time in Prolog. Note that the first argument of *dappend* is changed when the query is solved.[2]

Let us now consider what happens if we use difference lists with bottom-up evaluation. Suppose we have the following rule for computing paths, as part of a larger program:

$$path(X, Y, L) : -\ path(X, Z, L1), edge(Z, Y), dappend(L1, dlist([Z|Y|V], V), L).$$

The *path* facts store potentially long paths represented as difference lists. The first problem in making an inference is that if we directly use the difference list stored in the given *path* fact and try to append an edge to it, the difference list in the given *path* fact will get damaged, and cannot be used to make further derivations. The evaluation mechanism (Opt-NG-SN evaluation) presented in this chapter uses a term representation with binding environments for variables, and a "persistent versioning" scheme to address the problem of modifying shared variables without damaging stored facts (Section 5.3).[3]

[2]*dappend* is destructive in the sense that it changes its first input argument. In this example, the first argument can still be interpreted as representing $[1, 2]$ by "subtracting" the second list, but it cannot be meaningfully used in further calls to *dappend* until the destructive update is undone during backtracking.

[3]By not damaging stored facts, we get the same effect as undoing of the update on backtracking. However, persistent versioning does not solve the problem that the first argument of *dappend* is changed after unification, This problem is inherent in the difference list representation, and is present with top-down evaluation too.

The second problem is less obvious, and is present with Magic rewriting as well as with its variants (MT-TR, MGU Magic Templates and MGU MTTR rewriting, and Alexander Templates). The Supplementary Magic rewriting of the given rule (ignoring queries on *path* and *edge*) is as follows:

$$\begin{aligned}
&R1 : sup_{1,0}(X, Y, Z, L, L1) :- query(path(X, Y, L)), path(X, Z, L1), edge(Z, Y).\\
&R2 : query(dappend(L1, dlist([Z|Y|V], V), L)) :- sup_{1,0}(X, Y, Z, L, L1).\\
&R3 : path(X, Y, L) :- sup_{1,0}(X, Y, Z, L, L1), dappend(L1, dlist([Z|Y|V], V), L).\\
&R4 : dappend(dlist(X, Y), dlist(Y, V), dlist(X, V)) :-\\
&\qquad\qquad query(dappend(dlist(X, Y), dlist(Y, V), dlist(X, V))).
\end{aligned}$$

Let us concentrate on the generation of queries and answers on *dappend*. The third argument of the $sup_{1,0}$ fact is a list of nodes on a path, and can be quite large. From the $sup_{1,0}$ fact we generate a query on *dappend*. In general, we have to rename variables in facts and rules before unification, in order to avoid name clashes. We can be smart and rename the rule and the smaller facts, and avoid renaming the larger facts. Such a renaming works well for rules $R2$ and $R4$, and we can create an answer to the query on *dappend* (we use the "persistent versioning" scheme to avoid damaging the stored *path* fact while doing this). But now we have to unify two potentially large facts, one for $sup_{1,0}$ and one for *dappend* with rule $R3$. Renaming either fact can be linear in the size of the fact. After renaming, the problem of actually unifying the renamed facts with the rule still remains.

We call the step of unifying an answer fact and a supplementary fact with a rule body as *answer-return unification*. Prolog evaluation does not perform any unification when an answer is returned to a query, and hence does not perform any unification equivalent to the answer-return unification. In general, unification takes time linear in the size of the terms. (For the case of ground terms, we can use hash-consing to store precomputed values to speed up unification; no such technique is known for the general case.)

To reduce the cost of answer-return unification, Opt-NG-SN evaluation stores part of the state of the computation along with each fact, and maintains information about the state through auxiliary identifiers stored with facts (Section 5.4). Using the stored state information, it is able to reduce the time cost of answer-return unifications performed by bottom-up evaluation to almost a constant per unification in many cases (in all cases, if subsumption-checking is not used). $\Box$

Table 2 shows a comparison between various costs in unoptimized bottom-up evaluation of an MGU MTTR rewritten program and Prolog evaluation. After presenting our optimization techniques, we present a similar comparison of Prolog with optimized bottom-up evaluation.

5.2 Basics

In this chapter we consider query evaluation on definite clause programs.[4] As the first stage of bottom-up query evaluation, the given program and query are rewritten using MGU MTTR rewriting (Section 3.4). Our evaluation mechanism is designed to evaluate the rewritten programs efficiently.

[4]We can extend this class to cover certain forms of negation such as stratified negation [CH85, ABW88] or modular negation [Ros90], by using our optimizations in conjunction with evaluation techniques such as "Ordered Search" [RSS92a]. We do not discuss this issue.

Operation	Bot. Up (No Opt.)	Prolog
Unification a. Answer-return b. Other	 $O($ size of terms $)$ $O($ size of terms $)$	 $O(1)$ $O($ size of terms $)$
Indexing a. Answer-return b. Other	 $O(\sum_{facts}$ size of fact $)$ $O(\sum_{facts}$ size of fact $)$	 $O(1)$ $O(\sum_{facts}$ size of fact $)$
Subsumption Checking	—	—
Creation of head fact	$O($ size of fact $)$ (Cannot instantiate shared variables)	$O(1)$

Table 2: Bottom-Up Evaluation using MGU MTTR rewriting vs. Prolog

We note that programs rewritten using MGU MTTR rewriting contain the meta-predicate goal_id. The semantics of such programs was discussed in Section 3.2. We use the term *program predicate* to denote base predicates (i.e. those defined in a database) as well as derived predicates (i.e. those defined in the program); meta-predicates (i.e., the goal_id predicate) are not considered program predicates.

5.2.1 Preprocessing

To make our discussion and analysis simpler, we assume that all non-equality literals in rules of the program have as arguments only distinct free variables. This can be achieved by the following straightforward transformation, without any increase in the time complexity of either Prolog evaluation of bottom-up evaluation. Suppose we have a rule:

$$R : p(t_{0,1}, t_{0,2}, \ldots, t_{0,n0}) :- q_1(t_{1,1}, t_{1,2}, \ldots, t_{1,n1}), q_2(t_{2,1}, t_{2,2}, \ldots, t_{2,n2}), \ldots, q_k(t_{k,1}, t_{k,2}, \ldots, t_{k,nk}).$$

We transform the above rule into the following rule, where each $X_{i,j}$ is a new variable, distinct from any variables in the rule.

$$\begin{aligned} R :\ & p(X_{0,1}, X_{0,2}, \ldots, X_{0,n0}) :- X_{0,1} = t_{0,1}, X_{0,1} = t_{0,2}, \ldots, X_{0,n0} = t_{0,n0}, \\ & X_{1,1} = t_{1,1}, X_{1,1} = t_{1,2}, \ldots, X_{1,n1} = t_{1,n1}, q_1(X_{1,1}, X_{1,2}, \ldots, X_{1,n1}), \\ & X_{2,1} = t_{2,1}, X_{2,1} = t_{2,2}, \ldots, X_{2,n1} = t_{2,n1}, q_2(X_{2,1}, X_{2,2}, \ldots, X_{2,n2}), \\ & \ldots \\ & X_{k,1} = t_{k,1}, X_{k,1} = t_{k,2}, \ldots, X_{k,n1} = t_{k,n1}, q_k(X_{k,1}, X_{k,2}, \ldots, X_{k,nk}). \end{aligned}$$

It is straightforward to verify that this transformation does not result in any increase in the time complexity of Prolog* evaluation.[5] In particular, it does not affect the use of tail-recursion optimization. This transformation introduces equality literals. We assume that equality is a base predicate with a single fact "$= (X, X)$". Further, we assume queries are not generated for the equality predicate, and instead the optimization of MGU MTTR rewriting for uses of base predicates (described in Section 3.4.1) is applied to occurrences of the equality predicate. This optimization allows the replacement of occurrences of $answer(ID, q(\overline{t}))$ by $q(\overline{t})$ in the bodies of rules, if q is a base predicate. No queries are generated for literals where this replacement is performed.

[5]We assume that the size of the program is fixed. Thus, although the above transformation can defeat rule indexing techniques used by Prolog, the loss of speed is by at most a constant factor.

As a result of this preprocessing, body occurrences of literals of the form $sup_{i,j}(\overline{t_i})$, $answer(ID, p_i(\overline{t_i}))$, or $query(p_i(\overline{t_i}), ID)$ are such that $\overline{t_i}$ is a tuple of distinct variables. This preprocessing is not critical, but simplifies the discussion considerably.

5.3 Representation of Terms and Facts

The representation used for terms is important in bottom-up evaluation. Subterms are shared between different goals and answers, both in bottom-up evaluation and in Prolog evaluation. Prolog uses a tuple at a time backtracking strategy, and hence it can destructively modify variable bindings, and on backtracking it can undo the modifications in order to perform further derivations. On the other hand, bottom-up evaluation can generate several facts from a given fact (for instance, several query facts may be generated from a a given query fact), and the facts may need to co-exist. Thus several instantiations of a variable may also need to coexist. Hence destructive modification of variable bindings is ruled out. We note that this problem also exists for non-depth-first evaluation strategies such as parallel implementations of Prolog. We describe below the term representation we use in our evaluation mechanism. The core of the representation is a "fully-persistent versioned" binding environment for variables.

A *binding environment (bindenv)* stores bindings for variables. A variable in *bindenv* may be free, or may be bound to a structure $structure'$ (which is possibly an atomic value). Variables within $structure'$ are also interpreted in *bindenv*. A bindenv differs from a substitution in the way in which it is interpreted. A variable X in the binding environment

$$\{X \rightarrow f(Y), Y \rightarrow a\}$$

is interpreted as being bound to $f(a)$ by dereferencing variables completely, whereas a variable X in a substitution $\{X/f(Y), Y/a\}$ is interpreted as being bound to $f(Y)$.

We represent a fact as a pair $\langle structure, bindenv\rangle$. Here *bindenv* is a binding environment that records the current binding of each variable present in *structure* (and perhaps other variables as well). The following is an example of our term representation:

$$\langle g(W, Y), \{Y \rightarrow X, Z \rightarrow 4, W \rightarrow f(Z, Z)\}\rangle$$

This represents the fact $g(f(4,4), X)$.

During rule application we allow variable bindings to span bindenvs; such bindings are of the form $\langle structure', bindenv'\rangle$. Variables within $structure'$ are interpreted in $bindenv'$. We do not allow such bindings in facts, for reasons that we note later in this section.

Given a fact f, we use $f.structure$ to refer to the structure of f, and $f.bindenv$ to refer to the bindenv of f. Thus $f = \langle f.structure, f.bindenv\rangle$. We use the notation $\langle s, e\rangle$, where s is a term, to denote s interpreted in bindenv e.

Definition 5.3.1 We say that terms $\langle s1, e1\rangle \equiv \langle s2, e2\rangle$ if both terms represent exactly the same term.

Given a fact $f = \langle f.structure, f.bindenv\rangle$, the variables in $f.structure$ are said to be *directly accessible* from $f.structure$. By looking up the bindings of these variables in $f.bindenv$, more variables are reachable (transitively). All such variables are said to be *accessible* from $f.structure$. Note that $f.bindenv$ may contain bindings for variables are not accessible (directly or indirectly) from $f.structure$.

The *variables in the fact f* are those variables that are accessible from $f.structure$. The *free variables in the fact f* are those variables that are accessible from $f.structure$, and are free. The *bound variables in the fact f* are those that are accessible from $f.structure$, and are bound. $\Box$

Binding environments are implemented using "fully persistent versions of data structures" [DSST86, Die89]. When applied to bindenvs represented as arrays, a fully persistent versioning scheme permits us to carry out the following operations efficiently:

1. Create a new child version of an existing bindenv (which itself may have been created as a child version of another bindenv, and so on). The new version has the same bindings as the old version when it is created, but any changes made to the new version will not affect the old version.
2. Add a new variable to a version of a bindenv.
3. Lookup the binding of a variable in a version of a bindenv.
4. Change the binding of a variable in a version of a bindenv.

Variable names (internally) are just numbers, and looking up the binding of a variable is achieved by indexing the array. A null entry in the array represents a variable that is not bound. Adding a variable is equivalent to extending the array by adding a new variable binding. We assume that each version of a bindenv keeps track of the highest numbered variable in it, so that new variables can be added to a bindenv version.

In this chapter, whenever we consider the time complexity of evaluation, we assume that the versioning scheme of Dietz [Die89] is used. Using Dietz's scheme, operation (1) can be done in constant time, and operations (2), (3) and (4) can be done in time $O(min(\log\log m, \log n))$, where m is the total number of versions of bindenvs that have been created and n is the number of versions of the variable that have been modified. For brevity, where several variables are versioned, we use the notation $\mathcal{V}$ (defined below).

Definition 5.3.2 ($\mathcal{V}$) Consider an evaluation of a program. Let $\{V_1, V_2, \ldots\}$ be the variables used in the evaluation. Let n_i denote the number of versions of V_i that have been modified, and let m denote the total number of versions of bindenvs that have been created. Then $\mathcal{V}$ denotes $\max_i(\min(\log\log m, \log n_i))$. $\Box$

We do not discuss the details of representation of versioned bindenvs, and refer the reader to [DSST86, Die89]. We noted earlier that the representation we use for facts does not allow bindings of the form $(structure, bindenv)$. This restriction is because do not know how to create versions of facts efficiently using the representation without this restriction — the problem is related to the problem of confluent versioning (see, e.g., [DST90]).

5.3.1 Context Identifiers for Facts

With each supplementary and initial_query fact we store two identifiers. The first identifier field is called *cont_id*, which stands for "context identifier". Loosely speaking, this field stores a unique identifier for the supplementary / initial_query fact. The semantics of this field are made more precise later. The second identifier field is called *par_id*, which stands for "parent context identifier". Again loosely speaking, this field is used to store the context identifier of the context (i.e., the supplementary/initial_query fact) that resulted in the generation of a query, and where answers to the query will be used. With all facts other than supplementary and initial_query facts, we store only the *par_id* field.

5.3.2 Discussion

We did not require updates on a parent version of a bindenv to be seen by child versions that have been created earlier. However, such a feature could be useful for lazy path-compression on chains of variable bindings. Such path-compression can be implemented if desired (at no extra cost) using the fully-persistent versioning schemes described in [DSST86, Die89].

There is a variant of Dietz's versioning scheme ([Die89]), with an access cost of $\log n$ for each variable, where n is the number of versions of the variable that have been modified. This scheme has lower constant costs than the $\mathcal{V}$ access cost scheme. An alternative way of implementing bindenvs is as a balanced search tree. Searching for or modifying a variable binding takes $O(\log n)$ time, where n is the number of variables in the bindenv. Search trees can be made fully persistent using the techniques or Driscoll et al.[DSST86], at no extra access cost. The idea is fairly simple — whenever a node is changed by an operation, the path from the root to the node is copied, so that the changes are not seen by other versions of the bindenv.

There is a "virtual copy" scheme due to D.H.D. Warren ([War83], cited in [Per85]) that allows creation of versions of bindenvs represented as tries. It differs from the schemes mentioned above in that it is not "fully persistent" — if you make an update on a parent version, it will not be seen by any child versions created earlier. However, it is simpler than the fully-persistent versioning techniques. We have implemented this scheme in the CORAL deductive database system [RSS92b].

Several schemes have been proposed for representing variable bindings in the context of OR-Parallel Prolog evaluation. It is pointed out in [GJ90] that efficient implementation of the following operations is important for OR-parallel Prolog evaluation: (1) access time to find variable bindings, (2) environment creation time, and (3) task switching time. Each of these operations has an analogue in BU-evaluation. (We note that task-switching as described in [GJ90] differs from the analogous operation in BU-evaluation, in which rule instantiations can be carried out essentially independently, except for some concurrency control to prevent conflicting updates to persistent bindenvs. Hence, it appears that the lower bounds on the costs of these operations shown in [GJ90] do not apply to BU-Evaluation.) Using Dietz's versioning scheme, we get bounds of $O(\mathcal{V})$ for operations (1) and (2), and constant time for (3).

5.4 How to Apply a Rule

The basic operation in bottom-up evaluation is the application of a rule to produce new facts. In this section we present an algorithm to apply a rule, with several optimizations to handle non-ground facts more efficiently. We assume that the rules to be evaluated are those generated by the MGU MTTR rewriting presented in Section 3.4. The conceptual steps in applying a rule using a single fact were described in Section 4.2. In this section we concentrate on the details of how these steps are implemented.

Procedure Apply_Rule is shown below. It essentially performs a left to right nested loops join. We describe informally some of the procedures that it uses, and present the details of these procedures later in this section.

```
Procedure ApplyRule( $R$ ).
Let the rule to be applied be:
```

```
$R : p(\overline{t})\!: -q1(\overline{t_1})[, q2(\overline{t_2})]$.
    /* [ ] denotes an optional argument */

1. Let $r\_env$ = a new empty (non-persistent) bindenv for the variables in the rule.
2. Fetch facts for $q1$. /* q1 must be a base/derived predicate */
   For each fetched fact $\langle str1, env1\rangle$ do the following:
   2.1. Set $env1'$ = new version of $env1$.
   2.2. Unify ( $\langle q1(\overline{t_1}), r\_env\rangle$, $\langle str1, env1'\rangle$).
        During this unification, variables in the rule are bound preferentially.
   2.3. If the unification in the Step 2.1 succeeds, Then
        2.3.1. If $q2$ is a program predicate, Then
               a. Fetch $q2$ facts that unify with $\langle q2(\overline{t_2}), r\_env\rangle$.
               b. For each fetched fact $\langle str2, env2\rangle$ do the following:
                    Execute Smart_Unify $(R, \langle str1, env1\rangle, \langle str2, env2\rangle, \langle R', r\_env'\rangle)$.
                    If Smart_Unify failure, continue with the next fetched fact.
                     Else Insert_Head_Fact( $\langle R', r\_env'\rangle$)
        2.3.2. Else
               Rename_and_Reunify( $R, \langle str1, env1\rangle, \langle R', r\_env'\rangle$)
               If $q2$ is a meta-predicate, evaluate $q2(\overline{t_2})$.
                    /* Else the rule has only one literal */
               Insert_Head_Fact( $\langle R', r\_env'\rangle$).
   2.4. Undo bindings in $r\_env$.
end Apply_Rule.
```

An important point to note in Apply_Rule is the creation of versions of bindenvs. Version creation ensures that the unification operations do not affect any stored facts. We do not present details of Unify, but describe it informally. Unify unifies its two arguments — bindings in the two environments are updated to create the unified result. During unification, variables in its first argument are bound preferentially.[6] Details of the indexing technique used to retrieve facts are discussed later.

Smart_Unify unifies the two fetched facts with their respective body literals. Notice that the fact bindenvs that it is called with are the original fact bindenvs, that *do not* incorporate the changes due to the unification in Step 2.1 of Apply_Rule. The same is true for Rename_and_Reunify. Unification in Step 2.1 of Apply_Rule creates variable bindings that cross bindenvs. Since we do not allow such bindings in facts, we ignore the bindings created in Step 2.1 once facts have been fetched in Step 2.3.1.a. Smart_Unify incorporates our main optimization ideas, and is described in detail in Section 5.4.1. Renaming of variables in the rule and the facts may need to be done in the course of Smart_Unify. Smart_Unify returns a renamed and instantiated version R' of the rule R. (Instead of renaming rule variables during rule application, their renaming may be deferred to the point when the head fact is created. We discuss this optimization in Section 5.9.)

[6]In other words, if two free variables, one from the left argument and one from the right argument are to be unified, the variable from the left argument is bound to the other variable.

In the case when there is only one non-meta literal in the body of the rule, Smart_Unify is not called. Rename_and_Reunify is instead used to rename rule variables in a manner similar to Smart_Unify, and to create a renamed and instantiated version R' of rule R.

In general, the rule can contain the meta-literal goal_id. Such a literal is evaluated after the literal to its left has been evaluated (it is important that meta-literals are evaluated after literals to their left are evaluated). The goal_id literal generates a single answer each time it is evaluated. Insert_Head_Fact creates a head fact, and inserts it into the appropriate relation. It checks for subsumption of the fact (if required; in some cases subsumption-checking is not used) before inserting it.

Let us now consider procedure Smart_Unify.

```
Procedure Smart_Unify $(R, \langle s1, env1\rangle, \langle s2, env2\rangle, \langle R', r\_env'\rangle)$.
    /** Due to MGU MTTR rewriting, any rule with two program predicates
        has a query/supplementary/initial_query literal and an answer literal in
        its body. Hence one of $\langle s1, env1\rangle$ and $\langle s2, env2\rangle$ is a query/
        supplementary/initial_query fact, and the other an answer fact.
    **/
    1. If $\langle s1, env1\rangle$ is the query/supplementary/initial_query fact,
        set $s = \langle s1, env1\rangle, a = \langle s2, env2\rangle$
        else set $s = \langle s2, env2\rangle, a = \langle s1, env1\rangle$.
    2. If $R$ is a supplementary (Type 2) rule,
        Then execute Return_Unify $(R, s, a, \langle R', r\_env'\rangle)$.
        If it succeeds, return success.
    3. Return Rename_Fact_and_Unify$(R, s, a, \langle R', r\_env'\rangle)$.
end Unify.
```

Smart_Unify makes use of knowledge about MGU MTTR rewriting. In a MGU MTTR rewritten program, any rule with two program predicates in its body must be either a Type 2 (supplementary) rule or a Type 6 (base fact) rule. In either case the body of the rule has a query/supplementary/initial_query literal and an answer literal. Let us ignore for now the call to Return_Unify, and assume that Smart_Unify calls Rename_Fact_and_Unify (we will come back to Return_Unify later).

Rename_Fact_and_Unify renames variables so that the variable names used in the two facts and the rule are disjoint. First consider the case that the rule is not a Type 0 rule Q_{R3}. The main optimization here is that variables in the query/supplementary fact are not renamed. Variables in the rule and in the answer fact are renamed instead. The reason for renaming facts in this manner is discussed later. After the variables are renamed, the (renamed) facts are unified with the renamed rule. The renamed rule R', interpreted in bindenv r_env' is the result of unifying the facts with the rule. Note that since both the facts and the rule used the same bindenv after renaming, all bindings in the bindenv are local to the bindenv (and hence in the form required in Section 5.3).

If the rule is a Type 0 rule Q_{R3}, a further optimization is used. Such a rule creates answers for the user's query from facts for *answer*, and is of the following form:

$$Q_{R3} : q(\overline{A}) \text{:} -initial\_query(\_, ID, \_), answer(ID, q(\overline{A})).$$

Note that the arguments of *initial_query* are all "don't care" ('_') except for the ID argument. The value stored in the ID argument is an integer, and the variable ID gets bound to this value. Hence no renaming is needed for *initial_query* facts used in the rule.

Note that r_env' is the bindenv from the versioned query/supplementary fact, and hence a version descendant of the bindenv of the original query/supplementary fact. The identifiers stored in the bindenvs of facts are updated by Update_Context_Ids. We discuss details of updates to the identifiers later. The values in these identifier fields are used in Return_Unify.

```
Procedure Rename_Fact_and_Unify($R, s, a, \langle R', r\_env' \rangle$)
    1. Set $r\_env'$ = a new version of $s.bindenv$.
    2. Set $s' = \langle s.structure, r\_env' \rangle$.
    3. If $R$ is not a Type 0 rule $Q_{R3}$
         Set $a'$ to be a fully dereferenced version of $a$, with free variables renamed
              with numbers starting from just above the highest numbered variable
              in $r\_env'$.
         Add all new variable to $r\_env'$.
    4. Set $a'.bindenv = r\_env'$.
    5. Let $R'$ be a renamed version of $R$ with variable names starting
         from after the highest numbered variable in $r\_env'$.
         Add all variables in $R'$ to $r\_env'$.
    6. Unify the query/supplementary literal in $\langle R', r\_env' \rangle$ with $s'$,
         preferentially binding variables in $R'$.
    7. Unify the answer literal in $\langle R', r\_env' \rangle$ with $a'$,
         preferentially binding variables in $R'$.
    8. If the unifications fail, Then return failure.
    9. Update_Context_Ids( $R', r\_env', s$).
    10. Return success.
end Rename_Fact_and_Unify.
```

In the case that Smart_Unify is not called from ApplyRule, Rename_and_Reunify is called instead. Rename_and_Reunify renames the variables in the rule so that they are disjoint from the variables in the sole body fact, and redoes the unification performed in Step 2.2. The bindenv r_env' is now a child of the bindenv of the fact $\langle str1, env1 \rangle$.

Insert_Head_Fact inserts the derived fact into the appropriate relation, after performing subsumption-checking if required. Before doing so, it dereferences variables in the fact.

```
Procedure Rename_and_Reunify($R, \langle str1, env1 \rangle, \langle R', r\_env' \rangle$)
```

```
        1. Set $r\_env'$ = new version of $env1$.
        2. Let $R'$ be a renamed version of $R$ with variable names starting
              from after the highest numbered variable in $r\_env'$.
              Add all variables in $R'$ to $r\_env'$.
        3. Unify the body literal of $\langle R', r\_env' \rangle$ with the fact $\langle str1, r\_env' \rangle$.
        4. Update_Context_Ids( $R', r\_env', \langle str1, env1 \rangle$).
end Rename_and_Reunify

Procedure Insert_Head_Fact( $\langle R', r\_env' \rangle$)
        1. If subsumption-checking is to be done, and the head fact of
                    $\langle R', r\_env' \rangle$ is subsumed by an existing fact
              Then return.
        3. Set $h$ to be a dereferenced version of the head of $R'$.
        5. Insert $\langle h, r\_env' \rangle$ into the appropriate relation.
end Insert_Head_Fact.
```

Ignoring the call to Return_Unify, it is not hard to see that Apply_Rule is correct (i.e., it generates all and only those facts that follow (using mgus) from the rule and the given facts). In all cases, versions of fact bindenvs are created so that existing facts are not affected by the actions of Apply_rule. Since rules in the MGU MTTR rewritten program have at most two body literals, and the first literal is either a base or derived literal, for all such rules Apply_Rule performs the unification of facts with the rule in a fairly straightforward manner if we ignore Return_Unify. We describe the actions of Return_Unify and prove correctness in a more formal fashion later.

5.4.1 Context Identifiers and Return-Unification

The unification of the answer fact with the answer literal in rules that use supplementary predicates (Type 2 rules) has no counterpart in Prolog. This answer-return unification is done implicitly by Prolog while generating the answer fact, since answer facts are not shared by different calls; Prolog does not have to perform a unification on returning an answer. In bottom-up evaluation answer facts have to be explicitly unified with a rule in order to generate new facts. It is important that this operation be done efficiently in bottom-up evaluation; else bottom-up evaluation could be much slower than Prolog.

Procedure Return_Unify performs answer-return unification in $O(\mathcal{V})$ time whenever it succeeds. It uses information stored with facts, that is maintained by procedure Update_Context_Ids. We describe the procedures below. We refer to the test made in Step 1 of Procedure Return_Unify as the *test for return-unification*, and the actions performed by Return_Unify as *return-unification*.

```
Procedure Update_Context_Ids( $R', r\_env', fact$)
        Let $head$ be the head of $\langle R, r\_env' \rangle$.
        1) Switch Type(R):
```

```
    Type 0: /* Initialization rules*/
        Set $head.par\_id = 0$;
        If the head of $R$ is $initial\_query$
            Set $head.cont\_id =$ new Context identifier.
        Else set $head.par\_id = fact.cont\_id$.
    Type 1: /* supplementary rule from query; fact = query fact*/
        Set $head.par\_id = fact.par\_id$
        Set $head.cont\_id =$ new context identifier.
    Type 2: /* supplementary rules; fact = supplementary fact */
        Set $head.par\_id = fact.par\_id$.
        Set $head.cont\_id =$ new context identifier.
    Type 3: /* answer rules; fact = supplementary fact */
        Set $head.par\_id = fact.par\_id$.
    Type 4: /* query rules, except for last literal;
                fact = supplementary fact */
        Set $head.par\_id = fact.cont\_id$.
    Type 5: /* query rules, for last literal; fact = supplementary fact */
        Set $head.par\_id = fact.par\_id$.
    Type 6: /* answer rules, for base predicates; fact = query fact */
        Set $head.par\_id = fact.par\_id$.
end Update_Context_Ids
```

We shall show several interesting properties about the propagation of context identifiers. The essential idea is that each supplementary fact (which can be viewed as a "context") has a distinct *cont_id* value. For Type 4 rules, the *par_id* of the query fact generated is set to the *cont_id* of the body supplementary fact. Such query facts will result in answers being generated for the query, and used with the body supplementary fact. Answers that have this value in their *par_id* fields will be used with this supplementary fact, and share bindenvs with it in a manner made precise later. The *par_id* value, in some sense, identifies the representation of the fact. There can be more than one copy of each fact, each with its own representation and its own *par_id* value. With subsumption-checking, all but one copy of each fact are eliminated.

For Type 5 rules, the *par_id* of the query fact generated is set to the *par_id* of the body supplementary fact. Such a query implements tail-recursion, and any answer generated will not be used with the body supplementary fact. Instead it will be used with a parent context (supplementary/initial_query fact) whose *cont_id* is equal to the *par_id* of the body supplementary fact.

The context identifiers are quite different from the *lcont* and *lcid* identifiers used in QSQR evaluation [Vie86, Vie88]. We discuss the differences in Section 5.11.

```
Procedure Return_Unify ($R, s, a, \langle R', r\_env' \rangle$)
    /* $s$ is a supplementary fact, and $a$ an answer fact. */
    1. If $s.cont\_id \neq a.par\_id$
          Then return failure.
    2. Set $r\_env'$ = new version of $a.bindenv$.
    3. Let $R'$ be a renamed version of $R$ with variable names starting
          from after the highest numbered variable in $r\_env'$.
          Add all variables in $R'$ to $r\_env'$.
    4. Bind each variable in the supplementary literal of $\langle R', r\_env' \rangle$
          to the corresponding argument of $s.structure$.
          /* This step is well defined as described below. */
    5. Bind each variable in the answer literal of $\langle R', r\_env' \rangle$
          to the corresponding argument of $a.structure$.
          /* This step is well defined as described below. */
          /* This may change some variable bindings that were made in Step 4. */
    6. Update_Context_Ids( $R', r\_env', s$).
    7. Return success.
end Return_Unify
```

Each argument of the supplementary literal is a distinct variable. Hence the concept of having for each variable in the supplementary literal a "corresponding argument" in the fact (Step 4 of Return_Unify) is well-defined. It is harder to see that the concept is well-defined for Step 5. Such a literal is of the form $answer(ID, q(\overline{X}))$, where $\overline{X}$ is a tuple of distinct variables, due to the preprocessing described in Section 5.2.1. The concept is well defined only because all facts used with the above literal are of the form $answer(id, q(\overline{a}))$.[7] The arguments "corresponding" to the variables in $\overline{X}$ are the arguments of $q(\overline{a})$ in the above fact.

The bindings created by Return_Unify when it succeeds are such that $\langle s, r\_env' \rangle$ is the same as the supplementary literal of $\langle R', r\_env' \rangle$, and $\langle a, r\_env' \rangle$ is the same as the answer literal of R', r_env'. That is, the actions of Return_Unify compute a unifier for the (renamed) rule, the (renamed) supplementary fact and the answer fact. Further, the unifier is a most general unifier. Return_Unify makes use of the information about the fact representation that is stored in the *par_id* and *cont_id* fields, in order to compute the unifier efficiently.

The idea is roughly as follows (we prove correctness formally later). When a query fact q is generated from a supplementary fact s, $q.bindenv$ is a new version of $s.bindenv$. Hence it inherits all variable bindings that are in $s.bindenv$. Suppose computation proceeds and an answer is generated for the query fact. If the test in Step 1 of Return_Unify succeeds, then the bindenv of the answer fact is a descendant of the bindenv of the supplementary fact (as we shall show). The updates to the bindenv are such that a free variable may become bound, but once a variable is bound, its binding does not change. Thus, if we replace the bindenv of the supplementary fact s by $a.bindenv$, the resultant fact s' is an instance of s. Thus bindenv replacement unifies the supplementary and answer facts with the rule body. Most importantly, bindenv replacement can be done very fast — in $O(\mathcal{V})$ time. A full unification (which would have to be done in the absence of the

[7]If we had facts of the form $answer(id, Y)$, the concept of corresponding arguments for the variables in $\overline{X}$ is ill-defined.

information about the fact representation) could take time linear in the size of the terms to be unified.

5.4.2 Examples

We now present examples of the use of our techniques in the evaluation of the *dappend* and *append* programs.

Example 5.4.1 Appending two difference lists can be done in time $O(\mathcal{V})$ with our term representation. We illustrate the use of Return_Unify using the following program. We have added a literal $test(1)$ to the end of the rule in order to suppress tail-recursion optimization for the call to *dappend*; the addition of this literal helps illustrate our techniques better.

$$\begin{array}{l} path(L1, L2, L) :- dappend(L1, L2, L), test(1) \\ dappend(dlist(X, Y), dlist(Y, V), dlist(X, V)). \end{array}$$

The preprocessed form of the above program is as follows:

$$\begin{array}{ll} path(L1, L2, L) & :- dappend(L1, L2, L), test(1). \\ dappend(V1, V2, V3) & :- V1 = dlist(X, Y), V2 = dlist(Y, V), V3 = dlist(X, V). \end{array}$$

The MGU MTTR rewriting of this part of the program is as follows (we have applied some optimizations to simplify the program; in particular, we have unfolded some uses of goal_id and =; also, we leave out Type 0 rules for simplicity).

$$\begin{array}{ll} R1 : sup_{1,0}(HId, L1, L2, L, ID, A) : & -query(path(L1, L2, L), ID, A), \\ & \mathsf{goal\_id}(dappend(L1, L2, L), ID). \\ \multicolumn{2}{l}{R2 : query(dappend(L1, L2, L), ID, answer(ID, dappend(L1, L2, L))) :-} \\ & sup_{1,0}(HId, L1, L2, L, ID, A). \\ R3 : sup_{1,1}(HId, L1, L2, 0, A) & :- sup_{1,0}(HId, L1, L2, ID, A), answer(ID, dappend(L1, L2, L)). \\ R4 : A & :- sup_{1,1}(HId, L1, L2, 0, A), test(1). \\ & \\ R5 : A & :- query(dappend(V1, V2, V3), ID, A), \\ & V1 = dlist(X, Y), V2 = dlist(Y, V), V3 = dlist(X, V). \end{array}$$

Suppose we have a query fact

$$query(path(dlist([a|b|X], X), dlist([c|Y], Y), P), 0, answer(P)) : 0$$

and a base fact $test(1)$. Evaluation of the program on these facts is depicted in Figure 7. The *par_id* of each fact is shown following the fact. For supplementary facts, the *cont_id* is shown following the *par_id*. We use pointers from facts to their bindenvs in the figure. Several facts point to some of the bindenvs — this notation should be interpreted as each fact having its own version of the bindenv, and is done only to keep the figure concise.

