Representing and Querying Complex Information in the Coral Deductive Database System

Divesh Srivastava

Technical Report #1205

December 1993

REPRESENTING AND QUERYING COMPLEX INFORMATION IN THE CORAL DEDUCTIVE DATABASE SYSTEM

by

Divesh Srivastava

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

Doctor of Philosophy
(Computer Sciences)

at the
UNIVERSITY OF WISCONSIN – MADISON
1993

Abstract

The inadequacy of relational databases for new classes of applications has led to considerable research directed at enhancing the modeling capability of the database and the expressive power of the query language. The development of deductive databases was aimed at providing a declarative, potentially complete query language based on Datalog. However, Datalog lacks features such as aggregation and negation, does not support the manipulation of numeric values, and inherits the relational data model with its limited modeling power. Our goal in this thesis is to resolve these limitations of Datalog and to demonstrate that powerful and practical database query languages based on Datalog can be designed and efficiently implemented.

We present a Magic-sets based bottom-up evaluation technique, Ordered Search, that can be used to efficiently evaluate programs with left-to-right modularly stratified aggregation and negation; this class of programs includes useful applications such as bill-of-materials. We provide theoretical results to demonstrate that our technique is more efficient than previous bottom-up evaluation techniques. We also give performance results from the implementation of Ordered Search in the Coral deductive database system showing the practicality of this evaluation technique.

We propose a program transformation technique, Constraint-rewrite, which propagates arithmetic constraints, such as $Cost \leq 100$, specified in a program. By considering semantic manipulation of constraints, our techniques significantly extend earlier work. The Constraint-rewrite transformation can be combined with the Magic-sets transformation to efficiently propagate both constant binding and constraint binding information. We also develop a uniform framework that integrates our results and the earlier related work in the literature.

We design a deductive, object-oriented language, Coral++, that combines the data modeling features of C++ and the querying capability of Coral—two existing languages—with minimal changes to either, and yields a powerful combination of the object-oriented and deductive paradigms. We describe our implementation strategy for Coral++, which effectively uses the existing Coral run-time system and the C++ compiler to support the object-oriented features of the Coral++ data model and query language.

Acknowledgements

I have enjoyed working with Prof. Raghu Ramakrishnan during my graduate studies at the University of Wisconsin–Madison. Raghu is an excellent Ph.D. advisor, always encouraging me to pursue any research idea that caught my fancy. At the same time, he tried to make sure that both my feet were firmly planted on the ground by prodding me to implement my ideas to test their practicality. Despite having a very busy schedule, Raghu always had time for discussions whenever I knocked on his door and I thank him for that.

Working with the CORAL team—Raghu Ramakrishnan, S. Sudarshan, Praveen Seshadri and Per Bothner—has been an enlightening experience, and I have come to believe that a small group of researchers can design and successfully implement a fairly large software project. I have also learned a lot from my collaborators in research: Catriel Beeri, David Kemp, Jeffrey Naughton, Raghu Ramakrishnan, Praveen Seshadri, Peter Stuckey, S. Sudarshan, among others. During my years as a graduate student, I collaborated extensively with Sudarshan in doing research and writing papers, and we have had many interesting debates and heated arguments over the merits of competing ideas; it has always been a lot of fun. I have had many interesting discussions with Shaul Dar during my last year at Madison; he also helped me considerably in preparing for my final oral defense and proof-reading portions of my thesis.

Several other professors have played important roles in broadening my research horizons. Yannis Ioannidis, Kenneth Kunen and Jeffrey Naughton were always willing to listen to me and give valuable feedback on my research ideas in databases and logic programming, Olvi Mangasarian explained many concepts of linear programming in terminology that I could comprehend, Anne Condon and Deborah Joseph got me interested in theoretical aspects of computer science, and Michael Byrd, Arda Denkel, Ellery Eells, Malcolm Forster, LaVerne Shelton, Elliott Sober and Leora Weitzman inculcated in me a lasting interest in philosophy.

I would like to thank Charles Fischer, Kenneth Kunen, Olvi Mangasarian, Jeffrey Naughton, Raghu Ramakrishnan, and Marvin Solomon for sitting in on my Ph.D. preliminary and/or final committees. Marvin Solomon, in particular, made many excellent suggestions to ensure that my thesis could be read and understood by researchers other than myself.

Several research grants supported me financially during my years as a graduate student, and I would like to gratefully acknowledge the following grants: a David and Lucile Packard Foundation Fellowship in Science and Engineering, a Presidential Young Investigator Award with matching grants from Digital Equipment Corporation, Tandem and Xerox, NSF grants DCR-8402565, IRI-8804319 and IRI-9011563, and an IBM Faculty Development Award.

Contents

Chapter 1

Introduction

1.1 Motivation

Current databases typically represent data as flat relations. Queries on such databases are expressed in languages such as SQL. However, flat relations are often insufficient to naturally model real-world information, and SQL has a limited expressive power. For instance, in an airline application, relational databases cannot naturally represent a sequence of flights between two cities, and one cannot express the query "find the cheapest sequence of flights between two given cities" in SQL. Similarly, in an engineering/CAD application, relational databases cannot naturally model a part-subpart hierarchy, and SQL cannot be used to express the bill-of-materials query "find the quantity of each basic part required to assemble a given complex part." In recent years considerable research has aimed at providing sophisticated features for the natural modeling of real-world information, and enhancing the expressive capability of the query language. The major directions this research has taken have been deductive databases and object-oriented databases.

Deductive databases allow information to be explicitly represented in non-first normal form relations, and provide a facility for generalized recursive queries based on logic programming formalisms. Deductive query languages, like relational query languages, are declarative—they let the users specify *what* the answers to a query should be, rather than *how* these answers should be computed. This makes it easy for the users to specify queries, but it places the burden of query evaluation on the system. Thus deductive database systems must have powerful optimization techniques to efficiently answer queries (see [8, 66, 92] for a discussion). Existing deductive database systems (Aditi [93], CORAL [65, 68], Glue-NAIL [54, 62], and LDL [22, 59], among others) incorporate many different optimization strategies for answering queries.

One of the most important optimization strategies used in these deductive database systems is Magic-sets [7, 11, 64, 73, 80]. Magic-sets is a general program transformation technique that seeks to restrict the computation to facts that are relevant to answering the query, where a fact is considered relevant if it would be computed in a top-down evaluation of the given query. However, bottom-up evaluation of recursive programs based on Magic-sets has many advantages over a top-down evaluation scheme such as Prolog. It is sound and complete with respect to the declarative semantics of least Herbrand models (see Lloyd [50], for instance) for positive programs. When used in conjunction with Semi-naive evaluation [5, 6, 10], it is guaranteed to prevent repeated computation. Further, it is more efficient than Prolog in many database applications, because it is set-oriented. However, Magic-sets evaluation has its limitations. The evaluation is neither sound nor complete in the presence of negation or aggregation, which are very useful features of database query languages. Also, while the Magic-sets optimization can propagate "constant" binding information (e.g., $X = 5$) from a query into the program such that the transformed program may be efficiently evaluated, the propagation of "constraint" binding information (e.g., $X < 5$) leads to the computation of constraint facts [39, 64], which are expensive to manipulate. The major goal of this thesis is to address and resolve these limitations of Magic-sets based bottom-up evaluation of deductive database queries.

Although deductive databases offer a powerful query facility, they typically support a structural data model, making the representation of complex information difficult. In contrast, *object-oriented* databases allow users to model information using the sophisticated type system of a general-purpose programming language, giving the users a more powerful information modeling capability. However, manipulation of information in object-oriented databases is typically done using an imperative programming language, where the user has to specify not only *what* the answers to a query should be, but also *how* these answers should be computed. This makes specification of queries harder in object-oriented databases than in relational databases or in deductive databases. Another goal of this thesis is to demonstrate that the advantages of object-oriented database languages and deductive database languages can be combined in a clean and practical manner.

1.2 Outline of Thesis

In Chapter 2 we present notation and background material needed for the understanding of this thesis. Most of the basic terminology is given in Section 2.1. The Semi-naive evaluation technique is reviewed in Section 2.2. In Section 2.3, we describe the Magic-sets transformation.

The broad aim of this thesis is to show that powerful and practical database query languages based on logic programming can be designed and efficiently implemented. In Chapter 3 we describe a memoing evaluation technique, Ordered Search, which can evaluate a class of programs with negation and aggregation, known as left-to-right modularly stratified programs [74], more efficiently than other memoing techniques in the literature. We provide both complexity results and performance results to demonstrate the efficiency of the Ordered Search evaluation. Here is a motivating example. (Details of the syntax are presented in Chapter 2.)

Example 1.1 (Bill-of-materials) The following program is representative of a class of problems known as "bill-of-materials" problems.

$$\begin{array}{ll}
bom(Part, sum(<C>)) & :-\ subpart\_cost(Part, SubPart, C). \\
subpart\_cost(Part, Part, Cost) & :-\ basic\_part(Part, Cost). \\
subpart\_cost(Part, Subpart, Cost) & :-\ assembly(Part, Subpart, Quantity), \\
 & \quad bom(Subpart, TotalSubcost), \\
 & \quad Cost = Quantity * TotalSubcost.
\end{array}$$

The database consists of the two relations $assembly(Part, Subpart, Quantity)$ and $basic\_part(Part, Cost)$. Each fact in the relation $assembly(Part, Subpart, Quantity)$ indicates the quantity of a subpart in a composite part, and the relation represents a hierarchy of parts. Each fact in the relation $basic\_part(Part, Cost)$ indicates the cost of a basic part (i.e., a part that has no subparts) in the hierarchy of parts.

The above program computes the total cost of a composite part by adding the total costs of all its subparts. It computes two relations, $bom(Part, Totalcost)$ and $subpart\_cost(Part, Subpart, Cost)$, using the information in the database relations. Each fact in the relation $bom(Part, Totalcost)$ contains the total cost of a (basic or composite) part in a complex assembly, and each fact in the relation $subpart\_cost(Part, Subpart, Cost)$ indicates the cost of a subpart in a composite part (taking into account the quantity of subparts in the composite part).

Magic-sets based bottom-up evaluation would not be able to evaluate this program correctly, since the semantics of the program requires that for a given part a, all facts of the form $subpart\_cost(a,\_,\_)$ be computed before attempting to compute the fact $bom(a,\_)$. This requires imposing an order on the sequence of derivations, and Magic-sets evaluation cannot guarantee such an order. The evaluation technique proposed in Chapter 3, Ordered Search, works on the Magic-sets transformed program, and maintains "dependency information" between "subgoals" to ensure that the derivations are carried out in the desired order, and the correct answers to a query are computed efficiently. □

In Chapter 4 we describe an optimization technique, Constraint-rewrite, which propagates constraints (such as $X < 5$) specified in a program, in such a way that the transformed program fully utilizes the constraint information present in the original program. The Magic-sets evaluation cannot utilize all the constraint information present in such program-query pairs without computing non-ground constraint facts [39, 64], which are expensive to manipulate. With the Constraint-rewrite optimization, only "constraint relevant" facts are computed, and if the evaluation of the original program computes only ground facts, then so does the evaluation of the transformed program. Here is a motivating example. (Details of the syntax are presented in Chapter 2.)

Example 1.2 (Flight Connections) The following program is representative of a class of problems known as "path computation" problems.

$$\begin{array}{ll} cheap\_flight(S,D,C) & :-\ flight(S,D,C), C \leq 150. \\ flight(Src,Dst,Cost) & :-\ leg(Src,Dst,Time,Cost), Cost > 0. \\ flight(S,D,C) & :-\ flight(S,D1,C1), flight(D1,D,C2), C = C1 + C2. \end{array}$$

The database consists of the relation $leg(Src, Dst, Time, Cost)$, and each fact in this relation indicates the duration and the cost of a single leg flight between two cities Src and Dst. The above program computes cheap flights (i.e., flights whose total cost does not exceed \$150) between cities, where the total cost of a flight is defined as the sum of the costs of the legs of the flight.

If the query is to obtain all cheap flights between two given cities, one would like to compute only "constraint relevant" $flight$ facts: clearly, $flight$ facts that have $Cost >$ 150 are not relevant to answering this query and, hence, need not be computed.

Magic-sets evaluation would not be able to propagate the constraint $Cost \leq 150$ without computing "constraint facts." Using constraint facts in an evaluation involves symbolic manipulation of constraints, and is hence likely to be much more expensive than using only ground facts in the evaluation. Constraint-rewrite is able to propagate the constraint $Cost \leq 150$, and the bottom-up evaluation of the transformed program computes only ground facts.

Further, the Constraint-rewrite optimization can be combined with the Magic-sets transformation to efficiently propagate constant binding information in addition to propagating constraint binding information. Given any query on *cheap_flight* (i.e., any pattern of bound arguments), the bottom-up evaluation of the transformed program computes only *flight* facts with $Cost \leq 150$, that are reachable from the query. □

In Chapter 5 we present a deductive object-oriented language, Coral++, that combines the advantages of object-oriented database languages and deductive database languages. The central observation is that object-oriented features such as abstract data types, encapsulation, inheritance and object identity are essentially extensions of the data model. We can achieve a clean integration of these features in a deductive query language by making the deductive language draw values from a richer set of domains, and by allowing the use of the facilities of the deductive language to maintain, manipulate and query collections of objects of a given type. Coral++ combines the data modeling features of C++ and the querying capability of Coral—two existing languages—with minimal changes to either, and yields a powerful combination of the object-oriented and deductive paradigms. Here is a motivating example. (The syntax is discussed in Chapters 2 and 5.)

Example 1.3 (Computing Department Budgets) A university database maintains information about various departments as well as information about employees. Some type declarations in Coral++ syntax are given below.[1]

```
class employee ;     /* forward declaration */
class department {
public:
     employee   *head ;
     int   budget( ) ;
} ;
class person {
public:
     char   *name ;
```

[1]The Coral++ type declarations have the same syntax as C++ class declarations.

```
};
class employee : public person {
public:
     int    salary ;
     department    *dept ;
};
```

In this example, the budget attribute of class department is a "computed" attribute, which is computed, say, by adding the funding the department receives from several sources. The Coral++ program for the query, "Find all departments where the sum of the salaries of the employees has exceeded the department budget, and the head of the department is named John" is:

$$\begin{array}{lll} interesting\_dept(D) & :- & sum\_sals(D,S), \\ & & S > ((department*)D) \rightarrow budget(), \\ & & ((department*)D) \rightarrow head \rightarrow name = "John". \\ sum\_sals(D, sum(<S>)) & :- & employee(E), D = ((employee*)E) \rightarrow dept, \\ & & S = ((employee*)E) \rightarrow salary. \end{array}$$

The ability to use inheritance and computed attributes in specifying a query demonstrates the advantage of supporting an object-oriented data model in conjunction with a declarative query language. □

We summarize our contributions in Chapter 6. Appendix A contains detailed algorithms for Ordered Search evaluation and Constraint-rewrite optimization. These techniques are described intuitively in the main body of the thesis. Appendix B describes the fold/unfold transformations that are used by the Constraint-rewrite optimization.

Chapter 2

Preliminaries

In this chapter we present some of the basic terminology required for the understanding of the rest of this thesis. We also describe some basic query evaluation and program optimization techniques used in deductive database systems.

2.1 Syntax and Semantics

2.1.1 Syntax

We consider a first-order language that has a countably infinite set of variables, and countable sets of function and predicate symbols, these sets being pairwise disjoint. We denote variables by strings of characters, possibly subscripted, starting with an upper case letter, e.g. X, Y_1. We denote function and predicate symbols by strings of characters, possibly subscripted, starting with a lower case letter, e.g. f, p_2. We assume, without loss of generality, that with each function symbol f and each predicate symbol p is associated a unique non-negative integer 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* symbol.

Our language also includes the unary arithmetic function symbol $-$, the binary arithmetic function symbols $+, -, *$ and $/$, the binary equality predicate symbol $=$, the binary arithmetic predicate symbols $<, \leq, >$ and $\geq$, and the numeric constant symbols, e.g., 5 and 6.75.

We now define the syntax of programs and queries, in a series of steps. Program semantics is discussed in the next section.

Definition 2.1 (Arithmetic Term) An *arithmetic term* is defined as follows:

- A variable, or a numeric constant symbol is an arithmetic term.
- If t_1 and t_2 are arithmetic terms, then $(t_1 + t_2), (t_1 - t_2), (t_1 * t_2), (t_1/t_2)$, and $(-t_1)$ are arithmetic terms. □

Often, in using arithmetic terms, we omit full parenthesization. In such cases, we assume the usual precedence and associativity of the arithmetic function symbols. For instance, the arithmetic term $((X_1 + (2 * Y_1)) + 3.5)$ may be written as $X_1 + 2 * Y_1 + 3.5$.

Definition 2.2 (Term) A *term* is defined as follows:

- An arithmetic term is a term.
- Any constant symbol is a term.
- If f is an n-ary function symbol, and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term. □

Definition 2.3 (Grouping Term) If t is a term, and *agg* is an aggregate function (for instance, min, max, sum, or *count*), then $agg(< t >)$ is a *grouping term.* □

A *tuple* of terms and grouping terms $(t_1, \ldots, t_n)$ is sometimes denoted simply by the use of an overbar, e.g. $\bar{t}$.

Definition 2.4 (Arithmetic Constraint) If t_1 and t_2 are arithmetic terms, then $t_1 = t_2, t_1 < t_2, t_1 \leq t_2, t_1 > t_2$ and $t_1 \geq t_2$ are *arithmetic constraints.* □

In this thesis, unless noted otherwise, "constraint" refers to an arithmetic constraint.

Definition 2.5 (Atom) If p is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is an *atom.*

A *literal* is either an atom $p(t_1, \ldots, t_n)$ or a negative atom *not* $p(t_1, \ldots, t_n)$. □

Definition 2.6 (Grouping Atom) A *grouping atom* has the form $p(t_1, \ldots, t_n)$, where p is an n-ary predicate symbol, some of the t_i's are terms, and the others are grouping terms. □

Definition 2.7 (Rule) A *rule* in our language has the form:

$$r : A : - c_1, \ldots, c_m, L_1, \ldots, L_n.$$

A is an atom or a grouping atom, $m, n \geq 0$, $c_1, \ldots, c_m$ are constraints, and $L_1, \ldots, L_n$ are (positive or negative) literals. Optionally, a rule may be named as above, using the prefix "r :," where r is a constant symbol. If A is a grouping atom, then we require n to be greater than 0.

We refer to A as the *head* of the rule and refer to $c_1, \ldots, c_m, L_1, \ldots, L_n$ as the *body* of the rule. All variables in the rule are assumed to be universally quantified at the front of the rule.

A *Horn rule* is one where the head is an atom, and the body does not have any constraints or negative literals.

If the body of the rule has no literals, we may refer to the rule as a *fact*. Facts that have constraints in the body are referred to as *constraint facts*. If there are no constraints in the body of a fact, we may omit the ": –" symbol. (Note that a fact cannot be a grouping atom.) □

Definition 2.8 (Program) A *program* is a collection of rules. A *Horn program* is one with only Horn rules. A *Datalog program* is a function-free Horn program. □

Definition 2.9 (Range-restricted Program) A rule r in a program is said to *range-restricted* if every variable in the rule occurs in a positive body literal.

A program P is said to be *range-restricted* if every rule in the program is range-restricted. □

Definition 2.10 (Query) A *query* Q is of the form:

$$Q : ?\ q(\bar{t}).$$

The predicate q is referred to as the *query predicate*. □

Definition 2.11 (EDB/IDB) A predicate p is said to be an *extensional database* (EDB) predicate or *base* predicate of a program, if p does not appear in the head of any rule that has at least one body literal.