The main points to note in the figure are the following. When using rules $R1$, $R2$, and $R5$, there is only one derived predicate in the rule body. No renaming is done except for rule variables. Unification is straightforward, and the facts shown are created. The bindenvs get progressively refined, and more variables are added to the bindenv (we have used the optimization of deleting rule variables if they are not referred

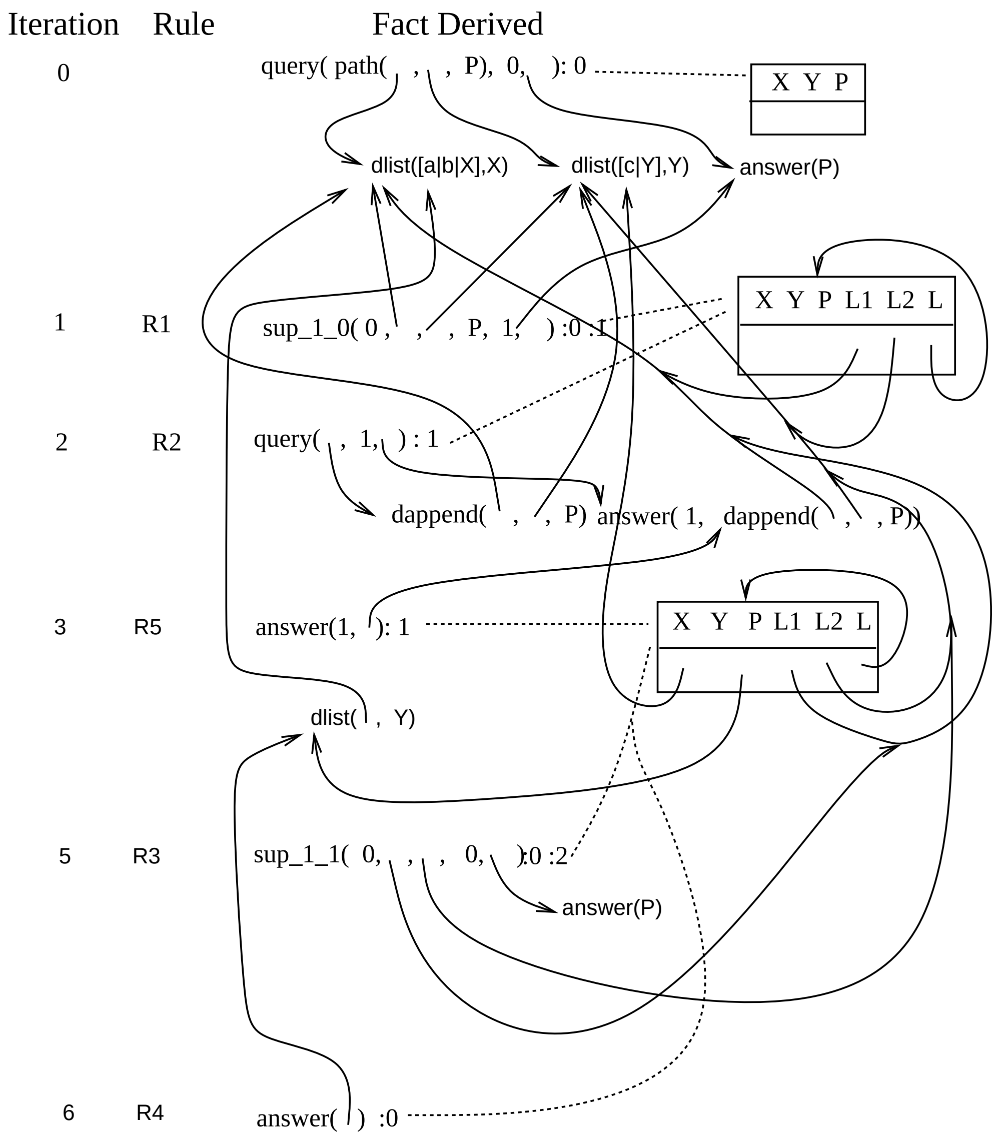

Figure 7: Evaluation of Program That Uses *dappend*

to in the head fact created after dereferencing). The *par_id* fields are shown for all facts. The *cont_id* fields are shown for supplementary facts.

We now come to the use of rule $R3$. Return_Unify succeeds on this rule since the *par_id* of the answer fact is equal to the *cont_id* of the supplementary fact. The bindenv of the answer fact is such that if the original supplementary fact is interpreted in the new bindenv, the variable P is bound to the result of *dappend*ing the two given lists. This is because X and P have been bound appropriately in the answer fact bindenv. Since Return_Unify succeeds, no renaming is required, and unification takes $O(\mathcal{V})$ time. This step would have taken time proportional to the size of the difference lists, had bottom-up evaluation without our optimizations been used. Finally rule $R4$ is used to create an answer to the query we were given. Here, the base fact $test(1)$ would have been renamed if it had variables. The supplementary fact is not renamed.

Overall, the time cost of the evaluation shown is $O(\mathcal{V})$, regardless of the sizes of the difference lists. □

Example 5.4.2 The append program is defined as follows.

$$append([], X, X).$$
$$append([H|T], L, [H|L1]) :- append(T, L, L1).$$

Preprocessing generates the following program (we have simplified the program a little to keep the example concise):

$$append(X1, X2, X3) :- X1 = [\,], X2 = X3.$$
$$append(X1, L, X2) \quad :- X1 = [H|T], X2 = [H|L1], append(T, L, L1).$$

The MGU MTTR rewriting of the original program was discussed in Section 3.4.2. The MGU MTTR rewriting of the modified program is very similar, provided we treat the equality literals as base literals. We omit the program for brevity.

We note that each rule in the rewritten program has only one derived literal in the body. Hence only rule variables are renamed. Unification costs are $O(\mathcal{V})$ per inference[8] — the derived literals all have as arguments distinct variables, and the unifications performed by the equality literals are all straightforward. Return unification is not important for this program, but variables in the *query* facts are modified and the versioned term representation is critical for efficiency. Overall the cost of evaluation of a query on *append* is $O(n \cdot \mathcal{V})$, where the first argument of the query is a list of length n. □

5.5 Correctness of Apply_Rule

We present a sequence of lemmas that are used to show that the actions performed by Return_Unify are correct, and hence Apply_Rule is sound. We first show that given a supplementary fact s, if for any fact f, $f.par\_id = s.cont\_id$, then $f.bindenv$ is a descendant of $s.bindenv$. Further, we show that there is a query q generated from the supplementary fact, and any variable that is not accessible from q has the same bindings in $f.bindenv$ as in $s.bindenv$. This property is proved formally in Lemmas C.1.1 and C.1.2 in Appendix C.1. The idea is to show using induction on lengths of derivation sequences, that if *par_id* value of a fact is inherited from the *cont_id* field of a supplementary fact, then so is the bindenv.

[8]We assume that occur checks are not used.

Lemma 5.5.1 *Suppose that there is a query fact*

$$q = query(p_i(\overline{a_i}), id1, answer(id1, p_i(\overline{a_i})))$$

generated by a Type 4 rule (i.e., from a non-tail-recursive literal), and an answer fact $a = answer(id1, p_i(\overline{b_i}))$. *Suppose also that* $q.par\_id = a.par\_id$. *Let* q_str2 *denote the last argument of* $q.structure$. *Then*

$$\langle q\_str2, a.bindenv\rangle \equiv \langle a.structure, a.bindenv\rangle$$

□

The detailed proof is presented in Appendix C.1. The basic idea is to show that if the conditions of the lemma hold, the structure in the answer fact is a dereferenced version of the structure in the query fact. The formal proof is by induction on lengths of derivation sequences, and uses a case analysis of the different rule types.

Lemma 5.5.2 *Suppose that Return_Unify succeeds on rule R with facts s and a. Then* $\langle R', r\_env'\rangle$ *is an mgu of* R' *with (a renamed variant of)s and a.* □

The detailed proof is presented in Appendix C.1. The idea is that if Return_Unify succeeds, the instantiated rule generated is generated by a most general unifier. We show that the instantiation is a unifier essentially by using Lemma 5.5.1. We then show that it is a most-general unifier by showing that any bindings introduced by the unifier are necessary for unification.

5.5.1 Soundness and Completeness of Evaluation

By Lemma 5.5.2, Return_Unify performs unification correctly. The rest of Apply_Rule is relatively straightforward. Note that whenever we modify variables in a fact bindenv, we have ensured that the version we use is a new version, and hence none of these steps affect stored facts. Thus we have the following theorem.

Theorem 5.5.3 *Let* P^{MGU_T} *be a MGU MTTR rewritten program and R a Semi-Naive version of a rule in* P^{MGU_T}. *Then a call to Apply_Rule(R) generates all and only those facts that follow from R using the set of facts available in the relations.* □

We call a version of Semi-Naive evaluation (described in Section 2.2.3) that uses procedure Apply_Rule to perform rule application as *Opt-NG-SN evaluation.* We call the query evaluation technique that first rewrites the program and query using MGU MTTR rewriting, and then evaluates it using Opt-NG-SN evaluation as *Opt-NGBU evaluation.*

From the above theorem, and the soundness and completeness of Semi-Naive evaluation, we have the following result.

Theorem 5.5.4 *Let* P^{MGU_T} *be a MGU MTTR rewritten program. Then Opt-NG-SN evaluation of* P^{MGU_T} *is such that (1) any fact generated is subsumed by facts in the least model of* P^{MGU_T}, *and (2) every fact in the least model of* P^{MGU_T} *is subsumed by the facts generated.* □

From the soundness and completeness results of MGU MTTR rewriting (Theorems 3.4.1 and 3.4.2), we then have the following theorem.

Theorem 5.5.5 *Let P be a program and Q a query on the program. Let P^{MGU_T} be the program generated from P and Q by MGU MTTR rewriting. Then Opt-NG-SN evaluation of P^{MGU_T} is such that (1) Every fact generated as an answer for Q is an answer to Q, and (2) Every answer to Q is subsumed by the set of answers generated.*

5.6 Cost of Optimized Evaluation

We now examine the costs of the basic steps in bottom-up evaluation that are not present in top-down evaluation without memoing. Note that the extra costs mentioned below are also incurred by top-down evaluations that perform memoing. We have discussed versioning and its cost, in Section 5.3, and have looked at the cost of extra unifications, in Section 5.4.1.

Indexing of Facts: We index supplementary and answer facts using hash-indices on the *goal-id* fields. The indexing was discussed in Section 3.3.4. Retrieving facts can be done in constant time per indexing operation and retrieved fact, and fetches only facts that will unify. Inserting facts into the index can be done in constant time.

Subsumption Checking: For the purpose of comparison of Opt-NGBU evaluation with Prolog* evaluation, we assume that no subsumption-checking is done.

Subsumption-checking of non-ground facts is in general costly, but provides benefits by avoiding repeated computation, and is is important in many cases. We discuss the costs and benefits of subsumption-checking in Section 5.8.

5.6.1 Cost of Inferences Using Apply_Rule

We now examine the costs associated with inferences made using Procedure Apply_Rule. In general, when unifying a variable with a term, we need to perform an "occur check" to ensure that the variable is not present within the term. Most implementations of Prolog do not perform the occur check, and unifying a variable with a term takes constant time. In our context, the unification would take $O(\mathcal{V})$ time. However, if we do perform occur checks in a naive fashion, Return_Unify would take time linear in the size of the facts. But we can show that an occur check is not necessary for soundness in Return_Unify. This is because the rule literals have distinct variables that are not present in the facts; all the unification operations in Return_Unify bind a rule variable to a term in one of the facts, and hence no occur checks are needed.

The following proposition is straightforward.

Proposition 5.6.1 *Procedure Return_Unify runs in $O(\mathcal{V})$ time.* $\square$

We also have the following lemma.

Lemma 5.6.2 *Suppose that an MGU MTTR rewritten program is evaluated using Opt-NG-SN evaluation without subsumption-checking. Then every call to Return_Unify succeeds.* $\square$

The proof is presented in Appendix C.2. The essential idea is that the goal-id values and *par_id* values are propagated through facts in lock-step if subsumption-checking is not used. A supplementary fact and an answer fact unify only if their goal-id values are the same. We show that if they are the same, then the *cont_id* of the supplementary fact will be the same as the *par_id* of the answer fact, and hence Return_Unify will always succeed.

As a result of the preprocessing, every body occurrence of literals of the form $sup_{i,j}(\overline{t_i})$ or $answer(ID, p_i(\overline{t_i}))$ is such that $\overline{t_i}$ is a tuple of distinct variables. Arguments similar to those for Return_Unify then show that occur checks are not needed for Rename_and_Reunify. Hence procedure Rename_and_Reunify can be implemented to run in $O(\mathcal{V})$. Inserting facts into a hash-index on the goal-id field takes constant time. Hence procedure Insert_Head_Fact runs in time $O(\mathcal{V})$ + the cost of subsumption-checking (if it is performed). In all cases below, we assume that the cost of renaming a rule and adding its variables to the appropriate bindenv is $O(\mathcal{V})$.

Suppose Return_Unify is called, and succeeds on a rule instantiation. Then the time taken by Apply_Rule for that rule instantiation, ignoring the time taken for subsumption-checking, is $O(\mathcal{V})$.

If Return_Unify is not called, or is called and fails, there are four cases based on the type of the rule.

1. The first case is when there is only one literal in the rule body. In this case, the cost of evaluation is essentially the cost of unification of the fact and the literal. Due to our preprocessing, the arguments of the literal are distinct variables, and evaluation takes $O(\mathcal{V})$ time.
2. The second case is when the second literal in the rule body is an answer literal. If no subsumption-checking is performed, Return_Unify always succeeds for this case. We discuss the costs if subsumption-checking is performed, in Section 5.8.
3. The third case is when the second literal in the rule body uses the meta-predicate goal_id. If subsumption-checking is not performed, the meta-predicate *goal-id* runs in constant time per call.

   The first literal in the rule uses a supplementary predicate, and has as arguments distinct variables. Hence unification for this literal can be done in time $O(\mathcal{V})$. The total cost of rule instantiation is $O(\mathcal{V})$ in the absence of subsumption-checking.
4. The fourth case is when the second literal in the rule body is a base literal. Rename_Facts_and_Unify renames the base fact. Prolog evaluations based on structure copying make copies of base facts when using them, and have the same overhead. Typically, base facts is assumed to be of constant size.

5.7 A Comparison With Prolog*

We now perform a detailed comparison of the costs of Opt-NGBU query evaluation (i.e., MGU MTTR + Opt-NG-SN evaluation) without subsumption-checking and Prolog evaluation. We make the following simplifying assumption:

A1: Given terms a, $a1$ and b, if a is equivalent to $a1$ then the time taken to unify a and b is the same as the time taken to unify $a1$ and b.

Two terms may be equivalent, but may be represented by different structures. The actual structures created depend on the details of the unification algorithm; for instance, if the unification algorithm delays dereferencing of variables when performing unification, the resultant representation is different from the representation if dereferencing is always used. The main tradeoff is that in some cases dereferencing of unused terms is avoided by delayed dereferencing, but on the other hand some variables may have to be dereferenced several times. We do not factor this low level decision into our comparison. We also ignore the effects of path-compression when dereferencing variables. Path compression can result in a substantial improvement in speed for some programs, if it is performed. Most Prolog implementations do not perform it, since it complicates the maintenance of trail information. Path compression can be implemented in bottom-up evaluation using fully-persistent bindenvs (see Section 5.3.2). We assume it is not done in either case, in order to simplify the discussion.

We also assume that bottom-up evaluation as well as Prolog* evaluation use the same indexing technique for base relations.

Attempted derivations in Opt-NG-SN evaluation are split into cases based on the rule type, and we use the same mapping that was used to prove Theorem 4.3.1, to show that for each derivation in Opt-NGBU query evaluation (without subsumption-checking), Prolog evaluation has an action of "almost" the same cost.

The details of the comparison are presented in Appendix C.3. We sketch the basic idea below. Attempted derivations using Type 0, Type 3, Type 4 and Type 5 rules are shown to take $O(\mathcal{V})$ time each. If there are n such derivations, the mapping shows that Prolog performs $\Omega(n)$ actions, each of at least unit cost. The cost of an attempted derivation using such a rule is primarily the cost of evaluation of the equality literal. We show that Prolog* evaluation performs an equivalent unification action. For Type 2 rules that have an *answer* literal in the body, Return_Unify always succeeds, and hence the cost of an attempted derivation $O(\mathcal{V})$. For Type 2 rules that have a goal_id literal in the body, the cost of an attempted derivation is $O(\mathcal{V})$, which is mapped to an action of Prolog* evaluation that takes at least unit time. For Type 2 rules that have a base literal (for e.g., an equality literal) in the body, we show that any attempted derivation is mapped to an action of Prolog* evaluation that performs the same indexing operations and unification. Thus the loss of speed of due to Opt-NGBU evaluation is at most a factor of $O(\mathcal{V})$ in this case. We then have the following theorem.

Theorem 5.7.1 *Let P be a program, and Q a query. Given any database, suppose the cost of Prolog* evaluation of Q is t units of time.*[9] *Opt-NGBU evaluation without subsumption-checking evaluates the query on the given database in time $O(t \cdot \mathcal{V})$. (The size of the program is not taken into account in this time complexity measure.)* $\Box$

The proof of the above theorem is presented in Appendix C.3.

Table 3 summarizes a comparison of the cost of various steps in Opt-NGBU evaluation without subsumption-checking with the corresponding costs in Prolog evaluation. This table may be contrasted with Table 2 to see the benefits of Opt-NG-SN evaluation.

With Dietz's versioning technique [Die89], $\mathcal{V}$ is $O(\log\log t)$ for the following reason. As we noted in Section 5.3, $\mathcal{V}$ is $O(\log\log n)$, where n is the number of versions of bindenvs that are created. Each attempted derivation creates at most three bindenv versions. The number of actions performed by Prolog* evaluation is at most t since each action has at least unit cost. Hence the number of attempted derivation steps is at most $c_1 \cdot t + c_2$, for constants c_1 and c_2 that are independent of t (Theorem 4.3.1). The $O(\log\log t)$ bound on $\mathcal{V}$ then follows.

[9]Where each action of Prolog* evaluation takes at least unit time.

Operation	Bot. Up (Opt.)	Prolog
Unification		
a. Answer-return	$O(\mathcal{V})$	$O(1)$
b. Other	$O(\mathcal{V}\cdot$ size of terms $)$	$O($ size of terms $)$
Indexing		
a. Answer-return	$O(1)$	$O(1)$
b. Other	$O(\mathcal{V} \cdot \sum_{facts}$ size of fact $)$	$O(\sum_{facts}$ size of fact $)$
Subsumption Checking	—	—
Creation of head fact	$O(1)$	$O(1)$

Table 3: Opt-NGBU Evaluation (without subsumption-checking) vs. Prolog

Corollary 5.7.2 *Let P be a program, and Q a query. Given any database, suppose the cost of Prolog* evaluation of Q is t units of time. Opt-NGBU evaluation without subsumption-checking evaluates the query on the given database in time $O(t \cdot \log\log t)$. (The size of the program is not taken into account in this time complexity measure.)* $\square$

The above result shows that the cost of memoing facts (ignoring the cost of checking for subsumption) can be made quite small, in the sense of time complexity. The optimization techniques we developed in this chapter are of considerable theoretical importance, since they help us establish the above result. This result assumes that the size of the program is a constant, We can relax the assumption that the size of the program is constant, as discussed briefly in Section 5.9.

The question of how bottom-up and top-down methods compare is considered important, and has been under investigation by several researchers [Ull89a, Bry90, Ram88, Sek89]. Most of this research, with the exception of [Ull89a], has restricted itself to comparisons in terms of the number of facts generated or the number of inferences made. Our result carries the comparison of top-down and bottom-up methods farther than the results of Ullman [Ull89a].

1. Our result extends the class of programs considered from safe Datalog to full logic programs.
2. Our result compares bottom-up evaluation with a sophisticated model of Prolog evaluation, which incorporates tail-recursion optimization, unlike earlier work.

We remind the reader that our analysis ignores constant costs, and the effect of factors such as virtual memory.

5.8 Subsumption Checking in Bottom-Up Evaluation

In general, subsumption-checking is a costly operation, and we are not aware of efficient subsumption-checking techniques for the case of arbitrary non-ground facts. However, there are special cases for which subsumption-checking can be done efficiently.

For ground facts, subsumption is the same as equality, and hash-consing [Got74, SG76] can be used to perform subsumption-checking in constant time in many cases.[10] In Section 5.10 we discuss an efficient

[10]Efficient hash-consing requires that the ground terms are built up from smaller ground terms. This is not true if ground terms can be created by instantiating variables within an existing non-ground term.

subsumption-checking technique for a restricted class of non-ground facts. We use this technique to optimize Semi-Naive evaluation for a restricted class of programs.

Without subsumption-checking, Semi-Naive evaluation is still sound and complete. However, derivations can be repeated, and in the worst case a computation that terminates with subsumption-checking may loop for ever, repeating derivations, if subsumption-checking is not used.

Subsumption-checking can be done for some predicates and not for others, and need be done only if the benefits from avoiding repeated computation and possible avoiding of infinite loops is worth the cost. For instance, there are cases where *query* facts (after adornment as in [BR87b]) are ground, although answer facts may not be ground. In many such cases, it suffices to perform subsumption-checking on the ground *query* facts.

If subsumption-checking is performed, Return_Unify may fail for for some answer-return unifications. These unifications occur with Type 2 rules of the following form:

$$\ldots : -sup_{j,i}(\ldots, ID, A), answer(ID, \ldots).$$

Our indexing scheme ensures that any answer fact and supplementary facts fetched by indexing will unify with the rule. Suppose facts $sup_{j,i}(\overline{v}, id, ans)$ and $answer(id, p(\overline{b}))$ are fetched by indexing. If Return_Unify does not succeed with these facts, it means that the answer fact was generated from a query fact that is equivalent to (but not the same as) the query fact that was generated using $sup_{j,i}(\overline{a}, id, ans)$. This means that the query fact generated from $sup_{j,i}(\overline{a}, id, and)$ was eliminated by subsumption-checking.

Prolog would have solved the above repeated subgoal, whereas bottom-up evaluation with subsumption-checking avoids the repeated computation. The benefit of avoiding repeated computation has to be balanced against the cost of checking for subsumption, and the cost of rule application when Return_Unify fails. The second of these costs is as follows.

Let n be the size of the answer fact. Then the fact can be renamed in time $O(n \cdot \mathcal{V})$. Unification can be done in time linear in the size of the result of unification, and the overall cost of rule instantiation is $\mathcal{V}$ times the size of the answer fact after unification. If the computation of the answer fact would have taken more time than $\mathcal{V}$ times the size of the answer fact after unification, copying the fact is no more expensive than recomputing it. To decide if subsumption-checking for goals is useful in a given context, we also have to include the cost of subsumption-checking.

5.9 Optimizations and Discussion

The evaluation technique we described can be extended and optimized in several different ways. The techniques described in this chapter can be adapted in a straightforward manner to work with the MGU Magic Templates rewriting instead of MGU MTTR rewriting. In fact, MGU Magic Templates can be considered a special case of MGU MTTR rewriting where the last literals in the rules are not treated as tail-recursive (and Type 4 query rules generated for these literals instead of Type 5 query rules). The proofs of correctness change a little due to the differences in the form of the query facts generated.

We can use adornments (see e.g. [BR87b]) with MGU MTTR rewriting (or with MGU Magic rewriting) under some restrictions on how adornments are generated. The idea is as follows. Consider any argument of a literal that is a a free variable that appears nowhere else in or before the literal in the rule. Any query fact created from this literal will have a distinct free variable in such an argument. Only such arguments may be considered free when adorning a literal. Arguments adorned f are projected out of the query literals and facts. The proof that Return_Unify works correctly becomes a little more complicated, but Opt-NG-SN evaluation does work correctly if the adornment is done subject to the above restrictions.

Rule variables are added at the end of bindenv of the head fact. We dereference head variables before creating the head fact. This often (always, in the case when the program generates only ground facts) results in none of the rule variables being referenced from the head fact that is created. In case none of the rule variables are referenced from the created head fact, we can drop these variables from the versioned bindenv (and the number of the highest numbered variable in the bindenv changes appropriately).

Dropping rule variables from the bindenv of the head fact can be quite useful for the following reason. Throughout our discussion we assumed a loss in efficiency of $O(\mathcal{V})$ compared to Prolog* evaluation. In the case of range-restricted programs, where no non-ground facts are generated, the optimization of removing unreferenced rule variables from the bindenv results in bindenvs that have no variables at all. Such bindenvs need not be stored explicitly. Hence we can evaluate such a program without any $O(\mathcal{V})$ overhead. Similarly, if if variables in non-ground facts are not instantiated (for example in the append program on non-ground lists), $O(\mathcal{V})$ reduces to $O(1)$.

In the discussion earlier, we assumed that the rule is renamed during Apply_Rule. The renaming need not be done explicitly, but can be achieved by a two step process. During rule application we maintain a separate bindenv for rule variables. During unification we maintain a trail of variable bindings. When creating the head fact, we add the rule variables to the bindenv of the head fact, and use the trail to back-patch all variables that were bound to rule variables. We can combine this optimization with the optimization mentioned above, to avoid creating slots for rule variables in the bindenv of the head fact.

The creation of a new version of $env1$ in Step 2.1 of Apply_Rule is not really necessary in a sequential implementation. Unify binds variables in its first argument preferentially. For preprocessed MGU MTTR rewritten programs, in the call to Unify in Step 2.2 of Apply_Rule, only variables in the rule get bound; variables in the fact bindenv are not affected. Thus we do not need to create a new version of $env1$ in this step; if required, a version of $env1$ is created later by $Rename\_Facts\_and\_Unify$.

Theorem 5.7.1 assumes that the size of the program is a constant. The primary reason for this assumption is that each iteration of Semi-Naive evaluation applies all the rules, and may make only one derivation. The mapping of costs described in Section 4.2 depends on this assumption.

We can relax this assumption using a rule indexing scheme for MGU MTTR (and MGU Magic Templates) rewritten programs. We do not go into details, but the idea is as follows. We keep track of δp_i^{old} relations that are non-empty, and use these to index rules that can be used to make derivations using these relations. There are only a constant number of Type 0 rules, and we do not need to index them. For Type 1 and Type 6 rules, we can use any indexing technique that Prolog uses to find rules that unify with a subgoal. For Type 2 rules that use an *answer* literal, we can use the goal-identifier field to directly index supplementary or answer facts, and use only rules for which matching supplementary and answer facts are available. Other Type 2 rules have only one derived relation — the supplementary relation. We use the non-empty δ supplementary relations to index such rules. Semi-naive rewritten Type 3, Type 4 and Type 5 rules always succeed in making an inference if there is a fact for the δ relation in their body; indexing such rules using the non-empty δ

relations is straightforward.

Using this rule indexing technique for MGU MTTR and MGU Magic Templates programs, along with the optimizations described in Section 4.2.2, we can (a) extend Theorem 4.3.1 to remove the assumption that the size of the program is a constant, (b) extend the model of Semi-Naive evaluation to show that the cost of evaluation can be completely mapped to the cost of attempted derivations, even without the assumption of constant program size, and (c) extend Theorem 5.7.1 to remove the assumption that the size of the program is a constant.

5.9.1 More Example Programs

We present a brief analysis of the benefits of our optimization techniques on some example programs. We have implemented our optimization techniques on the CORAL deductive database system [RSS92b], and we present some preliminary performance figures.

Example 5.9.1 Consider the well-known program to append lists (Example 3.4.1), with a query involving non-ground lists.

The following table presents performance numbers on lists of the specified lengths. The number of distinct variables in the list is shown in parentheses. The column "Unoptimized" refers to evaluation without the Apply_Rule optimizations we described in this chapter. The column "Optimized" refers to evaluation using the optimizations described in this chapter but with MGU Magic rewriting. The column "Tail-Rec" refers to evaluation using MGU MTTR rewriting and the optimizations described in this chapter.

Dataset	Unoptimized	Optimized	Tail-Rec
Length 25 (3 vars)	.31	.19	.08
Length 50 (3 vars)	0.98	.35	.15
Length 100 (3 vars)	3.85	.67	.30
Length 100 (25 vars)	3.87	.69	.30
Length 100 (ground)	.44	.55	.30

The numbers show that for ground lists, optimized evaluation with MGU Magic rewriting is not much worse than unoptimized evaluation, while optimized evaluation using MGU MTTR rewriting is faster than both these. For non-ground lists, the time cost of optimized evaluation grows linearly with the size of the lists, while for unoptimized evaluation, the cost grows approximately quadratically. □

Example 5.9.2 This program illustrates the use of difference lists in a program that is best evaluated bottom-up. We assume that the query is $?path(1, X, C, P)$ (the single source shortest path problem).

$path(X, Y, C, dlist([Y|D], D)) :- edge(X, Y, C).$
$path(X, Y, C1 + C2, P) \quad :- path(X, Z, C1, P1), edge(Z, Y, C2), dappend(P1, dlist([Y|D], D), P).$
@aggregate_selection $groupby(path(X, Y, C, P)[X, Y], min(C)).$
$dappend(dlist(X, Y), dlist(Y, V), dlist(X, V)).$

This program keeps track of the vertices in paths that it computes, and stores the list as a difference list to allow efficient concatenation of edges to the list. Prolog is not suitable for evaluation of this program (or other path programs) since it can get into infinite loops with cyclic data.

The use of the aggregate_selection annotation and its efficient implementation is discussed in Chapter 6. We discuss this example ahead of that chapter in order to illustrate the use of non-ground data-structures in a program that is best evaluated bottom-up. The annotation

@aggregate_selection $groupby(path(X, Y, C, P)[X, Y], min(C))$

in the program specifies that for answer facts for the predicate *path*, for each value for X and Y, only facts with minimum value for C should be retained.

For the sake of brevity, we omit the rewritten version of the program, but assume that the aggregate selection on *path* is also used for *answer* facts for *path* (i.e., facts of the form $answer(id, path(\ldots))$). We use MGU Magic Templates with adornments; tail-recursion optimization is not useful for this program since for the rules defining *path*, the last literal is not recursive to *path*.

Due to the use of adornments, *query* facts for *path* are ground, and store only bindings for the first argument of *path*. We use subsumption-checking for such facts. Further, supplementary facts generated from these query facts are ground, and as a result, we do not need to rename *path* facts when applying rules in the rewritten program.[11] The aggregate selection is used to prune *answer* facts for *path*; no other subsumption-checking is done for answer facts. Subsumption-checking is not done for *dappend* facts either.

Without the use of difference-lists, we could either generate the paths in reverse order using list *cons*, which is unappealing, or we could use *append* instead of *dappend*, which would cost $O(V)$ time per append. Let the evaluation time of a version of this program using ordinary lists, and *cons* instead of *append* be $O(f(E, V))$.[12] Then the evaluation time of the program using *append* would be $O(V \cdot f(E, V))$, which is considerably slower if V is large.

If we used a naive version of rule application, the cost of creating new lists by *dappend*ing two difference lists will take time linear in the size of the lists, which can be $O(V)$. This would lead to a time complexity of $O(V \cdot f(E, V))$.

Using our optimizations, each query generated for *dappend* is solved using the rule for *dappend* in time $O(\mathcal{V})$, and Return_Unify succeeds for the answer-return unification, when answers to queries on *dappend* are generated, and takes time $O(\mathcal{V})$. This leads to an overall time complexity of $O(\mathcal{V} \cdot f(E, V))$.

The number of versions of bindenvs that are created is $O(f(E, V))$, and the number of *path* facts that are computed is $O(f(E, V))$. Hence, if we use Dietz's versioning scheme, $\mathcal{V}$ is $O(\log \log f(E, V))$. Each bindenv has $O(V)$ variables since the maximum path length is $O(V)$. Hence, if we use Warren's versioning scheme (as we do in our implementation) $\mathcal{V}$ is $O(\log V)$. In either case, the time complexity is not much worse than the time complexity of evaluation using ground lists with *cons*, and has the benefit of generating path lists in the correct order.

We ran two variations of this program on the CORAL system. Both variations used the query ?$path(X, Y)$. The first used a difference list representation, and the second used an ordinary list representation, but

[11]Our implementation detects which facts are ground and tries to avoid renaming non-ground facts. Hence this optimization is incorporated automatically.

[12]The time complexity depends on the number of distinct shortest paths between pairs of nodes. If we store only one shortest path between each pair of nodes, $f(E, V) = E \cdot V$, as we show in Example 6.6.2. This can be improved to $E \cdot \log V$ by using other optimizations, as we show in Example 6.6.3.

used *cons* rather than *append*. The second variation generated only ground facts, but generated path lists in reverse order. The ground program ran in 0.6 seconds on a sample dataset, while the non-ground program ran in 0.8 seconds. Thus the loss of speed due to the non-ground data-structure is reasonably small (33%), while providing the benefit of printing out paths in the correct order. □

5.10 Bottom-Up vs. Prolog* for a Restricted Class of Programs

In this section we present a summary of results from Sudarshan and Ramakrishnan [SR92b] that compare bottom-up evaluation using MTTR rewriting with Prolog* evaluation for a subclass of range-restricted programs. All facts generated by a range restricted program are ground. However, MTTR rewriting (as also MGU MTTR rewriting) of a range-restricted program results in a program that is not range-restricted (even if the adornment step of [BR87b] is performed). The comparison is presented in two parts — the first in terms of number of derivations, and the second in terms of time complexity for programs that generate only a restricted kind of non-ground facts.

In Section 3.1 we showed that Magic rewriting could perform some unnecessary inferences if non-ground facts are generated. However, if only ground facts are generated, an equivalent of Theorem 4.3.1 holds for the case of MTTR rewriting, provided subsumption checking is used in the evaluation. This is shown by the following theorem, from [SR92b].

Theorem 5.10.1 *[SR92b] Let P be a range-restricted program and Q a query on P. Let P^T be the MTTR rewriting of P with query Q. Then there is a mapping M of derivations in the Semi-Naive evaluation of P^T to actions of the Prolog* evaluation of Q on P, such that not more than three different derivations of bottom-up evaluation of P^T are mapped to the same action of Prolog* evaluation.* □

Theorem 4.3.1 showed that even if subsumption-checking is not performed, Semi-Naive evaluation of an MGU MTTR rewritten program performs no more than a constant factor worse than Prolog* in terms of number of actions. However, Semi-Naive evaluation without subsumption checking of an MTTR rewritten program could loop when Prolog terminates, as the following program (adapted from [NR91]) illustrates.

Example 5.10.1 Consider the following program.

$q() :- p(a), p(a), r(a).$
$p(a).$
$r(a).$
Query: ?-$q()$.

The MTTR rewritten form of the above program has (among other rules) the following rules.

$R1 : query(q(), q()).$
$R2 : query(p(a), p(a)) :- query(q(), A).$
$R3 : A \quad :- query(p(a), A).$
$R4 : query(p(a), p(a)) :- query(q(), A), p(a).$

A query fact $query(p(a), p(a))$ is generated using rules $R1$ and $R2$. If subsumption-checking is not performed, rules $R3$ and $R4$ enter into an infinite loop. Rule $R3$ uses the newly generated fact $query(p(a), p(a))$ and generates fact $p(a)$. Rule $R4$ uses the newly derived fact $p(a)$[13] (which is not eliminated since subsumption checking is performed), and generates a fact $query(p(a), p(a))$. This fact is not eliminated either, and the cycle repeats. $\Box$

We now define a class of facts that we call NGSF facts, and define a class of programs that we call NGSF programs, that generate only NGSF facts. Later in the section we summarize details of an evaluation technique for NGSF programs.

Definition 5.10.1 (NGSF Facts and Programs) [SR92b] Let p be a predicate in program P. A fact $p(\bar{t})$ is said to be *non-ground structure free* (NGSF) iff each argument of the fact is either a ground term or a variable. The definition is easily extended to tuples $\bar{t}$.

We extend this definition to allow limited forms of structure introduced by MTTR rewriting (similar extensions can be used with MGU MTTR rewriting). A fact of the form $query(p(\bar{t}), q(\bar{s}))$ (resp. $sup_{i,j}(\bar{u}, q(\bar{s}))$) is said to be NGSF iff $p(\bar{t})$ and $q(\bar{s})$ (resp. $\bar{u}$ and $q(\bar{s})$) are NGSF.

We say that a program is *non-ground structure free* (NGSF) iff every fact derived in an NSN evaluation[14] of the program is non-ground structure free. $\Box$

For example, $p(f(a, g(b)), X)$, $query(p(X, Y), q(X, Y))$, $sup_{i,j}(f(a), p(f(a), X))$, and $p(X, g(c, g(c, g(c))), X)$ are non-ground structure free, but the facts $p(f(X))$ and $query(p(f(X)), q(X))$ are not.

For the class of NGSF programs, one can perform each of the basic operations in bottom-up evaluation (unification, indexing and subsumption-checking) at unit cost, as described in Sudarshan and Ramakrishnan [SR92b]. Since variables are present only at the outermost level of facts, renaming of variables can be done with unit time cost. Unification is then done at unit cost by using hash-consing [Got74, SG76]. A scheme for indexing based on "pattern-forms" that encode the patterns of variables in each fact is presented in [SR92b], and it is shown that indexing of relations can be done at unit cost per retrieved fact for NGSF programs. The same basic pattern-form scheme is used for subsumption-checking, and it is shown that for NGSF facts, subsumption-checking can be done at an amortized cost of $O(1)$ per fact. Finally, these results are put together, and the model of semi-naive evaluation presented in Section 4.2 is used to show that the cost of evaluation is $O(1)$ per fact derived.

We now define a class of programs form which the MTTR rewriting is NGSF.

Condition Strongly NGSF Evaluable: We say that a program P with query Q is *strongly NGSF evaluable* if P satisfies the following condition:

1. P is range restricted
2. For every rule in P, for every literal $p(\bar{t})$ in the body of the rule, any variable that appears with an enclosing function symbol in $p(\bar{t})$ also appears in a literal to the left of $p(\bar{t})$ in the rule.
3. Those variables in the head of the rule that appear only in the last literal in the body of the rule do not appear with enclosing function symbols in the head of the rule.
4. The query on the program does not have any variables that are enclosed in function symbols. $\Box$

[13]Recall that if a rule R has a derived literal in its body, in Semi-Naive evaluation each derivation that uses R must use a newly generated fact for one of the derived literals in the body.

[14]The order in which derivations are made in a bottom-up evaluation is not deterministic. This leads to a non-determinism in the set of facts computed, if subsumption-checking is used. To avoid this problem, we use NSN evaluation in this definition.

The intuition behind this condition can be understood as follows: First, all facts produced by the program are ground. Second, evaluation of the query on the program will not create any subgoal containing non-ground structures. Third, when tail recursion optimization is used on these programs, structures in answers to subgoals on tail-recursive predicates will not be used to "build" larger structures in the head of the rule. Without the restriction provided by the Part 3 of Condition Strongly NGSF Evaluable, P^T may compute facts with large non-ground structures that are hard to handle in bottom-up evaluation.

If P with query Q is Strongly NGSF Evaluable, then P^T (the MTTR rewritten version of the program) is non-ground structure free. (The definition of strongly NGSF evaluable is overly strict, since it does not make use of bindings provided by queries, and can be weakened.) We then have the following theorem.

Theorem 5.10.2 *[SR92b] Suppose we are given a program P that is Strongly NGSF Evaluable. Let t_P be the running time of a Prolog* evaluation of P, and let t_B be the running time of Semi-Naive evaluation of P^T (the MTTR rewritten version of the program). Then there is some constant c, that is independent of t_P and t_B (but may be dependent on the arity of predicates in P, and the textual size of P) such that $t_B \leq c * t_P$.*
$\square$

The above result also holds using MGU MTTR rewriting, although only the case of MTTR rewriting is considered in [SR92b].

Contrast the above theorem with Theorem 5.7.1. The above theorem shows that for a subclass of range-restricted programs (that properly contains range-restricted Datalog), bottom-up evaluation with subsumption checking, in the worst case, can be only a constant factor slower than Prolog*. Theorem 5.7.1 provides a weaker bound, but applies to all definite clause programs. It is easy to find examples (in the subclass) for which the behavior of Prolog* is much worse than that of bottom-up evaluation (for some programs Prolog* does not terminate, although bottom-up evaluation does).

5.11 Related Work

We are not aware of any work related to optimizing semi-naive evaluation for the case when non-ground facts are generated. However, there has been some related work in the area of top-down evaluation with memoization, and in the linguistics community.

D. S. Warren [War89] describes the XWAM, an implementation of memoization for Prolog. The XWAM uses a depth-first search, coupled with memoization of subgoals and answers to avoid repeated computation. The techniques described there are for the case of ground Datalog programs. There is a brief mention of possible extensions to the scheme to programs that generate variables, by using bindarrays. However, no details are provided.

Pereira [Per85] describes an implementation of parsers for unification based grammer formalisms. In these parsers, complex phrase types are built by incremental refinement of phrase types. A naive implementation copies phrase types; by using "virtual copy memory" (i.e., versioned memory), Pereira shows how to reduce the cost of copying the phrase types. There is a problem in this context that corresponds to the renaming problem; for special cases of grammers, "renaming" can be avoided, but in general it must be performed. Thus there is no equivalent in this context to the return unification optimizations that we present.

The context identifiers we use are quite different in function from the *lcont*, and *lcid* identifiers used in QSQR evaluation [Vie86, Vie88]. The *lcont* and *lcid* identifiers correspond to the goal identifiers that we use. If a goal is generated more than once, it is given the same goal-id. The goal-id of the supplementary fact (context) is that of the goal that is generated from it, and the goal-id of an answer is the goal-id of the goal that generated it. In contrast, each supplementary fact has a different *cont_id*. The *par_id* of a query is inherited from a supplementary fact that generated it. But since a query may be generated independently from several different supplementary facts, only one of the copies of the query (with its associated *par_id* value) is retained, if subsumption-checking is used. The same is true of *par_id* values for answer facts.

5.11.1 Memoization for Other Evaluation Schemes

The optimizations described in this chapter work at the level of rule application, and are essentially independent of the control strategy used during evaluation. They can be applied to other memoing evaluation schemes such as QSQR [Vie86, Vie88] and Alexander [RLK86, Sek89]. They can also be used in conjunction with techniques that order the inferences made in a bottom-up evaluation (e.g., [RSS90, RSS92a, GGZ91, SR91]).[15]

It is also possible to use the idea of persistent versioning to implement memoization of goals and answers in Extension Tables [Die87].[16] However, since Extension Tables uses the basic tuple-at-a-time depth-first mechanism of Prolog, the connection between goals and answers is implicitly maintained and "return unifications" are not explicitly performed. (A negative consequence is that the method is not complete even though it does memoing.) Since variables in the run-time stack and the heap may have to be versioned, it appears that fairly large portions of memory have to be versioned. In Opt-NGBU evaluation, we can avoid versioning rule variables in most cases, and for programs that (with adornment) generate only ground facts and queries, all bindenvs are empty, and have no versioning costs. While these considerations do not affect the asymptotic cost of versioning, the constant overheads for versioning are likely to be higher for Extension Tables.

QSQR (like its extension QoSaQ) is a top-down evaluation strategy that is closely related to the bottom-up evaluation of Supplementary Magic programs. QSQR is set-oriented, and represents goals and answers explicitly, much like Supplementary Magic. QSQR has been implemented for Datalog, for which versioning is not important since large data structures are not created. However, there appears to be no inherent problem in using QSQR for general logic programs. The techniques we developed in this chapter (as well as the corresponding analysis) can be applied with minor modifications to QSQR.

5.12 Conclusion

The results in this chapter are significant in two ways. First, they provide an efficient memoing technique for programs that generate non-ground facts. This is significant since naive techniques for handling non-ground facts in memoing evaluations are inefficient, and we do not know of any other optimizations that are useful in this context. The cost benefits are illustrated by the programs that we discussed. Second, they extend

[15]Some of these techniques modify Magic rewriting in minor ways. Corresponding changes may need to be made in our optimization technique.

[16]We discuss the ET_{interp} algorithm since it has better asymptotic properties; similar remarks apply to the ET* algorithm, which additionally repeats some computation.

our understanding of the similarities between top-down and bottom-up further than previous results, which considered only programs that generated only ground facts. We have shown that bottom-up evaluation is asymptotically close to Prolog even in the worst case (within a factor of $\log\log m$, where m is bounded by the cost of Prolog evaluation). There is much current research in the area of persistent versioning schemes. If a more efficient versioning scheme is developed, we can reduce the overheads in our scheme correspondingly. We have implemented the optimization techniques described in this chapter (modulo tail-recursion optimization) on the CORAL deductive database system.

Chapter 6

Optimization of Aggregation

In this chapter we develop an optimization technique for bottom-up evaluation, using a notion of relevance of facts to some aggregate operations such as *min* and *max*. Our notion of relevance (Section 6.3) can be seen as an extension of the notion of relevance used in optimizations such as Magic sets rewriting [BMSU86, BR87b, Ram88]. The optimization technique consists of two parts — a rewriting technique that "pushes" aggregate selections into rules in the program (Sections 6.4 and 6.5), and an evaluation technique that makes use of aggregate selections when evaluating the rewritten program (Section 6.6).

The optimization technique is able to detect many facts as irrelevant, and avoids using them to make derivations. As an example of the power of our techniques, we start with a naive program to find shortest paths, and show how our optimization techniques deduce the "optimality principle" for this program. The optimized evaluation of this program is equivalent to Dijkstra's algorithm (Example 6.6.3).

6.1 Introduction

Database query languages such as SQL provide aggregation operations that let one compute aggregate values over sets of answers. The use of aggregation with recursive queries has been considered by several researchers (e.g., [BNR+87, MPR90]), and has been implemented in LDL [NT89]. Generalized forms of transitive closure with aggregation are a restricted form of recursive queries with aggregation (and can be expressed using the notation of LDL [NT89]). Several researchers (e.g., [RHDM86, ADJ88, CN89, Ede90]) have considered optimizations for this special class of programs. The advantages of the richer language of recursive queries with aggregation is clear, but unless effective optimization techniques are developed, the performance of specialized systems based on supporting the limited class of generalized transitive closure queries cannot be matched. In this chapter we consider optimizations of recursive queries with aggregate operations.

Consider the (very naive) program shown in Figure 8, for computing the lengths of shortest paths between nodes in the relation $edge(X,Y,C)$, where C is the length of an edge from X to Y. It essentially enumerates all path lengths and chooses shortest path lengths among them. The notation $s\_p\_length(X,Y,min(\langle C\rangle))$ in the head of rule $R1$ denotes that for each value of X,Y all possible C values that are generated by the body of the rule are collected in a set, and the *min* aggregate operation is applied on the set of values. For each value of X and Y, an *s_p_length* fact is created with the result of the *min* operation as the third argument.

$R1:\ s\_p\_length(X,Y,min\langle C\rangle): -path(X,Y,C).$
$R2: path(X,Y,C1): -\ path(X,Z,C), edge(Z,Y,EC), C1 = C + EC$
$R3: path(X,Y,C)\ \ : -\ edge(X,Y,C).$
Query: ?-$s\_p(X,Y,C)$.

Figure 8: Program Simple_ShortCost

$R1: shortest\_path(X,Y,P,C)\quad : -\ s\_p\_length(X,Y,C), path(X,Y,P,C).$
$R2: s\_p\_length(X,Y,min\langle C\rangle)\quad : -\ path(X,Y,P,C).$
$R3: path(X,Y,[edge(Z,Y)|P],C1): -\ path(X,Z,P,C), edge(Z,Y,EC), C1 = C + EC.$
$R4: path(X,Y,[edge(X,Y)|nil],C): -\ edge(X,Y,C).$
Query: ?-$s\_p(X,Y,P,C)$.

Figure 9: Program Simple_ShortPath

The formulation of the program as above is desirable since it is declarative, can be queried in many different ways and is easy to write. It is easily augmented with additional constraints such as "the edges all have a given label" (for instance, flights on United Airlines alone must be considered), or "there must be no more than three hops on the flight". The standard bottom-up evaluation of such a program is extremely inefficient since it constructs paths of every possible length in the graph, and generates an infinite number of facts with cyclic graphs. In contrast, the above problem can be solved in polynomial time using either Warshall's algorithm or Dijkstra's shortest path algorithm (see [AHU74]). It can also be evaluated efficiently if it is expressed using specialized operators for transitive closure ([RHDM86, ADJ88, CN89]).

We propose an optimization technique for bottom-up evaluation, using a notion of relevance of facts to some aggregate operations such as *min* and *max*. Our notion of relevance can be seen as an extension of the notion of relevance used in optimizations such as Magic sets rewriting [BMSU86, BR87b, Ram88]. To demonstrate the power of our techniques, we use a more complex version of the shortest path program, that actually computes paths (Figure 9), and informally present some of the basic ideas behind our optimization technique, in the following example.

Example 6.1.1 Consider Program Simple_ShortPath (Figure 9). The predicate $path(X,Y,P,C)$ is defined to compute paths between each pair of nodes X,Y, with P being a list of nodes on the path, and C being the length of the path. The predicate $s\_p\_length(X,Y,C)$ defined in rule $R2$ finds the length C of the shortest path from X to Y for each pair of nodes X,Y. The use of s_p_length in rule $R1$ selects *path* facts corresponding to shortest paths.

Aggregate operation *min* has the property that non-minimal values in a set are unnecessary for the aggregate operation on the set. Using this property, we can deduce that a fact $path(a,b,p1,c1)$ is relevant to the rule defining the query predicate *shortest_path* only if there is no fact $path(a,b,p2,c2)$ such that $c2 < c1$. We use tests called *aggregate selections* to check whether a fact is relevant; conditions such as the above are used in the tests.

The rewriting (automatically) deduces an aggregate selection on the occurrence of the predicate *path* in rule $R2$; only facts with minimum length values satisfy the aggregate selection. It then "pushes" this aggregate selection into rules that define *path*, and propagates the selections through the program.

The rewriting algorithm outputs a program containing aggregate selections on the predicates. For Program Simple_ShortPath, the main difference between the rewritten program and the original program is that every occurrence of *path* in the rewritten program has an aggregate selection that selects shortest paths. We discuss the rewritten program after introducing the notation used to express aggregate selections.

The evaluation phase of our technique makes use of the aggregate selection on *path*, and deletes facts on which the aggregate selection test fails (namely all non-minimal paths for each pair of nodes). We can optimize the evaluation further by ordering the use of facts in the evaluation: we hide newly generated facts, and expose after each iteration the *path* fact with minimum length among all hidden *path* facts.

Ordering the use of facts as above, reduces the time complexity to the same as that of Dijkstra's algorithm ($O(E \cdot \log V)$ on a graph with E edges and V nodes), if we store only one shortest path between each pair of nodes. We discuss details in Section 6.6.2. The evaluation mechanism also works (with a higher time complexity) when edge lengths are negative, so long as there are no negative length cycles. $\Box$

Recently Ganguly et al. [GGZ91] independently examined Datalog programs with *min* or *max* aggregate operations. Their work addresses problems that are similar to those that we consider, but the approaches are different and the techniques are complementary. We present a comparison of our techniques with those of Ganguly et al. in Section 6.8.1, and describe several advantages of our approach. Knuth [Knu77] generalizes Dijkstra's algorithm to deal with a class of "superior context free grammers". Our evaluation technique generalizes Knuth's techniques in a very natural manner, and the algorithms reduce to Knuth's algorithms in the special case of "superior context free grammers". We discuss this issue briefly in Section 6.8.1.

We note that the evaluation techniques presented in this chapter are orthogonal to the optimization techniques for non-ground facts that were presented in Chapter 5, although we restrict the use of non-ground terms with aggregation. We discuss this issue briefly in Section 6.8. There are programs, such as the program in Example 5.9.2, where both optimizations are of use.

The rest of the chapter is organized as follows. We present basic definitions and background material in Section 6.2. Our notion of relevance is developed in Section 6.3, where we also introduce aggregate selections and constraints as a way of specifying relevance information. Techniques for propagation of aggregate selections and constraints through single rules are developed in Section 6.4. In Section 6.5 we present an algorithm to rewrite programs by propagating aggregate selections through the program, starting from the query. In Section 6.6 we show how to evaluate rewritten programs. We discuss extensions and related work in Section 6.8.

6.2 Definitions and Background Material

We use $Vars(t)$ to denote the set of variables that occur in a term t. Similarly, $Vars(\bar{t})$ denotes the set of variables that occur in a tuple of terms $\bar{t}$. Given a domain D, we use the notation $M(D)$ to denote multisets[1] of elements from D.

We define the *predicate dependence (PD) graph* of a program as the digraph whose nodes are the predicates of the program, and whose edges are defined as follows: There is an edge from predicate a to predicate b if

[1] Also referred to as bags.

there is a rule defining b that uses predicate a in its body. The *strongly connected components (SCCs) of a program* are the strongly connected components of its predicate dependence graph. The *reduced PD graph* of a program is defined as the result of collapsing together all nodes in the PD graph that belong to the same SCC. The reduced PD graph is acyclic, and defines a partial ordering of the SCCs of a program. We say that an SCC $S1$ is *lower than* SCC $S2$ if $S1$ precedes $S2$ in the partial ordering defined by the reduced PD graph.

Example 6.2.1 Consider the following program.

$R1 : p(X) :- q(Y), d(X,Y).$
$R2 : q(X) :- p(Y), e(X,Y).$
$R3 : p(X) :- r(X).$

This program has four SCCs — one containing q and p, one containing d, one containing e, and one containing r. We use the set of predicates in an SCC to refer to the SCC. The partial ordering of the SCCs is as follows: each of $\{d\}$, $\{e\}$ and $\{r\}$ precedes $\{p, q\}$. □

We view an aggregate function as any function $agg\_f : M(D) \rightarrow D1$ for some domains D and $D1$. Note in particular that $D1$ could be the same as D, as is the case for most aggregate functions we consider, such as *min*, *max*, *sum* etc. We also allow $D1$ to be $M(D)$, thereby allowing aggregate functions such as $least_k$, that returns a multiset of the least k elements from a given multiset. Note that if we consider the domain to be 2^D, i.e., sets of elements from D rather than multisets of elements from D, the aggregate functions are still applicable, since $2^D \subset M(D)$.

The *cardinality* of an element in a multiset is defined in a straightforward manner. Multiset membership is defined using cardinality:

$$s \in S \equiv cardinality(s, S) \neq 0$$

Similarly, multiset containment is defined as

$$S1 \subseteq S2 \equiv \forall s \in S1, cardinality(s, S1) \leq cardinality(s, S2)$$

Other set operations $\subset, \supset, \supseteq$ and $=$ are similarly extended to multisets. Set difference "$-$" is extended to multisets by the following definition:

$\forall s$, if $cardinality(s, S1) - cardinality(s, S2) \geq 0$,
    then $cardinality(s, S1 - S2) = cardinality(s, S1) - cardinality(s, S2)$
    else $cardinality(s, S1 - S2) = 0$.

We also define a binary operation $\setminus$ as follows:

$\forall s \in S2$, $cardinality(s, S1 \setminus S2) = 0$
$\forall s \notin S2, cardinality(s, S1 \setminus S2) = cardinality(s, S1)$

In this chapter we consider definite clause programs extended with the aggregation operations that we describe later in this section. We assume that the programs are range-restricted.[2] This means that only

[2]That is, every variable in the head of each rule also appears in the body of the rule, and all facts are ground.

ground terms can be generated, which is reasonable for the most part in the context of aggregate functions, since the result of aggregate functions (consider, for example, *min*) on non-ground values is usually not well-defined. Non-ground terms are useful in arguments of facts that are not aggregated upon, and in Section 6.8 we discuss how the above restriction can be relaxed in some cases. We assume that program transformations such as Magic Sets have already been carried out; their use is largely orthogonal to the optimizations described in this chapter.

In order to define the semantics of a program, we have to first define a universe for the program. In this we follow [BNST91, BRSS92], where extended Herbrand universes are defined. Such universes allow terms that are sets; the universes are easily extended to allow multisets (for instance by encoding multisets as sets of ordered pairs $\langle element, cardinality\rangle$). Note that although we allow the generation of multisets through aggregation, we assume that the relations in the program are sets of facts, and not multisets (i.e., duplicate elimination is done when evaluating the program). It is not hard to relax this assumption, but this assumption simplifies the discussion.

The syntax that we use for aggregation with set/multiset grouping is very similar to that used in LDL [NT89] for grouping. The syntax is as follows.

$$p(\overline{t}, agg\_f\langle Y\rangle) :- q(\ldots).$$

We refer to the argument $agg\_f\langle Y\rangle$ as the *grouping argument*, and Y as the *grouping variable*. The variables in $\overline{t}$ are referred to as *group-by variables*. For simplicity in describing our algorithms, we assume that there is at most one grouping argument in the head of a rule, and we usually show the grouping argument as the last argument of the head of the rule. These restrictions are easy to relax since every program can be rewritten to be in the required form. For simplicity, we also assume that there is at most one literal in the body of a rule that uses aggregation. This assumption can be relaxed using a straightforward rewriting.

Definition 6.2.1 We say that the fact $p(\overline{a}, v)$ *follows from* a rule

$$p(\overline{t}, agg\_f\langle Y\rangle) :- q(\ldots).$$

and a given set of facts I if the following holds:

1. Let S be the set of instantiations of the variables in the rule s.t. the instantiation of $q(\ldots)$ is in I. Let $\overline{X} = Vars(\overline{t})$, and $\overline{x}$ an instantiation of the variables in $\overline{X}$ that maps $\overline{t}$ to $\overline{a}$.

   Then $\pi_Y \sigma_{\overline{X}=\overline{x}} S \neq \emptyset$, and $v = agg\_f(\pi_Y \sigma_{\overline{X}=\overline{x}} S)$, where π is a multiset projection (i.e., it does not do duplicate elimination).
2. The set of facts I contains all true q facts, (and all other q facts are false).

Given a rule R that uses aggregation, and a set of facts I, we define $T_R(I)$ as the set of all facts that follow from R and I. $\Box$

Example 6.2.2 Suppose we have a rule

$$p(X, Y, min\langle C\rangle) :- q(X, Y, C).$$

Given that the set of all true q facts is $\{q(1,2,3), q(1,2,5), q(1,3,4)\}$, the facts $p(1,2,3)$ and $p(1,3,4)$ follow using the rule. $\Box$

The semantics used in LDL differs from the above in that π is a set projection operator. For aggregate operations such as *max* or *min* the difference in semantics is irrelevant. For other aggregate operations such as *sum*, we can get the set projection semantics by using $agg\_f\langle set\langle Y\rangle\rangle$ instead of $agg\_f\langle Y\rangle$.

A program is said to be *stratified* if for every rule in the program that uses aggregation in the head, every predicate used in the body is in a lower SCC than the SCC of the head predicate.

Example 6.2.3 The following program is a minor variant of the program in Example 6.2.1. The SCCs of this program are the same as those of the program in Example 6.2.1.

$$
\begin{aligned}
&R1 : p(X) &&:- \; q(Y), d(X,Y).\\
&R2 : q(X) &&:- \; p(Y), e(X,Y).\\
&R3 : p(min\langle X\rangle) &&:- \; r(X).
\end{aligned}
$$

This program is stratified since in rule $R3$, the body predicate r is in a lower SCC than the head predicate p. If we replaced r by q, the program would not be stratified, since p and q in rule $R3$ are in the same SCC. $\square$

We make the following assumption:

A0: All programs considered in this chapter are stratified.

Definition 6.2.1 requires that the set of all true facts for the body predicate be available before aggregation can be used. For stratified programs, this can be done in a fairly straightforward fashion. We describe the semantics of stratified programs informally below. See [BNST91] for a formal definition. We define the semantics SCC by SCC, proceeding in a total order consistent with the partial ordering of the SCCs. The semantics for each SCC defines the set of true facts for predicates in the SCC. The semantics of base predicates is given by the set of facts in the database. Now consider an SCC S_i, and suppose we have defined the semantics for all lower SCCs. Definition 6.2.1 now defines T_R for rules in S_i that use aggregation, since the set of all true facts for lower SCCs is fixed by the semantics of the lower SCCs. For other rules, T_R is as defined in Section 2.1.3. Define

$$T_{S_i}(I) = \cup_{R \in S_i} T_R(I)$$

The semantics of SCC S_i is defined to be the least fixpoint of T_{S_i}, given the set of facts for lower SCCs.[3] The semantics of the program is defined to be the union of the semantics of each SCC in the program.

A stratified program can be evaluated using Semi-Naive evaluation, an SCC at a time, in a total order of SCCs that is consistent with the partial order of the SCCs [BNST91, BNR$^+$87]. We describe this process briefly. Assume that all SCCs that precede a given SCC S_i have been evaluated. This is trivially true for SCCs that have only base predicates. We can now evaluate SCC S_i using Semi-Naive evaluation as follows. For all predicates in lower SCCs, the fixpoint has been evaluated, and all such predicates are considered as base predicates for the purpose of Semi-Naive evaluation. Rule application is generalized to handle aggregation in a straightforward fashion, since all body predicates in a rule with aggregation are defined in lower SCCs, and hence are treated as base predicates. Semi-Naive rewriting and Semi-Naive evaluation are performed as usual, for the rules in SCC S_i. Semi-Naive evaluation of the program terminates when all SCCs in the program have been evaluated.

[3]We can show that T_{S_i} is monotone and continuous, and hence it has a least fixpoint [BNST91].

Semantics can be given to programs that use non-stratified aggregation, and there are evaluation mechanisms for several such classes of programs. We do not explore this issue here, but refer the reader to [Ros90, KS91, RS92, BRSS92, RSS92a, Van92] for more details.

6.3 Views of Relevance In Logic Programs

The idea of relevance of facts to a query is used by Prolog and other top-down evaluation techniques, as well as by program rewriting techniques such as Magic [BR87b, Ram88]. Suppose we have a rule

$$R : p(\overline{t}) \text{:} - q_1(\overline{t_1}), q_2(\overline{t_2}), \ldots, q_n(\overline{t_n}).$$

Assume for simplicity that we have a left-to-right rule evaluation (in the fashion of Prolog). Then a fact $q_i(a_i)$ is relevant if there is an instantiation

$$R' : p(a) \text{:} - q_1(a_1), q_2(a_2), \ldots, q_i(a_i).$$

of (the head and first i body literals of) R such that the head fact $p(a)$ is relevant, and all instantiated facts $q_1(a_1), \ldots, q_{i-1}(a_{i-1})$ have been derived. Thus, the notion of relevance is local to a rule and to a set of facts that can instantiate it.

In contrast, in the shortest path problem we can decide that a particular fact $path(a, b, p1, c1)$ is irrelevant if a shorter path (fact) has been found. Such information is "global", in the sense that relevance depends on facts other than those used to instantiate a rule. We develop this notion of relevance for programs with aggregate operations in the rest of this section, in three steps. (1) If agg_f is an aggregate function and S a multiset of values, we consider when some values in S can be ignored without affecting $agg\_f(S)$ (Section 6.3.1). (2) We use the ideas of step 1 to define when a fact is relevant (Section 6.3.2). (3) We introduce *aggregate selections* and *aggregate constraints* as a way of explicitly identifying irrelevant facts (Section 6.3.3).

6.3.1 Relevance and Aggregate Functions

Given a multiset of values and an aggregate function on the multiset, not all the values may be needed to compute the result of the aggregate function. For instance, if the aggregate function is min, no value except the minimum value is needed. We now formalize the notion of values being unnecessary for aggregate functions.

Definition 6.3.1 (Incremental Aggregate Selector (IncSel) Functions) Let agg_f be an aggregate function $agg\_f : M(D) \rightarrow D1$, for some domains D and $D1$. We say that agg_f is an *incremental aggregate selector* (*IncSel*) function if there exists a function $unnecc_{agg\_f} : M(D) \rightarrow 2^D$ such that

1. $\forall S \in M(D), \forall S1, (S \backslash unnecc_{agg\_f}(S)) \subseteq S1 \subseteq S \Rightarrow agg\_f(S1) = agg\_f(S)$
2. $unnecc_{agg\_f}$ is monotone. i.e., $\forall S1 \subseteq S2,$ s.t. $S2 \in M(D)$, $unnecc_{agg\_f}(S1) \subseteq unnecc_{agg\_f}(S2)$
3. $\forall S \in M(D), unnecc_{agg\_f}(S) = unnecc_{agg\_f}(S \backslash unnecc_{agg\_f}(S))$
4. $unnecc_{agg\_f}$ does not map all elements of $M(D)$ to $\emptyset$.

□

Given a multiset of facts S, the set of facts $unnecc_{agg\_f}(S)$ is "unnecessary" in the following sense: Values in $unnecc_{agg\_f}(S)$ can be dropped from S without affecting the result of $agg\_f(S)$, due to Part 1 of the above condition. Part 2 of the above condition lets us detect unnecessary values before the entire multiset of values is computed—when we have computed some $S1 \subset S$, any value detected as unnecessary for agg_f on $S1$ is also guaranteed to be unnecessary for agg_f on S; a value that is necessary for $S1$ may however be unnecessary for S. Part 3 of this condition ensures that if a value is detected to be unnecessary for an aggregate operation on a multiset, it will continue to be detected as unnecessary if we discard unnecessary values from the multiset[4]. Part 4 of the condition ensures that the definition of IncSel functions is not trivialized by the use of a trivial $unnecc_{agg\_f}$ function.

Consider an IncSel function agg_f on domain $M(D)$. There may be more than one possible function $unnecc_{agg\_f}$ as required by the definition of IncSel functions.

Definition 6.3.2 ($\mathbf{unnecessary_{agg\_f}}$) For each incremental aggregate selector function agg_f that is allowed in our programs, a function $unnecc_{agg\_f}$ (as above) is chosen, and is denoted by $unnecessary_{agg\_f}$.

The function $necessary_{agg\_f} : M(D) \to 2^D$ is defined as

$$necessary_{agg\_f}(S) = set(S \backslash unnecessary_{agg\_f}(S))$$

□

We do not consider how this choice is made, but assume it is made by the designer of the system based on the following criterion. Given two such functions f and g, we say $f \geq' g$ iff $\forall S \subseteq D, f(S) \supseteq g(S)$; clearly $>'$ (the strict version of $\geq'$) is an (irreflexive) partial order. Preferably, a function that is maximal under the (irreflexive) partial order $>'$ is chosen.

Note that $unnecessary_{agg\_f}(S)$ could be infinite. We do not construct $unnecessary_{agg\_f}(S)$, but require that we can efficiently test for the presence of a value in $unnecessary_{agg\_f}(S)$, for finite S.

Example 6.3.1 The function min on reals, with $unnecessary_{min}(S) = \{x \in D \mid x > min(S)\}$ is an IncSel function. The function max on reals with $unnecessary_{max}$ symmetrically defined is also an IncSel function. Other examples (with the functions $unnecessary_{agg\_f}$ appropriately defined), include the aggregate function that selects the kth largest element of a multiset for some constant k, and the aggregate function that sums up the k largest elements of a multiset. □

Assumption 6.3.1 *In the rest of this chapter we assume that the optimization techniques are applied only on IncSel functions.* □

We also assume that a suite of IncSel functions and the corresponding functions $unnecessary_{agg\_f}$ are given to us. In an actual implementation we would expect the system implementor to define such as suite of functions.

[4]Part 3 of Condition IncSel is used in Theorem 6.6.1 to show that inferences are not repeated. None of the other results require aggregate functions to satisfy this condition.

6.3.2 Relevance of Facts

We now use the notion of necessity with respect to an aggregate function in defining our extended notion of relevance of facts. The semantics for a program defines a model for the program, and our notion of relevance is defined with respect to this model (which we call the *intended model of the program*).

Definition 6.3.3 (Relevance of Facts) Consider a program P with a query on it. A fact $q(\overline{a})$ is *relevant to the query* iff one of the following is true:

1. $q(\overline{a})$ is an answer to the query, or
2. $q(\overline{a})$ occurs in the body of an instantiated rule without aggregation in the head such that every literal in the body is true in the intended model, and the head fact of the rule is relevant to the query, or
3. There is a (ground) fact $p(\overline{a}, v)$ that is relevant to the query, and a rule R in the program

   $$R : p(\bar{t}, agg\_f\langle Y\rangle)) \colon - q(\overline{t_1}).$$

   such that

   (a) Let S be the set of all possible substitutions σ such that $\overline{t_1}[\sigma] = \overline{a_1}$, and $q(\overline{t_1})[\sigma]$ is true in the intended model. Let $Y[S]$ denote the multiset of values for Y generated by substitutions in S. Then $v = agg\_f(Y[S]))$.