A predicate p is said to be an *intensional database* (IDB) predicate or *derived* predicate of a program, if p appears in the head of a rule with at least one body literal. □

The motivation for separating the EDB predicates from the IDB predicates is that program optimizations are applied only to the IDB predicates, and not to the EDB predicates. This is important in the database context since the set of EDB facts can be very large. However, the distinction is artificial, and we may choose to consider (a subset of) EDB facts to be part of a program.

2.1.2 Semantics

We now define the meaning of programs with constraints that have no grouping atoms or negative body literals. (We consider the meaning of programs with grouping atoms and negative body literals in Chapter 3.)

We use the word *ground* as a synonym for "variable-free," when referring to terms, atoms, literals, rules, etc.

Definition 2.12 (Herbrand Universe) The *Herbrand universe* of a program P is the set of all ground terms that may be formed from the constant symbols and function symbols appearing in P.[1] $\square$

Definition 2.13 (Herbrand Base) The *Herbrand base* of P is the set of all ground atoms with predicate symbols from P whose arguments are in the Herbrand universe of P. $\square$

Definition 2.14 (Relation) An n-ary *relation* R is a collection of n-tuples. $\square$

Definition 2.15 (Interpretation) Given a program P, an interpretation I of P consists of:

- A domain D.
- A mapping from each constant symbol in P to an element of domain D.
- A mapping from each n-ary function symbol in P to a function from D^n to D.
- A mapping from each n-ary predicate symbol in P to a relation in D^n. $\square$

[1]Note that we exclude the arithmetic function symbols from being used in the construction of the Herbrand universe.

Definition 2.16 (Herbrand Interpretation) Given a program P, an *Herbrand interpretation* of P is an interpretation with the Herbrand universe as the domain, with each constant symbol mapped to itself in the Herbrand universe, with each n-ary function symbol f interpreted as a mapping from n-tuples $(t_1, \ldots t_n)$ of terms from the Herbrand universe to the term $f(t_1, \ldots, t_n)$, and with each n-ary predicate symbol interpreted as a mapping to some n-ary relation on the Herbrand universe. □

The interpretations that we consider for assigning a meaning to a program treat arithmetic function symbols, arithmetic predicate symbols, and the equality predicate symbol in a special manner. These "interpreted" function symbols and predicate symbols are given their standard meaning. For instance, the meaning of the "$\leq$" arithmetic predicate symbol is a binary relation of numbers of the form, (i, j), such that $i \leq j$. In the rest of this thesis, interpretations that assign the standard meaning to arithmetic function and predicate symbols are loosely referred to as Herbrand interpretations.

Definition 2.17 (Substitution) A *substitution* is a mapping from the set of variables to the set of terms. Substitutions are denoted by lower case Greek letters θ, σ, φ, etc. Substitutions are easily generalized to apply to terms, atoms, arithmetic terms, constraints, and other syntactic objects of the language.

A substitution σ is *more general* than a substitution θ if there is a substitution φ such that, for all terms t, $\theta[t] = \varphi[\sigma[t]]$.

Two terms t_1 and t_2 are said to be *unifiable* if there is a substitution σ such that $\sigma[t_1] = \sigma[t_2]$. The substitution σ is said to be a *unifier* of t_1 and t_2. Note that if two terms have a unifier, they have a most general unifier (*mgu*) that is unique up to renaming of variables. □

Definition 2.18 (Constraint Satisfaction) A constraint $c(\bar{t})$ is said to be *satisfied* by a substitution σ if $\sigma[c(\bar{t})]$ evaluates to **true** under the standard interpretation of the arithmetic functions and predicates. □

For instance, the constraint $X < 5$ is satisfied by the substitution $\{X \rightarrow 4\}$, since $4 < 5$ evaluates to **true** under the standard interpretation of $<$, whereas the constraint $X < 5$ is not satisfied by the substitution $\{X \rightarrow 7\}$.

We now define when a rule is *true* in an interpretation for different kinds of rules.

Definition 2.19 (Truth of a Rule in an Interpretation) First, consider rules without grouping atoms in their heads. Let r be a rule of the form:

$$r : A :- c_1, \ldots, c_m, B_1, \ldots, B_n, not\ D_1, \ldots, not\ D_k.$$

$A, B_1, \ldots, B_n, D_1, \ldots, D_k$ are atoms, and $c_1, \ldots c_m$ are constraints. Let I be an Herbrand interpretation, and σ be a substitution that maps all variables of r to elements of the Herbrand universe. The rule r is said to be *true* in interpretation I for substitution σ if $\sigma[A]$ is present in I whenever:

- Each $c_i, 1 \leq i \leq m$ is satisfied by σ,
- Each $\sigma[B_i], 1 \leq i \leq n$ is present in I, and
- Each $\sigma[D_i], 1 \leq i \leq k$ is absent from I.

Now consider rules whose head is a grouping atom. For simplicity, assume that the body contains only one positive literal, and that the head contains only one grouping term of the form $agg(< Y >)$, where agg is an aggregate function, and Y is a variable.[2] Let r be a rule of the form:

$$r : p(\bar{t}, agg(< Y >)) : - p_1(\overline{t_1}).$$

Let $\overline{Z}$ be the variables (possibly including Y) that occur in $\bar{t}$. Let I be an Herbrand interpretation, and σ be a substitution that maps all variables of r to elements of the Herbrand universe. The rule r is said to be *true* in interpretation I for substitution σ if $p(\bar{s}, a)$ is present in I whenever:

- $\sigma[p_1(\overline{t_1})]$ is present in I,
- $\sigma[\bar{t}] = \bar{s}$, and
- a is the result of applying the aggregate function agg to the multiset $\{\eta[Y] \mid \eta \in S_\sigma\}$, where S_σ is the set of all ground substitutions η (on variables in $\overline{t_1}$) such that $\eta[\bar{t}] = \sigma[\bar{t}]$, and such that $\eta[p_1(\overline{t_1})]$ is present in I. $\Box$

Intuitively, grouping terms are similar to the "group-by" construct in SQL. The variables in $\bar{t}$ can be thought of as the "group-by" variables, and Y can be thought of as the "grouped" variable. Given values for each of the group-by variables, we first collect in a multiset *all* possible values of the grouped variable from p_1 facts that are present in the interpretation I, and have the same values for the group-by variables. Then, we apply the aggregate function to this multiset. If the resulting head fact is also in the interpretation, the rule with the grouping atom as its head is said to be *true*.

[2]All programs described in this thesis satisfy these conditions. Programs that do not satisfy these conditions can be rewritten, as in Beeri et al. [12], to satisfy these conditions, without changing the meaning of programs.

Definition 2.20 (Herbrand Model) Consider a program P. An Herbrand interpretation I is said to be an *Herbrand model* of a program P if each of the rules of P is true, for every substitution σ that maps variables of the rule to elements of the Herbrand universe. □

Intuitively, an Herbrand model of a program P is an interpretation that is "closed" under the reading of each rule of the program as an "if – then" statement in logic.

In an Herbrand interpretation (or model), the mappings for constant symbols and function symbols are fixed; only the mappings of predicate symbols vary. An Herbrand model can hence be viewed as a collection of relations, one corresponding to each uninterpreted predicate symbol.

Definition 2.21 (Minimal Model) An Herbrand model $M1$ is a *subset* of Herbrand model $M2$ if each relation in $M1$ is a subset of the corresponding relation in $M2$. Model $M1$ is a *proper subset* of model $M2$ if it is a subset of $M2$, and at least one relation in $M1$ is a proper subset of the corresponding relation in $M2$.

An Herbrand model $M1$ of program P is *minimal* if no proper subset of $M1$ is a model of P. □

The following result is a consequence of the results of Jaffar and Lassez [36].

Theorem 2.1 *Each program P without grouping atoms or negative body literals has a unique minimal Herbrand model.* □

Definition 2.22 (Meaning of a Positive Program) The *meaning* of a program without grouping atoms or negative body literals is given by its unique minimal Herbrand model. □

We define the meaning of programs with grouping atoms and negative body literals in Chapter 3.

Often, we are interested only in the answers to a particular query Q on a program P, rather than in the meaning of the full program P. We now define what we mean by an answer to a query.

Definition 2.23 (Answer to a Query) Consider a program P, and query Q:

$$Q : ?\ q(\bar{t}).$$

The answer to query Q is the collection of tuples matching $\bar{t}$ in the relation corresponding to the query predicate q in the meaning of program P. □

2.2 Bottom-up Evaluation

2.2.1 Naive Evaluation

Consider a program P and a database D. (The set of all relations corresponding to EDB predicates is referred to as the *database*.) The *Naive bottom-up evaluation* (see Bancilhon [6], for instance) of $\langle P, D\rangle$ proceeds in iterations. In each iteration, each of the rules in the program is independently "applied" on the set of available facts, and the set of available facts is updated at the end of the iteration. (The database D constitutes the initial set of available facts.) The Naive evaluation terminates when no new facts can be computed for any of the derived predicates of the program.

We now describe the application of a rule. For simplicity, we consider only range-restricted programs; the bottom-up evaluation of such programs compute only ground facts. In Chapter 4, we briefly consider non-range-restricted programs.

Definition 2.24 (Rule Application) First, consider a program rule without a grouping atom in its head:

$$r : p(\bar{t}) :- c_1, \ldots, c_m, p_1(\overline{t_1}), \ldots, p_n(\overline{t_n}), not\ p_{n+1}(\overline{t_{n+1}}), \ldots, not\ p_k(\overline{t_k}).$$

A *derivation* of a p fact using rule r consists of two steps:

- First, choose a (ground) p_i fact for each $p_i(\overline{t_i})$, $1 \leq i \leq n$, from the available facts to instantiate the variables in the body of rule r, such that each of the constraints $c_j, 1 \leq j \leq m$ in the body of rule r is satisfied, and none of the instantiated facts $p_i(\overline{t_i}), n+1 \leq i \leq k$ are present in the set of available facts. Let σ denote the substitution that instantiates the variables of the rule.
- Then the fact $\sigma[p(\bar{t})]$ is derived.

Note that our requirement of range-restricted programs allows negation in the body of a rule to be processed using "set-difference." In the absence of such a requirement, rule application could have required taking the complement of a relation with respect to the (Herbrand) base of the program.

Next, consider a program rule with a grouping atom in its head. By assumption, this is of the form:

$$r : p(\bar{t}, agg(< Y >)) :- p_1(\overline{t_1}).$$

Let $\overline{Z}$ be the variables (possibly including Y) that occur in $\bar{t}$. A *derivation* of a p fact using rule r consists of two steps:

Iteration	*Derivations made*
1	{ **r1: anc(1,2), r1: anc(2,3), r1: anc(4,5)** }
2	{ **r2: anc(1,3), r3: anc(1,3),** $r1 : anc(1,2), r1 : anc(2,3), r1 : anc(4,5)\}$
3	$\{r2 : anc(1,3), r3 : anc(1,3), r1 : anc(1,2), r1 : anc(2,3), r1 : anc(4,5)\}$

Table 2.1: Derivations in a Naive Evaluation of $\langle P, D\rangle$

- First, choose all (ground) p_1 facts from the available facts, that are instances of $\overline{t_1}$, such that the p_1 facts agree on the variables in $\overline{Z}$. Let σ denote the substitution that instantiates the variables of $\overline{Z}$.
- Let S denote the multiset of all Y values, one for each of the p_1 facts chosen in the previous step. Let a be the result of applying the aggregate function *agg* on S. Then the fact $p(\sigma[\overline{t}], a)$ is derived.

An *application* of rule r consists of making *all* possible derivations that can be made using rule r and the set of available facts. □

The following result is straightforward.

Theorem 2.2 *The* Naive bottom-up evaluation *of a program without grouping atoms or negative body literals computes the meaning of the program.* □

The Naive evaluation of a program on a database is extremely inefficient since it repeats derivations; each derivation made in an application of a rule is repeated in every subsequent application of the same rule. Example 2.1 below illustrates this inefficiency.

Example 2.1 (Naive Evaluation) Consider the following program P and query Q:

$r1 : anc(X, Y) :- par(X, Y).$
$r2 : anc(X, Y) :- par(X, Z), anc(Z, Y).$
$r3 : anc(X, Y) :- anc(X, Z), anc(Z, Y).$
$Q : ?\ anc(1, X).$

Let the database D contain the facts:

$par(1, 2). \quad par(2, 3). \quad par(4, 5).$

A Naive bottom-up evaluation of $\langle P, D\rangle$ would result in the derivations of *anc* facts as shown in Table 2.1. (The rule used to derive a fact is also indicated, and new derivations are shown in bold-face.) Only $anc(1, 2)$ and $anc(1, 3)$ are answers to the query Q. Note that each derivation made in an iteration of Naive bottom-up evaluation is repeated in subsequent iterations. □

2.2.2 Semi-naive Evaluation

Semi-naive evaluation [5, 6, 10, 27, 61] is an optimization of Naive evaluation, such that no derivation is repeated in the bottom-up evaluation. For simplicity, we consider only range-restricted programs; the bottom-up evaluation of such programs compute only ground facts.

Given a program P, the Semi-naive evaluation of P also proceeds in iterations. There are essentially two components to the Semi-naive evaluation of a program.

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

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

   $$r : p(\overline{t}) :- l_1(\overline{t_1}), \ldots, l_n(\overline{t_n}), not\ p_{n+1}(\overline{t_{n+1}}), \ldots, not\ p_m(\overline{t_m}).$$

   where $l_i(\overline{t_i}), i \leq n$ are base literals, and $p_i, i > n$ are predicates defined in P the following Semi-naive rewritten rule is obtained from r:

   $$r1 : \delta p^{new}(\overline{t}) :- l_1(\overline{t_1}), \ldots, l_n(\overline{t_n}), not\ p_{n+1}(\overline{t_{n+1}}), \ldots, not\ p_m(\overline{t_m}).$$

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

   For each rule in P of the form:

   $$\begin{aligned} r : p(\overline{t}) :- {} & p_1(\overline{t_1}), \ldots, p_n(\overline{t_n}), not\ p_{n+1}(\overline{t_{n+1}}), \ldots, not\ p_m(\overline{t_m}), \\ & l_{m+1}(\overline{t_{m+1}}), \ldots, l_k(\overline{t_k}). \end{aligned}$$

where $p_1, \ldots, p_m$ are predicates defined in P, $n > 0$ and $l_i(\overline{t_i}), i > m$ are (positive or negative) base literals, the following n Semi-naive rewritten rules are obtained from r, if $n > 0$:

$$\begin{aligned}
r1 : \delta p^{new}(\overline{t}) :- \; & \delta p_1^{old}(\overline{t_1}), p_2(\overline{t_2}), \ldots, p_n(\overline{t_n}), \\
& not\ p_{n+1}(\overline{t_{n+1}}), \ldots, not\ p_m(\overline{t_m}), l_{m+1}(\overline{t_{m+1}}), \ldots, l_k(\overline{t_k}). \\
r2 : \delta p^{new}(\overline{t}) :- \; & p_1^{old}(\overline{t_1}), \delta p_2^{old}(\overline{t_2}), \ldots, p_n(\overline{t_n}), \\
& not\ p_{n+1}(\overline{t_{n+1}}), \ldots, not\ p_m(\overline{t_m}), l_{m+1}(\overline{t_{m+1}}), \ldots, l_k(\overline{t_k}). \\
& \vdots \\
rn : \delta p^{new}(\overline{t}) :- \; & p_1^{old}(\overline{t_1}), p_2^{old}(\overline{t_2}), \ldots, p_{n-1}^{old}(\overline{t_{n-1}}), \delta p_n^{old}(\overline{t_n}), \\
& not\ p_{n+1}(\overline{t_{n+1}}), \ldots, not\ p_m(\overline{t_m}), l_{m+1}(\overline{t_{m+1}}), \ldots, l_k(\overline{t_k}).
\end{aligned}$$

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

2. The second component is a technique to apply the rewritten rules and update these differentials, ensuring that each derivation is made *exactly once* in the bottom-up evaluation.

   Initially, all relations other than those corresponding to EDB (or, base) predicates are empty. In evaluating P, the first iteration consists of applying each of the Semi-naive (both non-recursive and recursive) rewritten rules in P. Subsequent iterations consist of applying only the recursive Semi-naive rules. The evaluation of P proceeds by iterating until no new facts are computed for any of the predicates defined in P. After applying the Semi-naive rules in an iteration, the extensions of the Semi-naive relations for each p_i are updated using Procedure SN-Update in Figure 2.1.

In Semi-naive bottom-up evaluation, the differential versions of predicates are used to distinguish newly generated facts from older facts. Consider a predicate p in the program P.

- The relation corresponding to the differential predicate δp^{new} in the Semi-naive transformed program contains facts generated in the "current" iteration; these facts are not available to the evaluation.

- The relation corresponding to the differential predicate δp^{old} contains facts generated in the "immediately prior" iteration; these facts were made available to the evaluation at the end of the previous iteration.

```
procedure SN-Update($p_i$)
{
    /* After applying Semi-naive rules in an iteration */
    $p_i^{old} := p_i^{old} \cup \delta p_i^{old}$.
    $\delta p_i^{old} := \delta p_i^{new} - p_i^{old}$.
    $p_i := p_i^{old} \cup \delta p_i^{old}$.
    $\delta p_i^{new} := \phi$.
}
```

Figure 2.1: Procedure SN-Update

Iteration	*Derivations made*
1	$\{r1 : anc(1,2), r1 : anc(2,3), r1 : anc(4,5)\}$
2	$\{r2 : anc(1,3), r3 : anc(1,3)\}$
3	$\{\}$

Table 2.2: Derivations in a Semi-naive Evaluation of P

- The relation corresponding to the differential predicate p^{old} contains facts generated two or more iterations prior to the current iteration.
- The relation corresponding to the differential predicate p in the Semi-naive transformed program contains facts generated one or more iterations prior to the current iteration, i.e., $p = p^{old} \cup \delta p^{old}$.

At every stage of the evaluation, the set of relations p_i^{old}, for all i, has the property that every derivation that uses only these facts has been made. This can be seen from the nature of the δ terms in the bodies of the Semi-naive rules, and the order of updates of the various Semi-naive relations by Procedure SN-Update.

The following result is straightforward:

Theorem 2.3 *The* Semi-naive bottom-up evaluation *of a program without grouping atoms or negative body literals computes the meaning of the program, and does not repeat any derivations.* □

Example 2.2 (Semi-naive Evaluation) Consider again the program P of Example 2.1. Using Semi-naive evaluation, the sequence of derivations made is shown in

Table 2.2. Note that each derivation made in the Naive evaluation is also made in the Semi-naive evaluation. However, the Semi-naive evaluation does not repeat any derivations, i.e., all derivations are new derivations. But if a fact ($anc(1,3)$ in this case) is derived by two *different* derivations, each of these derivations is made in the Semi-naive evaluation.

Such "redundant" derivations can be avoided by recognizing that every fact derived using rule $r2$ is also derived using rule $r3$, and vice versa. Consequently, a scheme that prunes redundant derivations (Helm [33], for instance) could recognize this and never apply rule $r3$; the derivation of $r3 : anc(1,3)$ is not made in such a case. Given a program that has redundant derivations, its evaluation using Semi-naive *does not* eliminate such derivations. □

2.2.3 Dependency Graphs

Both the Naive and the Semi-naive evaluation strategies can be refined by taking the structure of the program into account. In this section we formalize the syntactic structure of the program based on the "predicate dependency graph."

Definition 2.25 (Strongly Connected Component) A directed graph $G = (V, E)$, where V is the vertex set and $E \subseteq V \times V$ is the edge set, is said to be *strongly connected* if there is a path, using the directed edges in E, between every pair of distinct vertices in V.

Given a directed graph G, a subgraph $G1$ of G is said to be a *strongly connected component* (SCC) of G if $G1$ is a maximal subgraph in G that is strongly connected. Note that the SCCs of a directed graph G partition the vertices of G.

The *condensation* of G with respect to the SCCs is the graph $G' = (V', E')$ obtained with vertex set V' being the set of SCCs $\{S_1, \ldots, S_k\}$ of G. An edge $(S_i, S_j) \in E', i \neq j$, if there is an edge $(V_k, V_l) \in E$, where V_k is a vertex in S_i and V_l is a vertex in S_j. The condensation G' of G reflects the SCC structure of G, and is acyclic.

We say that $S_1 \prec S_2$, if there is a path from S_1 to S_2 in G'. (In particular, $S_i \prec S_i$, for all i.) □

Note that the "$\prec$" relation on a condensed graph is a partial order, i.e., it is reflexive, anti-symmetric and transitive.

Definition 2.26 (Predicate Dependency Graph) Given a program P with predicates $Pred = \{p_1, \ldots, p_n\}$, the *predicate dependency graph* of P is the directed graph

$G = (Pred, E)$, where $(p_i, p_j) \in E$ iff p_i occurs (positively or negatively) in the body of a rule defining p_j.

A predicate p_i *depends upon* a predicate p_j if there is a path from p_j to p_i in the predicate dependency graph. A predicate p_i is *recursive with* a predicate p_j if p_i depends upon p_j and p_j depends upon p_i. $\Box$

We refer to the SCCs of the predicate dependency graph of a program as the SCCs of the program. With each SCC S of a program P, we also associate all rules in P defining the predicates in S. Note that if SCC S_i contains a predicate used in a rule in SCC S_j, then $S_i \prec S_j$. Given a program P, its SCCs $S_1, \ldots, S_m$ are said to be in *topological order*, if whenever $S_i \prec S_j$, then $i \leq j$.

The Naive and Semi-naive evaluation strategies can be refined by evaluating one strongly connected component (SCC) at a time, in a topological ordering of the SCCs. In the SCC-by-SCC Semi-naive evaluation of a program, predicates that are not defined in an SCC S are treated as "base" predicates for the Semi-naive rewriting of S, and only predicates defined in S are updated using Procedure SN-Update after each iteration in the evaluation of S.

2.3 Magic-sets Transformations

Given a program P and query Q, the Naive and Semi-naive bottom-up evaluation techniques compute the meaning of the full program P in order to answer the query. If we are interested only in the answers to the query Q, these evaluation techniques can be very inefficient, and a considerable amount of effort within the deductive database community has been devoted to automatically specializing evaluation techniques for efficiently computing answers to a query. It is beyond the scope of this thesis to review them all, and we refer the reader to [66, 92] for a survey. In this section, we focus attention of the Magic-sets approach to improving the efficiency of answering queries. The Magic-sets transformation is important because it can result in significant improvements, and at the same time it is generally applicable (see Kemp [43] for details).

One of the main objectives of efficient query evaluation is to avoid computing the entire model of the original program in answering a query. Top-down evaluation techniques, such as Prolog, usually achieve this objective by using answers to subgoals to bind arguments of other subgoals, and hence restrict the model to be computed to facts relevant to answering a query (or goal). Magic-sets transformations [7, 11, 64, 73, 80] are used to imitate top-down computations using bottom-up computation. The major advantage they provide over Naive and Semi-naive bottom-up evaluation is that they allow a bottom-up computation to be specialized with respect to the query, thus improving the efficiency of answering queries.

The intuition behind the Magic-sets transformation is to compute a set of auxiliary (or, magic) predicates that contain the subgoals set up in a top-down evaluation of the original program. The rules in the program P are then modified by attaching additional literals that act as filters and prevent the rule from generating "irrelevant" facts. There are three distinct steps to the Magic-sets transformation of a program P and query Q.

- First, *new* rules that define "magic" predicates are added to the Magic-sets transformed program MP. Intuitively, applications of these rules in a bottom-up evaluation of the transformed program compute subgoals set up in a top-down evaluation of the original program P.

  In the top-down evaluation of program P, subgoals on a predicate p may have some arguments bound to ground terms, while other arguments may contain free variables. The former arguments are referred to as *bound* arguments, and the latter arguments are referred to as *free* arguments of the subgoals on predicate p.[3] In the Magic-sets transformed program MP, the arity of the magic predicate m_p (corresponding to predicate p in program P) is the number of bound arguments in the subgoals set up on predicate p in a top-down evaluation of P.[4]

- Second, *modified* versions of the rules of program P are added to the Magic-sets transformed program MP. Each rule of program P is modified by attaching an additional "magic" literal to the body of the rule. These magic literals correspond to the subgoals set up on the *heads* of rules in a top-down evaluation of P.

  These magic literals act as filters and prevent the rule from generating irrelevant facts, where a fact is considered *relevant* only if it is an answer to a subgoal set up in a top-down evaluation of P.

- Finally, a "seed' magic fact that corresponds to the query Q is added to the Magic-sets transformed program MP.

[3]Ramakrishnan [64] generalizes these notions of "bound" and "free" arguments of a subgoal to capture larger classes of binding patterns. The interested reader is referred to [55, 56, 64] for a detailed discussion.

[4]We assume that all subgoals set up on a predicate p in a top-down evaluation have the same set of bound and free arguments. This assumption can be easily relaxed using the notion of "predicate adornments," and the interested reader is referred to [11] for a detailed discussion.

Because of the correspondence between the subgoals set up in a top-down evaluation of the original program P, and the magic facts computed in a bottom-up evaluation of the Magic-sets transformed program MP, we often refer to magic facts as subgoals in this thesis. Similarly, there is a correspondence between the answers to subgoals set up in a top-down evaluation of P, and the non-magic facts computed in a bottom-up evaluation of MP; hence, we often refer to these non-magic facts as answer facts in this thesis. The interested reader is referred to [17, 69, 86, 91] for details on the duality between top-down and bottom-up evaluation.

Note that the first two steps of the Magic-sets transformation can be performed at "compile-time," while the final step can be performed only at "run-time," when the actual query is known. We formally define the Magic-sets transformation next, and then provide an example of the Magic-sets transformation.

Definition 2.27 (Magic-sets Transformation) Let P be a program and Q be a query of P. The *Magic-sets transformation* results in a new program MP obtained as follows. Initially, MP is empty.

1. Create a new predicate m_p for each predicate p in P, where the arity of m_p is the number of bound arguments in the subgoals set up on predicate p in a top-down evaluation of P.

2. For each rule in P, add the *modified version* of the rule to MP. If a rule has head $p(\overline{s})$, the modified version of this rule is obtained by adding the literal $m\_p(\overline{s}^1)$ to the body, where $\overline{s}^1$ denotes the tuple of argument positions of $\overline{s}$ that are bound in the top-down evaluation of P.

3. For each rule r in P with head $p(\overline{s})$, and for each body literal $q(\overline{t})$, add a *magic rule* to MP. The head is $m\_q(\overline{t}^1)$. The body contains the literal $m\_p(\overline{s}^1)$, and all the literals in the body of r that are to the left of $q(\overline{t})$.

4. Create a *seed* fact $m\_q(\overline{c}^1)$ from the query Q. □

Example 2.3 (Magic-sets Transformation) Consider again the program P and query Q of Example 2.1, reproduced below:

$$
\begin{array}{l}
r1 : anc(X,Y) :- par(X,Y). \\
r2 : anc(X,Y) :- par(X,Z), anc(Z,Y). \\
r3 : anc(X,Y) :- anc(X,Z), anc(Z,Y). \\
Q : ?\ anc(1,X).
\end{array}
$$

Iteration	*Derivations made*
1	$\{mr4 : m\_anc(1)\}$
2	$\{r1 : anc(1,2), mr1 : m\_anc(2), mr2 : m\_anc(1)\}$
3	$\{r1 : anc(2,3), mr1 : m\_anc(3), mr2 : m\_anc(2), mr3 : m\_anc(2)\}$
4	$\{r2 : anc(1,3), r3 : anc(1,3), mr2 : m\_anc(3), mr3 : m\_anc(3)\}$
5	$\{mr3 : m\_anc(3)\}$
6	$\{\}$

Table 2.3: Derivations in a Semi-naive Evaluation of MP

In a top-down, Prolog-style evaluation of $\langle P, Q\rangle$, each subgoal on the *anc* predicate has its first argument bound to a constant, and the second argument is a free variable. Consequently, the magic predicate *m_anc* is a unary predicate.

The Magic-sets transformation of this program-query pair results in the following program MP:

$$\begin{aligned}
r1:&\quad anc(X,Y) :- m\_anc(X), par(X,Y).\\
r2:&\quad anc(X,Y) :- m\_anc(X), par(X,Z), anc(Z,Y).\\
r3:&\quad anc(X,Y) :- m\_anc(X), anc(X,Z), anc(Z,Y).\\
mr1:&\; m\_anc(Z) :- m\_anc(X), par(X,Z).\\
mr2:&\; m\_anc(X) :- m\_anc(X).\\
mr3:&\; m\_anc(Z) :- m\_anc(X), anc(X,Z).\\
mr4:&\; m\_anc(1).
\end{aligned}$$

Rules $mr1, mr2$ and $mr3$ of MP are the new rules defining magic predicates; rules $r1, r2$ and $r3$ of MP are modified versions of the rules of P, and the fact $mr4$ is the seed magic fact obtained from the query Q.

Each of the rules $mr1, mr2$ and $mr3$ is obtained from one of the *anc* literals in the bodies of rules of P as follows: rule $mr1$ is obtained from the $anc(Z,Y)$ literal in rule $r2$ of P, rule $mr2$ is obtained from the $anc(X,Z)$ literal in rule $r3$ of P, rule $mr3$ is obtained from the $anc(Z,Y)$ literal in rule $r3$ of P. To understand the structure of the rules defining the magic predicates, it is useful to recollect the steps in a top-down evaluation of program P. For instance, given a subgoal $?anc(a,Y)$ on the head of rule $r2$, a top-down evaluation would set up the "new" subgoal $?anc(b,Y)$, if there was a fact $par(a,b)$ in the database. This is *exactly* what an application of rule $mr1$ of the transformed program MP achieves: if there is a magic fact $m\_anc(a)$ (corresponding to the subgoal $?anc(a,Y)$ in the top-down evaluation of P) and the fact $par(a,b)$ is present in the database, an application of rule $mr1$ would compute the magic fact $m\_anc(b)$ (corresponding to the subgoal $?anc(b, Y)$ in the top-down evaluation of P). The other rules defining the magic predicates can be understood in a similar manner.

Each of the rules $r1, r2$ and $r3$ in MP is obtained from the corresponding rule in P by adding the magic literal $m\_anc(X)$ to the body of the rule. Intuitively, these literals act as filters and ensure that an $anc(a, b)$ fact is computed in the bottom-up evaluation of the transformed program only if there is a magic fact $m\_anc(a)$, i.e., the subgoal $?anc(a, Y)$ had been set up in the top-down evaluation of P. Consequently, the anc facts computed by the transformed program are all "relevant" to answering the query.

We now present the sequence of derivations made in a Semi-naive bottom-up evaluation of MP; this sequence is shown in Table 2.3. Note that the evaluation of MP results in avoiding the derivation of $anc(4, 5)$, since this is "irrelevant" to computing the answers to the query $anc(1, X)$. $\square$

Supplementary Magic-sets transformation is an optimization of Magic-sets transformation that, essentially, performs some common sub-expression elimination automatically, at the cost of computing additional (supplementary) predicates. We refer the reader to [11, 64] for a detailed discussion of the Supplementary Magic-sets transformation, and give an example here.

Example 2.4 (Supplementary Magic-sets Transformation) Consider the Magic-sets transformed program MP of Example 2.3. Note that $m\_anc(X), par(X, Z)$ occur as the first two body literals in each of the rules $r2$ and $mr1$. Also, $m\_anc(X), anc(X, Z)$ occur as the first two body literals in each of the rules $r3$ and $mr3$.

The Supplementary Magic-sets transformation infers the presence of these common sub-expressions because of the nature of the Magic-sets transformation, and performs common sub-expression elimination from the modified version of a rule r in the original program and the magic rules obtained from rule r, to optimize the evaluation of the transformed program. The Supplementary Magic-sets transformation of the program P

and query Q of Example 2.1 is the following program:

$$\begin{array}{lll}
r1: & anc(X,Y) & :- m\_anc(X), par(X,Y). \\
r2: & anc(X,Y) & :- sup\_1(X,Z), anc(Z,Y). \\
r3: & anc(X,Y) & :- sup\_2(X,Z), anc(Z,Y). \\
sr1: & sup\_1(X,Z) & :- m\_anc(X), par(X,Z). \\
mr1: & m\_anc(Z) & :- sup\_1(X,Z). \\
mr2: & m\_anc(X) & :- m\_anc(X). \\
sr3: & sup\_2(X,Z) & :- m\_anc(X), anc(X,Z). \\
mr3: & m\_anc(Z) & :- sup\_2(X,Z). \\
mr4: & m\_anc(1). &
\end{array}$$

The use of the literal $sup\_1(X,Z)$ eliminates the common sub-expression $m\_anc(X)$, $par(X,Z)$ in the bodies of rules $r2$ and $mr1$, and the use of the literal $sup\_2(X,Z)$ eliminates the common sub-expression $m\_anc(X), anc(X,Z)$ in the bodies of rules $r3$ and $mr3$. (Note that the Supplementary Magic-sets transformation does not replace the sub-expression $m\_anc(X), par(X,Z)$ in the body of rule $r1$ by $sup\_1(X,Z)$.) $\square$

Chapter 3

Ordered Search

3.1 Background

Magic-sets based bottom-up evaluation of queries on deductive databases has many advantages over an evaluation scheme such as Prolog. It is sound and complete with respect to the declarative semantics of least Herbrand models for positive Horn clause programs. In particular, it is able to avoid infinite loops by detecting repeated (possibly cyclic) subgoals (i.e., magic facts). Further, in many database applications, it is more efficient than Prolog due to its set-oriented evaluation. However, the completely set-oriented, breadth-first search strategy of bottom-up evaluation has certain disadvantages. For example, to evaluate several classes of programs with negation (or aggregation), it is necessary to order the inferences; in essence, all answers to a negative subgoal must be evaluated before making an inference that depends upon the negative subgoal. A completely breadth-first search strategy (Ross [74, 75], for instance) would have to maintain redundant subgoal dependency information to achieve this.

In this chapter we present a memoing technique to order the use of generated subgoals, that is a hybrid between pure breadth-first and pure depth-first search. The technique, called Ordered Search, works on the Magic-sets transformed program, and is able to maintain subgoal dependency information efficiently, while detecting repeated subgoals, and avoiding infinite loops. The technique also avoids repeated computation and is complete for Datalog. Ordered Search can be used to evaluate programs with left-to-right modularly stratified negation and aggregation more efficiently than with any previously known bottom-up technique.

3.1.1 Motivating Examples

Example 3.1 (Computing Even Numbers) Consider the program $\langle P_{even}, Q_{even}\rangle$ below:

$$\begin{array}{ll} r1: & even(X) :- succ(X, Y1), succ(Y1, Y), even(Y). \\ r2: & even(X) :- succ(X, Y), not\ even(Y). \\ r3: & even(0). \\ & succ(1,0).\ \ succ(2,1).\ \ \ldots\ \ succ(n, n-1). \\ & Q_{even} : ?\ even(m). \end{array}$$

where $m \leq n$.

Intuitively, in the meaning of the above program the answer to the query should be *yes* if m is an even number, and *no* if m is an odd number. (The meaning of programs with negative body literals is formally defined in Section 3.3.) A bottom-up evaluation of the above program would apply the program rules iteratively until no new facts are computed. Initially the *even* relation is empty. The first application of rule $r2$ would compute facts of the form $even(i)$ for every $succ(i, j)$ fact that is present in the database. Clearly this is *not* the desired meaning of the program.

A Prolog evaluation of the above program would however compute the correct answer to the query. Given the query (or, goal) $?even(3)$, for instance, a Prolog evaluation first sets up the subgoal $?even(1)$ (using rule $r1$), which sets up the subgoal $?not\ even(0)$ (using rule $r2$). This subgoal fails because of rule $r3$, and consequently the subgoal $even(1)$ fails. On backtracking, the subgoal $?not\ even(2)$ is set up (using rule $r2$), which sets up the subgoal $?even(0)$ (using rule $r1$). This subgoal succeeds, and consequently the subgoal $?not\ even(2)$ fails. As a result, the goal $?even(3)$ fails, which is consistent with the desired semantics. Intuitively, the reason that Prolog evaluation correctly evaluates this program is that it implicitly sets up "dependencies" between subgoals, and computes answers to a negative subgoal before making an inference that "depends on" the negative subgoal. (We make precise this notion of "depends on" in Section 3.3.)

Although bottom-up evaluation of the original program does not compute the correct answers to the query, one may enquire whether the bottom-up evaluation of the Magic-sets transformed program computes the desired meaning. The bottom-up evaluation of the Magic-sets transformed program *does* compute the subgoals (i.e., magic facts) set up in the top-down evaluation. However, it does not maintain "dependencies" between subgoals and consequently the evaluation of the Magic-sets transformed program also computes incorrect answers to queries.

Ross [75] proposed a modified Magic-sets rewriting of $\langle P_{even}, Q_{even}\rangle$ in conjunction with a bottom-up method for evaluating the rewritten program. The rewritten program has rules that define predicates of the form $dp(even(X), even(Y))$ and $dn(even(X), even(Y))$, which correspond to positive and negative dependencies, respectively, between subgoals set up in the Prolog evaluation. The bottom-up method proposed by Ross explicitly computes all the subgoal dependency information and ensures that, if there is a fact $dn(even(i), even(j))$ (i.e., there is a negative dependency between subgoals $?even(i)$ and $?even(j)$ in the top-down evaluation), the answer to the subgoal $?even(j)$ is computed *before* computing the answer to the subgoal $?even(i)$. Ross' approach on this program would take $O(m^2)$ space and make $O(m^2)$ derivations since it would compute and store *transitive* dependencies between subgoals.

Our technique, Ordered Search, also modifies Magic-sets rewriting and bottom-up evaluation to compute "dependencies" between subgoals, and to ensure that if there if a negative dependency between subgoals $?even(i)$ and $?even(j)$ in the top-down evaluation, then the answer to the subgoal $?even(j)$ is computed *before* computing the answer to the subgoal $?even(i)$. However, our approach is quite different from that proposed by Ross. Our variant of the Magic-sets rewriting (described in Section 3.3.3) introduces "guard literals" before each negative body literal in the Magic-sets transformed program. These guard literals are satisfied only when *all* the answers to the subgoal corresponding to the negative body literal have been obtained. Subgoals are stored on a stack-like data structure, and the order in which the subgoals are stored and made available to the bottom-up evaluation corresponds to the dependencies between subgoals. In effect, only information about direct dependencies is stored; information about transitive dependencies can be inferred. Hence, Ordered Search would use $O(m)$ space and make $O(m)$ derivations in computing the query answer. (For more details, see Example 3.8.)