   (b) There is a $\sigma \in S$ s.t. $q(\overline{a}) = q(\overline{t_1})[\sigma]$, where $Y[\sigma] \in necessary_{agg\_f}(Y[S])$.

□

A fact is said to be *irrelevant to the query* if it is not relevant to the query. In future, we simply say relevant (resp. irrelevant) when we mean "relevant to the query" (resp. "irrelevant to the query").

Example 6.3.2 Consider a program with one rule

$$R : p(X, min\langle Y\rangle) \colon - q(X, Y).$$

and facts $q(5,4), q(5,6)$ and $q(5,3)$. Let the query on the program be $?p(X,Y)$. Fact $p(5,3)$ is generated as an answer. With $X = 5$, the set of facts that match the body of the rule have Y values of $3, 4$ and 6, of which only 3 is necessary for *min*. Hence $q(5,3)$ is relevant to the query. $q(5,3)$ is a base fact, and no facts are used to derive it. Therefore there are no other relevant facts. Hence $q(5,4)$ and $q(5,6)$ are irrelevant to the query, while $q(5,3)$ is relevant.

Also, by the above definition, for the shortest path length program (Figure 8) all *path* facts, except those corresponding to shortest paths, are irrelevant. This can be seen by working backwards from answers to the query. Facts for the predicate *s_p_length* are the only facts that are directly relevant (by Part 1 of the definition). Of the *path* facts used to derive these facts, the only relevant ones are shortest paths (by Part 3 of the definition). By examining the rules for *path*, we can verify that any *path* fact that is used to derive a shortest path, and is relevant by Part 2 of the definition, is itself a shortest path. □

Our extended notion of relevance is very tight, and in general we may not be able to determine the relevance of a fact without actually computing the intended model of the program. The techniques we present will use sufficient but not necessary conditions to test for irrelevance. During the evaluation of some programs we may generate a fact, and later discover that it is irrelevant, for instance when some other "better" fact is generated. Once a fact is found to be irrelevant, by "withdrawing" the fact we may be able to determine that other facts generated using it can no longer be generated, and hence can also be "withdrawn". The cost of such cascading withdrawals could be very high, and so we confine ourselves to only discarding irrelevant facts. Although not "withdrawing" computation could result in some additional irrelevant computation, the gains in efficiency from our optimization without "withdrawing" computation can still be significant.

6.3.3 Aggregate Constraints and Selections

We now introduce some concepts that allow us to specify relevance information. Informally, *sound aggregate selections* are used to specify tests for relevance of facts—if there is a sound aggregate selection on a predicate in our rewritten program, and a fact for the predicate does not satisfy the selection, the fact is irrelevant. Aggregate selections are introduced by our rewriting algorithm and the information is used by our evaluation algorithm. The syntax (using a variant of Starburst SQL *groupby*) and semantics of aggregate selections are described in the next few definitions.

Definition 6.3.4 (Atomic Aggregate Selection) An *atomic aggregate selection* has the following syntax:

$$c(\overline{u}) : groupby(p(\overline{t}), [\overline{X}], agg\_f(Y))$$

Here $c(\overline{u})$ denotes a literal or a conjunction of literals, and $\overline{X}$ a set of variables such that $\overline{X} \subseteq Vars(\overline{t})$. We must have $Y \in Vars(\overline{t})$, and agg_f must be an IncSel function.

Consider a program P with an associated intended model. Given the set of facts for predicate p in the intended model of P, we have a set of instantiations of $\overline{t}$. Since $\overline{X} \subseteq Vars(\overline{t})$ and $Y \in Vars(\overline{t})$, for each value $\overline{d}$ of $\overline{X}$ in the set of instantiations of $\overline{t}$, we have a corresponding multiset of values for Y; we denote this multiset by $S_{\overline{d}}$. We construct (conceptually) a relation $unnecc\_agg(\overline{X}, Y)$ with a tuple $(\overline{d}, e)$ for each $\overline{d}$, and each $e \in unnecessary_{agg\_f}(S_{\overline{d}})$.

Let $c(\overline{a})$ be a ground conjunction. We say that $c(\overline{a})$ *satisfies the atomic aggregate selection* s_i iff there exists a substitution σ such that (1) $c(\overline{a}) = c(\overline{u})[\sigma]$, (2) σ assigns ground terms to all variables in $Vars(\overline{u}) \cup \overline{X} \cup \{Y\}$, and (3) $(\overline{X}, Y)[\sigma]$ is not in $unnecc\_agg$ [5]. $\square$

In the above definition, the variables in $[\overline{X}]$ are called *group-by variables* and the variable Y is called the *grouping variable* in the atomic aggregate selection. The variables in the set $((Vars(\overline{t}) - \overline{X}) - \{Y\})$ are local to the groupby, and cannot be quantified or instantiated from outside the groupby.

[5]Note that the relation $unnecc\_agg$ could be infinite. To actually perform the test, we could take an instantiation of Y, and test if it is in $unnecessary_{agg\_f}(\overline{X})[\sigma]$ without actually constructing the whole (possibly infinite) set $unnecessary_{agg\_f}(\overline{X})[\sigma]$, or the (possibly infinite) relation $unnecc\_agg$.

Example 6.3.3 The following is an example of an atomic aggregate selection:

$$path(X, Y, P, C) : groupby(path(X, Y, P1, C), [X, Y], min(C))$$

In the above atomic aggregate selection, the group-by variables are X and Y, and the grouping variable is C. We have not specified where the literal $path(X, Y, P, C)$ occurs — it could be, for instance, a literal in a rule body, or it could be taken to refer to facts for the predicate *path*; we shall make the use of the literal more precise in succeeding definitions.

Suppose the set of facts in *path* is

$path(a, b, [a, c, b], 20).$
$path(a, b, [a, b], 30).$
$path(a, c, [a, c], 10).$
$path(c, b, [c, b], 10).$

The ground literal $path(a, b, [a, b], 30)$ does not satisfy the aggregate selection — the literal binds the group-by variables X, Y to a, b, and the C values for this group are 20 and 30; hence, 30 is irrelevant for the *min* aggregate function on this group. However, $path(a, b, [a, c, b], 20)$ satisfies the aggregate selection. The ground literal $path(d, e, [d, e], 200)$ satisfies the selection, since there is no fact, in the set of facts for *path*, that binds the group-by arguments X, Y to d, e, and hence no value is classified as irrelevant for this group. $\Box$

Definition 6.3.5 (Aggregate Selection) An *aggregate selection* s is a conjunction of atomic aggregate selections, $s = (s_1 \wedge s_2 \wedge \ldots \wedge s_n)$. A ground conjunction $c(\overline{a})$ *satisfies an aggregate selection* $s = (s_1 \wedge s_2 \wedge \ldots \wedge s_n)$ iff it satisfies each of the atomic aggregate selections s_i individually. $\Box$

We use the short form $c(\overline{u}) : g1 \wedge g2$ to denote $(c(\overline{u}) : g1) \wedge (c(\overline{u}) : g2)$. We often say "the aggregate selection s on the body of R" to denote the aggregate selection $c(\overline{u}) : s$, where $c(\overline{u})$ is the body of rule R. Note that a conjunction of aggregate selections is also an aggregate selection.

Our approach to rewriting the program consists of placing aggregate selections on literals and rule bodies in the program in such a fashion that if a fact/rule instantiation does not satisfy the aggregate selection it is guaranteed to be irrelevant. Hence we define the concept of sound aggregate selections formally below.

Definition 6.3.6 (Sound Aggregate Selection) An aggregate selection s is a *sound aggregate selection on the body of a rule* R iff only irrelevant facts are produced by instantiations of the body of R that do not satisfy s.

An aggregate selection s is a *sound aggregate selection for a literal* $p(\overline{t})$ in the body of a rule R iff only irrelevant facts are produced by instantiations of R that use for literal $p(\overline{t})$ any fact $p(\overline{a})$ that does not satisfy s.

An aggregate selection s is a *sound aggregate selection on a predicate* p iff any fact $p(\overline{a})$ is irrelevant if it does not satisfy s. $\Box$

Given a sound aggregate selection on a literal/rule, we can (partially) test during an evaluation whether a fact or an instantiated rule satisfies it. The extension of each predicate p at that point is a subset of the extension of p in the intended model of the program. Since the aggregate functions are incremental aggregate selectors, an answer of "no" at that point means that the answer would be "no" in the intended model of the program, and hence the fact/instantiation is irrelevant. However, an answer of "yes" is conservative, since the fact/instantiation may be detected to be irrelevant if all facts in the intended model were available.

Example 6.3.4 Consider an aggregate selection

$$path(X,Y,P,C) : groupby(path(X,Y,P,C),[X,Y],min(C))$$

Suppose we have two facts $path(a,b,\_,2)$ and $path(a,b,\_,3)$ at a point in the computation. Then we know that $path(a,b,\_,3)$ does not satisfy the selection. Later in the computation we may derive a fact $path(a,b,\_,1)$. At this point we find that $path(a,b,\_,2)$ also does not satisfy the selection. □

We define sound aggregate constraints next—they differ slightly from sound aggregate selections, and we use them in our rewriting algorithm to generate aggregate selections.

Definition 6.3.7 (Sound Aggregate Constraint) An aggregate selection s is a *sound aggregate constraint* for predicate p iff every fact that can be derived for p satisfies the aggregate selection s. □

The following are technical definitions that we use primarily to ensure that the aggregate selections that we generate can be tested efficiently. The motivation is that the fact/rule instance on which we have an aggregate selection must bind all the variables in the aggregate selection.

Definition 6.3.8 (Free Variables) The *free* variables of an atomic aggregate selection

$$c(\overline{u}) : groupby(p(\overline{t}),[\overline{X}],agg\_f(Y))$$

are the variables in the set $(Vars(\overline{X}) \cup \{Y\})$. The other variables in an atomic aggregate selection are *bound* variables (since the semantics of atomic aggregate selections quantifies these variables within the scope of the atomic aggregate selection).

The free variables of aggregate selection $s = s_1 \wedge \ldots \wedge s_n$ are those variables that are free in at least one of the atomic selections aggregate s_i. □

Definition 6.3.9 (Restrictions of Aggregate Selections) An atomic aggregate selection s_i is said to be *restricted* to a given set V of variables if every free variable in s_i occurs in V. Let $s = (s_1 \wedge s_2 \wedge \ldots \wedge s_n)$. Then

$$restriction(s,V) = \wedge\{s_i \mid s_i \text{ is restricted to } V\} \quad \Box$$

Example 6.3.5 Consider the following selection:

$$s = c(\overline{u}) : groupby(path(X,Y,P,C),[X,P],min(C))$$
$$\wedge\, groupby(path(X,Y,P,C),[X,Y],min(C))$$

The free variables of s are X, Y, P and C, and

$$restriction(s,\{X,Y,C\}) = c(\overline{u}) : groupby(path(X,Y,P,C),[X,Y],min(C)) \quad \Box$$

$R1: shortest\_path(X,Y,P,C) \quad :- s\_p\_length(X,Y,C), path\_s1(X,Y,P,C).$
$R2: s\_p\_length(X,Y,min\langle C\rangle)) \quad :- path\_s1(X,Y,P,C).$
$R3: path\_s1(X,Y,[edge(Z,Y)|P],C1) :- path\_s1(X,Z,P,C), edge(Z,Y,EC), C1 = C + EC.$

$R4: path\_s1(X,Y,[edge(X,Y)|nil],C) :- edge(X,Y,C).$
Selections::
$s1 = path\_s1(X,Y,P,C): groupby(path\_s1(X,Y,P,C),[X,Y],min(C)).$

Figure 10: Program Smart

6.4 Generating Aggregate Constraints and Selections

We present a quick overview of the next few sections of the chapter . We develop our algorithm for propagating relevance information in two steps. (1) In this section we present a collection of techniques for generating sound aggregate selections. (2) In Section 6.5, we present our main rewriting algorithm, Algorithm Push_Selections, which uses these techniques as subroutines. In Section 6.6, we examine an evaluation mechanism that can take advantage of sound aggregate selections on predicates that are generated by the rewriting mechanism.

As a preview of what the techniques can achieve, consider Program Simple_ShortPath (Figure 9). The result of rewriting is Program Smart, shown in Figure 10. The rewritten program uses a new predicate *path_s*1 which is a version of *path*, with the sound aggregate selection $s1$ on it. The predicate *path* itself is not present in the rewritten program. The other predicates have no aggregate selections on them. The selection $s1$ tells us that *path_s*1 facts that are not of minimum length between their endpoints are irrelevant. Deleting such facts during the evaluation leads to considerable time benefits, and is discussed in Section 6.6.2.

In the first part of this section we describe an initial set of techniques for generating aggregate constraints and selections. The techniques are shown below. Technique C1 describes a way of deducing sound aggregate constraints on predicates. Techniques BS1, BS2 and BS3 describe three ways to generate sound aggregate selections on the bodies of rules. Technique LS1 describes a simple way of deducing sound aggregate selections on literals. In Sections 6.4.1 and 6.4.2 we present a more sophisticated analysis that helps us to derive further sound aggregate selections on body literals. We note that this set of deduction rules is not complete; in Section 6.4.3 we show that it is undecidable in general whether a body literal satisfies a sound aggregate selection.

Technique C1: (Generating Aggregate Constraints)

Suppose that there is only one rule defining p, and it is of the form:

$p(\overline{t}, agg\_f\langle Y\rangle): -q(\overline{t_b})$

Let $\overline{X} = Vars(t)$, and let agg_f be an IncSel function such that

$$\forall S \in M(D), agg\_f(S) = necessary_{agg\_f}(S)$$

Then $p(\overline{t},Y): groupby(q(\overline{t_b}),[\overline{X}],agg\_f(Y))$ is a sound aggregate constraint on p.

Technique BS1: (Generating Aggregate Selections from Aggregate Constraints)

Suppose we have a rule of the form

$head(\overline{t_h})\colon -c(\overline{t_b}), p(\overline{t})$

and suppose there is an aggregate constraint on p of the form: $p(\overline{t_1}) : s$ where all free variables in s are included in $Vars(\overline{t_1})$. Suppose there exists a renaming σ of variables in $\overline{t_1}$ such that $p(\overline{t}) = p(\overline{t_1})[\sigma]$. Then $s[\sigma]$ is a sound aggregate selection on the body of the rule.

Technique BS2: (Generating Aggregate Selections from Aggregate Operations)

Suppose we have a rule of the form

$p(\overline{t}, agg\_f\langle Y\rangle)\colon -q(\overline{t_b})$

where agg_f is an IncSel function. Let $\overline{X} = Vars(\overline{t})$. Then

$$groupby(q(\overline{t_b}), [\overline{X}], agg\_f(Y))$$

is a sound aggregate selection on the body of rule R.

Technique BS3: (Generating Aggregate Selections from Other Aggregate Selections)

Consider a rule of the form

$p(\overline{t_h})\colon -body(\overline{t_b})$.

Suppose the head predicate p has a sound aggregate selection $p(\overline{t}) : s$ on it, where all free variables in s are included in $Vars(\overline{t})$.

Suppose there exists a renaming σ of free variables in s, and a substitution θ of other variables in t such that $p(\overline{t_h}) = p(\overline{t})[\sigma][\theta]$. Then $s[\sigma]$ is a sound aggregate selection on the body of the rule.

Technique LS1: (Generating Aggregate Selections On Literals)

Let s be a sound aggregate selection on the body of a rule R, and let $p(\overline{t})$ be a literal in the body of R. Then

$$p(\overline{t}) : restriction(s, Vars(\overline{t}))$$

is a sound aggregate selection on the literal $p(\overline{t})$ in the body of R.

The intuition behind Techniques C1 and BS2 is straightforward. Techniques BS1 and BS3 use an existing aggregate selection/constraint to generate a new aggregate selection. To translate an aggregate constraint $p(\overline{t_1}) : s$ on a predicate p into an aggregate selection on a rule that uses the predicate in the body, one can compute an mgu θ of $p(\overline{t_1})$ with a literal in which the predicate is used. The selection $s[\theta]$ is a sound aggregate selection on the body of the rule, since every fact that is used for the literal must unify with the literal, and must satisfy the aggregate constraint (see Theorem 6.4.1 for a formal proof). In fact, Technique BS1 uses a renaming σ on variables in the aggregate selection/constraint, rather than an arbitrary mgu σ. The use of renamings is not needed for correctness, but is done in order to restrict the set of aggregate selections that can be generated by our rewriting technique, thereby helping us ensure that our rewriting algorithm terminates.

Given an aggregate selection $p(\overline{t_1}) : s$ on a predicate p, we can compute an mgu θ of $p(\overline{t_1})$ with the head of a rule that defines p. Any fact generated by the rule must unify with the head, and if it does not satisfy the aggregate selection, it is irrelevant. Hence $s[\theta]$ is a sound aggregate selection on the body of the rule (see Theorem 6.4.1 for a formal proof). Technique BS3 generates aggregate selections as above, but restricts the mgu to be a renaming, for the same reasons as those described for Technique BS1.

We have the following theorem showing soundness of the above deduction techniques. A formal proof of the theorem may be found in Appendix D.

Theorem 6.4.1 *The aggregate selections generated by Techniques C1, BS1, BS2, BS3, and LS1 are sound aggregate selections.* $\square$

Example 6.4.1 Consider Program Simple_ShortPath (Figure 9). Using Technique C1 and rule $R2$ we get the aggregate constraint

$$s\_p\_length(X,Y,C) : groupby(path(X,Y,P,C),[X,Y],min(C))$$

on the predicate *s_p_length*. Using this aggregate constraint with rule R1, Technique BS1 deduces the following sound aggregate selection on the body of rule $R1$:

$$groupby(path(X,Y,P,C),[X,Y],min(C))$$

Using Technique BS2 we get the following sound aggregate selection on the body of rule $R2$:

$$groupby(path(X,Y,P,C),[X,Y],min(C))$$

If we had a sound aggregate selection

$$path(X,Y,P,C) : groupby(path(X,Y,P,C),[X,Y],min(C))$$

on the head predicate of rule $R3$, Technique BS3 would derive the following sound aggregate selection on the body of rule $R3$:

$$groupby(path(X,Y,P,C1),[X,Y],min(C1))$$

From these sound aggregate selections on the bodies of $R1$ and $R2$, using LS1, we deduce the sound aggregate selection

$$path(X,Y,P,C) : groupby(path(X,Y,P,C),[X,Y],min(C))$$

on the literal $path(X,Y,P,C)$ in the body of the rule $R1$, and the sound aggregate selection

$$path(X,Y,P,C) : groupby(path(X,Y,P,C),[X,Y],min(C))$$

on the literal $path(X,Y,P,C)$ in the body of the rule $R2$. $\square$

6.4.1 Pushing Aggregate Selections

We now look at another way of generating aggregate selections on rule body literals. But first we present some definitions. Aggregate functions such as *min* and functions as $+$ or $*$ interact in a particular fashion, and we use this interaction to generate sound aggregate selections on literals in the bodies of rules.

Definition 6.4.1 (Distribute Over) Let fn be a total function $fn : D \times D \times \ldots \times D \rightarrow D$ that maps n-tuples of values from D to a value in D. Define $s\_fn(U) = \bigcup\{fn(\overline{t}) \mid \overline{t} \in U\}$. Let agg_f be an aggregate function $agg\_f : M(D) \rightarrow D1$. Let $S_1, S_2, \ldots S_n$ be elements of $M(D)$, and let $S = S_1 \times S_2 \times \ldots \times S_n$.

$unnecessary_{agg\_f}$ is said to *distribute* over fn iff for every $(w1, w2, \ldots, wn) \in S$, and for every i, $1 \le i \le n$,

$$w_i \in unnecessary_{agg\_f}(S_i) \Rightarrow fn(w1, w2, \ldots, wn) \in unnecessary_{agg\_f}(s\_fn(S))$$

□

Example 6.4.2 For example $unnecessary_{min}$ distributes over "+" for reals and integers, and over $*$ for positive reals and positive integers, but does not distribute over $*$ for arbitrary reals.

$unnecessary_{min}$ also distributes over the function $min(a_1, a_2, \ldots, a_n)$, and further, surprisingly, also over $max(a_1, a_2, \ldots, a_n)$. In fact, *min* distributes over any function that is monotone non-decreasing on its arguments. $unnecessary_{max}$ behaves exactly like min on the above functions. Let *sum_highest_k* denote the aggregate function that sums the highest k values (for some fixed k). Then $unnecessary_{sum\_highest\_k}$ distributes over "+" on reals. □

We assume that the system implementor provides a set of pairs (agg_f, fn) for common aggregate functions agg_f and arithmetic functions fn such that $unnecessary_{agg\_f}$ distributes over fn. We discuss briefly in Section 6.8 how to extend the idea of distributes over to allow different aggregate functions for each S_i in the above definition.

Technique PS1 shows a way of deriving aggregate selections on literals in rule bodies by making use of distribution of aggregate functions over ordinary functions.

Technique PS1: (Generating Aggregate Selections on Literals)

Let R be a rule of the form
$R:\ p_h(\overline{t_h}) :- \ldots, p_i(\overline{t_i}, Wi), \ldots, Y = fn(W1, \ldots, Wn)$
such that there is no aggregate operation in the head of R. Suppose

1. There is a sound atomic aggregate selection on the head of R, of the form $groupby(p_h(\overline{t_h}), [\overline{X}], agg\_f(Y))$
2. $unnecessary_{agg\_f}$ distributes over fn,
3. Each of $W1, \ldots, Wn, Y$ are distinct variables,
4. Each Wi appears exactly once in the literal $p_i(\overline{t_i}, Wi)$, and appears in no literal other than $Y = fn(W1, \ldots, Wn)$.
5. Y does not appear in any other literal in the body of the rule, and does not appear in $\overline{X}$.

Then for each literal $p_i(\overline{t_i}, Wi)$ in the body of the rule, the following is a sound atomic aggregate selection on the literal:

$$p_i(\overline{Z}, W_i) : groupby(p_i(\overline{Z}, Wi), [\overline{Z}], agg\_f(Wi))$$

where Z is a tuple of distinct variables of the same arity as $\overline{t_i}$.

Theorem 6.4.2 *Technique PS1 is sound.* □

The proof of this theorem may be found in Appendix D.

Example 6.4.3 Suppose we have a sound atomic aggregate selection

$$groupby(path(X, Y, C), [X, Y], min(C))$$

on the head of the following rule:

$$path(X, Y, C) \colon -path(X, Z, C1), edge(Z, Y, C2), C = C1 + C2.$$

Technique PS1 derives a sound aggregate selection of the form

$$groupby(path(X, Z, C1), [X, Z], min(C1))$$

on the body literal *path*.

Now suppose we have a sound atomic aggregate selection

$$groupby(path(X, Y, P, C), [X, Y], min(C))$$

on the head of rule $R3$ of Program Simple_ShortPath. Technique PS1 derives a sound aggregate selection of the form $groupby(path(X, Z, P, C), [X, Z, P], min(C))$ on the body literal $path(X, Z, P, C)$ in rule $R3$. However, this literal has a "stronger" sound aggregate selection $groupby(path(X, Z, P, C), [X, Z], min(C))$. In Section 6.4.2 we see how the stronger selection can be derived. □

6.4.2 Extended Techniques for Pushing Selections

The selections generated by Technique PS1 are too weak in the following sense. Often there are arguments of literals that need not be introduced in the group-by variables of the aggregate selection generated, as is illustrated in Example 6.4.3. The deduction technique can be extended using the following idea. In the proof of Technique PS1, we partitioned the multiset S_Y based on the values of variables other than $\{W1, \ldots, Wn, Y\}$, and we showed that within each partition we have a cross product of the S_{Wi} values. This cross product is important for distributing *agg_f* over fn. We can make the partitions of coarser granularity by not including some variables in the partition, and yet have a cross product as above. We can then generate stronger sound aggregate selections on body literals. We first present some definitions that help in the generalization.

Definition 6.4.2 (Cross-Partitioning Variables) Consider a rule

$$R : p_h(\overline{t_h}) :- \ldots, p_i(\overline{t_i}, Wi), \ldots, Y = fn(W1, \ldots, Wn).$$

with an aggregate selection

$$groupby(p_h(\overline{t_h}), [\overline{X}], agg\_f(Y))$$

that satisfy the conditions of PS1. A set of variables $\mathcal{V}$ is said to be *cross-partitioning* if

1. $\{Y, W1, W2, \ldots, Wn\} \cap \mathcal{V} = \emptyset$.
2. Given any instantiation of the variables in $\mathcal{V} \cup \overline{X}$:
   (a) Let S denote the set of instantiations of $(W1, W2, \ldots, Wn)$ generated by successful instantiations of the rule with the given binding for $\mathcal{V} \cup \overline{X}$.
   (b) Let S_{Wi} denote the set of instantiations of Wi generated by successful instantiations of $p_i(\overline{t_i}, Wi)$ with the given binding for $\mathcal{V} \cup \overline{X}$.

   Then either S is empty, or $S = S_{W1} \times S_{W2} \times \ldots \times S_{Wn}$.

□

The set of all variables in the rule other than $\{Y, W1, W2, \ldots, Wn\}$ is a cross-partitioning set, as is shown by the proof of soundness of Technique PS1. However, there may be smaller sets of cross-partitioning variables.

Example 6.4.4 We continue with Example 6.4.1. Suppose we have a sound atomic aggregate selection $groupby(path(X, Y, P, C1), [X, Y], min(C1))$ on the head of rule $R3$:

$$R3 : path(X, Y, [edge(Z, Y)|P], C1) :- path(X, Z, P, C), edge(Z, Y, EC), C1 = C + EC.$$

Then the set of variables $\{X, Y, Z\}$ forms a cross-partitioning set. The reason is that with a given value for X, Y, Z, whatever values one finds for C using *path* can be used with whatever values that one gets for EC using *edge*. The value of P does not affect the cross-product. □

We discuss the issue of automatically determining sets of variables that are cross-partitioning, in Section 6.4.2.

Technique PS2: (Extended Technique For Generating Aggregate Selections on Literals)

Consider a rule R that with an aggregate selection satisfies the conditions of Technique PS1. Suppose some set $\mathcal{V}$ of variables is a cross-partitioning set. Define an argument of $p_i(\overline{t_i}, Wi)$ to be a *partitioning argument* if

1. it is not a variable, or
2. it is a variable that appears elsewhere in the same literal, or
3. it is a variable that is in $\overline{X} \cup \mathcal{V}$.

Let $\overline{Z}$ be a tuple of distinct variables of the same arity as $\overline{t_i}$. Let $\overline{Z'}$ be the set of variables in $\overline{Z}$ that correspond to partitioning arguments. Then the following is a sound atomic aggregate selection on literal $p_i(\overline{t_i}, Wi)$:

$$p_i(\overline{Z}, W_i) : groupby(p_i(\overline{Z}, Wi), [\overline{Z'}], agg\_f(Wi))$$

Theorem 6.4.3 *Technique PS2 is sound.* □

The proof of this theorem is presented in Appendix D.

Example 6.4.5 We continue with Example 6.4.4. Using Technique PS2, we deduce the following sound aggregate selection on the literal *path*:

$$path(X, Y, P, C) : groupby(path(X, Z, P, C), [X, Z], min(C))$$

This selection is "stronger" than the selection generated by Technique PS1, since it selects the minimum cost for each X, Z pair, rather than for each X, Z, P triple. □

Detecting Sets of Cross-Partitioning Variables

We now see how to determine a set of cross-partitioning variables for a rule. We first present some definitions. The idea behind the following definitions is that if a variable in a literal does not appear elsewhere in the rule or in the aggregate selection, we get a cross-product of S_{Wi} sets as in Definition 6.4.2, even if we do not include the variable in the cross-partitioning set.

Definition 6.4.3 (Strongly Non-Constrained Arguments) Suppose we are given a rule and an aggregate selection on the rule. Consider any literal in the rule. The *strongly non-constrained* arguments of the literal as those arguments that are distinct variables that (1) occur nowhere else in the body of the rule, and (2) do not appear as a free variable in the aggregate selection. □

Definition 6.4.4 (Strongly Non-Constrained Variables) Consider a rule R and an aggregate selection s as in Technique PS1. A variable in R is *strongly non-constrained* if it occurs in a non-constrained argument of some literal $p_i(\overline{t_i}, Wi)$. □

Proposition 6.4.4 *Consider a rule R and an aggregate selection s as in Technique PS1. Let $\mathcal{V}$ denote the set of all variables in the rule. Let $\mathcal{N}$ denote the set of non-constrained variables in the rule. Then $\mathcal{C} = \mathcal{V} - \mathcal{N} - \{W1, W2, \ldots, Wn, Y\}$ is a cross-partitioning set for rule R.* □

The proof of this proposition may be found in Appendix D.

Example 6.4.6 We revisit Example 6.4.4. Suppose we have a sound atomic aggregate selection

$$groupby(path(X, Y, P, C1), [X, Y], min(C1))$$

on the head of rule $R3$:

$$R3 : path(X, Y, [edge(Z, Y)|P], C1) : -\ path(X, Z, P, C), edge(Z, Y, EC), C1 = C + EC.$$

Then the third argument of $path(X, Z, P, C)$ is a non-constrained argument since it is a variable that does not appear elsewhere in the rule body. Hence P is a non-constrained variable. The set of variables $\{X, Y, Z\}$ forms a cross-partitioning set, as required by Proposition 6.4.4. □

6.4.3 An Undecidability Result

We note that the set of deduction rules we presented for sound aggregate selections is not complete. Depending on the actual facts for a predicate, it is possible that a literal has a sound aggregate selection on it, but the sound aggregate selection cannot be deduced syntactically. The following theorem shows that no set of deduction rules can be complete.

Theorem 6.4.5 *It is undecidable whether an aggregate selection is sound.*

Proof: Consider a rule

$$p(X, C) \colon -q(X, C), test(C).$$

and suppose we have a sound aggregate selection

$$p(X, C) : groupby(p(X, C), [X], min(C))$$

Then

$$q(X, C) : groupby(q(X, C), [X], min(C))$$

is a sound aggregate selection on the literal q iff for every X, the minimum value of C in $q(X, C)$ satisfies $test(C)$. However, with arbitrary logic programs, satisfiability is undecidable [SS82], and hence it is undecidable if the aggregate selection on the literal is sound. The theorem can be extended to aggregate selections on predicates, by letting the given rule be the only one that uses of q. □

It is conceivable that we can derive a set of rules that are complete for the class of deductions that use only (local) syntactic criteria. However, such a set of rules would be too weak in practise, as is illustrated by the program in Example 6.4.3. Here, distribution of min over $+$ depends critically on the semantics for $+$. Hence no deduction rule that used purely syntactic criteria would deduce the required selection.

6.4.4 Strength of Aggregate Selections

An aggregate selection s is *stronger than* an aggregate selection t (denoted as $s \geq t$), if whenever t classifies an instantiation as irrelevant, then so does s. Selections s and t are *equivalent* (in symbols, $s \equiv t$) if $s \geq t$ and $s \geq t$. Note that the ordering $>$ (i.e., the strict version of $\geq$) is an irreflexive partial ordering. It is not a total ordering since aggregate selections may be incomparable.

The following are sufficient conditions for an aggregate selection s to be stronger than t.

Compare_Aggregate_Selections(s, t):

1. Suppose s and t are atomic aggregate selections of the following form:

$$s = c1(\ldots) : groupby(p(\bar{t}), [\overline{X1}], agg\_f(Y))$$

$$t = c2(\ldots) : groupby(p(\bar{t}), [\overline{X2}], agg\_f(Y))$$

   (a) If $c1(\ldots) = c2(\ldots)$, and $Vars(\overline{X1}) \subseteq Vars(\overline{X2})$ then $s \geq t$.
   (Note that the first and third arguments of the above groupby's must be the same.)

   (b) If there is some substitution σ on the variables of $c1(\ldots)$ such that $c1(\ldots)[\sigma] = c2(\ldots)$, and $s[\sigma]$ is stronger than t, then s is stronger than t.

2. Suppose $s = c1(\ldots) : as_1 \wedge as_2 \wedge \ldots \wedge as_m$ and $t = c2(\ldots) : at_1 \wedge at_2 \wedge \ldots \wedge at_n$ where each as_i and at_j is atomic. Then $s \geq t$ if for each at_j there exists an as_i such that $c1(\ldots) : as_i \geq c2(\ldots) : at_j$.

Proposition 6.4.6 *The conditions in Compare_Aggregate_Selections(s,t) are sufficient conditions for s to be stronger than t.* □

The formal proof of the above proposition may be found in Appendix D.

6.5 The Aggregate Rewriting Algorithm

In this section we present a rewriting of the program based on the propagation of sound aggregate selections. The rewriting algorithm is somewhat similar to the adornment algorithm used in Magic sets rewriting (see [Ull89b]). When it detects that an occurrence of a predicate p in the body of a particular rule has a sound aggregate selection s on it, it creates a new labeled version p_s of p and notes that predicate p_s has aggregate selection s on it. That occurrence of predicate p is replaced by p_s, and by using aggregate selection s, (copies of) rules defining p are specialized to define p_s.

The rewriting algorithm is shown below. In Step 7 of the algorithm, s is a sound aggregate selection on the head of R', and this, along with any aggregate constraints on body predicates, may be used with techniques from Section 6.4 to generate new aggregate selections.

```
Algorithm Push_Selections($P, P^{as}$ )
Input:      Program $P$, and query predicate $query\_pred$.
Output:     Rewritten program $P^{as}$.
1)  Derive sound aggregate constraints on the predicates of the program
            using the deduction rules.
2)  Push $query\_pred\_nil$ onto $stack$.
3)  While $stack$ not empty do
4)      Pop $p\_s$ from the stack and mark $p\_s$ as seen.
5)      For each rule $R$ defining $p$ do
6)          Set $R'$ = a copy of $R$ with head predicate replaced by $p\_s$.
```

```
7)          Derive sound aggregate selections for each body literal $p_i$ of $R'$
                using the deduction rules.
8)          For each $p_i$ in the body of $R'$ do
9)              Let $si$ denote the conjunction of sound aggregate selections
                    derived for $p_i$; drop from $si$ any atomic aggregate selections
                    that are weaker than other atomic aggregate selections in $si$.
10)             If a version $p_i\_t$ of $p_i$ such that $t \leq si$ has been seen,
11)             Then choose one such, and set $si = t$ ;
12)             Else push $p_i\_si$ onto $stack$, and output selection $s_i$ on $p_i\_s_i$.
13)         Output a copy of $R'$, with each $p_i$ replaced by $p_i\_si$.
    End while.
End Algorithm.
```

Postprocessing 1: For each predicate p, for each version p_s of p, choose the weakest version p_t of p in the rewritten program such that $s \geq t$. Replace all occurrences of p_s in bodies of rules in the rewritten program by p_t. Finally, remove all rules that are not reachable from the query.

Postprocessing 2: Suppose we have a predicate q in the rewritten program, with an atomic aggregate selection $s = groupby(p(\overline{t}), [\overline{X}], agg\_f(Y))$ on it. If q is a version of p with aggregate selection s on it, rename p in the above selection to q. Otherwise, if p is absent from the rewritten program rename p in the selection with a predicate chosen as below: if a version p_s of p with aggregate selection s, exists, choose it. If not, select a version[6] p_s1 of p if any such version exists. If no p_s1 was found, p is not connected to the query predicate—drop the selection s from predicate q.

If in the rewritten program there are two versions of p, p_s and p_t such that $s > t$, there is no point in using the stronger version p_s — all the facts computed for p_s will be computed for p_t. Postprocessing 1 describes how to replace the stronger version of p by the weaker version.

As a result of the renaming of predicates followed by reachability analysis in Postprocessing 1, predicates used in aggregate selections may not be present in the rewritten program. Postprocessing 2 describes how to fix this problem.

Example 6.5.1 Applying this algorithm to Program Simple_ShortPath, we get the optimized program, Program Smart shown in Figure 10. The algorithm starts with the query predicate *shortest_path*. Creation of aggregate constraints, and pushing them into rules is done as discussed in earlier examples, and the operation of Algorithm Push_Selections is fairly straightforward. As a result of the rewriting we get the rules of Program Smart, but with *path_s*1 having the following sound aggregate selection on it:

$$path\_s1(X, Y, P, C) : groupby(path(X, Y, P, C), [X, Y], min(C)).$$

On postprocessing, we rename predicate *path* in the above selection to *path_s*1, to get Program Smart. To

[6]We omit details on how to make this choice.

get the benefits of the rewriting, the evaluation must make use of the aggregate selections present in Program Smart. We describe how to do this in the next section. $\Box$

Theorem 6.5.1 (Correctness of Rewriting) Let P be any program, and P^{as} the aggregate rewritten version of the program.

1. P^{as} and P are equivalent in the set of answers they generate for the query predicate.
2. The aggregate selection on each predicate in P^{as} is a sound aggregate selection on the predicate.

$\Box$

The proof of this theorem is presented in Appendix D The basic idea is that the deduction rules generate sound aggregate selections on body literals. The rewriting technique creates copies of the predicates of the body literals, such that all uses of the predicate have the aggregate selection on them, and hence the aggregate selection on the literal becomes an aggregate selection on new predicate.

Theorem 6.5.2 (Termination) Algorithm Push_Selections terminates on all finite input programs, producing a finite rewritten program. $\Box$

The above theorem shows that the generated program is finite. This is assured essentially because our deduction techniques bound the number of different aggregate selections that can be generated. The formal proof is presented in Appendix D.

The rewritten program could potentially be large, but, as is the case with the adornment algorithm for Magic sets rewriting, this is very unlikely to happen in practice—the rewritten program is likely to be not much larger than the original program. To ensure that the rewritten program is small we could adopt heuristics such as bounding the number of atomic aggregate selections in an aggregate selection to some fixed small value, or bounding the number of different aggregate selections on each predicate. We omit details here; these restrictions may increase the number of facts computed, but will not affect correctness.

Proposition 6.5.3 (Stratification) If the initial program is stratified w.r.t. aggregation, then the aggregate rewritten program is also stratified w.r.t. aggregation. $\Box$

Proof: We simply assign each predicate *p_s* to the same stratum as *p*. It can then be seen that every aggregation operation in the rewritten program respects this stratification. $\Box$

6.6 Aggregate Retaining Evaluation

In this section we see how to evaluate a rewritten program making use of aggregate selections on predicates. Essentially, once we know that a fact does not satisfy a sound aggregate selection on it we know that the fact is irrelevant, and any use if the fact will only generate irrelevant facts.

We define *Aggregate Retaining Evaluation (Agg-retaining Evaluation)* as a modification to Semi-Naive evaluation (Section 2.2.3) : At the end of each iteration of Semi-Naive evaluation, (in Step 2.2 of Algorithm SN_Evaluate) the following extra actions are performed:

1. Any fact that does not satisfy an aggregate selection is marked as deleted. Any fact marked deleted is not used in further derivations.
2. For each fact marked deleted, if

   (a) there is an aggregate selection with a groupby that uses the predicate of the fact, and
   (b) the fact affects the unnecessary set for the groupby,

   then the fact is retained for use in that groupby. Otherwise, the fact is discarded.

Part 2 of the above may seem hard to test. In fact, it is not critical that it be tested. Retaining a deleted fact that satisfies Part 2 of the condition above does not affect the derivations made later on. Moreover, there are straightforward sufficient conditions for it, such as the following.

If a fact for a predicate p fails a sound atomic aggregate selection

$$p(\bar{t}) : groupby(p(\ldots), [\ldots], \ldots)$$

(i.e., the groupby uses the same predicate p) discarding the fact will not affect the unnecessary set for this groupby. This is because Part 3 of condition IncSel ensures that if a value in a set is unnecessary for a set, discarding it will not affect the unnecessary value for the set. If all uses of p in atomic aggregate selections are of the above form, and a fact for p fails all the atomic aggregate selections, then discarding the fact will not affect the unnecessary set for the groupby's in any selection.

Example 6.6.1 Predicate $path\_s1$ in Program Smart has a sound aggregate selection

$$path\_s1(X,Y,P,C) : groupby(path\_s1(X,Y,P,C), [X,Y], min(C)).$$

If a fact is generated with any value for X and Y and another fact with the same value for X and Y already exists, we know that the one with the greater C value does not satisfy the aggregate selection. Agg-retaining evaluation of Program Smart discards facts with higher cost. If there is more than one stored fact with the same value for X, Y, C, the facts can differ only in their P value. If a fact fails the aggregate selection, it cannot affect the set of facts that are found irrelevant by the aggregate selection, and the fact can be discarded. □

The soundness and partial completeness of Agg-retaining evaluation are fairly straightforward to show. The main concern is termination. One might worry that Agg-retaining evaluation could discard a fact, then recompute it, and reuse it to make derivations since it does not recognize that it was used earlier. In the worst case, an infinite loop could result if this happens. The following theorem shows that this cannot happen. The essential idea is to show that once a fact is found irrelevant, it continues to be found irrelevant later in the computation. The proof of the theorem is presented in Appendix D.

Theorem 6.6.1 (Soundness, Completeness, Non-Repetition) Aggregate Retaining evaluation of P^{as} gives the same set of answers for *query_pred* as Semi-Naive evaluation of P, and does not repeat any inferences. Further, the Aggregate Retaining evaluation of P^{as} terminates whenever the Semi-Naive evaluation of P terminates. □

$R1 : s\_p\_length(X, Y, min\langle C\rangle)) : -\ path\_s1(X, Y, C).$
$R2 : path\_s1(X, Y, C1) \quad : -\ path\_s1(X, Z, C), edge(Z, Y, EC), C1 = C + EC.$
$R3 : path\_s1(X, Y, C) \quad : -\ edge(X, Y, C).$
Selections::
$s1 = path\_s1(X, Y, C) : groupby(path\_s1(X, Y, C), [X, Y], min(C)).$

Figure 11: Program Smart_ShortCost

6.6.1 Pragmatic Issues Of Testing Aggregate Selections

For concreteness, we let the set of aggregate functions that we consider in this section be the following: *min*, *max*, and for small integers k (up to some arbitrary number) the functions *least_k*, *highest_k*, *sum_of_least_k* and *sum_of_highest_k*.

Our selection propagating techniques ensure that all free variables in a groupby of an atomic aggregate selection also appear in the corresponding literal on which the selection is applied. When testing an atomic aggregate selection on a fact f, we have a unique ground instantiation of the group-by and grouping variables of the selection; the test of the aggregate selection can be performed efficiently for all the aggregate functions that we consider in this section.

If the test determines that fact f is irrelevant, f is discarded, else it is retained — for the aggregate functions we consider, discarding f does not affect the set of values that are classified as irrelevant. As the computation proceeds, the set of unnecessary values for the "group" to which f belongs (i.e., the set of facts with the same values in the grouped arguments) could grow larger, and this might enable us to determine that f is irrelevant, although this could not be detected earlier. By sorting the set of facts on the grouped arguments, this "re-testing" can be done efficiently. The cost of sorting is small for the aggregate operations we consider in this section; in the case of *max* or *min* aggregate operations there is at most one value stored for each group (however, there can be more than one fact with the same value).

Proposition 6.6.2 (Bounds on Performance) Given a program that uses only the aggregate operations considered in this section, and a database, let the time for Agg-retaining Evaluation of the program on the database be t_R, and let t_O be the time taken to evaluate the original program on the database. There is a constant k (independent of the database) such that $t_R \leq k * t_O$. $\square$

This means that Agg-retaining evaluation of the rewritten program can do at most a constant factor worse than Semi-Naive evaluation of the original program — the converse is not true.

Example 6.6.2 Given a graph with n nodes, the number of shortest paths between each pair of points may be exponential in n. Hence we cannot get a worst case time bound better than exponential in n for the shortest-path problem if we maintain all shortest paths as in Program Smart (Example 6.5.1). We instead consider two variants of this program below.

Consider Program Simple_ShortCost, from Section 6.1. This program does not maintain path information. The rewriting for this program is similar to the rewriting for Program Simple_ShortPath (Figure 9), and the rewritten program, Program Smart_ShortCost, is shown in Figure 11.

$R1 : s\_p\_length(s, Y, min\langle C\rangle)) : -\ path\_s1(s, Y, C).$
$R2 : path\_s1(s, Y, C1) \quad : -\ path\_s1(s, Z, C), edge(Z, Y, EC), C1 = C + EC.$
$R3 : path\_s1(s, Y, C) \quad : -\ edge(s, Y, C).$
Selections::
$\quad s1 = path\_s1(X, Y, C) : groupby(path\_s1(X, Y, C), [X, Y], min(C)).$

Figure 12: Program Smart_SingleSourceCost

Due to the aggregate selection, there can be at most $O(V^2)$ *path_s1* facts at any point in the evaluation. These facts can be used with the E edge facts. Rule $R2$ can be thought of as extending each edge backward, and each edge can be extended back to at most V nodes. Rule $R1$ can generate at most E *path* facts. This shows that $E \cdot V$ inferences are made per iteration, and hence there are at most $E \cdot V$ *path* facts used in each iteration. Each iteration then takes time $E \cdot V$, assuming that hash-based indices are used for *path* and *edge* facts. There are at most V iterations, since iteration i computes all shortest paths of length i. Thus, Agg-retaining evaluation of Program Smart_ShortCost takes time $O(E \cdot V^2)$.

To compute shortest paths from a single source, we can use a version of Simple_ShortCost where the variable in the first argument of each *path* literal in the program is bound to the source node s.[7] We do not show this program, but instead directly show its aggregate rewritten version, Program Smart_SingleSourceCost, in Figure 12. An analysis similar to that for Program Smart_ShortCost shows that Program Smart_SingleSourceCost runs in time $O(E \cdot V)$.

Note that the above bounds hold even if there are negative length edges, so long as there are no negative cycles in the *edge* graph.

In Sudarshan and Ramakrishnan [SR92a], we discuss extensions to aggregate functions to allow the program to specify that only one shortest path (chosen arbitrarily) is required. The rewriting algorithms are also extended to handle these extensions to aggregate functions. It is easy to modify Program Smart to get a program that computes shortest paths from a given source node. We can use these extensions to create a version of this program that selects a single shortest path. The extended Aggregate rewriting of this program generates a rewritten program that maintains at most one shortest path between the source node and each other node. We show in [SR92a] that precisely the same time bounds as for Program Smart_SingleSourceCost are applicable to the Agg-retaining evaluation of the rewritten program. □

6.6.2 Ordered Aggregate Retaining Evaluation

Consider the shortest path problem with a given starting point. Dijkstra's algorithm takes $O(E * log(V))$ time if we use a heap data structure to find the minimum cost path at each stage. However, Agg-retaining evaluation of Program Smart_SingleSourceCost (Example 6.6.2) takes $O(E \cdot V)$ time. We can get the effect of Dijkstra's algorithm by extending at each stage only the shortest path that hasn't been extended yet. In other words, we use only the *path* facts that are of minimal cost among those that haven't yet been used. This important observation is made in [GGZ91] and is used in their evaluation algorithm (see Section 6.8.1

[7]The Factoring transformation [NRSU89] and the Magic Sets rewriting on Program Simple_ShortCost, with a query having the first argument bound results is a similar program being generated. Aggregate Rewriting optimizes the resultant program successfully.

for a brief description). Their evaluation technique works for the class of "monotonic min programs" — see [GGZ91] for a precise definition. The basic idea behind their technique can be applied to a class of programs that we call *cost-inflationary* programs. These are defined below.

A cost domain is a domain with a partial ordering on it. A *cost predicate* is one with a distinguished argument called the cost argument, that takes values from the cost domain.

Definition 6.6.1 (Cost-Inflationary Programs) A strongly connected component (SCC) of a program is said to be *cost-inflationary-min* if either it has no aggregate selections, or all the following conditions hold:

1. All aggregate selections on predicates in the SCC use only the *min* aggregate operation.
2. Every predicate in the SCC has a cost argument.
3. For each predicate defined in the SCC, the *min* operation in each aggregate selection (if any) on it is on the cost argument of the predicate.
4. Every rule in the SCC is inflationary on the cost argument, i.e., for every successfully instantiated rule, the values in cost arguments of body predicates are less than the value in the cost argument of the head of the instantiated rule.

An SCC is said to be *cost-inflationary-max* if the conditions above hold, with min replaced with max, and less replaced with greater.

A program is said to be *cost-inflationary* if each of its SCCs is either cost-inflationary-min or cost-inflationary-max. □

We make use of an idea in [GGZ91] to derive an improved evaluation technique, *Ordered Aggregate Retaining Evaluation (Ordered-Agg Evaluation)* , for SCCs that have aggregate selections, and are cost-inflationary-min. The basic idea is to use lower cost facts before higher cost facts are used. We make use of a mechanism called sloppy-delta iteration for adapting Semi-Naive evaluation for fact orderings, described in [SKGB87]. All derived relations are split into a *visible* part and a *hidden* part containing facts that are not used to make derivations until they are moved into the visible part.

Ordered Aggregate Retaining Evaluation:

To evaluate cost-inflationary-min SCCs, we adapt Semi-Naive evaluation as follows.

1. Newly derived facts are put into the hidden parts of the respective relations. The facts in the hidden parts of relations are ordered based on the cost argument of the fact.
2. Whenever a fixpoint is reached with the visible parts of relations, we find the fact with the least cost from among all the facts in the hidden parts of relations and move it into the visible part.
3. Facts from the hidden as well as the visible relations are marked deleted (and possibly discarded) when an aggregate selection finds them to be irrelevant, as is done in Agg-retaining evaluation.

The technique for cost-inflationary-max SCCs is very similar to the above, with the ordering of elements reversed; we omit details. A cost-inflationary program is evaluated by using the appropriate version of

Ordered-Agg evaluation technique for each SCC, and evaluating the SCCs in a total order consistent with the partial ordering of the SCCs.

The effect of the above evaluation is exactly the same as if Ganguly et al.'s evaluation technique were used, for the case of cost-inflationary programs. The following example illustrates its benefits.

Example 6.6.3 The Aggregate rewritten single source shortest path cost program, Program Smart_Single-SourceCost, is shown in Figure 12.

All *path* facts generated by Program Smart_SingleSourceCost have the source node s as the first argument. Ordered-Agg evaluation of Program Smart_SingleSourceCost works as follows. First, all edges from s are used, and *path* facts created using rule $R3$. These path facts are hidden, and a local fixpoint is reached. Now a shortest path among the hidden *path* facts is selected and used. This generates new *path* facts, and all these are hidden. A local fixpoint is reached, and a shortest path among the hidden *path* facts is selected again. If there are two *path* facts to the same node, if one of them is of higher cost than the other, the aggregate selection using *min* deletes the fact of higher cost.

We assume that the edge weights are non-negative. The evaluation explores paths in order of increasing cost since edge weights are non-negative — any *path* fact generated must be of equal or higher cost than the *path* fact used to generate it. Thus when a hidden *path* fact is exposed, it is guaranteed to be a shortest path from s. Evaluation thus mimics Dijkstra's algorithm. The time complexity analysis is essentially the same as that used for Dijkstra's shortest path algorithm — the analysis is as below.

Suppose we use a heap data structure. The *min* aggregate selection ensures that for each node, only the minimum cost path from the source is retained. Thus only $O(V)$ path facts are present at any time.

Finding the overall shortest path at each step therefore takes $O(log(V))$ time, assuming a balanced heap is used. At each iteration a "minimal" node is chosen and the path to it is expanded. Thus some new facts are computed and added to the heap.

For the node that is chosen to be expanded in the next iteration, there can be no shorter path from the source, since every path that is computed hence will be longer (due to the assumption that edge weights are non-negative). Thus that node will never be chosen again to be expanded. Thus in V steps the algorithm terminates. At each step the edges from a node are examined, and the path to the node is expanded along each edge from the node. This can take a total of at most $O(E)$ time over all steps since each node is expanded exactly once, and at most $O(E)$ facts are added to the heap. Thus the heap operations take $O(E \cdot log(V))$ time. Thus the total time taken by Ordered-Agg evaluation of this program is $O(E \cdot log(V))$.

Note that even if edge weights are negative, the algorithm works correctly, and terminates provided there are no negative weight cycles. However, the time taken by the program may be exponential in the worst case.

Using the extensions described in Sudarshan and Ramakrishnan [SR92a], we can create a variant of Program Smart that computes only paths from a single source node, and maintains only one shortest path between each pair of nodes, and we can use the extended aggregate rewriting on this program. We show in [SR92a] that precisely the same time bounds as for Ordered-Agg evaluation of Program Smart_SingleSource-Cost are applicable to the Ordered-Agg evaluation of the rewritten program. □

Theorem 6.6.3 *Consider the Ordered Aggregate Retaining Evaluation of a cost-inflationary-min SCC. The* *evaluation is sound, and every fact that is used in the evaluation satisfies all aggregate selections on it. Further, the evaluation does not repeat derivations, and is complete and terminates if Agg-retaining evaluation terminates on the SCC.*

Proof: Soundness follows directly from the soundness of the aggregate selections.

At an intermediate fixpoint in an Agg-retaining evaluation, consider the fact with the least cost that has not been used yet, and satisfies any aggregate selections on it. Since the rules in the SCC are cost-inflationary-min, no fact with lesser cost can be derived hence. Therefore this fact definitely satisfies any *min* aggregate selection on the predicate (with respect to the complete set of facts).

Consider now an SCC for which Agg-retaining evaluation terminates. Since Agg-retaining evaluation terminates, there are only a finite number of facts that satisfy all aggregate selections present. At each intermediate fixpoint, a new such fact is chosen. Hence there are only a finite number of fixpoints. Now within an intermediate fixpoint, only derivations that use the selected fact can be made. Since this is a subset of the facts derived in Agg-retaining evaluation, each intermediate fixpoint terminates. Hence Ordered-Agg evaluation terminates on the SCC. Any deleted facts fails an aggregate selection and hence is irrelevant. Completeness then follows from the completeness of sloppy-delta iteration [SKGB87].

As in Agg-retaining evaluation, once a fact is found to fail an aggregate selection, it will continue to fail the aggregate selection. The non-repetition property follows from the non-repetition property of sloppy-delta iteration. □

The above theorem also shows that Ordered-Agg evaluation never makes more derivations than Agg-retaining evaluation for cost-inflationary programs. In turn, Agg-retaining evaluation makes no more inferences than Semi-Naive evaluation.

Ordered-Agg evaluation also works on programs that are not cost-inflationary. For instance, the shortest path program is not inflationary if there are negative cost edges. But even in this case, Ordered-Agg evaluation of Program Smart functions correctly, and terminates if there are no negative cost cycles, although it may not be very efficient if negative edges are present.

6.7 Examples

We now see some more examples of programs to which our techniques are applicable.

Example 6.7.1 The following program defines the earliest finish time of a task, given the finish times of preceding tasks.

$$\begin{array}{ll} R1 : e\_fin(X, max\langle T\rangle) & :-\ fin(X,T). \\ R2 : fin(X,T) & :-\ precedes(X,Y), fin(Y,T1), delay(X,D), T = T1 + D. \\ R3 : fin(X,T) & :-\ first(X), delay(X,T). \end{array}$$

This program can be optimized using our techniques, and in the resultant program fin is replaced by fin_s, where s is the aggregate selection

$$fin\_s(X,T) : groupby(fin\_s(X,T), [X], max(T))$$

The rules and other predicates are the same, but $finish$ facts that don't have maximal times are deduced to be irrelevant. We can extend this program to compute the critical path, and still apply our optimizations.

The aggregate rewritten program can be evaluated using Agg-retaining evaluation. We cannot use Ordered-Agg evaluation since the program is not cost-inflationary — it uses max, but the cost value of the head of a rule is greater than that of the body, whereas it should be less for cost-inflationary-max SCCs. Note that the evaluation of the program would take time $O(E \cdot V)$. If we ordered the use of facts such that a vertex is expanded only after all its predecessors have been expanded, we can do better. This can in fact be achieved by using the Ordered Search evaluation feature [RSS92a] provided in the CORAL deductive database system [RSS92b]. The time complexity of evaluation is then $O(E)$. □

Example 6.7.2 Consider the following program. Predicate $path2(X,Y,H,C)$ denotes a path where X and Y are source and destination, H denotes hops, and C denotes cost.

$R1 : p\_best(X,Y,H,C) \quad :- \; p\_few(X,Y,H), p\_short(X,Y,H,C).$
$R2 : p\_few(X,Y,min\langle H\rangle) \quad :- \; p\_short(X,Y,H,C).$
$R3 : p\_short(X,Y,H,min\langle C\rangle) :- \; path2(X,Y,H,C).$
/* ... Rules for $path2$ ... */
Query: ?-$p\_best(X,Y,H,C)$.

The program finds flights with the minimum number of hops, and within such flights, finds those with minimum cost. Our technique generates the sound aggregate selection on $path2$:

$$path2(X,Y,H,C) : groupby(path2(X,Y,H,C),[X,Y,H],min(C))$$

$$\wedge \quad groupby(path2(X,Y,H,C),[X,Y],min(H))$$

The rewritten program is the same as the original program (modulo renaming of predicates other than p_best), except for having the above sound aggregate selection on $path2$, as well as aggregate selections on p_few and p_best. In the evaluation of the rewritten program all paths that have more hops than the minimum for a given start and end point, as well as all paths that are not of minimum cost for a given start and end points and a given number of hops are discarded. □

Example 6.7.3 The following program can be used to find the cost of the cheapest three paths, and illustrates the ability of our techniques to handle aggregate operations other than min and max. We use the aggregate operation $least3$ that given a multiset, returns a multiset containing the three least values in the given multiset.[8]

$R1 : shortest3(X,Y,least3\langle C\rangle) :- \; path(X,Y,C).$
/* ... Rules for $path$ as in Figure 8 ... */
Query: ?-$shortest3(X,Y,C)$.

Aggregate operation $least3$ is an IncSel function, with $unnecessary_{least3}(S)$ defined as all values greater than the third lowest value in multiset S. Also, the function $unnecessary_{least3}$ distributes over "+". Hence our rewriting technique proceeds on the rules for $path$ in this program in a manner very similar to the shortest

[8]Since the Herbrand universe does not include multisets, we need to use an extended Herbrand universe when assigning a semantics to this program [BNR+87].

$R1 \quad nearest\_sg^{bff}(X, Y, min\langle D\rangle) : -query(nearest\_sg^{bff}(X)), sg^{bff}(X, Y, D).$
$R2 : sg^{bff}(X, Y, D) \quad : - \; query(sg^{bff}(X)), up(X, Z1), sg^{bff}(Z1, Z2, D1),$
$\qquad\qquad down(Z2, Y), D = D1 + 1.$
$R3 : sg^{bff}(X, Y, 1) \quad : - \; query(sg^{bff}(X)), flat(X, Y).$
$R4 : query(sg^{bff}(X)) \; : - \; query(nearest\_sg^{bff}(X)).$
$R5 : query(sg^{bff}(Z1)) : - \; query(sg^{bff}(X)), up(X, Z1).$
$R6 : query(nearest\_sg^{bff}(s)).$

Figure 13: Program Nearest_Same_Generation

cost program, and the rewritten rules are similar to the rules of Program Smart_ShortCost (Figure 11). except that *min* is replaced by *least*3. In the evaluation of the rewritten program, only the cheapest three paths between pairs of points are retained. □

Our optimization techniques are orthogonal to Magic rewriting [BR87b, BNR$^+$87] and are applicable to programs that cannot be expressed using transitive closure, as the next example shows.

Example 6.7.4 Consider Program Nearest_Same_Generation (from [GGZ91]) in Figure 13, that computes the "nearest" among all nodes in the "same generation" as a node s. Our techniques can be applied to optimize this program. This program has been rewritten using the Magic Templates transformation, with adornment [BR87b].[9]

The rewriting produces essentially the same program except that there is an aggregate selection $s = sg^{bff}(X, Y, D) : groupby(sg^{bff}(X, Y, D), [X, Y], min(D))$ on predicate sg^{bff}. In the evaluation of the rewritten program, for each X, Y pair only the fact $sg^{bff}(X, Y, D)$ such that D is minimum is retained. □

6.8 Discussion

We note that the evaluation techniques developed in this chapter are orthogonal to the optimization techniques developed in Chapter 5, but there are some restrictions on the use of non-ground facts with aggregation. We require that for all predicates, any arguments that are aggregated upon or used as a group-by argument of an aggregate operation or an aggregate selection must be ground. Other arguments can be non-ground — this does not affect our evaluation technique, although it can affect our rewriting technique. The optimization techniques developed here control the use of facts, and use tests for irrelevance. The optimization techniques developed in Chapter 5 work at the level of fact representation, and rule application. Example 5.9.2 illustrates a program for which both optimization techniques are useful.

Our rewriting techniques can be implemented using sufficient conditions for various tests as we mentioned in the course of the paper. In addition to this, our rewriting techniques provide a basis for human analysis of a program, with the subsequent introduction of aggregate selections by a human rather than a rewriting system. This is useful in cases where the required conditions are met, but the sufficient conditions are not powerful enough, or in systems where the rewriting algorithm has not been implemented. We then have a sound basis for the introduction of aggregate selections, rather than an ad hoc approach.

[9]The notation differs somewhat from that of Beeri and Ramakrishnan [BR87b]. Literals of the form $magic\_p(\ldots)$ in the rewriting of [BR87b] are written as $query(p(\ldots))$ in our notation.

Although, for simplicity, we only considered programs without negation, our results can be extended to deal with programs that use stratified negation. The Magic rewriting of a program with stratified negation or aggregation may not be stratified [BNR$^+$87]. Evaluation techniques have been developed for non-stratified programs generated by Magic rewriting of stratified programs, as well as for more general classes of non-stratified programs (see, e.g., [Ros90, KS91, RSS92a]). We believe our rewriting techniques can be generalized to handle some of these classes of programs, and the evaluation techniques used for these classes of programs can be generalized to use aggregate selections, just as we generalized Semi-Naive evaluation to use aggregate selections.

Van Gelder [Van92] considers programs with unstratified aggregation whose meaning is easier to understand if aggregation is pulled out of recursion. Our evaluation techniques can be viewed as complementing Van Gelder's ideas, by letting the user specify a stratified program, and automatically transforming it into one where aggregation has been pushed into recursion (through the use of aggregate selection).

In Section 6.4.1 we examined the case of $unnecessary_{agg\_f}$ distributing over a function. The definition of distributes over (Definition 6.4.1) can be extended in a straightforward manner by allowing a different function $unnecessary_{agg\_f\_i}$ for each argument of the function. This would let us distribute $unnecessary_{min}$ through to the first argument of "$-$", and get an $unnecessary_{max}$ function on the second argument of "$-$". Technique PS1 generalizes in a straightforward manner.

We can extend Definition 6.4.3 by allowing the strongly non-constrained arguments to have variables that occur as arguments of "non-constraining function" literals — functions that are total on the type of the variable, and whose results is 'assigned' to a variable that does not appear in the rule body or in the groupby variables of the aggregate selection. This can be further generalized to allow for function composition in the rule body, where the result of a function is used as an argument of another non-constraining function, The definition of non-constrained variables can then be correspondingly generalized by also defining all variables that appear only in non-constraining literals to be non-constrained variables. Proposition 6.4.4 generalizes correspondingly, and the basic idea in the proof remains unchanged.

Such an extension would be useful if, for instance, in Example 6.4.6, we had a literal

$$append(P, [edge(X, Y)], P1)$$

in the body, and $P1$ is used only in the head of the rule. *append* is a total function on the type list; this information could let us deduce that P is a non-constrained variable. Similarly, if we had extra arguments for *path*, for instance one that maintains the number of nodes in the path, we may be able to deduce that the argument is non-constrained. Such deductions are useful in generating stronger aggregate selections as in Example 6.4.5.

6.8.1 Related Work

Several papers in the past [RHDM86, ADJ88] addressed optimizations of generalized forms of transitive closure that allowed aggregate operations. Cruz and Norvell [CN89] examine the same problem in a generalized algebraic framework. On the other hand, we deal with a language that can express more general recursive queries with aggregation, and do not make use of any special syntax.

Knuth [Knu77] considers a class of problems that can be viewed in the framework of "superior context free grammers". Superior context free grammers can be viewed as cost-inflationary-min programs with one recursive binary predicate (call it g). A superior function is one whose value is greater than the values of each of its arguments. The cost argument of the head of each rule in the program is computed as a superior monotone non-decreasing function of the cost arguments of the body literals. The non-cost argument of p holds the name of the non-terminal in the superior context-free grammer. The problem is to solve the query

$$query(X, min\langle C\rangle) \colon -g(X, C).$$

Our rewriting techniques apply to this program, since *min* distributes over any monotone non-decreasing function, and we can use Ordered-Agg evaluation on it since the function is superior. The effect is exactly the same as using Knuth's algorithm. Knuth notes several applications of such grammers, such as finding the length of the shortest path, finding the expected number of comparisons in an optimum binary search tree (given probabilities of access of keys and gaps between keys), and optimum code-generation algorithms for compilers. Our evaluation technique generalizes this class, since we do not require the functions to be superior (although Ordered-Agg evaluation may not be applicable). Further, we allow arbitrary programs, which generalize the class of superior context-free grammers. For instance, since we allow the use of function symbols, we can find the optimum binary search tree (mentioned above), rather than just find the expected number of comparisons in the tree.

Recently Ganguly et al. [GGZ91] presented optimization techniques for monotone increasing (resp. decreasing) logic programs with *min* (resp. *max*) aggregate operations. Informally, there must be a single cost argument for each predicate in the program and the program must be monotone on this argument. They transform such a program into a (possibly unstratified) program with negation whose stable model yields the answers to the original program, but does not contain any irrelevant facts. They also present an efficient evaluation mechanism for computing the stable model for the transformed program, which is essentially equivalent to Dijkstra's algorithm for the case of shortest-path.

Our results were obtained independently of Ganguly et al. [GGZ91]. The results of Ganguly et al. complement this work in two important ways. Their idea of ordering of facts in the computation (which we have adapted and extended in Section 6.6.2) offers significant improvements in time complexity, and unlike our technique, theirs can handle monotonic min programs even if the use of *min* is unstratified.

Our techniques improve on those of Ganguly et al. in several ways. First, our techniques handle programs with multiple aggregate operations including *min* and *max*, least k, etc. Thus we can handle a program that maintains path information. Second, our techniques are applicable to stratified programs that are not monotonic. This means that we can handle problems such as the critical path problem. However, our techniques are not applicable to non-stratified programs. Third, we allow predicates with multiple cost arguments and allow multiple atomic aggregate selections on the same predicate. The use of these generalizations is illustrated in Examples 6.7.2 and 6.7.3, which cannot be handled by Ganguly et al.

We note that the rewriting techniques of Ganguly et al. only work efficiently with a version of the program that does not maintain actual paths. There are other common examples of programs that can benefit from our optimizations, although they cannot be handled by [GGZ91] since they are not cost inflationary. These include the shortest path problem with edges of negative weight, and the earliest finish time problem shown in Example 6.7.1. [10]

[10]This program uses *max* and is monotonically increasing, whereas Ganguly et al. require it to be monotonically decreasing.

6.9 Conclusions

We believe that evaluation with Aggregate Optimization will offer considerable time benefits for a significant class of stratified programs that use aggregate operations similar to *min* and *max*. We believe that given a technique such as that of Ganguly et al. [GGZ91], or of Beeri et al. [BRSS89] for evaluating special classes of unstratified programs, our optimization techniques can be adapted for such classes of programs, and can detect irrelevant facts using aggregate selections. Our optimization techniques may be useful for optimizing non-recursive queries, such as SQL queries, that use aggregate operations.

Chapter 7

Conclusion

In the first part of this thesis, we identified some problems with bottom-up evaluation of programs that generate non-ground terms. We presented a combination of an improved rewriting technique and an improved evaluation technique to address these problems. Our optimization techniques provide two benefits.

First, we were able to show that memoization can be done at a relatively low cost in terms of time complexity (a cost of a $\log\log$ factor with respect to Prolog evaluation) if we do not perform subsumption-checking. Whether or not to do subsumption-checking then becomes a matter of whether the cost of subsumption-checking is paid off by savings in terms of recomputation and improved termination properties. Unlike with the naive approach, no significant extra price (in the sense of time complexity) is paid either for storing facts, or for implementing a fair search strategy (breadth-first search instead of the depth-first search implemented by Prolog).

Second, the optimization techniques permit efficient bottom-up evaluation of programs that generate non-ground facts. We presented examples of programs that are best evaluated bottom-up, and use non-ground data-structures. This result is important since non-ground data-structures have been shown to be very useful in the context of Prolog evaluation, and we expect them to be of importance in the context of databases as well.