We describe other top-down and bottom-up techniques that can evaluate this program in Section 3.7. As an example, the technique of Morishita [53] would also use $O(m)$ space and make $O(m)$ derivations on this example. However, if rule $r1$ were removed from P_{even}, the technique of [53] would make $O(m^2)$ derivations, though it would still use only $O(m)$ space. Even on this modified program, Ordered Search would compute the answer to the query using $O(m)$ space and making $O(m)$ derivations. □

Example 3.2 (Working Parts) Consider the following program from Kemp et al. [40], which is modified from an example in Ross [74].

$$\begin{array}{ll}
working(X) & :-\ tested(X). \\
working(X) & :-\ essential\_part(X,Y), working(Y). \\
working(X) & :-\ part(X,Y), not\ has\_suspect\_part(X). \\
has\_suspect\_part(X) & :-\ part(X,Y), not\ working(Y).
\end{array}$$

The database consists of the relations *tested*, *part* and *essential_part*. The *part(Part, Subpart)* relation represents a hierarchy of parts, and the *essential_part(Part, Subpart)* relation is a subset of the *part(Part, Subpart)* relation. Parts that have been tested are present in the *tested(Part)* relation.

Intuitively, in the meaning of the above program, a part is working if it has been tested, or if it is composite (i.e., it has subparts) and an essential subpart of it is working, or else if it is composite and none of its subparts are suspect. A composite part is considered suspect if at least one of its subparts is not working. The last two program rules essentially encode the condition that a part is working if *all* of its subparts are working.

Suppose the *part(Part, Subpart)* relation is a complete binary tree with m nodes, the left half of the leaf nodes are in *tested(Part)*, and the *essential_part(Part, Subpart)* relation comprises the left branches in *part(Part, Subpart)*. Thus the entire left hand side of the tree is working. The query enquires whether the root of the tree is working.

Again, on this program, Ross' approach would take $O(m^2)$ space and make $O(m^2)$ derivations since it would compute and store all the dependencies between subgoals transitively.

Our technique, Ordered Search, would compute and store only information about direct dependencies; hence, it would use $O(m)$ space and make $O(m)$ derivations in computing the query answer. □

Ordered Search also correctly evaluates left-to-right modularly stratified programs with aggregation. This class of programs includes the important "bill-of-materials" problems.

Example 3.3 (Bill-of-materials) Consider again the program from Example 1.1.

$$\begin{array}{ll}
bom(Part, sum(<C>)) & :-\ subpart\_cost(Part, SubPart, C). \\
subpart\_cost(Part, Part, Cost) & :-\ basic\_part(Part, Cost). \\
subpart\_cost(Part, Subpart, Cost) & :-\ assembly(Part, Subpart, Quantity), \\
 & \quad bom(Subpart, TotalSubcost), \\
 & \quad Cost = Quantity * TotalSubcost.
\end{array}$$

This program computes the total cost of a composite part by adding the total costs of its subparts. Let the *assembly*$(Part, Subpart, Quantity)$ relation be a complete binary tree with m nodes, with each composite part containing exactly one copy of each of its subparts. Let the *basic_part*$(Part, Cost)$ relation correspond to the leaves of the binary tree. The query asks for the total cost of the root of the tree.

Both Ross' approach and Ordered Search would take $O(m)$ space and make $O(m)$ derivations in computing the answer to a top-level query. $\Box$

3.2 Outline of Chapter

The rest of this chapter is organized as follows.

Preliminaries are covered in Section 3.3. We define the subgoal dependency graph that characterizes the dependencies between subgoals in Section 3.3.1. The syntax and semantics of modularly stratified programs are reviewed in Section 3.3.2. The Ordered Search evaluation uses a variant of the Magic-sets transformation; this variant is described in Section 3.3.3.

The data structures and algorithms that are required to evaluate a program using Ordered Search are described in Section 3.4. This chapter describes the algorithms at an intuitive level, and detailed algorithms are presented in Appendix A.1. In Section 3.4.4, we present a detailed description of the Ordered Search evaluation for Example 3.1.

Section 3.4 makes several informal claims about the properties of the Ordered Search evaluation. The soundness, completeness, and complexity results of the Ordered Search evaluation are formalized and presented in Section 3.5. The Ordered Search evaluation strategy is implemented in the Coral deductive database system. In Section 3.6 we present performance results that demonstrate the practicality of the Ordered Search evaluation. We also provide some results to measure the overheads of the Ordered Search evaluation compared to Semi-naive evaluation for positive programs.

Related work is presented in Section 3.7. Finally, in Section 3.8, we consider how Ordered Search is useful in evaluating queries when just one answer is desired. In this section, we also briefly describe some of our other research in ordering inferences in a bottom-up evaluation.

3.3 Preliminaries

3.3.1 Subgoal Dependency Graph

For simplicity, we consider only programs with negation in the rest of this chapter. However, we discuss examples with aggregation to illustrate the utility of our evaluation technique. Aggregation is similar to negation in the problems it poses to program semantics and bottom-up evaluation, and we indicate the similarities between negation and aggregation where appropriate. We also assume that all programs are range-restricted; this ensures that negation can be processed using set-difference, and only ground facts are generated in the bottom-up evaluation.

Definition 3.1 (Non-floundering Program) A program P is *non-floundering* if each negative subgoal set up in a top-down evaluation of P is ground. $\square$

As with the technique of Ross [75], our technique deals only with non-floundering programs, and we assume that programs are non-floundering in the rest of this chapter.

We now define SLP-trees, which characterize top-down Prolog-style program evaluations.

Definition 3.2 (SLP-trees) ([75]) Let P be a non-floundering program, and let Q be a query. We define the SLP-tree T_Q for Q. The root of T_Q is Q. Each node in T_Q is a sequence of literals. If Q' is a node of T_Q then its children are obtained as follows:

1. If Q' is empty, then we call it a successful leaf.
2. Suppose that the leftmost literal L in Q' is positive. Let U_L be the set of program rules whose heads unify with L. The children of Q' are obtained by resolving[1] Q' with (a variant of) each of the rules in U_L over the literal L using most general unifiers. If there is no such U_L then Q' has no children, and is a failed leaf.
3. Suppose the leftmost literal L in Q' is negative, say *not* A. (By our assumption about the absence of floundering, L must be ground.) Recursively construct the SLP-tree T_A for A.

   – If T_A is successful, then Q' is a failed leaf.

[1]As in Prolog evaluation.

- If T_A is failed, then Q' has a single child that is formed by deleting L from Q'.
- If T_A is indeterminate, then Q' is an indeterminate leaf.

If T_Q has a successful leaf, then T_Q is *successful.* If every leaf of T_Q is failed, then T_Q is *failed.* Otherwise, T_Q is *indeterminate.* (This can happen because of an infinite sequence of subgoals set up in the top-down evaluation.)

A *branch* of T_Q is a path from the root of T_Q. We associate with each successful leaf V an *answer substitution*, which is the composition of the most general unifiers used along the branch to V. □

SLP-trees contain information about subgoals as well as answers to subgoals. Since we are interested only in the subgoals and dependencies between them, we first define a *reduced SLP-tree*, which contains all the information contained in SLP-trees about the subgoals.

Definition 3.3 (Reduced SLP-trees) Let P be a non-floundering program, let Q be an query, and let T_Q be the SLP-tree for Q. We define the *reduced SLP-tree* R_Q for Q.

The root of R_Q is Q. If the SLP-tree T_Q is a single node, then so is R_Q. Otherwise, the children of Q in R_Q are the same as the children of Q in T_Q. If the query Q' is any (non-root) node of R_Q, then its children are obtained as follows: Let Q'_1 be the node corresponding to Q' in T_Q.

- If Q'_1 has only one literal L, Q' has no children.
- Else, let $Q'_1 = L_1, L_2, \ldots, L_m$. Consider a path B in T_Q from Q'_1 down to a leaf of T_Q. Let Q'_2 be the highest node (i.e., closest to Q'_1) in the path B that is an instance of $L_2, \ldots, L_m$. (Intuitively, the path between node Q'_1 and node Q'_2 corresponds to the computation of an answer to the left-most subgoal L_1 of Q'_1.) Let $S_{Q'_1}$ be the set of all such Q'_2 obtained along different paths in T_Q from Q'_1 down.
  - If $S_{Q'_1}$ is empty, Q' has no children.
  - Else, each element of $S_{Q'_1}$ is a child of Q'. □

We now formally define the negation tree, which summarizes information contained in SLP-trees about the *negative* subgoals set up in a top-down evaluation of the original program.

Definition 3.4 (Negation Tree) ([75]) Given a subgoal G, the *negation tree* N_G is defined as follows: The nodes of N_G are subgoals, and the root of N_G is G. Let H be any node of N_G. For every atom A for which T_A is recursively constructed (in Step 3 of Definition 3.2) in constructing T_H, A is a child of H. □

Intuitively, the subgoal dependency graph of a program-query pair is an AND/OR directed graph that characterizes the (positive and negative) dependencies between subgoals set up in a top-down evaluation of the original program. We formally define the subgoal dependency graph in terms of SLP-trees, reduced SLP-trees and negation trees.

Definition 3.5 (Subgoal Dependency Graph) Let P be a non-floundering program, and let Q be a query. The *subgoal dependency graph* $S(P,Q)$ has two types of nodes: AND nodes and OR nodes. There is one OR node in $S(P,Q)$ for the leftmost literal of each node in T_H, for each H that is a node in the negation tree for Q.

The children of an OR node are unordered, and each OR node L (corresponding to a literal) in $S(P,Q)$ has one AND child for each branch in the reduced SLP-tree R_L. Consider an AND node N and the associated branch B_N in R_L. Let the leftmost literals in each node along the branch B_N be $L, L_{1,0}, L_{2,1}, L_{3,2}, \ldots L_{k,k-1}$. The AND node N has k children, ordered from left-to-right (to reflect the order in which subgoals are solved), where $L_{i,i-1}$ is the i'th child of N. □

Definition 3.6 (Depends On) We say that the solution of subgoal Q_1 *depends on* the solution of another subgoal Q_2, if there is a path from Q_1 to Q_2 in the subgoal dependency graph $S(P,Q)$. □

When the solution of subgoal Q_1 depends on the solution of subgoal Q_2, we loosely use the notation "subgoal Q_1 depends on subgoal Q_2". (Note that, unlike Ross [75], we do *not* distinguish between positive and negative dependencies.)

3.3.2 Modular Stratification: Syntax and Semantics

Definition 3.7 (Stratification) A program is *stratified* if there is an assignment of ordinal levels to predicates such that whenever a predicate occurs negatively in the body of a rule, the predicate in the head of that rule is of strictly higher level, and whenever a predicate appears positively in the body of a rule, the predicate in the head of that rule has at least that level. □

Example 3.4 (Stratified Negation) Consider, for instance, the following program.

$$\begin{array}{ll} r1 : nocyc(X,Y) & :- tc(X,Y), not\ tc(Y,X). \\ r2 : tc(X,Y) & :- edge(X,Y). \\ r3 : tc(X,Y) & :- edge(X,Z), tc(Z,Y). \end{array}$$

The database consists of the *edge* relation, which contains the edges in a directed graph. The relation *tc* (computed by the program) contains the transitive closure of the *edge* relation, and the *nocyc* relation contains pairs of vertices such that there are no cycles in the graph between those pairs. Intuitively, this program is stratified since the definition of the *nocyc* predicate depends negatively on the definition of the *tc* predicate, but the definition of the *tc* predicate does not depend on the definition of the *nocyc* predicate. The *edge* and *tc* predicates can be assigned an ordinal level of 0, and the *nocyc* predicate can be assigned an ordinal level of 1. This assignment is consistent with the definition of stratification. □

Unlike positive programs, programs with negative body literals cannot be assigned a unique minimal model semantics, because such programs may not have a unique minimal model. Consider the above program. If the *edge* relation consists of the single fact $edge(1,2)$, the program has two minimal models: $M1 = \{edge(1,2), tc(1,2), nocyc(1,2)\}$ and $M2 = \{edge(1,2), tc(1,2), tc(2,1), tc(1,1)\}$. (Each of the interpretations $M1$ and $M2$ is a minimal model because if any fact is removed from the interpretation, it ceases to be a model of the program.)

Example 3.5 (Stratified Aggregation) A problem similar to negation arises for programs with grouping atoms and aggregation; such programs may not have a unique minimal model. Consider the following program:

$$\begin{array}{l} p(X, sum(<Y>)) :- q(X,Y). \\ q(1,2). \end{array}$$

The following two interpretations are both minimal models of the program: $M3 = \{q(1,2), p(1,2)\}$ and $M4 = \{q(1,2), q(1,3), p(1,5)\}$. (Each of the interpretations $M3$ and $M4$ is a minimal model because if any fact is removed from the interpretation, it ceases to be a model of the program.)

The definition of stratification can also be extended to programs with grouping atoms and aggregation. Intuitively, a program with grouping atoms and aggregation is stratified if there is no cyclic dependency between predicates "through" aggregation. (This can be formalized using the notion of assignments of ordinal levels to program predicates as before.) The solutions proposed for stratified negation also apply to programs with stratified aggregation. □

Apt, Blair and Walker [2] and Van Gelder [94] independently proposed a semantics for stratified programs. We give a brief description of their semantics for stratified programs, and refer the reader to [2, 94] for formal definitions. The intuitive idea is to choose one of the minimal models of the entire program, based on the assignment of ordinal levels to predicates and an "iterated minimal model" construction procedure. Consider the above program defining the predicate *nocyc* and the following assignment of ordinal levels to program predicates: the *edge* and *tc* predicates are assigned an ordinal level of 0, and the *nocyc* predicate is assigned an ordinal level of 1.

First, the procedure constructs a unique minimal model for the predicates at ordinal level 0, i.e., the *edge* and *tc* predicates. (By definition, the rules defining these predicates, i.e., rules $r2$ and $r3$ along with the database facts for the *edge* relation, constitute a positive program, which has a unique minimal model.) Having fixed the meaning of the *edge* and *tc* predicates, the procedure constructs a unique minimal model for the predicates at ordinal level 1, i.e., the *nocyc* predicate. (The single rule defining this predicate, i.e., rule $r1$, can be thought of constituting a positive program, since the only negative body literal *not* $tc(Y, X)$ corresponds to a predicate not defined in the program; this can be effectively thought of as a "database" predicate, where *not* $tc(a, b)$ is true if and only if $tc(a, b)$ is not present in the tc relation.) If the *edge* relation consists of the single fact $edge(1, 2)$, the meaning of the program (according to this semantics) is given by $M1 = \{edge(1, 2), tc(1, 2), nocyc(1, 2)\}$.

The iterated minimal model construction procedure clearly depends on the assignment of ordinal levels to predicates. Given a stratified program, there may be several possible assignments of ordinal levels that are consistent with the definition of stratification. For instance, the following assignment of ordinal levels to the program in Example 3.4 is also consistent with the definition of stratification: the *edge* predicate is assigned an ordinal level of 0, the *tc* predicate is assigned an ordinal level of 1, and the *nocyc* predicate is assigned an ordinal level of 2.

One of the main results of Apt et al. [2] was the following theorem.

Theorem 3.1 *Consider a stratified program P and an assignment of ordinal levels to the predicates of P. The "iterated minimal model" procedure constructs a minimal model* *of the program P, and the model constructed is independent of the specific assignment of ordinal levels to the predicates of P.* □

As a consequence of the above result, we can assign a unique meaning to a stratified program.

Definition 3.8 (Meaning of a Stratified Program) Consider a stratified program P with grouping atoms and negative body literals. The *meaning* of the program is the result of the "iterated minimal model" construction procedure for any consistent assignment of ordinal levels to program predicates. □

Definition 3.9 (Herbrand Instantiation of a Program) The *Herbrand instantiation* of a program is the set of rules obtained by substituting terms from the Herbrand universe for variables in the rule in every possible way.

An *instantiated rule* is a rule in the Herbrand instantiation of a program. □

Note that the Herbrand instantiation of a program with a finite number of rules could have an infinite number of rules.

Definition 3.10 (Local Stratification) A program is *locally stratified* if there is an assignment of ordinal levels to *ground atoms* such that whenever a ground atom appears negatively in the body of an instantiated rule, the head of that rule is of strictly higher level, and whenever a ground atom appears positively in the body of an instantiated rule, the head has at least that level. □

Example 3.6 (Locally Stratified Negation) Consider, for instance, the following program.

$even(0).$
$even(s(X)) :- not\ even(X).$

This program computes even numbers, where the non-negative integer n is represented using n occurrences of the function symbol s. This program is not stratified since the definition of the predicate *even* depends negatively on itself. However, it is locally stratified. Intuitively, this is because in the Herbrand instantiation of the above program the number of occurrences of the function symbol s in the head of a rule is one more than the number of occurrences of the function symbol s in the only literal in the body of the rule. Consequently, if each ground atom has the same ordinal level as the number of occurrences of the function symbol s, the condition for local stratification would be satisfied. □

Przymusinski [63] proposed the perfect model semantics for the class of locally stratified programs. We do not present details of the perfect model semantics here, and refer the reader to [63] for formal definitions. As with stratified programs, the intuitive idea is to choose one of the minimal models of the entire program, based on the assignment of ordinal levels to ground atoms and an "iterated minimal model" construction procedure. For locally stratified programs, the iterated minimal model construction procedure operates on the Herbrand instantiation of the program, unlike stratified programs where the procedure operates on the original program directly.

Example 3.7 (Non-locally Stratified Negation) Consider the following variant of the program in Example 3.6.

$r1 : even(0).$
$r2 : even(X) :- s(X, Y), not\ even(Y).$
$r3 : s(1, 0).\ \ s(2, 1).\ \ s(3, 2).$

In the Herbrand instantiation of the above program, one could instantiate the rule $r2$ using the same substitution for the variables X and Y. Consequently, the program is not locally stratified. For instance, the following is an instantiated rule of the above program:

$$even(1) :- s(1, 1), not\ even(1).$$

No assignment of ordinal levels to ground atoms would be consistent with the definition of local stratification, since the ground atom $even(1)$ would have to be assigned a lower level than itself. □

Note however that the instantiated rule contains the ground atom $s(1, 1)$, which is not the head of any instantiated rule. Ross [74] introduced the class of modularly stratified programs, generalizing locally stratified programs, where information about the model of a lower SCC is utilized to give the program a semantics based on an assignment of ordinal levels to ground atoms.

Definition 3.11 (Reduction of an SCC) ([75]) Let S be an SCC of a program, and $Used_S$ be the set of predicates that occur in the bodies of rules of S, but that are not defined in S. Suppose $Used_S$ is fully defined by an interpretation M over the universe $\mathcal{U}$.

Form $I_{\mathcal{U}}(S)$, the instantiation of rules of S with respect to $\mathcal{U}$, by substituting terms from $\mathcal{U}$ for all variables in the rules of S in every possible way. Delete from $I_{\mathcal{U}}(S)$ all rules having a subgoal whose predicate is in $Used_S$, but which is false in M. From the remaining rules, delete all subgoals having predicates in $Used_S$ (which must be true in M) to leave a set of instantiated rules $R_M(S)$. We call $R_M(S)$ the *reduction of S modulo M*. □

Definition 3.12 (Modular Stratification) ([75]) Let $\prec$ be the dependency relation between SCCs of a program. We say the program P is *modularly stratified* if, for every SCC S of P,

1. There is a modularly stratified model M for the union of all SCCs $S' \prec S, S' \neq S$ and
2. The reduction of the rules of S modulo M is locally stratified, with a perfect model $M1$.

   The modularly stratified model for the union of S and all SCCs $S' \prec S$ is given by $M \cup M1$. □

Intuitively, a program is modularly stratified iff its SCCs are locally stratified once all instantiated rules with false subgoals that are defined in a "lower" SCC are removed. Note that the definition of modular stratification depends on the truth values of the predicates defined in lower levels. This property is unlike stratification, where checking that a program is stratified can be done syntactically.

Like stratification, modular stratification can be extended to handle programs with grouping atoms and aggregation. The intuition is similar, and the program in Example 3.3 is an example of a program with modularly stratified aggregation.

Recall that in a range-restricted program, a variable appearing in a negative body literal must also appear in a positive literal. Since the Magic-sets transformation propagates binding information in a left-to-right manner (reflecting a top-down evaluation), each of the variables in a negative body literal must appear to the left in a positive body literal. This condition can be easily satisfied by reordering the body literals in a range-restricted rule. However, this is not sufficient to guarantee freedom from infinite loops through negation in a top-down evaluation of the program. To guarantee this, we must make sure that only the bindings (for predicates in lower SCCs) that "make the reduction of an SCC locally stratified" are passed to negative body literals. The solution is to refine the notion of modular stratification to take account of the left-to-right order of evaluation.

Definition 3.13 (Rule Prefix) A *rule prefix* is formed from a rule with n subgoals in the body by deleting the rightmost m subgoals, where $0 \leq m \leq n$. □

Definition 3.14 (Left-to-right Modular Stratification) ([75]) Let $\prec$ be the dependency relation between SCCs of a program. We say the program P is *left-to-right modularly stratified* if, for every SCC S of P,

1. There is a left-to-right modularly stratified model M for the union of all SCCs $S' \prec S, S' \neq S$, and
2. The reduction of the set of all prefixes of rules of S modulo M is locally stratified, with a perfect model $M1$.

   The left-to-right modularly stratified model for the union of S and all SCCs $S' \prec S$ is given by $M \cup M1$. □

In the subgoal dependency graph for left-to-right modularly stratified programs there is no cyclic dependency involving a negative subgoal. The technique of Ross [75] as well as our technique makes essential use of this property in evaluating programs with left-to-right modularly stratified negation.

Several semantics have been proposed for general logic programs with negation and/or aggregation. These include the well-founded semantics [42, 87, 96, 97], the stable model semantics [31, 42], and the valid semantics [12, 87]. These semantics differ in the meaning they assign to a program when there are cyclic dependencies between subgoals through negation and/or aggregation. The class of programs we consider in this chapter, i.e., left-to-right modularly stratified programs, do not have such cyclic dependencies, and all these various semantics agree on the meaning of the program. In the sequel, we (arbitrarily) refer to the meaning of such programs as its *well-founded semantics.*

Each fact in the well-founded model of a left-to-right modularly stratified program has at least one *derivation tree* that indicates how the fact is derived. We now define derivation trees, which we use extensively in proofs of subsequent theorems.

Definition 3.15 (Derivation Tree) Consider a non-floundering program P and database D. *Derivation trees* are defined recursively and a derivation tree for a fact $p(\overline{c})$ is defined as follows:

- If p is a base predicate, the derivation tree consists of a single node with label $p(\overline{c})$, if this fact is present in D.
- If p is a derived predicate, let r be a rule in P defining p:

$$r : p(\overline{t}) :- p_1(\overline{t_1}), \ldots, p_k(\overline{t_k}), not\ p_{k+1}(\overline{t_{k+1}}), \ldots, not\ p_n(\overline{t_n}).$$

Let $d_i, 1 \leq i \leq k$, be facts with derivation trees T_i, let $d_i, k+1 \leq i \leq n$, be facts with no derivation trees, let θ be a unifier of $(p_1(\overline{t_1}), \ldots, p_k(\overline{t_k}))$ and $(d_1, \ldots, d_k)$, let $p(\overline{c}) = p(\overline{t})[\theta]$, and let $d_i = p(\overline{t_i})[\theta], i > k$.

Then the root of the derivation tree is a node labeled with $p(\overline{c})$ and r, and each $T_i, 1 \leq i \leq k$, is a child of the root. Also, for each $d_i, i > k$, "*not* d_i" is a child of the root.

A *derivation step* for fact $p(\overline{c})$ consists of the rule r, and $d_i, 1 \leq i \leq n$. $\Box$

3.3.3 Modified Magic-sets Rewriting

For the purpose of this chapter, we modify the Magic-sets rewriting as follows:

1. For each magic predicate *m_p* in the Magic-sets transformed program MP, we create a new predicate *done_m_p* with the same arity as *m_p*, which contains those subgoals on p all of whose answers have been computed.

2. For each rule r in MP, and for each negative literal, say *not* $q_i(\overline{t_i})$ in the body of r, we add the literal *done_m_*$q_i(\overline{t_i}^1)$ to the body of r just before the occurrence of *not* $q_i(\overline{t_i})$.

Intuitively, the literal *done_m_*$q_i(\overline{t_i}^1)$ will be satisfied only when the complete set of q_i answers matching $\overline{t_i}^1$ have been computed. Hence, this literal acts as a *guard* on the use of the subsequent negative q_i literal. In a similar fashion, we can also define the modified Supplementary Magic-sets rewriting. In the rest of this chapter, we use SMT(P, Q) to refer to the program obtained by this modified Supplementary Magic-sets transformation of program-query pair $\langle P, Q \rangle$.

Further, when we talk about the dependencies between subgoals in the rewritten program, we refer to the dependencies between subgoals in the original program, before the rewriting has been performed.

3.4 Ordered Search

We now describe our evaluation technique, which we call Ordered Search, that works on the transformed program obtained using Magic-sets or Supplementary Magic-sets rewriting. This technique generates subgoals and answers to subgoals asynchronously, as in bottom-up evaluation, but orders the use of generated subgoals in a manner reminiscent of top-down evaluation, and is in a sense a hybrid between pure (tuple-oriented) top-down evaluation and pure (set-oriented) bottom-up evaluation. We informally describe how Ordered Search works on a transformed program-query pair SMT(P, Q) and provide a detailed algorithmic description in Section A.1.

3.4.1 An Overview

The central data structure used by Ordered Search, the *Context*, is used to preserve "dependency information" between subgoals. Ordered Search can be understood as modifying Semi-naive bottom-up evaluation as follows:

1. Newly generated magic and supplementary facts (if any) are inserted in the *Context* instead of being directly inserted in the differential relations. Consequently, these facts are hidden from the evaluation. (Other newly generated facts are inserted in the differential relations, and made available to the evaluation, as usual.)

2. Magic and supplementary facts from *Context* are *selectively* inserted into the differential relations (i.e., made available for further use by the evaluation) when no new facts can be derived using the current set of facts available to the evaluation, i.e., a fixpoint has been reached. (When a fact in *Context* is made available to the evaluation, it is said to be "marked" on the *Context*.)

3.4.2 Data Structures: *Context*

The *Context* is a sequence of *ContextNodes*. Each *ContextNode* has an associated set of magic facts and supplementary facts, and each magic or supplementary fact is associated with a unique *ContextNode*. A *ContextNode* is said to be "marked" if any magic or supplementary fact associated with the *ContextNode* is marked. The sequence of marked *ContextNodes* is a subsequence of the sequence of *ContextNodes*.

In the rest of this chapter, when we use adjectives like "earlier", "later", etc. to refer to subgoals and *ContextNodes* in *Context*, we mean their position in the sequence and not the time (which might be different) at which these subgoals and nodes were inserted in the sequence.

We now intuitively describe the various operations performed on *Context*:

1. When a new magic or supplementary fact is inserted in *Context*, it is associated with a new *ContextNode*. Facts on *Context* are stored in an ordered fashion, such

that if magic fact Q_1 generates (i.e., depends on) the magic fact Q_2, then Q_2 is stored after or along with Q_1 in the *Context*.

2. On detecting a cyclic dependency between subgoals on the *Context*, the associated *ContextNodes* are collapsed into one *ContextNode*, and all the facts associated with these *ContextNodes* are now kept together. Thus, unlike the stack of subgoals in Prolog evaluation, cyclic dependencies are handled gracefully.

3. When all the answers to a subgoal have been computed, the subgoal is removed from the *Context*.

3.4.3 Algorithms

We give an intuitive description of the Ordered Search technique and in the process make several claims informally. These are formally stated and proved in Section 3.5.

Inserting Facts into *Context*

Newly generated magic and supplementary facts (obtained by applying the Semi-naive rules of the Magic transformed program) are inserted in the *Context* before they are selectively made available to the evaluation. When applying these rules, Ordered Search records which magic or supplementary fact was used to make each derivation. (From the form of rules in the (Supplementary) Magic-sets transformation, there is exactly one such fact.) Let Q_1 be a newly computed magic/supplementary fact derived from magic/supplementary fact Q_2.

- If Q_1 is a magic fact $m\_p(\overline{t_1})$ that has been completely evaluated, it will be present in the *done_m_p* relation.

  In this case, Ordered Search does not insert Q_1 in *Context*.

- Else, since magic/supplementary facts that have been made available for use but have not been completely evaluated are marked in the *Context*, we know that Q_2 occurs as a marked fact in a marked *ContextNode*.

  The fact Q_1 is now inserted in a new unmarked *ContextNode* immediately before the next marked *ContextNode* following the marked *ContextNode* associated with Q_2 in the sequence of *ContextNodes*. (If there is no such marked *ContextNode*,

Q_1 is inserted as the last *ContextNode* in the *Context*.) Thus, Q_1 is inserted after Q_2.

Since Q_2 depends on Q_1, "answers" to Q_1 could be used in computing "answers" to Q_2. Insertion, as above, is used to maintain dependency information between subgoals within the *Context* as a linear sequence. The order in which facts from *Context* are made available to the evaluation will ensure that Q_1 is made available to the evaluation before Q_2 is said to be completely evaluated.

Duplicate elimination is now performed in the *Context* to ensure that there is at most one copy of Q_1 in *Context*. If there is more than one unmarked copy of Q_1 in *Context* at this stage, only the "last" copy of Q_1 is retained. If there is a marked copy of Q_1 in *Context*, i.e., if Q_1 has already been made available to the evaluation, there are two possibilities:

- If the marked copy of Q_1 occurs after the unmarked copy, only the marked copy of Q_1 is retained in *Context*.
- If the unmarked copy of Q_1 occurs after the marked copy, Q_1 depends on itself. We have thus detected a cyclic dependency between the set of all marked subgoals in *Context* in between the two occurrences of Q_1. Ordered Search recognizes this and collapses this set of marked subgoals into the node of the marked copy of Q_1 in *Context*.

Collapsing marked subgoals into a single node when a cyclic dependency is detected is essential to the correctness of the technique in the presence of cycles in the subgoal dependency graph of the original program. (Note that in left-to-right modularly stratified programs there can be positive cyclic dependencies, but no negative ones.) Since all these subgoals (cyclically) depend on each other, we cannot guarantee that any of these subgoals is completely evaluated until we know that all of them have been completely evaluated.

Making Facts Selectively Available

Facts from *Context* are made available to the evaluation only when no new facts can be computed using the set of available facts. If the last *ContextNode* contains at least one unmarked (magic or supplementary) fact, Ordered Search chooses one such unmarked fact, marks it and makes it available to the evaluation by inserting it in the corresponding differential relation. (Note that this fact still remains in the *Context*.)

$$\begin{tikzcd}
\text{m\_even (4)} \arrow[dd, "+"'] \arrow[dr, "-"] & \\
& \text{m\_even (3)} \arrow[dl, "-"'] \arrow[dd, "+"] \\
\text{m\_even (2)} \arrow[dd, "+"'] \arrow[dr, "-"] & \\
& \text{m\_even (1)} \arrow[dl, "-"'] \\
\text{m\_even (0)} &
\end{tikzcd}$$

Figure 3.2: Subgoal Dependencies for Example 3.8

If all facts in the last *ContextNode* are marked, all the facts in the last *ContextNode* can be considered to be completely evaluated. Intuitively, the reason for this is that a set of facts on *Context* (that have been made available to the evaluation) can be considered to be completely evaluated if:

1. no new facts can be generated using the currently available set of facts (i.e., the iterative application has reached a fixpoint), and
2. every magic fact generated from these facts has been completely evaluated.

All these facts are removed from *Context* and all magic facts among these are inserted in the corresponding *done_m_p* relations. The last *ContextNode* is now removed from *Context*. Thus, when a magic fact $m\_p(\overline{t_1})$ on *Context* has been completely evaluated, it is moved to *done_m_p*.

3.4.4 Motivating Examples Revisited

We describe how Ordered Search can be used to evaluate Example 3.1.

Example 3.8 (Modular Negation) Consider the left-to-right modularly stratified program P_{even} from Example 3.1, and the query $?even(4)$. For this program-query pair, the dependencies between subgoals is shown in Figure 3.2.

Iteration No.	Relation	Facts
0	$Context$	$m\_even(4)$
1	$even$	$\{\}$
	m_even	$\{m\_even(4)\}$
	$done\_m\_even$	$\{\}$
	$Context$	$m\_even(4)^*, m\_even(2), m\_even(3)$
2	$even$	$\{\}$
	m_even	$\{m\_even(4), m\_even(3)\}$
	$done\_m\_even$	$\{\}$
	$Context$	$m\_even(4)^*, m\_even(3)^*, m\_even(2), m\_even(1)$
3	$even$	$\{\}$
	m_even	$\{m\_even(4), m\_even(3), m\_even(1)\}$
	$done\_m\_even$	$\{\}$
	$Context$	$m\_even(4)^*, m\_even(3)^*, m\_even(2), m\_even(1)^*,$ $m\_even(0)$
4	$even$	$\{even(0)\}$
	m_even	$\{m\_even(4), m\_even(3), m\_even(1), m\_even(0)\}$
	$done\_m\_even$	$\{\}$
	$Context$	$m\_even(4)^*, m\_even(3)^*, m\_even(2), m\_even(1)^*,$ $m\_even(0)^*$

Table 3.4: Evaluation of $\langle P_{even}, Q_{even}\rangle$ using Ordered Search

Iteration No.	Relation	Facts
5 Inner Loop Fixpoint	$even$	$\{even(0)\}$
	m_even	$\{m\_even(4), m\_even(3), m\_even(1), m\_even(0)\}$
	$done\_m\_even$	$\{m\_even(0)\}$
	$Context$	$m\_even(4)^*, m\_even(3)^*, m\_even(2), m\_even(1)^*$
6 Inner Loop Fixpoint	$even$	$\{even(0)\}$
	m_even	$\{m\_even(4), m\_even(3), m\_even(1), m\_even(0)\}$
	$done\_m\_even$	$\{m\_even(0), m\_even(1)\}$
	$Context$	$m\_even(4)^*, m\_even(3)^*, m\_even(2)$
7	$even$	$\{even(0), even(2)\}$
	m_even	$\{m\_even(4), m\_even(3), m\_even(1), m\_even(0), m\_even(2)\}$
	$done\_m\_even$	$\{m\_even(0), m\_even(1)\}$
	$Context$	$m\_even(4)^*, m\_even(3)^*, m\_even(2)^*$
8	$even$	$\{even(0), even(2), even(4)\}$
	m_even	$\{m\_even(4), m\_even(3), m\_even(1), m\_even(0), m\_even(2)\}$
	$done\_m\_even$	$\{m\_even(0), m\_even(1)\}$
	$Context$	$m\_even(4)^*, m\_even(3)^*, m\_even(2)^*$

Table 3.5: Evaluation of $\langle P_{even}, Q_{even}\rangle$ using Ordered Search (continued)

Iteration No.	Relation	Facts
9 Inner Loop Fixpoint	$even$	$\{even(0), even(2), even(4)\}$
	m_even	$\{m\_even(4), m\_even(3), m\_even(1), m\_even(0), m\_even(2)\}$
	$done\_m\_even$	$\{m\_even(0), m\_even(1), m\_even(2)\}$
	$Context$	$m\_even(4)^*, m\_even(3)^*$
10 Inner Loop Fixpoint	$even$	$\{even(0), even(2), even(4)\}$
	m_even	$\{m\_even(4), m\_even(3), m\_even(1), m\_even(0), m\_even(2)\}$
	$done\_m\_even$	$\{m\_even(0), m\_even(1), m\_even(2), m\_even(3)\}$
	$Context$	$m\_even(4)^*$
11 Inner Loop Fixpoint	$even$	$\{even(0), even(2), even(4)\}$
	m_even	$\{m\_even(4), m\_even(3), m\_even(1), m\_even(0), m\_even(2)\}$
	$done\_m\_even$	$\{m\_even(0), m\_even(1), m\_even(2), m\_even(3), m\_even(4)\}$
	$Context$	

Table 3.6: Evaluation of $\langle P_{even}, Q_{even} \rangle$ using Ordered Search (continued)

The Magic-sets transformed program is as follows:

$$\begin{array}{lll}
r1: & even(X) & :-\ m\_even(X), succ(X,Y1), succ(Y1,Y), even(Y). \\
r2: & even(X) & :-\ m\_even(X), succ(X,Y), done\_m\_even(Y), not\ even(Y). \\
r3: & even(0) & :-\ m\_even(0). \\
mr1: & m\_even(Y) & :-\ m\_even(X), succ(X,Y1), succ(Y1,Y). \\
mr2: & m\_even(Y) & :-\ m\_even(X), succ(X,Y). \\
mr3: & m\_even(4). & \\
\end{array}$$

$succ(1,0).\ \ succ(2,1).\ \ \ldots\ \ succ(n,n-1).$

The evaluation of the rewritten program using Ordered Search could proceed as shown in Tables 3.4, 3.5 and 3.6. The iteration number indicates the iteration of the inner "repeat ... until" loop in Procedure Ordered-Search in Section A.1. It computes and stores only information about direct dependencies as a linear ordering of the magic facts on *Context*; hence, the evaluation uses linear space and makes a linear number of derivations.

Ross [75] proposed a rewriting (SMR) of $\langle P_{even}, Q_{even}\rangle$ in conjunction with a bottom-up method for evaluating the rewritten program. This method explicitly stores all the subgoal dependency information for negative subgoals. For $m = 4$, SMR(P_{even}, Q_{even}) includes the following rules (by folding rules defining supplementary predicates) in addition to other rules that are not relevant for our purpose:

$$\begin{array}{lll}
mr1: & magic(even(Y),+) & :-\ magic(even(X),\_), succ(X,Y1), succ(Y1,Y). \\
mr2: & magic(even(Y),-) & :-\ magic(even(X),\_), succ(X,Y). \\
mr3: & magic(even(4),+). & \\
dr1: & dp(even(X),even(Y)) & :-\ magic(even(X),-), magic(even(X),\_), \\
 & & \quad succ(X,Y1), succ(Y1,Y). \\
dr2: & dn(even(X),even(Y)) & :-\ magic(even(X),-), magic(even(X),\_), \\
 & & \quad succ(X,Y). \\
dr3: & dp(P,even(Y)) & :-\ dp(P,even(X)), magic(even(X),\_), \\
 & & \quad succ(X,Y1), succ(Y1,Y). \\
dr4: & dn(P,even(Y)) & :-\ dp(P,even(X)), magic(even(X),\_), succ(X,Y). \\
\end{array}$$

Ross' algorithm to evaluate SMR(P_{even}, Q_{even}) would compute the facts shown in Table 3.7, in addition to other facts that are not relevant for our purpose.

Ross' approach on this program computes and stores the transitive dependencies in addition to the direct dependencies; consequently, it would use $O(m^2)$ space and make $O(m^2)$ derivations. □

Relation	Facts
$magic(even(X), -)$	$X = 3, 2, 1, 0$
$magic(even(X), +)$	$X = 4, 2, 1, 0$
$dp(even(X), even(Y))$	$(X, Y) = (3, 1), (2, 0)$
$dn(even(X), even(Y))$	$(X, Y) = (3, 2), (2, 1), (1, 0), (3, 0)$

Table 3.7: Evaluation of SMR(P_{even}, Q_{even}) using Ross' Technique

3.5 Theoretical Results about Ordered Search

All results in this section are applicable to programs with function symbols, except where stated otherwise. We also assume that only ground facts are generated in the evaluation. Ordered Search evaluation can be used to correctly evaluate programs with left-to-right modularly stratified negation and aggregation. However, for the sake of simplicity, we prove the correctness and complexity results of Ordered Search evaluation only for programs with negation. The generalization of the proofs to programs with left-to-right modularly stratified aggregation is straightforward.