The optimization techniques[1] have been implemented on the CORAL deductive database system. The extra cost added by the optimization techniques seems to be reasonably small for programs that generate only ground facts.

There have been some extensions to bottom-up evaluation that control the order of search (Ramakrishnan, Srivastava and Sudarshan [RSS92a]). However, it is still an open problem whether Prolog's depth-first control strategy can be simulated bottom-up (or by memoing top-down techniques), without a loss of efficiency for the case where all answers are required. A related issue is that of intelligent backtracking (see, e.g., [CD85]), which allows termination of computation for a subgoal before all answers to the subgoal have been generated. A restricted form of intelligent backtracking can be incorporated within the evaluation of a rule in bottom-up evaluation. The *choice* annotation [NT89, GPSZ91] as well as the *any* aggregate selection [SR92a] provide some of the benefits of intelligent backtracking across rules, in the context of bottom-up evaluation. How to provide the full benefits of intelligent backtracking is an open problem, and is related to the search strategy

[1]Modulo tail-recursion optimization

used. Another area of future work is efficient bottom-up evaluation of programs with negation in rule bodies (see, e.g. [Ros90, KSS91, RSS92a]).

In the second part of this thesis we developed optimization techniques that are useful for programs that use aggregate operations along with grouping operations. We developed a notion of relevance that extends the notion used by Magic rewriting, and presented an evaluation technique, based on aggregate selections, that makes use of this extended notion of relevance. We presented a rewriting technique that can deduce aggregate selections; it is powerful enough to deduce the "optimality principle" for the shortest path program. The examples we presented illustrate the importance of control of deduction. We also identified a class of programs for which there is an efficient control strategy based on ordering the use of facts. The shortest path program falls into this class, as does the larger class of superior context free grammars, and the evaluation technique generalizes Dijkstra's shortest path algorithm.

Future work in this area includes studying the effect of control on evaluation. Extending the set of rules for deducing aggregate selections is also of importance. Another area of interest is to develop techniques to "push" aggregate operations such as *sum* and *count* into rules, even though they do not provide aggregate selections. This could reduce the cost of evaluation considerably in many cases.

Appendix A

Proofs From Chapter 3

A.1 Proofs From Section 3.3

Lemma 3.3.1 Let P be any program, and Q a query on P. Consider a step in a derivation sequence for P_Q^{MGU} such that the evaluation prior to that step has property MGU-Prop.

Suppose a supplementary fact $sup_{j,i}(id, \overline{v_i}, id_{i+1})$ is derived at this step. Let $sup_{j,i}$ be a supplementary predicate generated from a rule R_j of P,

$$R_j : p(\overline{t}) \text{:} -p_1(\overline{t_1}), p_2(\overline{t_2}), \ldots, p_n(\overline{t_n}).$$

such that the body of R_j is non-empty.

Then there are facts $answer(id_1, p_1(\overline{a_1})), \ldots, answer(id_{i-1}, p_{i-1}(\overline{a_{i-1}}))$, and a fact $query(p(\overline{s}), id)$, such that

1. Each $id_m, 1 \leq m \leq i$, is the id of an mgu-subgoal generated from $?p(\overline{s})$, and
2. The substitution for variables of R_j specified by $\overline{v_i}$, is in

$$MGU(\langle p(\overline{t}), p_1(\overline{t_1}), \ldots, p_{i-1}(\overline{t_{i-1}})\rangle, \langle p(\overline{s}), p_1(\overline{a_1}), \ldots, p_{i-1}(\overline{a_{i-1}})\rangle)$$

Proof: We prove this by induction on i (where the supplementary predicate is $sup_{j,i}$).

The base case has $i = 0$. For this case, any $sup1_{j,0}$ fact must be generated using a rule

$$sup1_{j,0}(ID, \overline{V}, p_1(\overline{t_1})) \text{:} -query(p(\overline{t}), ID).$$

Since bottom-up evaluation uses mgus, in any fact created thus $p_1(\overline{t_1})$ is an mgu-subgoal generated from the subgoal with id ID, and the bindings of variables in $\overline{V}$ satisfies Part 2 of this lemma. The rule defining $sup_{j,0}$ merely replaces the goal with its id. Hence the claim holds for the basis case.

Now suppose it holds for all values up to some $k-1$, and consider k. Suppose a fact $sup1_{j,k}(id, \overline{v_k}, p_{k+1}(\overline{b_{k+1}}))$ is generated. It must be generated from a rule of the form:

$$sup1_{j,k}(I, \overline{V}, p_{k+1}(\overline{t_{k+1}})) \text{:} -sup_{j,k-1}(I, \overline{V}, I1), answer(I1, p_k(\overline{t_k})).$$

using some fact $sup_{j,k-1}(id,\overline{v_{k-1}},id_k)$ and some fact $answer(id_k,p_k(\overline{a_k}))$. Hence, by inductive assumption, there is a fact $query(p(\overline{s}),id)$, and there are facts

$$answer(id1,p_1(\overline{a_1})),\ldots,answer(id_{k-2},p_{k-2}(\overline{a_{k-2}})$$

that satisfy the necessary conditions, and $id_1,\ldots,$ id_{k-1} are ids of mgu-subgoals generated from the subgoal with id i.

Further, the bindings of variables in $\overline{v_{k-1}}$ correspond to an mgu for the rule prefix, by Part 2 of the lemma and induction hypothesis. By the statement of the lemma, $p_k(\overline{a_k}))$ is an mgu-answer to the subgoal with id id_k. When making a derivation mgus are used. From this and the structure of supplementary rules it follows that the the substitution for variables of R_j specified by $\overline{v_k}$, is in

$$MGU(\langle q(\overline{t}),p_1(\overline{t_1}),\ldots,p_k(\overline{t_k})\rangle\langle q(\overline{t}),p_1(\overline{a_1}),\ldots,p_k(\overline{a_k})\rangle)$$

Hence the variable bindings created satisfy Part 2 of the lemma.

Using arguments exactly the same as in the base case, the last argument of the generated $sup1_{j,k}$ fact is an mgu-subgoal, generated from id. And as before, the rule defining $sup_{j,k}$ replaces the subgoal by its id. Hence Part 1 of the lemma follows.

This concludes the induction step and the proof of this lemma. □

Lemma A.1.1 *Let P be any program, and Q a query on P. Consider a step in a derivation sequence for program P_Q^{MGU} such that the evaluation prior to that step has property MGU-Prop. Suppose a fact $answer(id,p(\overline{a}))$ is derived at this step.*

Then $p(\overline{a})$ is an mgu-answer to the subgoal with identifier id.

Proof: Such a fact can be generated using a rule of one of three forms. The first case is of rules of the form:

$$answer(I,h(\overline{t})){:}-sup_{j,0}(I,\overline{V},\_).$$

The proof is straightforward for such rules, since the $sup_{j,0}$ fact used in the body is generated by a most general unification of the subgoal with identifier id with the head of a rule with an empty body from P, and the $sup_{j,0}$ fact stores the variable bindings in $\overline{V}$. These variable bindings are used to create the head fact for the same rule from program P.

The second case is of rules of the form:

$$answer(I,h(\overline{t})){:}-sup_{j,n-1}(I,\overline{V},I1),answer(I1,p_n(\overline{t_n})).$$

The supplementary fact used in the body of the rule R must satisfy Lemma 3.3.1. The proof of this case then directly parallels the arguments in the induction step of the proof of Lemma 3.3.1.[1] We omit details, for brevity.

The third case is of Type 6 rules, which are of the form:

$$answer(I,h(\overline{t})){:}-query(b_i(\overline{X_i}),ID,answer(I,h(\overline{t})),b_i(\overline{X_i}).$$

[1]Note that this proof does not used induction — it merely uses the arguments from the induction step of the proof of Lemma 3.3.1.

Let the query fact used be $query(b_i(\overline{a_i}), id, answer(id1, h(\overline{s}))$ and $b_i(\overline{b_i})$. By induction hypothesis, $?b_i(\overline{a_i})$ is an mgu-query on b_i, and the rule application computes an mgu of $\overline{a_i}$ and (a variant of) $\overline{b_i}$. The result of the unification is then an mgu-answer, and by Part 2 of MGU-Prop, the result follows. $\Box$

Theorem 3.3.2 Given any program P and query Q, the bottom-up evaluation of P_Q^{MGU} has property MGU-Prop.

Proof: The proof is by induction on derivation sequences for P^{MGU}. The theorem holds trivially for the empty derivation sequence. Now suppose it holds prior to step m in a derivation sequence. By inductive assumption all subgoals generated earlier are mgu-subgoals, and all answers generated earlier are mgu-answers.

Consider first the case that a fact of the form $query(p_i(\overline{t_i}), id)$ is derived at step m. If the fact is generated from an *initial_query* fact, it is an mgu-subgoal by definition. Otherwise the fact must be generated using of a rule

$$query(p_i(\overline{t_i}), ID){:}-sup_{j,i-1}(HId, \overline{V}, ID).$$

with some fact $sup_{j,i-1}(hid, \overline{v}, id)$. By Part 2 of Lemma 3.3.1, $\overline{v}$ contains bindings generated from an mgu as required by the definition of mgu-subgoals. Hence $?p_i(\overline{t_i})$ is an mgu-subgoal. Now id is the identifier of the subgoal got by applying to $p_i(\overline{t_i})$ the substitution that is stored in $\overline{v}$ (using the rule defining $sup1_{j,i-1}$). Hence $?p_i(\overline{t_i})$ is an mgu-subgoal, with identifier id.

Now consider the case that a fact of the form $answer(id, p(\overline{a}))$ is derived in step m. It follows from Lemma A.1.1 and the induction hypothesis that $p(\overline{a})$ is an mgu-answer to a subgoal with identifier id.

This completes the induction step and the proof of this theorem. $\Box$

Theorem 3.3.3 Given any program P and query Q, the bottom-up evaluation of P_Q^{MGU} is complete with respect to Q, i.e., if a fact p that is an answer to Q is present in the least model of P, then p is subsumed by a fact computed in the bottom-up evaluation of P_Q^{MGU}.

Proof: We prove the following result: (p stands for any predicate, in the following) if a fact $query(p(\overline{b}), id)$ is available to the evaluation of P_Q^{MGU}, then for every fact $p(\overline{a})$ that unifies with $p(\overline{b})$, and is generated by a bottom-up evaluation of program P (the original program), evaluation of P_Q^{MGU} generates a fact $answer(id, p(\overline{c}))$ such that $p(\overline{c})$ subsumes $p(\overline{a})[mgu(\overline{a}, \overline{b})]$. Given this fact, an answer $p(\overline{c})$ will be generated by rule Q_{R3}. The theorem then follows from the completeness of bottom-up evaluation of P.

Assume that this result is not true, and consider the shortest derivation sequence of P such that a fact $p(\overline{a'})$ produced in the derivation sequence contradicts this property. If p is a base predicate, a Type 6 rule would generate the required answers using the base facts.

We now consider the case that p is a derived predicate. Let $p(\overline{a}) = p(\overline{b})[mgu(\overline{a'}, \overline{b})]$. Now, consider the rule R_j (in P) whose instance R''_j is used to derive $p(\overline{a'})$. Let R_j and $R'_j = R''_j[mgu(\overline{a'}, \overline{b})]$ be as follows:

$$R_j : p(\overline{t}){:}-p_1(\overline{t_1}), p_2(\overline{t_2}), \ldots, p_n(\overline{t_n}).$$

$$R'_j : p(\overline{a}){:}-p_1(\overline{a_1}), p_2(\overline{a_2}), \ldots, p_n(\overline{a_n}).$$

Let θ be such that $R'_j = R_j[\theta]$.

Claim: If a fact $query(p(\overline{b}), id)$ is made available, such that $\overline{b}$ subsumes $\overline{a}$, then for each $0 \leq i \leq n-1$ the evaluation of P_Q^{MGU} generates

(1) a fact $sup_{j,i}(id, \overline{v_i}, id_{i+1})$ such that the variable bindings in $\overline{v_i}$ subsume the bindings of variables in θ, and
(2) a fact $answer(id_{i+1}, p_{i+1}(\overline{c_{i+1}}))$ such that $\overline{c_{i+1}}$ subsumes $\overline{a_{i+1}}$.
Proof of Claim: We prove this claim by induction on i. The claim is trivial in case the body of R_j is empty, and we assume that this is not the case.

We consider the basis case first. The subgoal $?p(\overline{b})$ must unify with $p(\overline{t})$ since $\overline{a}$ is an instance of $\overline{t}$ as well as an instance of $\overline{b}$. Hence the body of the rule defining $sup1_{j,0}$ unifies with $query(p(\overline{b}), id)$, and a fact $sup1_{j,0}(id, \overline{v_0}, p_1(\overline{d_1}))$ is generated such that the bindings in v_0 subsume the bindings of variables in θ. From this fact, a fact $sup_{j,0}(id, \overline{v_0}, id_1)$ is generated, and this fact too satisfies part 1 of the claim.

A fact $query(p_1(\overline{d_1}), id_1)$ then gets generated from $sup_{j,0}(id, \overline{v_0}, id_1)$, such that $p_1(\overline{d_1})$ is the result of applying the substitution in v_0 to $p_1(\overline{t_1})$. Hence $p_1(\overline{d_1})$ subsumes $p_1(\overline{a_1})$.

By the outer assumption, an mgu-answer $answer(id_1, p_1(\overline{c_1}))$ such that $p_1(\overline{c_1})$ is at least as general as $p_1(\overline{a_1})$ must be generated by the evaluation of P_Q^{MGU}. This finishes the proof of the basis case.

Now we look at the inductive step for this claim. Suppose that for values $0 \ldots i$ this claim holds. We then have facts $sup_{j,i}(id, \overline{v_i}, id_{i+1})$ and $answer(id_{i+1}, p_{i+1}(\overline{c_{i+1}}))$ that satisfy the claim. These are then used to get an instantiated supplementary rule of the following form:

$$sup1_{j,i+1}(id, \overline{v_{i+1}}, id_{i+2}) \colon - sup_{j,i}(id, \overline{v_i}, id_{i+1}), answer(id_{i+1}, p_{i+1}(\overline{c_{i+1}})).$$

This instantiated rule then generates the fact $sup1_{j,i+1}(id, \overline{v_{i+1}}, id_{i+2})$. Now, $p_{i+1}(\overline{c_{i+1}})$ subsumes $p_{i+1}(\overline{a_{i+1}})$, and the bindings in $\overline{v_i}$ subsume the bindings in θ. Hence the unifier for the supplementary rule subsumes θ, and these bindings are stored in $\overline{v_{i+1}}$. This in turn leads to the generation of a fact $sup_{j,i+1}(id, \overline{v_{i+1}}, id_{i+2})$ that satisfies Part (1) of the claim.

Arguments similar to those in the base case show that this leads to the generation of a fact $query(p_{i+2}(\overline{b_{i+2}}), id_{i+2})$ such that $p_{i+2}(\overline{b_{i+2}})$ subsumes $p_{i+2}(\overline{a_{i+2}})$. As before, by the outer induction hypothesis, an mgu-answer $answer(id_{i+2}, p_{i+2}(\overline{c_{i+2}}))$ such that $p_{i+2}(\overline{c_{i+2}})$ is at least as general as $p_{i+2}(\overline{a_{i+2}})$ must be generated by the evaluation of P_Q^{MGU}.

This completes the induction step and the proof of the claim.
End Proof of Claim.

If the body of R_j is empty, the body of the rule defining $sup_{j,0}$ unifies with $query(p(\overline{b}), id)$, and a fact $sup_{j,0}(id, \overline{v_0})$ with variable bindings being an mgu of $p(\overline{b})$ with $p(\overline{t})$ is generated. This mgu is used to generate a head fact $answer(id, p(\overline{c}))$. Hence $p(\overline{c})$ subsumes $p(\overline{a})$, and both parts of the claim are satisfied.

If the body of R_j is not empty, the claim above shows that there are facts $sup_{j,n-1}(id, \overline{v_{n-1}}, id_n)$ and $answer(id_n, p_n(\overline{c_n}))$ that satisfy the conditions of the claim. There must be a rule defining $answer(ID, p(\overline{t}))$ using $sup_{j,n-1}$ and answers for p_n. Arguments similar to those in the induction step show that this rule then derives derives a fact $answer(id, p(\overline{c}))$ such that $p(\overline{c})$ subsumes $p(\overline{a})$.

This leads to a contradiction with the assumption, and completes the proof of this theorem. □

A.2 Proofs From Section 3.4

Proposition A.2.1 *Let P be any program, and Q a query on P. In any evaluation of P^{MGU_T}, for any fact of the form $query(p, id0, a)$ or of the form $sup_{i,j}(id0, \overline{v}, id1, a)$ that is derived, argument a must be of* *the form answer$(id, q(\ldots))$ for some predicate q.* □

Proof: (Sketch) This result follows the structure of rules in P^{MGU_T} through a simple induction on derivations. □

The following lemma provides some intuition behind the variable bindings stored in the supplementary facts.

Lemma A.2.2 *Let P be any program, and Q a query on P. Consider a step in a derivation sequence for P^{MGU_T} such that the evaluation prior to that step satisfies property MGU_T-Prop.*

Suppose a supplementary fact $sup_{j,i}(id, \overline{v_i}, id_i, a)$ is derived at this step. Let

$$R_j : p(\overline{t}) \text{:} -p_1(\overline{t_1}), p_2(\overline{t_2}), \ldots, p_n(\overline{t_n}).$$

be the rule of the original program P from which $sup_{j,i}$ was generated.

Then there are facts $answer(id_1, p_1(\overline{a_1})), \ldots$, $answer(id_{i-1}, p_{i-1}(\overline{a_{i-1}}))$, and a fact $query(p(\overline{s}), id, A)$, such that

1. *Each $id_m, 1 \leq m \leq i$ is the id of an mgu-subgoal generated from $?p(\overline{s})$, and*
2. *The substitution for variables of R_j specified by $\overline{v_i}$, is in*

$$MGU(\langle p(\overline{t}), p_1(\overline{t_1}), \ldots, p_{i-1}(\overline{t_{i-1}})\rangle, \langle p(\overline{s}), p_1(\overline{a_1}), \ldots, p_{i-1}(\overline{a_{i-1}})\rangle)$$

Proof: The structure of the supplementary rules in $P_Q^{MGU\_T}$ is exactly the same as the structure of the supplementary rules in P_Q^{MGU}, except that they carry an additional variable A. In the bodies of the supplementary rules, A is used only in the supplementary literals. Thus A does not affect any of the arguments in the proof of Lemma 3.3.1, and the proof holds unchanged. For brevity, we do not repeat the proof here. □

Lemma A.2.3 *Let P be any program and Q a query on P. Consider a step in a derivation sequence for P^{MGU_T} such that the evaluation prior to that step satisfies property MGU_T-Prop.*

Then the fact generated in this step also satisfy the conditions of MGU_T-Prop.

Proof: Consider first Part 1 of MGU_T-Prop. Any fact $answer(id, p(\overline{a}))$ must be generated using either a Type 3 rule or a Type 6 rule. Consider first a Type 3 rule

$$A \text{:} -sup_{j,0}(I, \overline{V}, \_, A).$$

using some fact $sup_{j,0}(hid, \overline{v}, \_, answer(id, p(\overline{a})))$,

By Lemma A.2.2 there must be a fact $query(q(\overline{s}), id', answer(id, p(\overline{a'})))$ such that $\overline{v}$ is an mgu of $q(\overline{s})$ with the head of a rule with empty body. Thus applying the substitution specified by $\overline{v}$ to the subgoal $?q(\overline{s})$, we get an mgu-answer for $?q(\overline{s})$. Hence by Part 2 of MGU_T-Prop, the result of applying the substitution specified by $\overline{v}$ to $p(\overline{a'})$ is an mgu-answer to id. From the rule defining $sup_{j,0}$ it is easy to see that $p(\overline{a})$ is the resultant answer.

Next consider a Type 6 rule

$$A \text{:} -query(b_i(\overline{X_i}), ID, A), b_i(\overline{X_i}).$$

Given a fact $query(b_i(\overline{a_i}), id, answer(id1, q(\overline{a})))$ and a fact for b_i, the mgu θ of the rule with the (renamed) facts results in $b_i(\overline{X_i})$ being instantiated to an mgu-answer of $b_i(\overline{a_i})$. By Part 2 of MGU_T-Prop, the instantiated $q(\overline{a})$ is an mgu-answer to the query with identifier $id1$. This completes the proof of this part of the lemma.

Now consider Part 2 of MGU_T-Prop. Any *query* fact must be derived either using a Type 4 rule of the form:

$$query(p_i(\overline{t_i}), ID1, answer(ID1, p_i(\overline{t_i}))) \text{:} -sup_{i-1}(ID, \overline{V}, ID1).$$

or a Type 5 rule of the form:

$$query(p_n(\overline{t_n}), ID1, A) \text{:} -sup_{n-1}(ID, \overline{V}, ID1, A).$$

In the case of Type 4 rules, the result follows trivially.

Consider now a fact $query(p_n(\overline{s}), id', answer(id, q(\overline{a})))$ generated using a Type 5 rule. Suppose this was generated using a fact $sup_{n-1}(id0, \overline{v}, id1, answer(id, q(\overline{a})))$. By Lemma A.2.2, there must be some facts:

$$answer(id1, p_1(\overline{a_1})), \ldots, answer(id_{i-1}, p_{i-1}(\overline{a_{i-1}}),$$

and a fact $query(p(\overline{s'}), id0, answer(id, q(\overline{a'})))$ such that $?p_n(\overline{s})$ is a mgu-subgoal generated from subgoal $?p(\overline{s'})$. Also, it is then easy to see that with

$$\theta \in MGU(\langle p(\overline{t}), p_1(\overline{t_1}), \ldots, p_{i-1}(\overline{t_{i-1}})\rangle, \langle p(\overline{s'}), p_1(\overline{a_1}), \ldots, p_{i-1}(\overline{a_{i-1}})\rangle)$$

$answer(id, q(\overline{a})) = answer(id, q(\overline{a'}))[\theta]$.

Now if $p_n(\overline{s})[\gamma]$ is an mgu-answer to $?p_n(\overline{s})$ (where γ is a substitution on the variables in $p_n(\overline{s})$), then we would have $p(\overline{s'})[\theta][\gamma]$ to be an mgu-answer to the subgoal $?p(\overline{s'})$. Then by Part 2 of Property MGU_T-Prop, and the fact $query(p(\overline{s'}), id0, answer(id, q(\overline{a'})))$, $answer(id, q(\overline{a'}))[\theta][\gamma]$ is an mgu-answer to the subgoal with identifier id. But this implies that $answer(id, q(\overline{a}))[\gamma]$ is an mgu-answer to the subgoal with identifier id. This then completes the proof of this part of the lemma. □

Theorem 3.4.1 Consider any program P with query Q. Then a bottom-up evaluation of P^{MGU_T} has property MGU_T-Prop.

Proof: We prove this by induction on derivation sequences for P^{MGU_T}. Each step in the derivation using rules other than Q_{R2} and Q_{R3} derive facts of the form $sup_{i,j}$, $query(\ldots)$, or $answer(\ldots)$.

For the basis case, the first fact in the derivation sequence must be generated from rule Q_{R1}, and this satisfies property MGU_T-Prop. Now assume that the evaluation up to some step a in the derivation sequence satisfies this property. By Lemmas A.2.2 and A.2.3, the fact derived in step a also satisfies property MGU_T-Prop. This completes the induction. □

Lemma A.2.4 *Let P be any program, and Q a query on P. Consider any fact $p(\overline{a})$ generated by a bottom-up evaluation of program P. Suppose a fact*

$$query(p(\overline{b}), id0, answer(id, q(\overline{t})))$$

is available to the evaluation of $P_Q^{MGU\_T}$, such that $p(\overline{a})$ unifies with $p(\overline{b})$. Let $\theta \in MGU(p(\overline{b}), p(\overline{a}))$ (wlog we assume that $\overline{b}$ and $\overline{t}$ share no variables with $\overline{a}$). Then bottom-up evaluation of $P_Q^{MGU\_T}$ generates a fact $answer(id, q(\overline{t}))[\gamma]$ such that γ subsumes θ.

Proof: The proof is by induction on derivation sequences in P. Consider a derivation sequence, and a step s in the sequence such that the lemma holds for every fact derived prior to s in the sequence. Let $p(\overline{a})$ be the fact derived in step s of the sequence.

If $p(\overline{a})$ does not unify with $p(\overline{b})$, the lemma holds for this step in a trivial fashion. Hence we consider the case where they do unify.

If p is a base predicate, the lemma follows in a straightforward manner, since $p(\overline{a})$ would be used in a Type 6 rule with the query fact.

We now consider the case that p is a derived predicate. Let $p(\overline{a'}) = p(\overline{b})[\theta]$. Now, consider the rule R_j (in P) whose instance R''_j is used to derive $p(\overline{a})$. Let R_j and $R'_j = R''_j[\theta]$ be as follows:

$R_j : p(\overline{t}){:} -p_1(\overline{t_1}), p_2(\overline{t_2}), \ldots, p_n(\overline{t_n}).$

$R'_j : p(\overline{a'}){:} -p_1(\overline{a_1}), p_2(\overline{a_2}), \ldots, p_n(\overline{a_n}).$

Let σ be such that $R''_j = R_j[\sigma]$, so that $R'_j = R_j[\sigma][\theta]$.
Claim: If the fact $query(p(\overline{b}), id0, answer(id, q(\overline{s})))$ is made available, then

1. for each $0 \leq i \leq n-1$ the evaluation of P_Q^{MGU} generates a fact

   $sup_{j,i}(id, \overline{V}, id_{i+1}, answer(id, q(\overline{s})))[\theta_i]$

   such that θ_i subsumes $[\sigma][\theta]$, and
2. for each $0 \leq i \leq n-2$ the evaluation of P_Q^{MGU} generates a fact

   $answer(id_{i+1}, p_{i+1}(\overline{c_{i+1}}))$

   such that $\overline{c_{i+1}}$ subsumes $\overline{a_{i+1}}$.
3. the evaluation of P_Q^{MGU} generates a fact

   $query(p_n(\overline{t_n}), id_n, answer(id, q(\overline{t})))[\theta_{n-1}]$

   where θ_{n-1} subsumes $[\sigma][\theta]$.

Proof of Claim: The claim in trivial in case the body of R_j is empty; the rest of this proof assumes that the body of R_j is non-empty. We prove Parts 1 and 2 of the claim by induction on i.

We first consider the basis case. The subgoal $?p(\overline{b})$ must unify with $p(\overline{t})$ since $\overline{a'}$ is an instance of $\overline{t}$ as well as an instance of $\overline{b}$. Hence the body of the rule defining $sup1_{j,0}$ unifies with $query(p(\overline{b}), id0, answer(id, q(\overline{s})))$ and a fact $sup1_{j,0}(id, \overline{V}, p_1(\overline{t_1}), answer(id1, q(\overline{s}))[\theta_0]$ is generated, such that θ_0 subsumes $[\sigma][\theta]$. It is easy to show that a fact for $sup_{j,0}$ that satisfies part 1 of the claim will then be generated in the bottom-up evaluation.

If $p_1(\overline{t_1})$ is not the last literal in the body of the rule, it is easy to show that a fact of the form $query(p_1(\overline{t_1}), id_1, answer(id_1, p_1(\overline{t_1})))[\theta_0]$, where $\overline{t_1}[\theta_0]$ subsumes $p_1(\overline{a_1})$, is generated from this supplementary fact. By the outer induction hypothesis, an mgu-answer $answer(id_1, p_1(\overline{c_1}))$ such that $p_1(\overline{c_1})$ is at least as general as $p_1(\overline{a_1})$ must be generated by the evaluation of P_Q^{MGU}. This finishes the proof of the basis case.

For the induction step, assume that parts (1) and (2) of the claim hold for $0 \leq i \leq k < n-1$. We can then show that Part 1 of the claim holds for $i = k+1$, and if $k < n-2$, we can show that Part 2 holds for $i = k+1$. The proof parallels that used in Theorem 3.3.3, since the structure of the rules is similar except for the rule for the last literal. We omit the details.

This completes the proof of (1) and (2). A fact of the form $query(p_n(\overline{t_n}), id_n, answer(id, q(\overline{s})))[\theta_{n-1}]$ must be produced by a Type 5 rule. Part 3 of the claim then follows from Part 1 of the claim and from the structure of Type 5 rules.

End proof of claim

If the body of the rule is empty, it is easy to show (in a manner similar to the base case of the above claim) that a fact is produced for $sup_{j,0}$, and a Type 3 rule then generates a fact $answer(id, q(\overline{t}))[\gamma]$ that satisfies the properties required by this theorem.

If the body of the rule is not empty, the claim above shows that a fact $query(p_n(\overline{t_n}), id_n, answer(id, q(\overline{s})))[\theta_{n-1}]$ is generated. We know that some fact $p_n(\overline{c_n})$ that subsumes $p_n(\overline{a_n})$ is generated in the derivation sequence, before step s (wlog assume it does not share variables with other facts/rules). Hence by the induction hypothesis, a fact $answer(id, q(\overline{s}))[\theta_{n-1}][\delta]$ such that δ subsumes $mgu(p_n(\overline{c_n}), p_n(\overline{t_n}[\theta_{n-1}]))$ is generated. But θ_{n-1} subsumes $[\sigma][\theta]$ and $\overline{c_n}$ subsumes $\overline{a_n}$, and hence $[\theta_{n-1}][\delta]$ subsumes $[\sigma][\theta]$. Let γ be the projection of $[\theta_{n-1}][\delta]$ on the variables in $\overline{s}$. Hence $answer(id, q(\overline{s}))[\gamma] = answer(id, q(\overline{s}))[\theta_{n-1}][\delta]$, and this fact is generated by bottom-up evaluation. Since σ does not affect the variables in $\overline{s}$, γ subsumes θ.

This completes the induction step, and the proof of the theorem. $\square$

Theorem 3.4.2 Given any program P and query Q, the bottom-up evaluation of $P_Q^{MGU\_T}$ is complete with respect to Q, i.e., if the bottom-up evaluation of P generates a fact p that is an answer to Q, then p is subsumed by a fact computed in the bottom-up evaluation of $P_Q^{MGU\_T}$.

Proof: Let $Q = ?q(\overline{t})$, and let its identifier be id. Then rules Q_{R1} and Q_{R2} generate a fact $query(q(\overline{t}), id, answer(id, q(\overline{t})))$. From Lemma A.2.4, the corresponding *answer* facts will be generated by the evaluation of $P_Q^{MGU\_T}$. The required answers to the query will then be generated by rule Q_{R3} using these facts. $\square$

Appendix B

Proofs From Chapter 4

We start by describing some assumptions we make and some notation that we use. We assume that each derivation step has a unique identifier, and we label facts derived by SN evaluation with the identifier of the derivation step that generated the fact. We use the notation $p() : k$ to denote a derivation of fact $p()$ with label k. (This label is ignored for the purpose of subsumption-checking; if subsumption-checking is used and a fact is generated twice (with different labels) only one copy of the fact with one label is stored and used in derivations.) In a similar fashion, we label actions (such as generation of a query or answer) performed by Prolog* in order to distinguish between multiple occurrences of the action.

We use the concept of labeled derivation steps and labeled attempted derivation steps (Section 4.2). In the rest of this section, we consistently use the term *derivations* (resp. *attempted derivations*) to refer to labeled derivation steps (resp. labeled attempted derivation steps).

The evaluation of a program rewritten using MGU MTTR rewriting generates goal-identifiers, given goals. We denote by $\mathsf{goal\_id}^{-1}$: goal-identifiers $\rightarrow$ goals as the inverse function of $\mathsf{goal\_id}$. That is, given a goal-identifier generated in an evaluation, it returns the original goal. Recall that in case subsumption-checking is not performed, the $\mathsf{goal\_id}$ meta-predicate is defined to return a different identifier on each call, but $\mathsf{goal\_id}^{-1}$ is well-defined.

We note that the mapping we specify is modulo renaming. For example, when we say that M maps a derivation $f1 = query(p(\bar{t}), \ldots) : k1$ to the generation of a subgoal $?p(\bar{t}) : k$ by Prolog*, we mean that there is a renaming of $f1$ such that its first argument is equal to the Prolog* subgoal $?p(\bar{t}) : k$. (Note that if we rename the Prolog* subgoal instead, we would have to perform a “global” renaming rather than just the given subgoal. Hence we rename the facts derived in bottom-up evaluation.) We assume that the rules in the original program are numbered $R_1, R_2, \ldots$. Recall that a predicate of the form $sup_{j,i}$ is derived from rule R_j.

We assume in the following lemma that the basic version of MGU MTTR rewriting is used, without any of the optimizations described in Section 3.4.1. For simplicity, the lemma assumes that the program uses no base predicate, and hence there are no Type 6 rules. After proving the lemma, we show how the proof can be extended to allow base predicates, and to incorporate some optimizations of MGU MTTR rewriting.

Lemma B.0.5 Let P be any (positive) logic program, and Q a query on P. Assume that P uses no base predicates. Let P^{MGU_T} be the MGU-MTTR rewriting of P with query Q. Then there is a mapping M of labeled attempted derivation steps in the Semi-Naive evaluation of P^T (with or without subsumption-checking) to actions of the Prolog* evaluation of Q on P, with the following properties.

1. **Goal identifiers:**

   M maps each goal identifier id to the Prolog* action $?\mathsf{goal\_id}^{-1}(id) : k1$ (this action is the generation of a subgoal).

2. **Type 0 rules - Q_{R2}, Type 4 and Type 5 rules:**

   Consider an attempted derivation using a Type 4 or Type 5 rule, or a Type 0 rule Q_{R2}. Such attempted derivations always succeed, and generate a fact

   $$f1 = query(p(\overline{t}), hid, answer(id, q(\overline{s}))) : k$$

   Then M maps the derivation $f1$ to a Prolog* action $?p(\overline{t}) : k1$. Further,

   (a) the return point of $?p(\overline{t}) : k1$ is a query that is equivalent to $M(id)$, and,

   (b) $q(\overline{s})$ is the instantiated return-point query, at the point that Prolog* generated the subgoal $?p(\overline{t}) : k1$.

   No two distinct labeled derivations of this type are mapped to the same Prolog* action.

3. **Type 3 rules:**

   Consider an attempted derivation using a Type 3 rule. Such a derivation always succeeds.

   M maps each derivation $answer(id, p(\overline{a})) : k$ to the generation of an answer $p(\overline{a})$ to a Prolog* query that is equivalent to $M(id)$.

   No two distinct labeled derivations of this type are mapped to the same Prolog* action.

4. **Type 0 rules - Q_{R1}:**

   Such a rule makes a derivation

   $$f1 = initial\_query(q(\overline{t}), hid, answer(id, q(\overline{t}))) : k$$

   M maps $f1$ to the Prolog* action $?q(\overline{t}) : k1$.

   There is only one such labeled attempted derivation, and it always succeeds.

5. **Type 0 rules - Q_{R3}:**

   Consider an attempted derivation using rule Q_{R3}. If the derivation succeeds, it derives

   $$f1 = q(\overline{a}) : k$$

   using a labeled fact $f2 = answer(id, q(\overline{a})) : k1$. M maps $f1$ to $M(f2)$.

   No two distinct labeled derivations of this type are mapped to the same Prolog* action.

   If the derivation fails, it must have used a labeled fact $f1$ for $initial\_query$. M maps the unsuccessful derivation to $M(f1)$. There is at most one such unsuccessful labeled derivation.

6. **Type 1 rules - 1:**

   Consider an attempted derivation using a Type 1 rule that has a predicate $query(\ldots)$ in the body. Let the fact used in the body be

   $$query(p(\overline{b}), hid, answer(id, q(\overline{s}))) : k1$$

   The derivation possibly derives a fact of one of the following forms:

   $$sup_{j,0}(hid, \overline{v}, 0, answer(id, q(\overline{a}))) : k$$

   $$sup_{j,0}(hid, \overline{v}, p(\overline{t}), answer(id, q(\overline{a}))) : k$$

   Now for the query fact, due to induction hypothesis, it must be the case that

   $$M(query(p(\overline{b}), hid) : k1) = ?p(\overline{b}) : k2$$

   M maps the labeled attempted derivation to the Prolog* action of unifying $?p(\overline{b}) : k2$ with the head of R_j.

   No two distinct labeled (successful/unsuccessful) attempted derivations of this type are mapped to the same Prolog* action.

   Further

   (a) the return point of $?p(\overline{b}) : k2$ is equivalent to $M(id)$, and,

   (b) $q(\overline{a})$ is the instantiated return-point query, just after Prolog* carries out the above unification.

   (c) the bindings stored in $\overline{v}$ are the bindings of the rule variables of R_j just after Prolog* carries out the above unification.

7. **Type 2 rules - 1:**

   Consider an attempted derivation using a Type 2 rule, where the body of the rule uses a fact $sup_{j,i}$. A head fact of the following form may be derived:

   $$sup1_{j,i}(hid, \overline{v}, p_{i+1}(\overline{s}), answer(id, q(\overline{a}))) : k$$

   If the derivation succeeds, M maps the labeled derivation to the Prolog* action of returning an answer to $?p_i(\overline{s}) : k1$, where

   (a) the query $?p_i(\overline{s})$ is generated from the ith literal of rule R_j, and the literal is not the last in the rule.

   (b) the bindings stored in $\overline{v}$ are the bindings of the rule variables of R_j at the point when the answer to $?p_i(\overline{s}) : k1$ was returned.

   (c) $q(\overline{a})$ is the instantiated version, at the point that the answer to the query is returned, of the return-point query of the call to R_j.

No two distinct labeled derivations of this type are mapped to the same Prolog* action.

If the derivation fails, there are two cases. If the derivation fails because there are no matching answer facts for a labeled supplementary fact $s : k3$, M maps the unsuccessful derivation to $M(s : k3)$. If the derivation fails because there are no matching facts for a labeled answer fact $a : k4$, M maps the unsuccessful derivation to $M(a : k4)$.

At most a constant number of failed labeled derivations of the above form are mapped to the same Prolog* action.

8. **Type 1 rules - 2 and Type 2 rules - 2:**

Consider an attempted labeled derivation using a Type 1 or Type 2 rule where the body of the rule has a literal $sup1_{j,i}$. Such a derivation always succeeds, and derives a fact of the form:

$$f = sup_{j,i}(hid, \overline{v}, nid, answer(id, q(\overline{a}))) : k$$

using a body fact of the form

$$f1 = sup1_{j,i}(hid, \overline{v}, p_{i+1}(\overline{s}), answer(id, q(\overline{a}))) : k1$$

Then $M(f)$ is defined to be $M(f1)$, which is the return of an answer to a query $?p_i(\overline{s}) : k1$. Further,

(a) the query $?p_i(\overline{s})$ is generated from the ith literal of rule R_j, and the literal is not the last in the rule.

(b) the bindings stored in $\overline{v}$ are the bindings of the rule variables of R_j at the point when the answer to $?p_i(\overline{s}) : k1$ was returned.

(c) $q(\overline{a})$ is the instantiated version, at the point that the answer to the query is returned, of the return-point query of the call to R_j.

(d) nid is the identifier of a subgoal $?p_i(\overline{s})$.

No two distinct labeled derivations of this type are mapped to the same Prolog* action.

Proof: The labeled derivations in the evaluation of P^T are totally ordered, such that each derivation uses only facts computed in earlier derivations. We use an induction on this sequence to prove the lemma. Note that at many points we say that a particular action will be performed by Prolog* evaluation. Such claims depend on the assumption that Prolog* evaluation terminates. In case Prolog* evaluation does not terminate, bottom-up evaluation can be no worse.

The base case is for an empty derivation sequence, and the induction hypothesis holds trivially. Now assume that there is a mapping M for labeled derivations up to step n, that satisfies the conditions of the lemma. We extend the mapping to step $n + 1$. We split the derivation in step $n + 1$ into several cases based on the type of the rule used.

For each rule type, we prove the corresponding claims. Goal-identifiers are generated only by the Type 0 rule Q_{R1}, Type 1 rules subcase 2 and Type 2 rules subcase 2. We prove the claims about goal-identifiers in the respective cases below.

Type 0 Rules - Q_{R1} and Q_{R3} : Consider a rule Q_{R1}. Such a rule generates a fact

$$initial\_query(q(\overline{t}), id, answer(id, q(\overline{t}))) : k$$

from the initial query. Let M map this derivation to the Prolog* action $?q(\overline{t}) : k0$ corresponding to the generation of the initial query. Further, we let M map id to the same Prolog* subgoal generation action.

Next, consider Q_{R3}, and suppose that the answer fact used in the body is $answer(id, q(\overline{a})) : k$. The derivation is mapped to $M(answer(id, q(\overline{a})) : k)$. Each fact $answer(id, q(\overline{a})) : k$ is used in at most one such derivation. By induction hypothesis, no other derivation of any fact $q(\overline{b})$ is mapped to this action.

Type 1 Rules - 1 :

This case covers labeled attempted derivations using Type 1 rules with a query literal in the body.

The attempted derivation must have used a rule of one of the forms below:

$$R_Q : sup1_{j,0}(\ldots) \colon -query(q(\overline{s}), ID, A).$$

$$R_Q : sup_{j,0}(\ldots) \colon -query(q(\overline{s}), ID, A).$$

and a fact $f1 = query(q(\overline{a}), id1, answer(id2, r(\overline{b}))) : k1$.

By induction hypothesis, $f1$ is mapped to a subgoal $?q(\overline{a}) : k3$. Now Prolog* evaluation will attempt to unify the subgoal $?q(\overline{a}) : k3$ with the head of rule R_j,[1] which is $q(\overline{s})$. We label this unification action as $k4$, and the attempted derivation is mapped to this unification action.

No other attempted derivation of this kind is mapped to this unification, since this is the only use of $f1$ with this rule, and no other derivation of a $query$ fact is mapped to $?q(\overline{a}) : k3$.

If the unification is successful, a fact is created in bottom-up evaluation, and Prolog* evaluation either returns an answer (if the rule is empty) or sets up a subgoal on the first body literal. The fact created by derivation in the two cases are respectively:

$$sup_{j,0}(hid, \overline{v_0}, 0, answer(id2, r(\overline{b'}))) : k$$

and

$$sup1_{j,0}(hid, \overline{v_0}, p_1(\overline{s}), answer(id2, r(\overline{b'}))) : k$$

The induction hypothesis shows that the return point of $?q(\overline{a}) : k3$ is equivalent to $M(id2)$. By induction hypothesis, $r(\overline{b})$ is the instantiated return-point subgoal when the subgoal $?q(\overline{a}) : k3$ is set up. The Prolog* unification of the query with the rule head $q(\overline{s})$ produces the same bindings for $\overline{a}$ and $\overline{b}$ as the unification of $f1$ with the body literal of R_Q. Hence $r(\overline{b'})$ is equivalent to the instantiated return-point query after the unification of $?q(\overline{a}) : k3$ with the head of R_j. It is easy to show from the structure of R_Q that the bindings stored in $\overline{v_0}$ are the bindings of the rule variables after Prolog* carries out the unification.

[1]Rule R_j is the rule in the original program from which R_Q is derived.

Type 2 Rules - 1 :

Consider an attempted derivation using a Type 2 rule, where the body of the rule uses the predicate $sup_{j,i-1}$.

First consider the case that the derivation succeeds. It must have used labeled facts of the following form:

$$f1 = sup_{j,i-1}(hid, \overline{v0}, nid, answer(id, q(\overline{s}))) : k1$$

$$f2 = answer(nid, p_i(\overline{a})) : k2$$

and a rule:

$$\begin{aligned} R_Q : sup1_{j,i}(HId, \overline{V_i}, p_{i+1}(\overline{t_{i+1}}), A) :- \; & sup_{j,i-1}(HId, \overline{V_{i-1}}, ID1, A), \\ & answer(ID1, p_i(\overline{t_i})). \end{aligned}$$

to derive a fact

$$f = sup1_{j,i}(hid, \overline{a_i}, p_{i+1}(\overline{s_{i+1}}), answer(id, q(\overline{s}))) : k0$$

Since this rule defines a $sup1_{j,i}$ predicate, the original rule say R_j from which the $sup1_{j,i}$ predicate was derived must have as ith body literal $p_i(\overline{t_i})$. Also we can show that supplementary fact $f1$ must have derived a query fact

$$f3 = query(p_i(\overline{s_i}), nid, answer(nid, p_i(\overline{s_i}))) : k3$$

where the goal-id of $?p_i(\overline{s_i})$ is nid. (If there is more than one derivation of this fact, all but one of them are eliminated by subsumption-checking.)

Now, by induction hypothesis, $f1$ is mapped to the either the successful unification of a query with the head of rule R_j, or to the return of an answer to $p_{i-1}(\overline{t_{i-1}})$, which is the $i-1$th literal in rule R_j. It is easy to show (from the induction hypothesis claim about variable bindings) that at this point in the evaluation, Prolog* would have generated a query $?p_i(\overline{s_i}) : k4$ from this literal. The query is not tail-recursive, and hence the return point of the query is the same as the point where the query is generated. Note that the goal-id of $?p_i(\overline{s_i}) : k4$ is nid, but $M(nid)$ may not be $?p_i(\overline{s_i}) : k4$, (although it is equivalent), if subsumption-checking is used.

By induction hypothesis on $f2$, the generation of answer $f2$ is mapped to the generation of an answer $p_i(\overline{a}) : k6$ to a query $f3$ that is equivalent $M(nid)$. But since the queries $M(nid)$ and $?p_i(\overline{s_i}) : k4$ are equivalent, so are $f3$ and $?p_i(\overline{s_i}) : k4$. Since $f3$ and $?p_i(\overline{s_i}) : k4$ are equivalent and both are not tail-recursive, each answer $p_i(\overline{a}) : k6$ generated for $f3$ can be mapped one-to-one to the generation of an answer $p_i(\overline{a}) : k7$ to $?p_i(\overline{s_i}) : k4$. Hence let this mapping map $M(f2)$ to the generation of an answer $p_i(\overline{a}) : k8$ for the query $?p_i(\overline{s_i}) : k4$.

We then define $M(f)$ to be the return of answer $p_i(\overline{a}) : k8$ to the query $?p_i(\overline{s_i}) : k4$.

No two such distinct derivations are mapped to the same Prolog* action since by induction hypothesis, (1) only $f1$ is mapped to the point in Prolog* evaluation just before the generation of $?p_i(\overline{s_i}) : k4$, and (2) no two derivations of $answer(nid, p_i(\overline{a}))$ are mapped to the same return of answer $p_i(\overline{a})$ to $?p_i(\overline{s_i}) : k4$ by Prolog* evaluation.

We show that the required bindings are stored in the generated answer fact as follows. By induction hypothesis, the bindings in $f1$ are the same as the rule bindings in Prolog* evaluation before $?p_i(\overline{s_i}) : k4$ is generated. Since the same answers are used in both cases, with the same literals, it is straightforward to show that the generated bindings are the same in f as when answer $p_i(\overline{a}) : k8$ is returned in Prolog* evaluation. Similar arguments also show that $answer(id, q(\overline{s}))$ is the instantiated return-point query at the point when the answer is returned.

This completes the case where the attempted derivation is successful. If the derivation fails, the mapping defined is straightforward. Since each fact is used in at most one unsuccessful derivation with each rule, it follows that at most a constant number of failed derivations are mapped to the same Prolog* action.

Type 1 and Type 2 Rules - 2 :

The derivation must have used a rule

$$R_Q : sup_{j,i}(\ldots) \text{:} -sup1_{j,i}(\ldots), \mathsf{goal\_id}(\ldots).$$

with a fact

$$f1 = sup1_{j,i}(hid, \overline{v_0}, p(\overline{s}), answer(\ldots)) : k0$$

to derive a fact $f = sup_{j,i}(hid, \overline{v_0}, nid, answer(\ldots)) : k5$.

We let M map f to $M(f1)$. This is the only use of $sup1_{j,0}(\overline{v_0}, \ldots) : k0$. Along with the induction hypothesis, this shows that no two distinct derivations of this kind are mapped to the same Prolog* action.

R_Q passes all arguments of $f1$ unchanged to f except that it replaces $p(\overline{s})$ by its goal-identifier nid. The other claims about the return point query and the query $?p_i(\overline{s})$ are shown directly by applying the induction hypothesis to $f1$, since these arguments are the same in f as in $f1$.

By induction hypothesis, $f1$ is mapped to a point in Prolog* evaluation where either a subgoal has been unified with a rule, or an answer has been returned to a literal in the rule. In either case, $p(\overline{s})$ is the instantiated literal that is next in the rule, and a subgoal $?p(\overline{s}) : k3$ will be generated. We define $M(nid)$ to be $?p(\overline{s}) : k3$.

Type 3 Rules: A Type 3 rule is of the form

$$A \text{:} -sup_{j,0}(HId, \overline{V}, \_, A).$$

Suppose a fact $f = answer(id, q(\overline{b})) : k0$ is derived using such a rule along with a fact

$$f1 = sup_{j,0}(HId, \overline{a}, 0, answer(id, q(\overline{s})) : k1$$

Now $sup_{j,0}(\overline{a}, answer(id, p(\overline{s}))) : k1$ is mapped to a Prolog* action that unifies a goal $?p(\overline{t}) : k2$ with the head of rule R_j. Let this action be labeled $k2$. This unification action must succeed, since the fact $sup_{j,0}(\overline{a}, answer(id, q(\overline{s}))$ was derived. Further, the body of the original rule must be empty (Type 3 rules are generated only from such rules). Hence Prolog* evaluation generates an answer at this stage.

By induction hypothesis on the $sup_{j,0}$ fact, (1) the return point of the query $?p(\overline{t}) : k2$ is equivalent to $M(id)$, and (2) $q(\overline{b})$ is the instantiated return-point query after the unification of $?p(\overline{t}) : k2$ with the head of rule R_j is performed. Hence Prolog$*$ evaluation generates an answer $q(\overline{b}) : k4$ for a return-point query that is equivalent to $M(id)$.

This is the only answer generated from $f1$, and by induction hypothesis, no two distinct labeled $sup_{j,0}$ facts are mapped to the same unification action. Hence no two distinct derivations of this kind are mapped to the same Prolog* action.

Type 0 Rules - Q_{R2}, Type 4 and Type 5 Rules :

We split this into three sub-cases, based on the rule type. We first consider the Type 0 rule Q_{R2}. This generates a query fact

$$query(p(\overline{t}), id, answer(id, p(\overline{t}))) : k$$

from the initial query. We let M map the derivation to the generation of the initial query $?p(\overline{t}) : k0$ by Prolog* evaluation.

The return point of this query is $?p(\overline{t}) : k0$, and this is equivalent to $M(id)$, since id is the goal-identifier for $p(\overline{t})$. This is the only derivation of this type that is mapped to this action of Prolog* evaluation. The remaining part of the claim for this case follows trivially since $?p(\overline{t}) : k0$ is the return-point query.

Next we consider Type 4 rules. These correspond to non-tail-recursive literals. Such a rule is of on form

$$query(p_i(\overline{t_i}), ID, answer(ID, p_i(\overline{t_i}))) \colon - sup_{j,i-1}(HId, \overline{V}, ID, A).$$

Let the generated fact be

$$f = query(p_i(\overline{s_i}), nid, answer(nid, p_i(\overline{s_i}))) : k$$

and let the fact used in the rule body be

$$f1 = sup_{j,i-1}(hid, \overline{v}, nid, answer(id, q(\overline{s}))) : k1$$

Now, by induction hypothesis, $M(f1)$ is mapped to a step where a query has been unified with a rule, or an answer has been returned for a literal, and the next literal in the rule is $p_i(\overline{t_i})$. The induction hypothesis also tells us that the variable bindings stored in $\overline{v}$ above are the same as the rule variable bindings. Hence Prolog* evaluation generates a query $?p_i(\overline{s_i}) : k3$. We let M map f to $?p_i(\overline{s_i}) : k3$.

Each supplementary fact is used in exactly one rule of this kind, and by induction hypothesis, no other supplementary fact is mapped to the same Prolog* action. Hence no other derivation of this type is mapped to the same action as Prolog* evaluation. The return point of this query is the query itself, since the literal is not tail-recursive, and the induction hypothesis on $f1$ shows that nid is the goal-id of $?p_i(\overline{s_i})$. Hence the claims about the return point and the instantiated return-point query follow in a straightforward manner.

Finally we consider Type 5 rules. Such a rule is of the form

$$query(p_i(\overline{t_i}), ID, A) \colon - sup_{i-1}(HId, \overline{V}, ID, A).$$

Let the derived fact be

$$f = query(p_i(\overline{s_i}), nid, answer(id, q(\overline{s}))) : k$$

and let the fact used in the body be

$$f1 = sup_{j,i-1}(hid, \overline{v}, nid, answer(id, q(\overline{s}))) : k1$$

The same arguments as for Type 4 rules show that there is a query $?p_i(\overline{s_i}) : k3$ generated by Prolog*, and we let M map f to $?p_i(\overline{s_i}) : k3$. The same argument as for Type 4 rules shows that no other derivation of this type is mapped to the same action of Prolog* evaluation. To show the claims about the return point we note the following. The literal for which the query is generated by a Type 5 rule is tail-recursive. Hence its return point is the same as that of the head of the rule. By induction hypothesis, this is equivalent to $M(id)$. Again, the induction hypothesis on $f1$ tells us that $q(\overline{s})$ is the instantiated return-point query at the point when Prolog* generates the subgoal $?p_i(\overline{s_i}) : k3$.

This completes the induction step, and the proof of the lemma. □

The above lemma was for the case that the program uses no base predicates. We can extend the lemma for the case of base predicates as follows. We use the optimizations described in Section 3.4.1 to treat all base literals as non-tail-recursive, and to not generate query or answer facts for these predicates. In particular, we can ensure that base literals that occur as the last literal in a rule are treated as non-tail-recursive by adding an extra $true()$ literal at the end of the rule body in the original program. Such a transformation does not affect number of actions performed by Prolog* evaluation significantly, and does not affect the time complexity of Prolog* evaluation.

As a result of the optimization, query rules and answer generation rules for base predicates are deleted. These deletions do not affect the mapping we described above. The only other change is that some Type 2 rules are simplified, and are now of one of the the following forms:

$$\begin{array}{ll} sup1_{j,i}(HId, \overline{V}, p_{i+1}(\overline{t_{i+1}}), A) & : -\ sup_{j,i-1}(HId, \overline{V}, ID1, A), p_i(\overline{t_i}). \\ sup_{j,i}(HId, \overline{V}, 0, A) & : -\ sup_{j,i-1}(HId, \overline{V}, ID1, A), p_i(\overline{t_i}). \\ sup_{j,i}(HId, \overline{V}, 0, A) & : -\ sup_{j,i-1}(HId, \overline{V}, ID1, A), \\ & \quad answer(ID1, p_i(\overline{t_i})) \end{array}$$

We classify all such rules under the case "Type 2 rules - 1". All the claims made for successful derivations in this case still hold, and the proof for this case works with minor modifications. We note that any facts that are used for the literal $p_i(\overline{t_i})$ in the above rules during bottom-up evaluation are also used in the corresponding stage in Prolog* evaluation. We now consider the case of failed derivations. Since Semi-Naive evaluation is used, and p_i is base or evaluable, any attempted derivation using such a rule uses a fact for $sup_{j,i-1}$, and sets up a query on $p_i(\overline{t_i})$. Prolog* evaluation would set up the same query. The derivation fails if there is no fact for p_i that unifies with the query. But in this case, Prolog* evaluation also fails on the same query. Hence the attempted derivation is mapped to the failed query attempt by Prolog* evaluation.

The above optimization and the extension of the mapping described above is important, in particular, when we consider the cost of evaluation in Chapter 5 — we introduce equality literals into rule bodies, and treat them as base predicates.

Thus we have the following theorem.

Theorem 4.3.1 Let P be a definite clause program, and Q be a query on the program. There are constants c_1 and c_2 (that may depend on the size of P) such that the following is satisfied.

Let P^{MGU_T} be the MGU MTTR rewriting of $\langle P, Q\rangle$. Given any database, let the number of labeled attempted derivation steps performed by a Semi-Naive evaluation (with or without subsumption checking) of P^{MGU_T} be n, and let the number of actions performed by Prolog* evaluation of query Q with the same database be m. Then $n < c_1 \cdot m + c_2$.

Proof: Lemma B.0.5 showed us that no two distinct labeled derivation steps using any rule type are mapped to the same action of Prolog* evaluation. Since there are only a finite number of rule types, at most a constant number of successful derivation steps are mapped to the same action of Prolog* evaluation. Lemma B.0.5 also showed that at most a constant number of unsuccessful labeled attempted derivation steps are mapped to any action of Prolog* evaluation.

To complete the proof of the theorem, we use the non-repetition property of Semi-Naive evaluation without subsumption-checking: no labeled derivation step is repeated in the evaluation (Theorem 4.2.1). (Semi-naive evaluation with subsumption-checking has a stronger non-repetition property, namely, no derivation step is repeated in the evaluation.) □

Appendix C

Proofs From Chapter 5

C.1 Proofs from Section 5.5

Lemma C.1.1 *Suppose that a MGU MTTR rewritten program is evaluated using Apply_Rule. Then*

1. *Given a supplementary/initial_query fact s and a query fact q, if $s.cont\_id = q.par\_id$, then $q.bindenv$ is a version descendant of $s.bindenv$.*
2. *Given a supplementary fact s and an answer fact a, if $s.cont\_id = a.par\_id$, then $a.bindenv$ is a version descendant of $s.bindenv$.*
3. *Given supplementary facts $s1$ and $s2$, if $s1.cont\_id = s2.par\_id$, then $s2.bindenv$ is a version descendant of $s1.bindenv$.*

Proof: The proof is by induction on the length of sequences of derivations used to derive a fact. The basis case is the derivation sequence of length 0, i.e. base facts, for which the lemma is satisfied trivially.

We make some observations before considering the induction case. For rules with one base/derived body literal (Types 1,3,4,5), it is easy to show that the bindenv of the head fact is a version child of the bindenv of the derived fact used in the body. For rules with two base/derived body literals (Types 4 and 6), where Return_Unify fails, we can see from procedures Rename_and_Unify that the bindenv of the head fact is a child of the bindenv of the supplementary/query fact used in the rule body. If Return_Unify is called and succeeds, the bindenv of the head fact is a version child of the bindenv of the answer fact..

For the induction step, consider a derivation sequence of length $n+1$, and assume that the claims are true for all facts with derivation sequences of length n or less. Consider the last step in the derivation sequence.

If the head fact derived is a supplementary fact or an initial_query fact, it is given a new *cont_id*, that is not present in other existing fact. Thus, parts 1 and 2 of the lemma are trivially satisfied. For part 3, the *par_id* of the derived supplementary fact $s3$ is set to the *par_id* of the query/supplementary fact $s2$ that derived it. If Return_Unify does not succeed, $s3.bindenv$ is a version child of $s2.bindenv$ (as observed earlier). Part 3 then follows from Part 3 of the induction hypothesis. If Return_Unify does succeed, $s3.bindenv$ is a version child of the bindenv of the answer fact. Further, the *par_id* of the answer fact is the same as the *cont_id* of the supplementary fact. Part 3 then follows from Part 1 of the induction hypothesis.

If the head fact derived is a query fact, the rule body has only one derived literal (which is a supplementary or initial_query literal). The rule must be a Type 1, Type 4 or a Type 5 rule. Parts 2 and 3 are trivially satisfied in all the above cases. We consider Part 1 of the lemma. In the case of Type 1 and Type 4 rules, the *par_id* of the query fact is set to the *cont_id* of the supplementary / initial_query fact. Since the fact used in the derivation is the only supplementary / initial_query fact with this *cont_id*, Part 1 of the lemma is satisfied. In the case of Type 5 rules, the *par_id* of the query fact is set to the *par_id* of the supplementary fact. The bindenv of the query fact is a child version of the bindenv of the supplementary fact. Part 1 then follows from Part 3 of the induction hypothesis.

If the head fact derived is an answer fact, Parts 1 and 3 follow trivially. For Part 2, the answer fact is generated from a supplementary fact using a Type 3 rule, or from a query fact using a Type 6 rule. The *par_id* of the answer fact is then set to the *par_id* of the query/supplementary fact, and the bindenv of the answer fact is a version child of that of the query/supplementary fact. Part 2 of the induction hypothesis then follows.

This covers all the cases, and completes the proof. □

Lemma C.1.2 *Let s and a be supplementary and answer facts such that s.cont_id = a.par_id. Then there is a query fact q generated by a Type 4 rule (i.e. the query is on a non-tail-recursive literal) using s, such that:*

1. *a.bindenv is a version descendant of q.bindenv, and*
2. *for all variables in q.bindenv other than those accessible from q, the bindings in a.bindenv are the same as the bindings in s.bindenv.*

Proof: From Lemma C.1.1, *a.bindenv* is a version descendant of *s.bindenv*. Each supplementary fact generates a query fact or an answer fact. In the case of Type 3 and Type 5 rules, the *par_id* of the generated fact is different from the *cont_id* of the supplementary fact. No two supplementary/initial_query facts have the same *cont_id*. Also, the *cont_id* of the supplementary fact is not passed on to any other fact but this sole query fact. If we assume that no query fact q is generated from s using a Type 4 rule, it is easy to show that *a.par_id* cannot be the same as *s.cont_id*. Hence there is such a query fact q generated.

Next we now show that for any fact f and any q as above, such that $f.par\_id = s.cont\_id$,

1. $f.bindenv$ is a version descendant of $q.bindenv$, and
2. Any variable in $q.bindenv$ that is not accessible in q has the same bindings in $f.bindenv$ as in $q.bindenv$, and is not accessible in $\langle f.structure, f.bindenv \rangle$.

We first note the following. Suppose we unify facts $f1 = \langle s1, env1 \rangle$ and $f2 = \langle s2, env1 \rangle$. Then the only variables that are modified by the unification are those that are accessible from either $s1$ or $s2$.

The proof is by induction on lengths of derivation sequences used to derive f. We note again that no two supplementary/initial_query facts have the same *cont_id*. For the basis case, a fact with derivation sequence of length 1 that has the same *par_id* value s q must be derived using q. In all cases of rules that use q, the bindenv of the head fact is a version descendant of the bindenv of the body fact. Further, any variable that is not accessible from $q.structure$ is also not accessible from $f.structure$, and is not modified by unification during the derivation.

For the induction step, we note that in all cases except for Type 2 rules, the bindenv of the head fact is a version descendant of the bindenv of the sole derived body fact. If the *par_id* of the generated fact is *q.par_id*, then the *par_id* of the derived body fact must also be *q.par_id*. Part (1) then follows from induction hypothesis. Also, the sole derived body fact is not renamed, and any new variables that are created by renaming other facts do not conflict with variables in *q.bindenv*. Hence Part (2) follows.

For Type 2 rules, if the *par_id* of the generated fact is *q.par_id*, then the *par_id* of the supplementary fact in the body must also be *q.par_id*. If Return_Unify fails, the bindenv of the head fact is a version child of the bindenv of the supplementary fact in the body, and Part (1) follows. That Part (2) follows can be shown by arguments similar to those used in the earlier case.

If Return_Unify succeeds, $f.bindenv$ is a version child of $a1.bindenv$ where $a1$ is the answer fact used in the rule body. But since Return_Unify succeeds, $a1.par\_id = s1.cont\_id$, where $s1$ is the supplementary fact used in the rule body. But by induction hypothesis, $a1.bindenv$ is a descendant of $s1.bindenv$. It follows by induction hypothesis that $f.bindenv$ is a version descendant of $q.bindenv$; this establishes Part (1).

We now consider Part (2). By induction hypothesis, $s1$ generates a query fact $q1$ using a Type 4 rule, and for any variable in $q1.bindenv$ that is not accessible from $q1$ the bindings in $q1.bindenv$ and $a1.bindenv$ are the same; also any such variable is not accessible from $a1$. But it is easy to show that any variable in $s1.bindenv$ that is not accessible from $s1$ is also not accessible from $q1$. Hence for any variable in $s1.bindenv$ that is not accessible from $s1$, the bindings are the same in $a1.bindenv$ as in $s1.bindenv$; also, any such variable is not accessible from $a1$. But $s1.par\_id = q.par\_id$. Hence, by induction hypothesis, any variable in $q.bindenv$ is not accessible from $s1.bindenv$. Hence the bindings for any such variables are the same in $f1.bindenv$ as in $q.bindenv$, and further any such variables are not accessible from $f1.structure$.

This completes the induction step and the proof of this part of the lemma. This also concludes the proof of this lemma. □

Lemma 5.5.1 Suppose that there is a query fact

$$q = query(p_i(\overline{a_i}), id1, answer(id1, p_i(\overline{a_i})))$$

generated by a Type 4 rule (i.e., from a non-tail-recursive literal), and an answer fact $a = answer(id1, p_i(\overline{b_i}))$. Suppose also that $q.par\_id = a.par\_id$. Let q_str2 denote the last argument of $q.structure$. Then

$$\langle q\_str2, a.bindenv\rangle \equiv \langle a.structure, a.bindenv\rangle$$

Proof: Since a Type 4 rule is used to derive q, q must have been generated from a supplementary fact s_q such that $q.par\_id = s\_q.cont\_id$.

We show that any query fact $q1$ s.t. $q1.par\_id = q.par\_id$ has q_str2 as the last argument of its structure, and any supplementary $s1$ such that $s1.par\_id = q1.par\_id$ has q_str2 as its last argument of its structure. The proof is by induction on lengths of derivation sequences.

We note that no two supplementary facts or initial_query facts have the same value for *cont_id*, since a new identifier is generated for each such fact.

For the basis case, any fact generated by a derivation sequence of length 1 and that has the same *par_id* field is a fact s for some predicate $sup1_{j,0}$ (generated using a Type 1 rule). The query fact is not renamed, and variables in the rule head are dereferenced, hence the last argument of $s.structure$ is q_str2.

For the induction case, assume that the induction hypothesis is true for all derivation sequences of length less than some k, and consider the last step in a derivation sequence of length k. We have a case analysis based on the type of the rule.

For Type 0 rules, the *par_id* of the generated fact must be the *cont_id* of an initial_query fact, which must be distinct from *s_q.cont_id*. For Type 1 rules, the last argument of the structure of the generated supplementary fact is the same as the last argument of the structure of the body fact after dereferencing (as was argued for the basis case). The *par_id* fields of the head fact is the same as that of the body fact. The result then follows from the induction hypothesis.

For Type 2 rules, we note that the last argument of the head literal appears only in the supplementary literal. Whether Return_Unify succeeds or not, the supplementary fact is not renamed. Arguments similar to the earlier arguments then show that the last argument of the structure of the head fact is the same as the last argument of the structure of the body supplementary fact. The *par_id* field of the head fact is the same as the *par_id* field of the body supplementary fact. The result then follows from the induction hypothesis.

Type 3 rules generate answer facts. Consider any answer fact $a1$ that is generated. The structure of Type 3 rules shows us that the head fact is a dereferenced version of q_str2, interpreted in bindenv *a.bindenv*. Hence the lemma holds for the answer fact $a1$.

Type 4 and Type 5 rules generate query facts. In the case of Type 4 rules, the *par_id* of the query fact is generated from the *cont_id* of some supplementary fact (different from *s_q*). Hence the *par_id* of the fact cannot be the same as *q.par_id*. In the case of Type 5 rules, the last argument is the same as the last argument of the body supplementary fact (since it is not renamed). The *par_id* of the head fact is the same as that of the body fact, and the result follows from induction hypothesis.

Type 6 rules generate answers from queries on base predicates. The query fact is not renamed, and an answer fact is generated. This case is similar to Type 3 rules, and the lemma holds for any answer facts generated.

This completes the case analysis and the proof of the lemma. $\Box$

Lemma 5.5.2 Suppose that Return_Unify succeeds on rule R with facts s and a. Then $\langle R', r\_env' \rangle$ is an mgu of R' with (a renamed variant of)s and a.

Proof: Since Return_Unify succeeds, $s.cont\_id = a.par\_id$. By Lemma C.1.1, the answer fact bindenv is a descendant of the supplementary fact bindenv, and hence can only be more refined. Thus bindenv replacement instantiates the supplementary fact further. We unify this fact with the rule by binding some variables to arguments of the fact (this is possible since the supplementary literal has as arguments only distinct variables). Let ra be the structure of the answer literal in the rule body. Let q_str2 denote the last argument of the structure of q. Then there is a query fact q generated using a Type 4 rule corresponding to the answer literal, such that $\langle q\_str2, q.bindenv \rangle$ is equivalent to $\langle ra.structure, q.bindenv \rangle$. Now, *q.bindenv* is an ancestor of r_env'. Hence $\langle ra.structure, r\_env' \rangle$ and $\langle q\_str2, r\_env' \rangle$ are equivalent.