3.5.1 Soundness, Completeness and Non-repetition

Recall that we use SMT(P, Q) to refer to the modified Supplementary Magic-sets transformation of a left-to-right modularly stratified program-query pair, as described in Section 3.3.3. We first show that an Ordered Search evaluation does not repeat derivation steps.

Theorem 3.2 *Suppose $\langle P, Q\rangle$ is a left-to-right modularly stratified program-query pair. An Ordered Search evaluation of SMT(P, Q) does not repeat derivation steps.*

Proof: In an Ordered Search evaluation of SMT(P, Q), rules are applied as in Semi-naive evaluation. Hence, to show that no derivation steps are repeated in the evaluation, we only need to show that the evaluation eliminates duplicates.

Duplication elimination of answers to subgoals as well as the *done_m* facts is performed during the updates to the delta relations, as in Semi-naive evaluation. (See Procedure Initialize-SN-Relations in Appendix A.1.) However, duplicate elimination of subgoals is *not* done during the updates to the delta relations. Recall that once a magic fact has been generated in an Ordered Search evaluation, it is present either in the *Context*, or in the *done_m* relation. Consider now a newly generated magic fact $m\_p(\overline{t})$.

If it is present in the *done_m_p* relation, it is not inserted in the *Context*. If it is not present in the *done_m_p* relation, it is inserted in the *Context*. Insertion of a subgoal in the *Context* along with duplicate elimination in the *Context* ensures that exactly one copy of a subgoal is retained in the *Context*. This concludes the proof of the result. □

As a corollary to the above result, we have:

Corollary 3.3 *Suppose $\langle P, Q\rangle$ is a left-to-right modularly stratified Datalog program-query pair. An Ordered Search evaluation of SMT(P, Q) terminates. Further, every subgoal that is generated in the Ordered Search evaluation is removed from the Context and moved to the corresponding done_m_p relation during the evaluation.* □

We now prove a couple of lemmas that formalize the claims we made while describing the Ordered Search algorithm. These lemmas are also used to prove the correctness of the Ordered Search evaluation.

Lemma 3.4 *Let $\langle P, Q\rangle$ be a left-to-right modularly stratified program-query pair. Consider an Ordered Search evaluation of SMT(P, Q).*

1. *If Q_1 is a marked subgoal in Context, and Q_2 is any subgoal in Context* after *Q_1, then Q_1 depends on Q_2.*
2. *If Q_1 is a marked subgoal in Context, and Q_2 is any subgoal in Context in the* same *ContextNode as Q_1, then Q_1 and Q_2 depend on each other.*
3. *If Q_1 and Q_2 are subgoals such that Q_1 generates Q_2 in the evaluation, and Q_1 is in the Context when it generates Q_2, then:*
   (a) *Q_2 is made available to the evaluation before Q_1 is removed from the Context.*
   (b) *Either Q_2 is removed from the Context before Q_1 is removed from the Context, or Q_1 and Q_2 are removed from the Context together.*

Proof: We prove the results by induction on the order of operations on subgoals in the *Context*. For the base case, the first operation on the *Context* is the insertion of the magic fact corresponding to the query Q. This subgoal is also marked, and Parts 1–3 of the result hold trivially in this case.

Consider now the induction step. There are three possible states of the *Context* to consider: (1) A newly derived subgoal is inserted into the *Context*, or (2) A fixpoint has been reached, and the last *ContextNode* contains an unmarked subgoal, or (3) A fixpoint has been reached, and the last *ContextNode* contains only marked subgoals. We prove the results for each of these three cases.

Case 1: A newly derived subgoal is inserted into the *Context*.

Induction steps in case 1: Only Parts 1 and 2 need to be shown.

Let Q_2 be the newly computed subgoal, derived from subgoal Q_1, that is inserted in the *Context*. From Theorem 3.2, we know that no derivation is repeated in the Ordered Search evaluation of SMT(P, Q). The insertion procedure inserts Q_2 as an unmarked subgoal after the marked subgoal Q_1, but before the next marked subgoal (if any) after Q_1, in the *Context*. (The fact that it is marked follows from the fact that it is available to the evaluation to generate Q_2.)

Suppose duplicate elimination does not collapse Q_2 into a marked *ContextNode*. From the induction hypothesis (Parts 1 and 2), each marked subgoal before Q_1 or in the same *ContextNode* as Q_1 in the *Context* depends on Q_1. Since Q_1 depends on Q_2, we have shown that each marked subgoal in the *Context* before Q_1 or in the same node as Q_1 in the *Context* depends on Q_2. This completes the induction step for Part 1.

Suppose duplicate elimination collapses Q_2 into a marked *ContextNode*. Collapsing occurs because there is a marked occurrence of Q_2 before Q_1 or in the same node as Q_1. From the induction hypothesis (Parts 1 and 2), each marked subgoal before Q_1 or in the same *ContextNode* as Q_1 in the *Context* depends on Q_1 and consequently, Q_2 depends on Q_1. Since Q_1 also depends on Q_2, we have detected a cyclic dependency between the set of all marked subgoals in *Context* between the two occurrences of Q_2. This completes the induction step for Part 2.

Case 2: A fixpoint has been reached, and the last *ContextNode* contains an unmarked subgoal.

Induction steps in case 2: Only Part 3a needs to be shown, since this is the step when an unmarked subgoal is made available to the evaluation.

Let Q_2 be the unmarked subgoal in the last *ContextNode* that is made available to the evaluation. Let Q_1 be the subgoal that generated this occurrence of Q_2. By the hypothesis of Part 3, Q_1 is present (as a marked subgoal) in the *Context*. Hence, Q_2 is made available to the evaluation before Q_1 is removed from *Context*. This completes the induction step for Part 3a.

Case 3: A fixpoint has been reached, and the last *ContextNode* contains only marked subgoals.

Induction steps in case 3: Only Part 3b needs to be shown, since this is the step when marked subgoals are removed from *Context*.

Let Q_2 be a marked subgoal in the last *ContextNode*. Let Q_1 be the subgoal that generated this occurrence of Q_2. By the hypothesis of Part 3, Q_1 is present (as a marked subgoal) in the *Context*. If Q_2 does not (directly or indirectly) generate Q_1, then Q_1 is present in a previous *ContextNode*, and hence Q_2 is removed from the *Context* before Q_1 is removed from the *Context*. If Q_2 (directly or indirectly) generates Q_1, then Q_1 and Q_2 are present in the same *ContextNode*, and are removed from the *Context* together. This completes the induction step for Part 3b.

This completes the proof of all the parts of the lemma. □

Definition 3.16 (Complete Evaluation of a Subgoal) A subgoal Q is said to be *completely evaluated* at a point in an evaluation, if all "answers" to Q have been computed by that point in the evaluation. □

Lemma 3.5 *Let $\langle P, Q\rangle$ be a left-to-right modularly stratified program-query pair. Consider an Ordered Search evaluation of SMT(P, Q). Let Q_0 be a subgoal in the Context that depends on subgoals $Q_1, \ldots, Q_m$, such that none of the subgoals $Q_1, \ldots, Q_m$ depends on Q_0. If $Q_1, \ldots Q_m$ have been completely evaluated at a point in the Ordered Search evaluation and have been removed from the Context, then Q_0 is removed from the Context only when it has been completely evaluated.*

Proof: In the subgoal dependency graph of a left-to-right modularly stratified program, there are no cycles through negative subgoals, although there can be cycles through positive subgoals. Consequently, if a subgoal Q_0 depends on a subgoal Q_1 negatively, then Q_1 does not depend on Q_0. By the hypothesis of the lemma, all such subgoals have been completely evaluated and have been removed from the *Context* (i.e., they are present in the corresponding *done_m* relations) in the Ordered Search evaluation. Consequently, a single fixpoint suffices to compute all the answers to Q_0, and an Ordered Search evaluation removes Q_0 from the *Context* only on reaching a fixpoint. This completes the proof of the lemma. □

The soundness and completeness results now follow from the exhaustive nature of the evaluation and the correctness of the Supplementary Magic-sets rewriting with the *done_m_p* literals as guards for negative body literals (referred to as SMT rewriting). Note that for programs with function symbols and negation, there is no effective procedure that can guarantee completeness in general. If there is an infinite sequence of subgoals, each depending on the next one in the sequence, and Ordered Search chooses to explore such an infinite path, it may not compute an answer to the original query, even if one exists. Such paths cannot exist for Datalog programs.

Theorem 3.6 *Let $\langle P, Q\rangle$ be a left-to-right modularly stratified Datalog program-query pair. An Ordered Search evaluation of SMT(P, Q) is sound and complete with respect to the well-founded semantics of $\langle P, Q\rangle$.*

Proof: Let $\langle P, Q\rangle$ be a left-to-right modularly stratified Datalog program-query pair, and $S(P, Q)$ be its subgoal dependency graph. We can associate ordinal levels with the subgoals in $S(P, Q)$ based on the SCCs in the subgoal dependency graph. Note that subgoals that depend on each other are given the same ordinal level, whereas if a subgoal Q_1 depends on subgoal Q_2, but Q_2 does not depend on Q_1, then Q_2 is given a lower ordinal level than Q_1. Recall that the subgoal dependency graph of a left-to-right modularly stratified program has no cycles through negative subgoals. Consequently, if subgoal Q_1 depends on subgoal Q_2 negatively, Q_1 has a higher ordinal level than Q_2. With each subgoal, we associate all the "answers" to the subgoal in the well-founded model of $\langle P, Q\rangle$, as well as all the instantiated rules of P having these answers as their head fact.
Claim 1: In an Ordered Search evaluation of SMT(P, Q), a subgoal Q is removed from the *Context* only when it has been completely evaluated.

We prove Claim 1 based on induction on the ordinal levels of subgoals and facts.

We say that $\langle P, m\_p(\overline{c}^1)\rangle$ and $\langle \mathrm{SMT}(P), m\_p(\overline{c}^1)\rangle$ are equivalent if P and $\mathrm{SMT}(P) \cup \{m\_p(\overline{c}^1)\}$ are equivalent with respect to all "answers" to $m\_p(\overline{c}^1)$.

Let $H1(n)$ be the proposition that for all subgoals $m\_p(\overline{c}^1)$ with ordinal levels less than or equal to n, $\langle P, m\_p(\overline{c}^1)\rangle$ and $\langle \mathrm{SMT}(P), m\_p(\overline{c}^1)\rangle$ are equivalent, and Claim 1 holds. As a basis, note that facts with an ordinal level of 0 correspond to base predicates, and the same sets of facts are present in the extensions of base predicates in P as well as in SMT(P). Claim 1 holds trivially. Hence, $H1(0)$ holds.

Assume that $H1(h)$ holds for some h, and consider any subgoal $m\_p_i(\overline{c_i}^1)$ with an ordinal level of $h+1$ in P. We prove the equivalence of $\langle P, m\_p_i(\overline{c_i}^1)\rangle$ and $\langle \mathrm{SMT}(P), m\_p_i(\overline{c_i}^1)\rangle$, and Claim 1 by induction on the heights of derivation trees of facts in P and SMT(P).

We now prove one direction of $H1(h+1)$. Let $H2(n)$ be the proposition that if a fact $p(\overline{c})$ has a derivation tree of height less than or equal to n in P, (1) there is a derivation tree in $\mathrm{SMT}(P) \cup \{m\_p(\overline{c}^1)\}$ for it, and (2) it is derived in an Ordered Search evaluation of $\mathrm{SMT}(P) \cup \{m\_p(\overline{c}^1)\}$. For the basis of $H2$, the set of facts with derivation trees of height one are simply base facts, and they also have derivation trees in SMT(P), and are derived in an Ordered Search evaluation of SMT(P). Hence $H2(1)$ holds.

Assume $H2(n)$ is true for some n, and consider any fact $p(\overline{c})$ with a derivation tree of height $n+1$ in P, where $p(\overline{c})$ has an ordinal level of $h+1$. Let the root of the derivation tree be $\langle p(\overline{c}), R\rangle$, where r is a rule in P, and the corresponding instantiated rule is:

$$r1 : p(\overline{c}) :- p_1(\overline{c_1}), \ldots, p_k(\overline{c_k}).$$

In the instantiated SMT(P), the corresponding rule has an occurrence of the literal $done\_m\_p_i(\overline{c_i}^1)$ before each negative p_i literal in the body of the rule.

For $i = 1, \ldots, k$, if $p_i(\overline{c_i})$ is a negative fact, the ordinal level of $p_i(\overline{c_i})$ is less than $h+1$, and it has no derivation tree in P. By hypothesis $H1(h)$, there exists no derivation tree for $p_i(\overline{c_i})$ in SMT(P) $\cup \{m\_p_i(\overline{c_i}^1)\}$, and if the fact $done\_m\_p_i(\overline{c_i}^1)$ is true, the subgoal $m\_p_i(\overline{c_i}^1)$ has been completely evaluated in the Ordered Search evaluation.

If $p_i(\overline{c_i})$ is not a negative fact, $p_i(\overline{c_i})$ has a derivation tree of height less than or equal to n in P. By hypothesis $H2(n)$, there exists a derivation tree in SMT(P) $\cup \{m\_p_i(\overline{c_i}^1)\}$ for this fact, and it is derived in an Ordered Search evaluation of SMT(P) $\cup \{m\_p_i(\overline{c_i}^1)\}$. Since the magic and *done_m* facts are not available *a priori*, we have to show that they can be computed in an Ordered Search evaluation of SMT(P) $\cup \{m\_p(\overline{c}^1)\}$. We show this by induction on the position of the occurrences of derived predicates in the body of $r1$.

Let $H3(m)$ be the proposition that the magic facts corresponding to the first m derived predicate occurrences in r, and *done_m* facts corresponding to the negative literals in the first m derived predicate occurrences in r are computed in an Ordered Search evaluation of SMT(P) $\cup \{m\_p(\overline{c}^1)\}$.

As a basis, consider the first derived fact in the body, say $p_l(\overline{c_l})$. By construction of SMT(P), there is a rule in SMT(P), say $r2$, with head $m\_p_l(\overline{c_l}^1)$, such that the body contains only base facts and $m\_p(\overline{c}^1)$. The corresponding facts in the body of the rule $r1$ can be used in the body of $r2$ to produce the fact $m\_p_l(\overline{c_l}^1)$. This is a derivation in SMT(P) $\cup \{m\_p(\overline{c}^1)\}$ in an Ordered Search evaluation. If this is a negative literal, $p_l(\overline{c_l})$ has an ordinal level less than $h+1$. From Corollary 3.3 and Lemma 3.4, $done\_m\_p_l(\overline{c_l}^1)$ is generated in the Ordered Search evaluation. The basis $H3(1)$ follows.

Assume $H3(j)$ holds for some j, and let $p_m(\overline{c_m})$ be the $j+1$'th derived predicate occurrence in the body of rule $r1$. By construction SMT(P) contains a rule, say $r3$, with head $m\_p_m(\overline{c_m}^1)$, such that the body contains only base predicates, $m\_p(\overline{c}^1)$, the first j derived predicate occurrences in r, and the *done_m* predicates corresponding to some of these derived predicates (those that are negative). In the Ordered Search evaluation of

SMT(P) $\cup$ $\{m\_p(\bar{c}^1)\}$, the *done_m* facts in the body of $r2$ are generated according to hypothesis $H3(j)$; non-negative facts in the body of $r2$ have derivation trees according to $H2(n)$; and by hypothesis $H1(h)$, no negative fact has a derivation tree. It follows that there is a derivation tree for $m\_p_m(\overline{c_m}^1)$ in SMT(P) $\cup$ $\{m\_p(\bar{c}^1)\}$, and it is derived in an Ordered Search evaluation. By similar arguments as the base case for $H3$, it can be shown that if $p_m(\overline{c_m})$ is negative, $done\_m\_p_m(\overline{c_m}^1)$ is also generated in the Ordered Search evaluation.

This completes the proof of $H3(j+1)$ as well as $H2(n+1)$. Since all the subgoals that depend on $m\_p(\bar{c}^1)$, but such that $m\_p(\bar{c}^1)$ does not depend on them, are fully evaluated, Lemma 3.5 ensures that $m\_p(\bar{c}^1)$ is completely evaluated before it is removed from the *Context*. This completes the proof of one direction of $H1(h+1)$.

We now prove the other direction of $H1(h+1)$. Let $H4(n)$ be the following proposition: for any non-magic fact $p(\bar{c})$ such that the ordinal level of $p(\bar{c})$ (in P) is less than or equal to $h+1$, if $p(\bar{c})$ has a derivation tree of height less than or equal to n in SMT(P)$\cup\{m\_p(\bar{c}^1)\}$, then there is a derivation tree for the same fact in P.

For the basis of $H4$, the set of facts with derivation trees of height one are simply base facts and also have derivation trees in P. Hence $H4(1)$ holds. Assume $H4(n)$ holds for some n, and consider any non-magic fact $p(\bar{c})$, with a derivation tree of height $n+1$ in SMT(P) $\cup$ $\{m\_p(\bar{c}^1)\}$. Let the root of this derivation tree be $\langle p(\bar{c}), R\rangle$. The rules in SMT($P$) for non-magic facts are obtained from rules in P by adding magic facts and *done_m* facts to the rule bodies. Hence, by dropping occurrences of magic facts from r we get a rule, say $r1$, in P.

Consider the rule r used to derive $p(\bar{c})$. By hypothesis $H1(h)$, no derivation trees exist in P for the non-magic negative facts in this rule. By hypothesis $H4(n)$, there exists a derivation tree in P for each of the non-negative non-magic facts in this rule. These non-magic facts can be used with $r1$ to get a derivation tree for $p(\bar{c})$ in P. This completes the proof of $H1(h+1)$, and hence the theorem follows. $\square$

In general, even if there are function symbols, Ordered Search evaluation is always sound with respect to the well-founded semantics, and is complete whenever it terminates.

3.5.2 Space and Time Complexity

In maintaining an auxiliary data structure, the *Context*, Ordered Search uses more space than Semi-naive bottom-up evaluation (which only needs to maintain differential relations). However, as the following results show, there is no increase in asymptotic space complexity compared to other bottom-up evaluation strategies. Intuitively, this is because Ordered Search computes no more facts (asymptotically) than the other bottom-up evaluation strategies, and the space used by the *Context* data structure is proportional to the space used by the subgoals computed.

Theorem 3.7 *Let $\langle P, Q\rangle$ be a positive program-query pair. Let the space taken to evaluate SMT(P, Q) in a Semi-naive bottom-up evaluation be S. Then, an Ordered Search evaluation of SMT(P, Q) takes space $O(S)$.*

Proof: From Theorem 3.2, we know that the same derivations are made in a Semi-naive bottom-up evaluation of SMT(P, Q) and an Ordered Search evaluation of SMT(P, Q). Consequently, the same set of subgoal and answer facts are computed by the two evaluation strategies.