(The above argument assumes that all variable bindings are stored in the supplementary literal. If this is not true, we have to treat any variables that are not stored in the supplementary literal separately, and show that the above equivalence holds for the corresponding arguments of ra after the bindings of rule variables created by Return_Unify.)

Lemma 5.5.1 shows that $\langle q\_str2, r\_env' \rangle$ and the answer fact are equivalent. Hence $\langle ra, r\_env' \rangle$ and the answer fact are equivalent. Hence bindenv replacement results in a correct unification.

We then need only to show that the unifier is most general. Any unifier for the rule would make the answer literal and the answer fact equivalent. Since the unifier does not instantiate the answer fact, it is as general as possible for variables in the answer fact and for variables in the instantiated answer literal. But the variables in the instantiated answer literal are exactly the variables accessible from the query. Variables in the supplementary fact that are not accessible from the query are left unchanged by the unifier; hence the unifier is as general as possible for these variables too. For variables in the rule body, any unifier would bind them to corresponding structures in the supplementary and answer fact. Variables in the rule head but not in the body are left unchanged by the unifier. Hence the unifier is as general as possible for these variables too. Hence the result follows. □

C.2 Proofs from Section 5.6

Lemma 5.6.2 Suppose that an MGU MTTR rewritten program is evaluated using Opt-NG-SN evaluation without subsumption checking. Then every call to Return_Unify succeeds.

Proof: If subsumption checking is not performed, each call to goal_id returns a new value. Now, the value is stored in the ID field of the supplementary fact. No other supplementary fact has this value in its ID field. Examining procedure Update_Context_Ids, we see that the supplementary fact is also given a new value for its *cont_id* field. For each goal-id value g, let $C(g)$ be the value of the *cont_id* field.

We claim that (1) for each query and answer fact a, if gid is the goal-id for a, then $a.par\_id = C(gid)$, and (2) for supplementary facts s, if the value stored in the HId field is gid, then $s.par\_id = C(gid)$. The proof is by induction on lengths of derivation sequences. For each type of rule we compare the propagation of the goal-id values by the rules, and the propagation of the *par_id* value by Update_Context_Ids, and find that these values are propagated in the same manner. This completes the induction step of the proof of the claim.

An examination of Type 2 rules shows that if a supplementary fact and an answer fact unify with the rule, the goal-id field of the answer fact and the ID field of the supplementary fact must be the same. But the above claim then shows that the *par_id* field of the answer fact is equal to the *cont_id* field of the supplementary fact. Hence Return_Unify always succeeds. □

C.3 Proofs from Section 5.7

Theorem 5.7.1 Let P be a program, and Q a query. Given any database, let the cost of Prolog* evaluation of Q be t units of time. Opt-NGBU evaluation without subsumption-checking evaluates the query on the given database in time $O(t \cdot \mathcal{V})$. (The size of the program is not taken into account in this time complexity measure.)

Proof: The proof is based on the mapping of attempted derivations in bottom-up evaluation to actions of Prolog* evaluation presented in Lemma B.0.5. We show that for each attempted derivation of cost $c \cdot \mathcal{V}$, there is an action of Prolog* evaluation that costs at least c Since not more than a constant number of derivations are mapped to the same action, the theorem follows. We use the following case analysis to prove the theorem. The case analysis parallels the case analysis used in Lemma B.0.5.

Type 0: Rules Q_{R1} and Q_{R2} make one derivation each. Each derivation is mapped to a Prolog action that takes at least unit time. It is straightforward to show that derivations using these rules take $O(\mathcal{V})$ time.

Rule Q_{R3} generates answers to the query, and each derivation using this rule is mapped to a Prolog* action that generates answers to the query. Due to the optimization used in Procedure Rename_and_Unify_Facts, no renaming of facts is done for this rule. The unification step is straightforward since the arguments of the answer literal are distinct free variables, and no occur check is required. Hence this step takes $O(\mathcal{V})$ time per answer to the initial query, while the corresponding Prolog* action takes at least $O(1)$ time.

Type 1 Rules - 1 :

These are Type 1 rules whose body uses a predicate *query*. Derivations using this kind of rules involve the unification of a query fact $query(h(\overline{a}), hid, ans)$ with a query literal $query(h(\overline{t}), HId, A)$. This derivation is mapped to the unification of $?h(\overline{a})$ with $h(\overline{t})$ by Prolog*.

Since A and HId appear nowhere else, they can be unified with the corresponding arguments in $O(\mathcal{V})$ time. Due to our assumption that the time taken for unification is independent of the exact structure of the terms, the unification of $h(\overline{t})$ with $h(\overline{a})$ costs the same, ignoring versioning costs, as the unification done by Prolog*. Factoring in the versioning overheads, we get an overhead of a factor of $O(\mathcal{V})$.

Type 2 Rules - 1 : These are Type 2 rules whose body uses a predicate $sup_{j,i}$. There are three subclasses of rules of this type.

The first subclass is of rules that use an *answer* literal in the body. Lemma 5.6.2 shows that Return_Unify succeeds whenever it is called for such rules. The time taken for successful derivations using such rules is $O(\mathcal{V})$. Any unsuccessful derivation using such a rule must use a supplementary fact for which there is no answer fact; any answer fact will have a supplementary fact with the same identifier value, and the derivation would be successful. The supplementary literal has as arguments distinct variables, and unification is straightforward. Hence the time taken for unsuccessful derivations using such rules is $O(\mathcal{V})$.

The second subclass is of rules have an equality literal in the body. Any attempted derivation using such a rule first unifies a $sup_{j,i}$ fact with the rule body (the unification always succeeds), and attempts to perform the unification needed to evaluate the equality literal. This attempted derivation is mapped to an equivalent unification action by Prolog* evaluation. Whether the derivation succeeds or not, it takes time at most $O(\mathcal{V})$ times that taken by Prolog* evaluation.

The third subclass is of rules that have a base literal in the body. An attempted derivation uses a fact for $sup_{j,i}$, performs an indexing operation on the base literal, and derives a head fact for each fetched base fact. For such rules, each attempted derivation is mapped to an action of Prolog* evaluation that indexes the base relation with the same bindings. We assume the same indexing technique is used in either case. We count the cost of fetching facts, and renaming and unifying the facts with the query on the base relation as part of the indexing cost. Bottom-up evaluation and Prolog* evaluation perform the same indexing operations, and hence the cost of the indexing operation is the essentially the same (modulo the $O(\mathcal{V})$ factor for accessing and binding variables, and creating bindenv versions for each fetched fact).

If no facts are fetched by the indexing operation in bottom-up evaluation, the cost of rule application (apart from the indexing cost) is $O(\mathcal{V})$, and we map this cost to the cost of the indexing operation. Since that cost is at least $O(\mathcal{V})$, and only one attempted derivation is mapped to each indexing operation, there is no change in the time complexity of the indexing operation, and we ignore the cost.

If facts are successfully fetched by the indexing operation, for each fact fetched, a successful derivation is made by bottom-up evaluation. The derivation is mapped to the return of an answer to the query on the base literal by Prolog* evaluation. The cost of the derivation is $O(\mathcal{V})$, since the cost of renaming and unifying the base fact has been counted with the cost of the indexing operation. The corresponding Prolog* action takes at least unit time, and hence the cost of the derivation is at most $O(\mathcal{V})$ times the cost of the Prolog* action.

Type 1 and Type 2 Rules - 2 :

Such rules derive a fact $sup_{j,i}$ using a fact $sup1_{j,i}$, and generate goal-id values through a call to goal_id. The unifications of the supplementary literal and supplementary fact can be done in $O(\mathcal{V})$ time since there are no repeated variables. The evaluation of goal_id takes constant time without subsumption checking. Overall, a successful derivation using a rule of this type takes $O(\mathcal{V})$ time. All attempted derivations using rules of this type are successful.

Type 3, Type 4 and Type 5 Rules :

Such rules have only one body literal, which has as arguments distinct variables. Hence a successful derivation using a rule of this type takes time $O(\mathcal{V})$. All attempted derivations using such rules are successful.

This completes the case analysis of all the rule types. □

Appendix D

Proofs From Chapter 6

Theorem 6.4.1 The aggregate selections generated by Techniques C1, BS1, BS2, BS3, and LS1 are sound aggregate selections.

Proof: The soundness of Techniques C1 and BS2 is very straightforward. For Technique BS1 we note the following. Since $p(\overline{t_1}) : s$ is a sound aggregate constraint on p, every fact in p must satisfy the constraint. Since all free variables in s are present in $\overline{t_1}$, σ is also a renaming of free variables in s (the bound variables do not matter since they are quantified within the atomic aggregate selections in s). Since $p(\bar{t}) = p(\overline{t_1})[\sigma]$, $p(\bar{t}) : s[\sigma]$ is equivalent to $p(\overline{t_1}) : s$. Hence for every successful rule instantiations, the variables must satisfy the constraint $s[\sigma]$. It follows (trivially) that every relevant rule instantiation satisfies the selection $s[\sigma]$.

Now consider Technique BS3. Every relevant fact for p satisfies $p(\bar{t}) : s$. Unifying this with the head of R, every relevant instantiation of R satisfies $s[\sigma][\theta]$, since $p(\bar{t})[\sigma][\theta]$ is equivalent to $p(\overline{t_1})$. Since θ does not affect variables in $s[\sigma]$, $s[\sigma] = s[\sigma][\theta]$. Hence, every relevant instantiation of rule R satisfies the selection $s[\sigma]$.

The proof of correctness of LS1 is straightforward — a restriction of a selection is weaker than the original selection, and every relevant rule instantiation satisfies the restriction. Since all free variables in the specified restriction occur in the literal, the restricted selection can be tested for the literal. If a fact is found irrelevant by the selection, clearly any rule instantiation using the fact in this literal will also be irrelevant, since it provides the same bindings for the free variables of the selection. □

Theorem 6.4.2 Technique PS1 is sound.

Proof: Consider any instantiation of the variables in $\overline{X}$. There are now several instantiations of the variables in the rule, that satisfy the rule body, and this defines a multiset S_Y of values for Y. Now further partition S_Y based on values of variables other than $\overline{X} \cup \{Y, W1, W2, \ldots, Wn\}$. Consider any partition S'_Y (in other words, consider the multiset of instantiations of Y, with all variables in the rule other than { Y, W1, W2, …, Wn } fixed). Since agg_f is an IncSel function, for any value y, if $y \in unnecessary_{agg\_f}(S'_Y)$ then $y \in unnecessary_{agg\_f}(S_Y)$.

Now each partition S'_Y defines a value for the variables in $\overline{t_i}$. Consider each literal $p_i(\overline{t_i}, Wi)$ Given a value for the variables in $\overline{t_i}$, the set of facts for p_i defines a set S_{Wi} of values for Wi. Also, each partition defines a set S_W of instantiations of the tuple $(W1, W2, \ldots, Wn)$. No two Wis appear in any literal other than $Y = fn(W1, \ldots, Wn)$. Given any tuple of values from $S_{W1} \times \ldots \times S_{Wn}$, there is a value for Y such

$Y = fn(W1, \ldots, Wn)$ is satisfied.

If the other literals in the rule are satisfied, within each partition S_W is equal to the cross product of the sets S_{Wi}. Otherwise S_W is empty, and no fact is generated using this instantiation for the variables other than { Y, W1, W2, ..., Wn }.

S'_Y is the set of Y values obtained by applying fn to the tuples in S_W. Since $unnecessary_{agg\_f}$ distributes over fn, if a value $wi \in unnecessary_{agg\_f}(S_{Wi})$, any Y value derived from it must be in $unnecessary_{agg\_f}(S'_Y)$ (this is trivially true if S_W is empty, since no Y value is derived). Hence it must also be in $unnecessary_{agg\_f}(S)$. Since we chose any instantiation of the variables in $\overline{X}$, this must be true for all instantiations of $\overline{X}$. Hence such a Wi value cannot not generate any relevant head fact. Therefore the aggregate selection generated for the literal is sound. □

Theorem 6.4.3 Technique PS2 is sound.

Proof: The proof essentially follows the proof of Theorem 6.4.2, with the set of partitioning variables different. Suppose we are given a binding for the variables $\overline{X} \cup \mathcal{V}$, where $\mathcal{V}$ is a cross-partitioning set of variables. With this binding, for each literal $p_i(\overline{t_i}, Wi)$, the set of facts for p_i define a set S_{Wi} of values for Wi. Also, this binding defines a set S_W of instantiations of the tuple $(W1, W2, \ldots, Wn)$ produced by successful instantiations of the rule. By the definition of cross-partitioning variables, either the cross products of the S_{Wi}'s is equal to S_W, or S_W is empty.

The partitioning arguments of a literal $p_i(\overline{t_i}, Wi)$ form a superset of the arguments that use variables $\overline{X} \cup \mathcal{V}$. Given a binding for $\overline{X} \cup \mathcal{V}$, we can extend the binding to get values for all variables in partitioning arguments of $p_i(\overline{t_i}, Wi)$. Let the multiset of instantiations of Wi defined by the given instantiation of the partitioning variables be M_{Wi}. Now, $M_{Wi} \subseteq S_{Wi}$, since (a) the arguments of $p_i(\overline{t_i}, Wi)$ that have cross-partitioning variables are defined to be partitioning arguments, and (b) the conditions of PS2 ensure that non-partitioning arguments of $p_i(\overline{t_i}, Wi)$ are distinct variables and will not constrain the set of successful instantiations of $p_i(\overline{t_i}, Wi)$.

Since agg_f is an IncSel function, $unnecessary_{agg\_f}$ is monotone, and any value found unnecessary for M_{Wi} will also be unnecessary for S_{Wi}, and any instantiation of $(W1, W2, \ldots, Wn)$ using such a value for Wi will result in an unnecessary value being generated for Y. Hence the aggregate selection generated is sound. □

Proposition 6.4.4 Consider a rule R and an aggregate selection s as in Technique PS1. Let $\mathcal{V}$ denote the set of all variables in the rule. Let $\mathcal{N}$ denote the set of non-constrained variables in the rule. Then $\mathcal{C} = \mathcal{V} - \mathcal{N} - \{W1, W2, \ldots, Wn, Y\}$ is a cross-partitioning set for rule R.

Proof: Consider any instantiation of the variables in $\overline{X} \cup \mathcal{C}$. For each p_i, let $\mathcal{N}_i$ denote the set of variables in $\mathcal{N}$ that appear in $p_i(\overline{t_i}, Wi)$. Let $\mathcal{M}_i = \mathcal{N}_i \cup \{Wi\}$. Let S_{Mi} (resp. S_{Wi}) denote the set of instantiations of variables in $\mathcal{M}_i$ (resp. Wi) generated by literal $p_i(\overline{t_i}, Wi)$, (with the given instantiation of $\overline{X} \cup \mathcal{C}$). Let S_M (resp. S) denote the set of instantiations of $\mathcal{N} \cup \{W1, \ldots, Wn\}$ (resp. $\{W1, \ldots, Wn\}$) generated by successful instantiations of the rule (with the given instantiation of $\overline{X} \cup \mathcal{C}$).

With the given instantiation of $\overline{X} \cup \mathcal{C}$, the body of the rule is either not satisfiable (in which case S is empty), or the literals other than the $p_i(\overline{t_i}, Wi)$ literals and $Y = fn(W1, \ldots, Wn)$ are satisfied (none of the variables in $\mathcal{N} \cup \{W1, W2, \ldots, Wn, Y\}$ appear in these literals). The case where S is empty is trivial. We consider the other case.

Each element in the cross product of the S_{Mi}'s defines an instantiation of the rule variables. But each such variable instantiation defines a successful instantiation of the rule: this is because the variables in $\mathcal{N}_i$ appear nowhere else in the rule body, and Wi appears only in $Y = fn(W1, \ldots, Wn)$, which has a successful instantiation for every value of $W1, \ldots, Wn$. Hence S_M is equal to $S_{M1} \times \ldots \times S_{Mn}$.

Now, for each value in S_{Wi} there is an element in S_{Mi} with the same value for Wi. Hence for each value in the cross product of the S_{Wi}s, there is an element in S_M with the same values for $(W1, \ldots, Wn)$.

But S is equal to the projection of S_M on to $(W1, W2, \ldots, Wn)$; similarly S_{Wi} is equal to the projection of S_{Mi} on to Wi. Hence $S = S_{W1} \times S_{W2} \times \ldots \times S_{Wn}$. $\Box$

Proposition 6.4.6 The conditions in Compare_Aggregate_Selections(s, t) are sufficient conditions for s to be stronger than t.

Proof: For case 1a, with more variables in the second argument of the groupby the multiset of values obtained for each group is smaller, and since $unnecessary_{agg\_f}$ is monotonic, the set of values detected to be unnecessary is smaller. Hence the set of facts detected to be unnecessary is also smaller for a weaker atomic aggregate selection. For case 1b, every value that unifies with $c2(\ldots)$ also unifies with $c1(\ldots)$, Hence any fact that is classified as irrelevant by t, also unifies with $c1(\ldots)$, and the multiset of values in its group in s is at least as large as the multiset for its group in t. Hence the result follows. It is easy to see that the test in case 2 is correct. $\Box$

Theorem 6.5.1 (Correctness of Rewriting) Let P be any program, and P^{as} the aggregate rewritten version of the program.

1. P^{as} and P are equivalent in the set of answers they generate for the query predicate.
2. The aggregate selection on each predicate in P^{as} is a sound aggregate selection on the predicate.

Proof: We first consider Part 1 of this theorem. Note that rules are not modified in the rewriting except to replace predicates by new versions of the predicate.

We first show that the answer set of P^{as} is covered by the answer set of P. We claim that for each fact $p\_s(\overline{a})$ derived in P^{as}, evaluation of P generates $p(\overline{a})$. Suppose not. Consider the shortest sequence of derivations in P^{as} that derives a fact for which this is not true, and consider the last derivation in this sequence. For each fact $p_i\_s_i(\overline{a}))$ used in this sequence, $p_i(\overline{a})$ (the version with the suffix $\_si$ dropped) is generated in P. If we drop the suffixes from the literals in the rule, we get a rule in P. Hence it follows that the corresponding head fact is generated in P, which contradicts the assumption.

The proof in the reverse direction is similar. We claim that for each fact $p(\overline{a})$ derived in P if p is reachable from the query predicate, then for each version p_s of p in P^{as}, a fact $p\_s(\overline{a})$ is generated in P^{as}. All rules reachable from the query are processed by the algorithm, since the rewriting algorithm performs a DFS of the reachability graph for the program (i.e., the graph with predicates as nodes, and an edge from a to b if a is used to define b). Hence there are rules in P^{as} for all predicates reachable from the query predicate. The rest of the argument then parallels the argument above, and is omitted for brevity.

Now consider Part 2 of the theorem. Consider a predicate p with an aggregate selection s on it. The aggregate selections deduced on the literals of rules defining q are sound, since the techniques for generating aggregate selections on literals are sound (Theorems 6.4.1,6.4.2, 6.4.3). Step 13 of the rewriting algorithm takes a literal $q(\ldots)$ with an aggregate selection $s1$ on it, and replaces the predicate q by q_s1 with aggregate selection $s1$ on q_s1. Hence all uses of q_s1 have the aggregate selection on them. Hence $s1$ is a sound aggregate selection on q_s1.

Preprocessing 1 replaces a predicate in a literal by another with a weaker aggregate selection. This will not result in any loss of derivations since every fact is relevant to the literal satisfies the stronger aggregate selection, and hence the weaker aggregate selection too. The reachability analysis and dropping of unreachable predicates does not affect the set of answers to the query.

Now consider Postprocessing 2. A version of p with an aggregate selection on it may have a subset of the facts in p if we discard facts that fail the selection. Due to monotonicity of the functions $unnecessary_{agg\_f}$, any value that is found unnecessary w.r.t. the subset would also be unnecessary w.r.t. the full set. Hence while the new selection may not be as strong as the original one, the renaming is guaranteed to be sound. □

Theorem 6.5.2 (**Termination**) Algorithm Push_Selections terminates on all finite input programs, producing a finite rewritten program.

Proof: The number of non-equivalent atomic aggregate selections that can be generated by the deduction rules we use is finite, for the following reason. Techniques C1 and BS2 generate only one aggregate constraint/selection per rule. Techniques PS1 and PS2 can generate only a finite number of atomic selections per literal, since they essentially choose a subset of arguments to group by, and an argument to apply the aggregate selection to. Techniques BS1 generates aggregate selections from aggregate constraints. Since the number of aggregate constraints is fixed, it generates only a finite number of atomic aggregate selections.

This leaves techniques BS3 and LS1. These generate no new groupby lists, except by renaming existing groupby lists. They generate atomic aggregate selections by applying these groupby lists to rule bodies and literals. Since the number of rule bodies and literals is fixed, these techniques generate only a finite number of atomic aggregate selections.

Given a finite number of atomic aggregate selections, the number of non-equivalent aggregate selections (formed by conjunctions of atomic aggregate selections) is also finite.

Hence after some point, the deduction rules can generate no new aggregate selection, the stack of predicate versions becomes empty, and the rewriting algorithm terminates. □

Theorem 6.6.1 (**Soundness, Completeness, Non-Repetition**) Aggregate Retaining evaluation of P^{as} gives the same set of answers for *query_pred* as Semi-Naive evaluation of P, and does not repeat any inferences. Further, the Aggregate Retaining evaluation of P^{as} terminates whenever the Semi-Naive evaluation of P terminates.

Proof: An aggregate selection on a predicate can be fully tested only at the end of the evaluation (after all facts have been computed). However the incremental nature of aggregate selections allows us deduce that some facts are irrelevant even during the course of the computation. If a fact for a predicate does not satisfy a sound aggregate selection on the predicate, it is guaranteed to be irrelevant to the query predicate — any derivation that can be made using it is guaranteed to be irrelevant. Hence the answers to the query are not affected if the fact is not used.

The only real concern is termination. Agg-retaining evaluation discards facts only when its discarding will not affect the unnecessary set for any atomic aggregate selection. Hence, if a fact is found to be irrelevant, it will continue to be found irrelevant for the rest of the evaluation. If such a fact is generated again, it will not be re-used. It follows from well-known soundness, completeness and non-repetition results on semi-naive evaluation (see eg. [MR89, RSS90]), that Agg-retaining evaluation does not repeat any inferences.

Now we consider the last part of the theorem. Agg-retaining evaluation of P^{as} makes no more inferences than semi-naive evaluation of P^{as}, and since it does not repeat inferences, it terminates whenever the semi-naive evaluation of P^{as} does. But semi-naive evaluation of P^{as} terminates whenever semi-naive evaluation of P does, and the last part of the theorem follows. $\square$

Bibliography

[ABW88] K. R. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan-Kaufmann, San Mateo, Calif., 1988.

[ADJ88] R. Agrawal, S. Dar, and H. V. Jagadish. On transitive closure problems involving path computations. Technical Memorandum, 1988.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[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.

[Bay85] R. Bayer. Query evaluation and recursion in deductive database systems. Unpublished Memorandum, 1985.

[BMSU86] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 1–15, Cambridge, Massachusetts, March 1986.

[BNR$^+$87] Catriel Beeri, Shamim Naqvi, Raghu Ramakrishnan, Oded Shmueli, and Shalom Tsur. Sets and negation in a logic database language. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 21–37, San Diego, California, March 1987.

[BNST91] Catriel Beeri, Shamim Naqvi, Oded Shmueli, and Shalom Tsur. Set constructors in a logic database language. *The Journal of Logic Programming*, pages 181–232, 1991.

[BR86] Francois Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 16–52, Washington, D.C., May 1986.

[BR87a] I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *Journal of Logic Programming*, 4(3), September 1987.

[BR87b] Catriel Beeri and Raghu Ramakrishnan. On the power of Magic. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 269–283, San Diego, California, March 1987.

[BRSS92] C. Beeri, R. Ramakrishnan, D. Srivastava, and S. Sudarshan. The valid model semantics for logic programs. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 91–104, June 1992.

[BRSS89] C. Beeri, R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Magic implementation of stratified programs. Manuscript, September 89.

[Bry90] Francois Bry. Query evaluation in recursive databases: Bottom-up and top-down reconciled. *IEEE Transactions on Knowledge and Data Engineering*, 5:289–312, 1990.

[CD85] J.H. Chang and A. M. Despain. Semi-intelligent backtracking of Prolog based on static data-dependency analysis. In *Proc. Symposium on Logic Programming*, pages 10–21, 1985.

[CDY90] Michael Codish, Dennis Dams, and Eyal Yardeni. Bottom-up abstract interpretation of logic programs. Technical Report CS90-24, Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Israel, 1990. A preliminary version appeared in the pre-ICLP90 Workshop on Abstract Interpretation, Eilat, June 1990.

[CGK89] D. Chimenti, R. Gamboa, and R. Krishnamurthy. Abstract machine for LDL. Technical Report ACT-ST-268-89, MCC, Austin, TX, 1989.

[CH85] Ashok K. Chandra and David Harel. Horn clause queries and generalizations. *J. Logic Programming*, 2(1):1–15, April 1985.

[CKW89] Weidong Chen, Michael Kifer, and Davis S. Warren. Hilog: A first-order semantics for higher-order logic programming constructs. In *Proceedings of the North American Conference on Logic Programming*, pages 1090–1114, 1989.

[CN89] I. F. Cruz and T. S. Norvell. Aggregative closure: An extension of transitive closure. In *Proc. IEEE 5th Int'l Conf. Data Engineering*, pages 384–389, 1989.

[Die87] Suzanne W. Dietrich. Extension tables: Memo relations in logic programming. In *Proceedings of the Symposium on Logic Programming*, pages 264–272, 1987.

[Die89] Paul F. Dietz. Fully persistent arrays. In *Workshop on Algorithms and Data Structures*, pages 67–74, 1989. (Appeared as LNCS 382).

[DSST86] James R. Driscoll, Neil Sarnak, Daniel Sleator, and Robert E. Tarjan. Making data structures persistent. In *Eighteenth Annual ACM Symposium on Theory of Computing*, 1986.

[DST90] James R. Driscoll, Daniel Sleator, and Robert E. Tarjan. Fully persistent lists with catenation. In *Symposium on Data Structures And Algorithms*, 1990.

[Ede90] Johann Eder. Extending SQL with general transitive closure and extreme value selections. *IEEE Trans. on Knowledge and Data Engineering*, 2(4):381–390, 1990.

[GGZ91] Sumit Ganguly, Sergio Greco, and Carlo Zaniolo. Minimum and maximum predicates in logic programming. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1991.

[GJ90] Gopal Gupta and Bharat Jayaraman. On criteria for Or-Parallel execution models of logic programs. In *Proceedings of the North American Conference on Logic Programming*, pages 737–756, 1990.

[Got74] E. Goto. Monocopy and associative algorithms in an extended lisp. Technical Report 74-03, Information Science Laboratory, Univ. of Tokyo, Tokyo, Japan, May 1974.

[GPSZ91] Fosca Giannotti, Dino Pedreschi, Domenico Sacca, and Carlo Zaniolo. Non-determinism in deductive databases. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Proceedings of the Second International Conference on Deductive and Object-Oriented Databases DOOD'91*, Munich, Germany, 1991. Springer-Verlag.

[GS91] P.A. Gardner and J.C. Shepherdson. Unfold/Fold transformations of logic programs. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic*. The MIT Press, 1991.

[Knu77] Donald E. Knuth. A generalization of Dijkstra's algorithm. *Information Processing Letters*, 6(1):1–5, February 1977.

[KS91] David Kemp and Peter Stuckey. Semantics of logic programs with aggregates. In *Proceedings of the International Logic Programming Symposium*, pages 387–401, San Diego, CA, U.S.A., October 1991.

[KSS91] David Kemp, Divesh Srivastava, and Peter Stuckey. Magic sets and bottom-up evaluation of well-founded models. In *Proceedings of the International Logic Programming Symposium*, pages 337–351, San Diego, CA, U.S.A., October 1991.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

[MFPR90a] I. S. Mumick, S. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic is relevant. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Atlantic City, New Jersey, May 1990.

[MFPR90b] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. Magic conditions. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 314–330, Nashville, Tennessee, April 1990.

[MPR90] Inderpal S. Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. Duplicates and aggregates in deductive databases. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, August 1990.

[MR89] Michael J. Maher and Raghu Ramakrishnan. Dèjá vu in fixpoints of logic programs. In *Proceedings of the Symposium on Logic Programming*, Cleveland, Ohio, 1989.

[MW88] David Maier and David S. Warren. *Computing With Logic.* The Benjamin Cummings Publishing Company Inc., 1988.

[NR91] Jeffrey F. Naughton and Raghu Ramakrishnan. Bottom-up evaluation of logic programs. In J-L. Lassez, editor, *Computational Logic: Essays in Honor of Alan Robinson.* The MIT Press, 1991.

[NRSU89] Jeffrey F. Naughton, Raghu Ramakrishnan, Yehoshua Sagiv, and Jeffrey D. Ullman. Argument reduction through factoring. In *Proceedings of the Fifteenth International Conference on Very Large Databases*, pages 173–182, Amsterdam, The Netherlands, August 1989.

[NT89] Shamim Naqvi and Shalom Tsur. *A Logical Language for Data and Knowledge Bases.* Principles of Computer Science. Computer Science Press, New York, 1989.

[O'K90] Richard A. O'Keefe. *The Craft of Prolog.* The MIT Press, 1990.

[Per85] Fernando Pereira. A structure-sharing representation for unification-based grammar formalisms. In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics*, pages 137–143, 1985.

[PW83] F.C.N. Pereira and D.H.D. Warren. Parsing as deduction. In *Proceedings of the twenty-first Annual Meeting of the Association for Computational Linguistics*, 1983.

[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.

[Ram90] Raghu Ramakrishnan. Parallelism in logic programs. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, San Francisco, California, 1990.

[RHDM86] A. Rosenthal, S. Heiler, U. Dayal, and F. Manola. Traversal recursion: A practical approach to supporting recursive applications. In *Proceedings of the ACM SIGMOD Conf. on Management of Data*, pages 166–176, 1986.

[RLK86] J. Rohmer, R. Lescoeur, and J. M. Kerisit. The Alexander method — a technique for the processing of recursive axioms in deductive database queries. *New Generation Computing*, 4:522–528, 1986.

[Ros90] Kenneth Ross. Modular Stratification and Magic Sets for DATALOG programs with negation. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 161–171, 1990.

[Ros91] Kenneth Ross. Modular acyclicity and tail recursion in logic programs. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1991.

[RS91] Raghu Ramakrishnan and S. Sudarshan. Top-Down vs. Bottom-Up Revisited. In *Proceedings of the International Logic Programming Symposium*, 1991.

[RS92] Kenneth Ross and Yehoshua Sagiv. Monotonic aggregation in deductive databases. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 114–126, 1992.

[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.

[RSS91] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. Rule ordering in bottom-up fixpoint evaluation of logic programs. Technical Report TR 1059, Computer Sciences Department, University of Wisconsin, Madison WI 53706, U.S.A., December 1991.

[RSS92a] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. Controlling the search in bottom-up evaluation. In *Joint Int'l Conf. and Symp. on Logic Programming 1992 (to appear)*, 1992.

[RSS92b] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. CORAL: Control, Relations and Logic. In *Proceedings of the International Conference on Very Large Databases*, 1992.

[RSS92c] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. Efficient bottom-up evaluation of logic programs. In J. Vandewalle, editor, *The State of the Art in Computer Systems and Software Engineering*. Kluwer Academic Publishers, 1992.

[Sek89] H. Seki. On the power of Alexander templates. In *Proc. of the ACM Symposium on Principles of Database Systems*, pages 150–159, 1989.

[SG76] M. Sassa and E. Goto. A hashing method for fast set operations. *Information Processing Letters*, 5(4):31–34, June 1976.

[SKGB87] H. Schmidt, W. Kiessling, U. Güntzer, and R. Bayer. Compiling exploratory and goal-directed deduction into sloppy delta iteration. In *IEEE International Symposium on Logic Programming*, pages 234–243, 1987.

[SR91] S. Sudarshan and Raghu Ramakrishnan. Aggregation and relevance in deductive databases. In *Proceedings of the Seventeenth International Conference on Very Large Databases*, September 1991.

[SR92a] S. Sudarshan and Raghu Ramakrishnan. Aggregation and relevance in deductive databases. In preparation (full version of [SR91]), 1992.

[SR92b] S. Sudarshan and Raghu Ramakrishnan. Top-Down vs. Bottom-Up Revisited. Manuscript in preparation (full version of [RS91]), 1992.

[SS82] J. Sebelik and P. Stepanek. Horn clause programs for recursive functions. In K. Clark and S.-A. Tarnlund, editors, *Logic Programming.* Academic Press, 1982.

[SS88] Seppo Sippu and Eljas Soisalon-Soinen. An optimization strategy for recursive queries in logic databases. In *Proceedings of the Fourth International Conference on Data Engineering*, Los Angeles, California, 1988.

[TS84] Hisao Tamaki and Taisuke Sato. Unfold/fold transformations of logic programs. In *Proceedings of the Second International Conference on Logic Programming*, pages 127–138, Uppsala, Sweden, July 1984.

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

[Ull89a] Jeffrey D. Ullman. Bottom-up beats top-down for Datalog. In *Proceedings of the Eighth ACM Symposium on Principles of Database Systems*, pages 140–149, Philadelphia, Pennsylvania, March 1989.

[Ull89b] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, 1989.

[Van92] A. Van Gelder. The well-founded semantics of aggregation. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 127–138, 1992.

[vEK76] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, October 1976.

[Vie86] Laurent Vieille. Recursive axioms in deductive databases: The query-subquery approach. In *Proceedings of the First International Conference on Expert Database Systems*, pages 179–193, Charleston, South Carolina, 1986.

[Vie87] Laurent Vieille. Database complete proof procedures based on SLD-resolution. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 74–103, 1987.

[Vie88] Laurent Vieille. From QSQ towards QoSaQ: Global optimizations of recursive queries. In *Proc. 2nd International Conference on Expert Database Systems*, April 1988.

[War83] David H. D. Warren. Logarithmic access arrays for prolog. Unpublished program, 1983.

[War89] David S. Warren. The XWAM: A machine that integrates Prolog and deductive database query evaluation. Technical Report Tec. Rep. 89/25, Department of Computer Science, SUNY at Stony Brook, October 1989.

[War92] David S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3), March 1992.