Let S_P be the space taken by the answer facts, and S_M the space taken by the subgoal facts, in a Semi-naive bottom-up evaluation of SMT(P, Q). Then, $S = S_P + S_M$. In an Ordered Search evaluation of SMT(P, Q), the space taken by answer facts remains unchanged. However, each magic fact $m\_p(\bar{c})$ could be stored twice: one copy in the *m_p* relation, and one copy, either in the *done_m_p* relation or in *Context*. (Note that no magic fact can appear both in the *done_m_p* relation and in *Context*.) Similarly, each supplementary fact could be stored twice: one copy in the supplementary relation, and one copy in the *Context*. Further, the space used by the *Context* data structure is proportional to the maximum number of subgoals on *Context*. Hence, the total amount of space taken by subgoal facts in an Ordered Search evaluation of SMT(P, Q) is $O(S_M)$. This concludes the proof of the result. □

Theorem 3.8 *Let $\langle P, Q\rangle$ be a left-to-right modularly stratified program-query pair. Let the space to evaluate $\langle P, Q\rangle$ using Ross' algorithm be S. Then, an Ordered Search evaluation of SMT(P, Q) takes space $O(S)$.*

Proof: Ross' rewriting of $\langle P, Q\rangle$ essentially contains all the rules of SMT(P, Q). (However, these rules in Ross' rewriting do not contain any *done_m* literals in rule bodies.) In addition, Ross' rewriting includes additional rules for inferring positive and negative dependencies between subgoals. Consequently, the set of subgoal and answer facts computed by an Ordered Search evaluation of SMT(P, Q) is the same as the set of subgoal and answer facts computed by Ross' evaluation. Ordered Search computes *done_m* facts in addition, but these take space proportional to the magic facts computed, and do not affect the asymptotic space complexity. Further, the space taken by the *Context* is proportional to the maximum number of subgoals in the *Context*. This concludes the proof of the result. □

Note that Ross' technique may use asymptotically more space than Ordered Search, since it stores transitive dependencies explicitly. For instance, in Example 3.1, Ross' algorithm uses $O(m^2)$ space, whereas Ordered Search uses $O(m)$ space. Our technique for evaluating left-to-right modularly stratified programs is *strictly better* than the algorithm in [75], in terms of the asymptotic space complexity.

We now compare the asymptotic time complexity of the Ordered Search technique with other bottom-up evaluation strategies.

Theorem 3.9 *Let $\langle P, Q\rangle$ be a positive program-query pair. Let the time taken (in terms of asymptotic derivation cost) to evaluate SMT(P, Q) in a Semi-naive bottom-up evaluation be T. Then, an Ordered Search evaluation of SMT(P, Q) takes time $O(T\alpha(T))$.*

Proof: From Theorem 3.2, we know that the same derivations are made in a Semi-naive bottom-up evaluation of SMT(P, Q) and an Ordered Search evaluation of SMT(P, Q). In order to obtain the total time taken by the Ordered Search evaluation in terms of the asymptotic cost of derivations, we need to obtain the cost of each derivation in the Ordered Search evaluation.

Unification of ground facts can be done in constant time using hash-consing for ground terms; indexing and insertion of ground facts in relations can also be done in constant time using hash based indexing (see [69]). Hence, the cost of each derivation depends on the operations on *Context*, and several of these operations are operations on sets: finding the node corresponding to a fact, taking the union of facts associated with nodes on *Context*, and deleting entire sets of facts associated with a *ContextNode*. These operations can be efficiently implemented using the union-find technique [89], with an amortized cost of $O(\alpha(N))$ per operation, where N is the total number of these operations on *Context*, and $\alpha(N)$ is the inverse Ackermann function. Further, the number of operations on *Context* is proportional to the number of derivations in the Ordered Search evaluation.

This completes the proof of the result. □

Theorem 3.10 *Let $\langle P, Q\rangle$ be a left-to-right modularly stratified program-query pair. Let the time to evaluate $\langle P, Q\rangle$ using Ross' algorithm be T. Then, an Ordered Search evaluation of SMT(P, Q) takes time $O(T\alpha(T))$.*

Proof: Ross' rewriting of $\langle P, Q\rangle$ essentially contains all the rules of SMT(P, Q). (In addition, Ross' rewriting includes additional rules for inferring positive and negative dependencies between subgoals.) Consequently, Ordered Search makes no more derivations than Ross' method. As before, the cost of each derivation is proportional to $\alpha(T)$. This completes the proof of the result. □

Since $\alpha(T)$ is very small even for very large values of T, Ordered Search compares favorably in asymptotic (space and time) complexity both to Semi-naive bottom-up evaluation for positive programs, and to Ross' evaluation of left-to-right modularly stratified programs.

Note, however, that Ross' algorithm may make asymptotically more inferences, and hence take asymptotically more time, than Ordered Search since it computes transitive dependencies. For instance, in Example 3.1, Ross' algorithm makes $O(m^2)$ inferences, whereas Ordered Search makes $O(m)$ inferences.

Corollary 3.11 *Let $\langle P, Q\rangle$ be a program-query pair. Then,*

1. *If $\langle P, Q\rangle$ is positive, an Ordered Search evaluation of SMT(P, Q) takes no more time (asymptotically) than the Semi-naive bottom-up evaluation of SMT(P, Q), and*

2. *If $\langle P, Q\rangle$ is left-to-right modularly stratified, then an Ordered Search evaluation of SMT(P, Q) takes no more time (asymptotically) than Ross' method to evaluate $\langle P, Q\rangle$.* □

3.6 Ordered Search in Practice

Ordered Search has been implemented in the Coral deductive database system [65, 68]. In this section, we briefly describe our experience with the Ordered Search evaluation strategy in Coral.

3.6.1 Modules in Coral

A Coral declarative program can be organized as a collection of interacting modules. Modules are the units of evaluation, and the high level module interface allows modules with different evaluation techniques to interact in a transparent fashion; the evaluation of each module is independent of the methods used to evaluate other modules.

Modules export the predicates that they define; a predicate exported by a module is visible to all other modules, and can be used by them in rule bodies. During the evaluation of a rule r in module A, if there is an occurrence of a predicate p_B exported by module B, a query is set up on module B taking into account the bindings on the variables in the p_B literal. The answers to this query are used iteratively in rule r; each time a new answer to the query is required, rule r requests for a new tuple from the interface to module B. The module interface makes no assumptions about the evaluation of the module. Module B may contain only database facts, or may have rules that are evaluated in any of several different ways. The module may choose to cache answers between calls, or choose to recompute answers. All this is transparent to the calling module. Similarly, the evaluation of the called module B makes no assumptions about the evaluation of calling module A. This orthogonality permits the free mixing of different evaluation techniques in different modules in Coral and is central to how different executions in different modules are combined cleanly.

Two basic evaluation strategies are supported in a module, namely pipelining and materialization. Pipelining uses facts "on-the-fly" and does not store them, at the potential cost of recomputation. Materialization stores facts and looks them up to avoid recomputation. Several variants of materialized evaluation are supported: for instance, the usual Semi-naive evaluation which is the default evaluation strategy for modules without negation and aggregation, and Ordered Search, which is chosen for modules with negation or aggregation.

3.6.2 Comparing Alternatives

The inter-module call strategy in Coral suggests an alternate way of evaluating programs with left-to-right modularly stratified negation. Given such a program, it can be automatically rewritten as follows. First, each negative literal *not* $p(\bar{t})$ in a rule body is replaced by *not* $p1(\bar{t})$, where $p1$ is a predicate not defined in the program, and $p1$ has the same arity as p. Second, for each such newly introduced predicate $p1$, we add a module that exports $p1$ and has a single rule of the form:

$$p1(\overline{X}) :- p(\overline{X}).$$

where $\overline{X}$ is a tuple of distinct variables. Finally, if the predicate p is not exported in the original program, p is exported from the module that defines it. The resulting (rewritten) program can now be evaluated without using Ordered Search in any module. Intuitively, the evaluation of the resulting program is correct since intermodule calls corresponding to the newly introduced predicates are made only for negative subgoals, and there is no cycle of negative subgoals in left-to-right modularly stratified programs.

$$\begin{aligned}
r1 &: working(X) &&:- tested(X). \\
r2 &: working(X) &&:- part(X,Y), not\ has\_suspect\_part(X). \\
r3 &: has\_suspect\_part(X) &&:- part(X,Y), not\ working(Y).
\end{aligned}$$

Figure 3.3: The *working* Program

Query distance	$working_1$ (Ordered Search)		$working_2$ (inter-module calls)		$working_3$ (optimized inter-module)	
	Time (secs)	Space (bytes)	Time (secs)	Space (bytes)	Time (secs)	Space (bytes)
10	0.332	12,288	0.258	303,104	0.168	188,416
100	3.109	81,920	2.808	3,014,656	1.437	1,765,376

Table 3.8: Evaluation Alternatives on *working* with Chain Data

In this section, we first compare the performance of the Ordered Search evaluation with this alternative for evaluating programs with left-to-right modularly stratified negation. All performance numbers in this chapter are obtained on a DECstation 5000 rated at 25 MIPS, with 24 megabytes of main memory, running Ultrix 4.2. We use the program in Figure 3.3 (from Ross [74]), and compare alternative evaluation strategies on different input data sets. We use the name $working_1$ to refer to the Ordered Search evaluation of the *working* program, the name $working_2$ to refer to the evaluation of the program rewritten as described above, and the name $working_3$ to refer to the evaluation of the *working* program, where rules $r1$ and $r2$ are in one module, and rule $r3$ is in a separate module. This program is equivalent to the *working* program in Figure 3.3, but it is more efficient than $working_2$. All programs use materialized evaluation.

Our first data set is a simple chain of length 100 for the *part*() relation, with the single sink node in the *tested*() relation. The space and time taken by the alternative evaluation strategies for two queries is shown in Table 3.8. The query distances are from the sink node of the chain. The Ordered Search evaluation (i.e., $working_1$) is about twice as slow as the optimized program relying on inter-module calls (i.e., $working_3$), and about 10% slower than the $working_2$ evaluation. However, the space used by the Ordered Search evaluation is considerably less than the other two evaluation strategies, which set up a large number of inter-module calls to evaluate the two queries.

Query level	$working_1$ (Ordered Search)		$working_2$ (inter-module calls)		$working_3$ (optimized inter-module)	
	Time (secs)	Space (bytes)	Time (secs)	Space (bytes)	Time (secs)	Space (bytes)
1	0.137	4,096	0.301	98,304	0.195	94,208
2	0.648	12,288	7.414	2,478,080	4.496	2,445,312
3	3.168	20,480	—	—	—	—

Table 3.9: Evaluation Alternatives on *working* with Tree Data

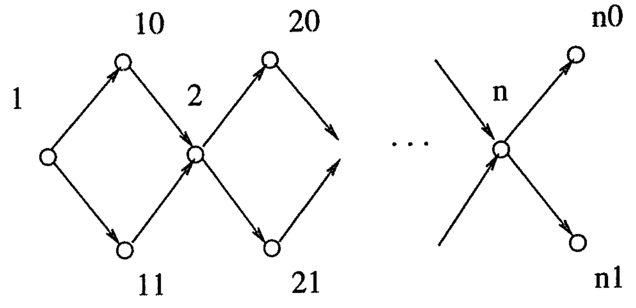

Figure 3.4: The DAG Data Set

Our second data set corresponds to a tree of height 3 with a fanout of 5 for the *part*() relation. All the leaf nodes of the tree are in the *tested*() relation. The space and time taken by the alternative evaluation strategies for three queries is shown in Table 3.9, where the leaf nodes of the tree are at level 0. On this data set, it can be seen that the Ordered Search evaluation is considerably faster than the evaluations that rely on inter-module calls. Also, it uses much less space. No numbers are reported for the evaluation strategies $working_2$ and $working_3$ for the query at level 3 of the tree, since the evaluation exhausted the free store.

With the chain and the tree data sets, neither of the evaluation strategies $working_2$ and $working_3$ set up repeated subqueries, or inter-module calls. When the data for the *part*() relation is a directed acyclic graph, i.e., a part could be a subpart of several different parts, the evaluation strategies relying on inter-module calls repeat computation, unlike the Ordered Search evaluation. This is illustrated by our third and final data set. This data set corresponds to a directed acyclic graph (DAG) for the *part*() relation, shown in Figure 3.4, for $n = 5$. Each of the leaf nodes of the DAG is in the *tested*() relation. The

Query	$working_1$ (Ordered Search)		$working_2$ (inter-module calls)		$working_3$ (optimized inter-module)	
	Time (secs)	Space (bytes)	Time (secs)	Space (bytes)	Time (secs)	Space (bytes)
5	0.070	4,096	0.070	16,384	0.051	16,384
4	0.168	4,096	0.375	114,688	0.211	106,496
3	0.277	8,192	1.406	532,480	0.871	483,328
2	0.383	8,192	5.789	1,974,272	3.472	1,904,640
1	0.469	12,288	23.604	7,729,152	14.151	7,581,696

Table 3.10: Evaluation Alternatives on *working* with DAG Data

$$\begin{aligned}
r1 : bom(Part, sum(<C>)) &:- subpart\_cost(Part, SubPart, C).\\
r2 : subpart\_cost(Part, Part, Cost) &:- basic\_part(Part, Cost).\\
r3 : subpart\_cost(Part, Subpart, Cost) &:- assembly(Part, Subpart, Quantity),\\
&\quad\ bom(Subpart, TotalSubcost),\\
&\quad\ Cost = Quantity * TotalSubcost.
\end{aligned}$$

Figure 3.5: The *bom* Program

space and time taken by the alternative evaluation strategies for five queries is shown in Table 3.10. Ordered Search evaluation does not repeat computation, and this results in its superiority over the evaluations that use inter-module calls (and consequently repeat computation).

The inter-module call strategy can also be used for evaluating programs with left-to-right modularly stratified aggregation. We compare the performance of the Ordered Search evaluation of the bill-of-materials program shown in Figure 3.5 (referred to as bom_1), with the evaluation of the *bom* program with rule $r1$ in one module and rules $r2$ and $r3$ in a separate module (referred to as bom_2), on three data sets. The evaluation of bom_2 relies upon the inter-module calling mechanism rather than Ordered Search to ensure that all *subpart_cost* facts are determined before the *bom* rule is applied. Thus, some operational reasoning is required to understand why this works as expected. Both programs use materialized evaluation.

Query distance	bom_1 (Ordered Search)		bom_2 (optimized inter-module calls)	
	Time (secs)	Space (bytes)	Time (secs)	Space (bytes)
10	0.297	4,096	0.145	57,344
100	2.808	16,384	1.370	520,192
1000	25.709	110,592	—	—

Table 3.11: Evaluation Alternatives on *bom* with Chain Data

Query level	bom_1 (Ordered Search)		bom_2 (optimized inter-module calls)	
	Time (secs)	Space (bytes)	Time (secs)	Space (bytes)
1	0.316	4,096	0.164	57,344
2	3.160	28,672	1.620	561,152
3	31.299	221,184	15.624	5,545,984

Table 3.12: Evaluation Alternatives on *bom* with Tree Data

Our first data set is a simple chain of length 1000 for the *assembly*() relation, with each subpart occurring in a part exactly once. The single sink node is in the *basic_part*() relation, with a cost of 1. Thus, all parts have a total cost of 1. The space and time taken by the alternative evaluation strategies for three queries is shown in Table 3.11. The query distances are from the sink node of the chain. The Ordered Search evaluation (i.e., bom_1) is about twice as slow as the optimized program relying on inter-module calls (i.e., bom_2). However, the space used by the Ordered Search evaluation is considerably less than the other evaluation strategy, which sets up a large number of inter-module calls to evaluate the queries. No numbers are reported for the evaluation strategies bom_2 for the query at distance 1000 from the sink node, since the evaluation exhausted the free store.

Our second data set corresponds to a tree of height 3 with a fanout of 10 for the *assembly*() relation, with each subpart occurring in a part exactly once. (Thus, there are 1110 facts in the *assembly*() relation.) All the leaf nodes of the tree are in the *basic_part*() relation, with a cost of 1. The space and time taken by the alternative evaluation strategies for three queries is shown in Table 3.12, where the leaf nodes of the tree are at level 0. Again, on this data set, it can be seen that the Ordered Search evaluation (i.e., bom_1) is about twice as slow as the evaluation (i.e., bom_2) that relies

Query	bom_1 (Ordered Search)		bom_2 (optimized inter-module calls)	
	Time (secs)	Space (bytes)	Time (secs)	Space (bytes)
9	0.098	4,096	0.051	16,384
7	0.270	4,096	0.242	106,496
5	0.426	8,192	1.203	475,136
3	0.617	8,192	4.781	1,970,176
1	0.773	8,192	19.655	7,909,376

Table 3.13: Evaluation Alternatives on *bom* with DAG Data

on inter-module calls. However, the Ordered Search evaluation uses much less space, as before.

With the chain and the tree data sets, the evaluation strategy bom_2 does not set up repeated subqueries, or inter-module calls. Our third data set is a directed acyclic graph for the *assembly*() relation, shown in Figure 3.4, for $n = 9$. Each of the leaf nodes of the DAG is in the *basic_part*() relation, with a cost of 1. The space and time taken by the alternative evaluation strategies for five queries is shown in Table 3.13. While the time taken by the Ordered Search evaluation grows linearly with the distance of the query from the sink node, the time and space taken by the inter-module call evaluation grows exponentially with the distance of the query from the sink node. As with negation, Ordered Search evaluation does not repeat computation, and this results in its considerable superiority, in both space and time, over the evaluation that uses inter-module calls, and consequently repeats computation.

3.6.3 Overheads of Ordered Search

Ordered Search evaluation uses more space and time than Semi-naive bottom-up evaluation for programs without negation or aggregation. There are two main reasons for the overheads of Ordered Search:

1. Ordered Search maintains an auxiliary data structure, the *Context*, to record dependencies between subgoals generated in the evaluation, and maintains *done_m* facts that correspond to subgoals that have been completely evaluated. As a consequence, Ordered Search uses more space and time than Semi-naive bottom-up evaluation, which only needs to maintain and update differential relations.

$$\begin{array}{ll} append([], X, X) & :- m\_append([], X). \\ append([X|Y], Z, [X|W]) & :- m\_append([X|Y], Z), append(Y, Z, W). \\ m\_append(Y, Z) & :- m\_append([X|Y], Z). \end{array}$$

Figure 3.6: Magic-sets Transformation of the *append* Program

List length	$append_1$ (Semi-naive)		$append_2$ (Semi-naive w/o SCC-by-SCC)		$append_3$ (Ordered Search)	
	Time (secs)	Space (bytes)	Time (secs)	Space (bytes)	Time (secs)	Space (bytes)
$400 + 400$	3.418	69,632	4.422	73,728	5.082	77,824
$2400 + 2400$	21.108	380,928	27.162	425,984	31.318	430,080

Table 3.14: Space and Time Overheads of Ordered Search on *append*

2. In an Ordered Search evaluation, *all* program rules are applied in each iteration, unlike a Semi-naive bottom-up evaluation of a positive program that partitions the rules of the program and evaluates it one strongly connected component (SCC) at a time. As a consequence, the number of Semi-naive rules (see Section 2.2) generated in the Ordered Search evaluation are more than the number of Semi-naive rules generated in the Semi-naive evaluation with the SCC-by-SCC optimization, While this does not affect the number of inferences made, it makes the processing less set-oriented, with the inferences distributed over a larger number of Semi-naive rule applications.

In this section, we examine the effect of these overheads on the efficiency of Ordered Search evaluation in the Coral system using a few example programs without negation or aggregation.

Our first program is the Magic-sets rewriting of the *append* program, which is given in Figure 3.6. This is a positive program, and consists of two strongly connected components, one containing the *m_append* predicate, and the other containing the *append* predicate. Table 3.14 gives the space and time taken to evaluate *append* using three different memoing strategies for answering queries where the first two arguments of *append* are bound, and the third argument is free: $append_1$ corresponds to an SCC-by-SCC Semi-naive bottom-up evaluation of the *append* program, $append_2$ corresponds to a Semi-naive

$$
\begin{aligned}
anc\_r(Ancestor, Descendant) &:- m\_anc\_r(Ancestor),\\
&\quad\ parent(Ancestor, Descendant).\\
sup\_2\_1(Ancestor, Child) &:- m\_anc\_r(Ancestor), parent(Ancestor, Child).\\
m\_anc\_r(Child) &:- sup\_2\_1(Ancestor, Child).\\
anc\_r(Ancestor, Descendant) &:- sup\_2\_1(Ancestor, Child),\\
&\quad\ anc\_r(Child, Descendant).
\end{aligned}
$$

Figure 3.7: SMT($P_{ancestor}$)

Tree level	anc_r_1 (Semi-naive)	anc_r_2 (Semi-naive w/o SCC-by-SCC)	anc_r_3 (Ordered Search)
	Time (secs)	Time (secs)	Time (secs)
2	0.539	0.676	1.312
3	6.589	7.578	14.737

Table 3.15: Time Overheads of Ordered Search on *anc_r*

bottom-up evaluation that evaluates the *append* program without the SCC-by-SCC optimization, and *append*$_3$ corresponds to an Ordered Search evaluation of the *append* program.

The Ordered Search evaluation of the *append* program is about 50% slower than the SCC-by-SCC Semi-naive, bottom-up evaluation, and takes about 10% to 15% more space. However, a significant fraction of these overheads is due to the absence of the SCC-by-SCC optimization in the Ordered Search evaluation, as is illustrated by the space and time numbers for the *append*$_2$ evaluation; the overheads due to the maintenance of additional data structures by Ordered Search contributes less than half of the space and time overheads.

The actual overheads of Ordered Search clearly depend on the data. In the *append* program, for instance, each iteration of the Semi-naive bottom-up evaluation (with or without the SCC-by-SCC optimization), as well as the Ordered Search evaluation computes exactly one new fact. Thus, the number of inferences made is the same as the number of rule applications. When the data is such that Semi-naive, bottom-up evaluation is more set-oriented than the Ordered Search evaluation, then Ordered Search has the overhead of *additional* rule applications needed to make the *same* set of inferences.

This is illustrated in our second program, which is the Supplementary Magic-sets rewriting of the right-linear "*ancestor*" program (without the right-linear, or factoring, optimization), given in Figure 3.7. The data corresponds to a 3-level tree with a fanout of 10. Table 3.15 gives the time taken to evaluate *anc_r* using three different memoing strategies for answering queries where the first argument is bound, and the second argument is free: anc_r_1 corresponds to an SCC-by-SCC Semi-naive, bottom-up evaluation of the *anc_r* program, anc_r_2 corresponds to a Semi-naive, bottom-up evaluation that evaluates the *anc_r* program without the SCC-by-SCC optimization, and anc_r_3 corresponds to an Ordered Search evaluation of the *anc_r* program. The queries ask for all descendants at two different levels of the tree; the query at level 2 has 110 answers, and the query at level 3 has 1110 answers. (The numbers in Table 3.15 exclude the time taken to print answers.) The Ordered Search evaluation is slightly more than twice as slow as the SCC-by-SCC, Semi-naive, bottom-up evaluation, and slightly less than twice as slow as the Semi-naive, bottom-up evaluation without the SCC-by-SCC optimization. The space overheads of the Ordered Search evaluation for the *anc_r* program are less than those for the *append* program.

3.6.4 Ordered Search and Persistence

Coral provides support for persistent data using the EXODUS client-server database toolkit [19]. Coral is the client process, and maintains buffers for persistent relations. Data stored using the EXODUS storage manager is paged into these buffers on demand, making use of the indexing and scan facilities of the storage manager.

The Ordered Search implementation is orthogonal to the implementation of persistent relations. Consequently, Ordered Search evaluation in Coral can be used with base and derived relations that are persistent. However, the current implementation assumes that the *Context* data structure itself fits in memory. This is usually not a problem since the size of the *Context* is proportional to the length of the longest path in the subgoal dependency graph, and not to the size of the database which can be much larger than the subgoal dependency graph. However, if the *Context* data structure does not fit in memory, it would have to be stored using the EXODUS storage manager. The main issue that arises is the structuring of the *Context* data structure, so that portions of it can be brought into memory on demand. We have not addressed this in the current Coral implementation.

3.7 Related Work

Ordered Search compares favorably with other top-down and bottom-up methods for evaluating logic programs in the literature. In earlier sections, we have presented a detailed comparison with Semi-naive bottom-up evaluation and with Ross' technique to evaluate left-to-right modularly stratified programs. We present a brief comparison with other techniques below.

3.7.1 Prolog

Ordered Search is sound, complete for Datalog and does not repeat derivations. Prolog is not complete even for Datalog, and may repeat derivations. Also, Prolog does not evaluate the class of left-to-right modularly stratified programs correctly.[2]

3.7.2 QSQR/QoSaQ and Extension Tables

Extension Tables [26] is similar to Prolog, except that it memos facts and subgoals and can detect loops. QSQR/QoSaQ [98, 99] is a top-down, memoing, set-oriented strategy that is closely related to bottom-up evaluation with Supplementary Magic rewriting. Like Prolog, these techniques cannot deal with left-to-right modularly stratified negation/aggregation. The tuple-oriented search strategy of the Extension Tables variant ET* is closer to Prolog, but it repeats computation.

Ross also describes how his approach can be used to adapt QSQR to deal with left-to-right modularly stratified negation. In this case as well, dependencies between subgoals are maintained transitively, and our previous comparisons also apply to this case.

3.7.3 Subquery Completion

A variant of QSQR, *subquery completion*, was described in Lefebvre [48] to deal with recursively defined aggregates. It uses the dependencies between subgoals maintained by QSQR to handle a class of acyclic programs with aggregation. However, this technique does not deal with programs that have cycles in the subgoal dependency graph of a strongly connected component with aggregates (even if the cyclic dependency is only between positive subgoals). Ordered Search allows positive cycles in the subgoal dependency graph, and deals with them by collapsing nodes in the *Context*, and declaring all

[2]Of course, a meta-interpreter can be written using Prolog to evaluate such programs.

the facts in a collapsed node to be completely evaluated once a fixpoint is reached. There is no analogue to this step in the technique of [48].

3.7.4 Techniques for computing the well-founded model

There are several query evaluation techniques in the literature that compute answers under the well-founded model. For example, WELL! [14] is based on global SLS-resolution, XOLDTNF [21] is an extension of OLDT resolution, GUUS [49] is based on the alternating fixpoint semantics, and the techniques of Kemp et al. [40, 41, 43] and Morishita [53] are based on alternating fixpoint semantics and Magic-sets. The class of programs handled by these techniques is larger than that handled by Ordered Search, but each of these techniques can repeat computation even for left-to-right modularly stratified programs. This can result in a loss of efficiency of evaluation.

There are other proposed techniques that control the order of inferences in a bottom-up evaluation in some way. Sloppy Delta Iteration [78] provides a way to "hide" facts until they are to be used. Techniques for hiding facts are used in [29, 85] to evaluate programs with aggregate operations efficiently. These results are only tangentially related to Ordered Search since the (motivation as well as the nature of the) orderings considered are quite different.

3.8 Discussion

We presented a memoing technique, Ordered Search, that is a hybrid between breadth-first and depth-first search, and we discussed how Ordered Search can be used to evaluate programs with left-to-right modularly stratified negation, that generate ground facts. Fully set-oriented computation causes problems for the evaluation of left-to-right modularly stratified programs, as illustrated by our comparisons with Ross [75]; it can result in an order of magnitude slow-down. Hence, it is important to provide some of the benefits of tuple-at-a-time computation with bottom-up evaluation, and Ordered Search does just this.

Ordered Search can also be used to evaluate programs with left-to-right modularly stratified aggregation; the algorithms described in Appendix A.1 and the Coral implementation [68] deal with programs with aggregation as well as programs with negation. Ordered Search can also be used for programs that compute non-ground facts; details are omitted from this thesis for the purpose of simplicity.

We now discuss how the search strategy of Ordered Search is useful in evaluating single answer queries (Section 3.8.1). We also briefly describe how a related strategy, of ordering rule applications to control the order in which inferences are performed, can be evaluated efficiently (Section 3.8.2).

3.8.1 Single Answer Queries

When only a single answer to the query is desired, the order in which facts are generated and used becomes important, and the depth-first search strategy of a top-down evaluation scheme such as Prolog can perform much better than the breadth-first search strategy of bottom-up evaluation methods. In this section, we show that Ordered Search can also be used for optimizing single-answer queries for linear programs by restricting the search space.

Example 3.9 (Obtaining a Single Answer) There are many cases where the user may want a single answer to a query. Consider, for example, the following program-query pair $\langle P_{path}, Q_{path} \rangle$.

$$\begin{array}{l} r1 : path(X, Y, [X, Y]) :- edge(X, Y). \\ r2 : path(X, Y, [X|P]) :- edge(X, Z), path(Z, Y, P). \\ edge(1, 2).\ \ edge(1, 3).\ \ edge(2, 1).\ \ edge(2, 4).\ \ edge(3, 4). \\ \quad ?\ path(1, 4, X). \end{array}$$

A top-down, tuple-oriented evaluation strategy, like Prolog, would set up a query on *path*, and solve the subgoals in a depth-first fashion. However, since there is a cycle in the *edge* relation, Prolog would not terminate on the given query.

One way of obtaining a single answer to the query is to evaluate the Magic-sets transformed program bottom-up until we get an answer to the query, and then terminate the evaluation. With this approach, subgoals are solved in parallel as they are generated.

Ordered Search solves subgoals in a depth-first fashion for this program, but since it performs memoing, it does not repeat computation, and terminates on this program. In general, it provides an alternative evaluation strategy to the breadth-first strategy of bottom-up evaluation. For many programs (the above program with the given data is one such) a depth-first search for one answer is much more efficient than a breadth-first search for one answer.

For this program-query pair, the subgoal dependencies are shown in Figure 3.8. Note that the subgoal dependency graph has a cycle; consequently, Prolog would not terminate on this example program-query pair.

$$\begin{tikzcd}
 & \text{m\_path }(1,4) \arrow[dl, bend right] \arrow[dr] & \\
\text{m\_path }(2,4) \arrow[ur, bend right] & & \text{m\_path }(3,4)
\end{tikzcd}$$

Figure 3.8: Subgoal Dependencies for *path* Program

Iteration No.	Relation	Facts
0	*path*	$\{\}$
	m_path	$\{\}$
	Context	$m\_path(1,4)$
1	*path*	$\{\}$
	m_path	$\{m\_path(1,4)\}$
	Context	$m\_path(1,4)^*, m\_path(3,4), m\_path(2,4)$
2	*path*	$\{path(2,4,[2,4])\}$
	m_path	$\{m\_path(1,4), m\_path(2,4)\}$
	Context	$\{m\_path(1,4)^*, m\_path(2,4)^*\}, m\_path(3,4)$
3	*path*	$\{path(2,4,[2,4]), path(1,4,[1,2,4])\}$
	m_path	$\{m\_path(1,4), m\_path(2,4)\}$
	Context	$\{m\_path(1,4)^*, m\_path(2,4)^*\} m\_path(3,4)$

Table 3.16: Ordered Search evaluation of $\langle P_{path}, Q_{path} \rangle$

The Magic-sets transformed program is straightforward and we do not describe it further. We describe the evaluation of $\langle P_{path}, Q_{path}\rangle$ using Ordered Search briefly in Table 3.16. Facts in *Context* marked with an * indicate facts made available to the evaluation, and facts in *Context* within { } indicate facts associated with a single *ContextNode*. Note that an answer is produced in iteration 3, as in the Semi-naive bottom-up evaluation of the Magic-sets transformation of $\langle P_{path}, Q_{path}\rangle$. However, the evaluation using Ordered Search has computed fewer facts than would be computed by pure bottom-up evaluation. Also note that a cycle was detected since *m_path*(1, 4) was derived from *m_path*(2, 4), and this magic fact occurs with an * earlier in *Context*. As a result, in iteration 2, several nodes in *Context* have been collapsed together. □

Recall that bottom-up evaluation of a Magic-sets transformed program generates subgoals and answers to the subgoals as in a top-down evaluation, although the order in which these are generated in the bottom-up evaluation may be quite different from a top-down evaluation. By ordering the newly generated facts in *Context*, Ordered Search makes facts selectively available to the evaluation in a manner considerably different from pure bottom-up evaluation. The order in which generated subgoals (magic facts) are selected to be used by Ordered Search is related to a top-down evaluation as described by the following result.

Proposition 3.12 *Suppose $\langle P, Q\rangle$ is a left-to-right modularly stratified program-query pair. In an Ordered Search evaluation of SMT(P, Q), the order in which magic facts are marked corresponds to a* depth-first traversal *(with marking) of the subgoal dependency graph of $\langle P, Q\rangle$ starting from Q.* □

The order in which Prolog explores the subgoal dependency graph also corresponds to a depth-first traversal, although Prolog does not "mark" nodes, and hence may repeat computation. After generating an answer for a subgoal generated from a rule literal, Prolog continues with the next rule body literal, before attempting to generate more answers for the first subgoal. Ordered Search, on the other hand, generates all answers for the first subgoal before trying to solve subgoals generated from the next rule body literal. Consequently, Prolog may perform a lot less computation than Ordered Search in obtaining a single answer to the query. For linear programs, however, delaying the availability of subgoals to the Ordered Search evaluation does not delay the computation of the first answer to the query (because of the asynchronous way in which answers are generated).

We conjecture that Ordered Search is most useful for computing single answers to a query for the class of linear programs that may have cyclic subgoals (and hence Prolog is not suitable).

3.8.2 Ordering Rules

The order of inferences in a bottom-up evaluation can also be controlled by using regular expressions over rules to specify orderings of the application of rules. Rule orderings are significant for several reasons.

1. Rule orderings have been proposed to prune redundant derivations and to allow the user to specify a desired semantics [33, 34, 35].

2. Rule ordering can result in increased efficiency.

   For example, in an SCC-by-SCC evaluation of a program, rules in lower SCCs do not need to be considered while applying rules in higher SCCs; this can improve the efficiency of evaluation. Ordering rules within an SCC can also improve efficiency by further reducing the number of rule applications. Although the orderings do not affect the number of inferences made, the processing becomes more set-oriented, with each rule application generating more facts.

3. Rule orderings have been proposed as a way of evaluating the (non-stratified) Magic-sets transformation of a stratified program [13].

The usual Semi-naive algorithm is capable of evaluating a limited number of rule orderings, e.g. those that result in an SCC-by-SCC evaluation. However, it cannot correctly evaluate regular expressions over rules that order rule applications within an SCC.

In [67], we present two fixpoint algorithms that address the issue of how to apply rules in a specified order without repeating inferences. One of them, General Semi-naive (GSN), applies a rule to produce new facts, and then immediately makes these facts available to subsequent applications of other rules (possibly in the same iteration). GSN is capable of dealing with a wide range of rule orderings but with a little more overhead than the usual Semi-naive evaluation. The other algorithm we present, Predicate Semi-naive (PSN), can utilize facts produced for a predicate p in the same iteration they have been derived in, although not always in the immediately following rule application. It handles a more restricted set of control expressions compared to GSN, but is cheaper than GSN. In fact, it has no additional overheads compared to the usual Semi-naive evaluation.

We also study rule orderings in detail in [67], and establish a close connection between cycles in rule graphs (which are a variant of rule/goal graphs defined in [8, 92]) and orderings that minimize the number of iterations and rule applications. We define what it means for a rule ordering to preserve a simple cycle, and show that a rule ordering that preserves all simple cycles in the rule graph (if such an ordering exists) is optimal within a certain class of rule orderings in minimizing the number of iterations, and hence the number of rule applications and joins.

Chapter 4

Propagating Constraint Selections

4.1 Background

Recently, there have been attempts ([9, 23, 39, 70], among others) to increase the expressive power of database query languages by integrating constraint paradigms with logic-based database query languages; such languages are referred to as *constraint query languages* (CQLs). Evaluating such programs can be expensive due to the manipulation of constraints, and hence optimizing such programs is very important. We consider the following problem in this chapter: How can we optimize a CQL program-query pair $\langle P, Q \rangle$ by propagating constraints occurring in P and Q? More precisely, the problem is to find a set of constraints for each predicate such that the following statements hold:

- Adding the corresponding set of constraints to the body of each rule defining a predicate yields a program P' such that $\langle P, Q \rangle$ is query equivalent to $\langle P', Q \rangle$ (on all input EDBs), and
- Only facts that are *constraint relevant* to $\langle P, Q \rangle$ are computed in a bottom-up evaluation of $\langle P', Q \rangle$ on an input EDB.

Constraint sets that satisfy the first condition are called query-relevant predicate (QRP) constraints; those that satisfy both conditions are called minimum QRP-constraints.[1]

[1]There is indeed a minimum QRP-constraint, as we show later. Since we treat a constraint as equivalent to its set of ground instances, this definition is independent of the exact representation of the constraint set.

The notion of *constraint relevance* is introduced to capture the information in the constraints present in P and Q. (We note that *every* fact for a predicate that appears in P or Q is constraint relevant if neither P nor Q contains constraints.) Identifying and propagating QRP-constraints is useful in two distinct situations:

- Often, it is possible to evaluate queries on CQL programs without actually generating constraint facts. The constraints in the program are used to prune derivations, and only ground facts are generated.

  If only Magic-sets transformation is used to optimize CQL programs, this could lead to the generation of constraint facts, even when the evaluation of the original program generates only ground facts.[2] A motivation for this work, as for Balbin et al. [4] and Mumick et al. [56], is to take advantage of the constraints present in the program to reduce the potentially relevant facts computed, and yet compute only ground facts during the bottom-up evaluation of the rewritten program.

- Even when constraint facts are generated, we may ensure termination in evaluating queries on CQL programs that would not have terminated if these constraints had not been propagated.

4.1.1 Motivating Examples

Let us describe some examples that motivate our results.

Example 4.1 (Flight Connections) Consider the following program P:

$$\begin{array}{ll}
r1: \mathit{cheaporshort}(S,D,T,C) & :-\ \mathit{flight}(S,D,T,C), T \leq 240. \\
r2: \mathit{cheaporshort}(S,D,T,C) & :-\ \mathit{flight}(S,D,T,C), C \leq 150. \\
r3: \mathit{flight}(Src,Dst,Time,Cost) & :-\ \mathit{leg}(Src,Dst,Time,Cost), \\
 & \qquad Cost > 0, Time > 0. \\
r4: \mathit{flight}(S,D,T,C) & :-\ \mathit{flight}(S,D1,T1,C1), \mathit{flight}(D1,D,T2,C2), \\
 & \qquad T = T1 + T2 + 30, C = C1 + C2.
\end{array}$$

[2]The Magic-sets algorithm described in Section 2.3 considers an argument of a subgoal "bound," only if it is ground. One could also treat an argument "bound," if it is potentially restricted; this is the approach taken by Ramakrishnan [64]. With this interpretation of bound arguments, it is possible for the Magic-sets transformation to generate non-range-restricted rules defining magic predicates; the bottom-up evaluation of such programs would compute non-ground constraint facts.

The query predicate is *cheaporshort*, and *Cost* and *Time* fields in *leg* are values drawn from the reals. Although P is a CQL program, each fact computed in the bottom-up evaluation of P is just a tuple of constants; no constraint fact is computed. Given a query, for instance:

$$?\ cheaporshort(madison, seattle, Time, Cost).$$

one would like to compute only relevant *flight* facts: clearly, *flight* facts that cost more than \$150 and take more than 240 minutes are *not relevant* to answering this query and, hence, need not be computed in answering the query.

Recall that Magic-sets seeks to restrict the computation to facts that are relevant to answering a query. The Magic-sets rewriting of the above program P can take one of two approaches.

1. Use the constraints in the bodies of rules such as $r1$ to restrict computation of magic facts. (In this case, an argument of a subgoal is treated as "bound" if it is potentially restricted, not necessarily ground.) For instance, the magic rule obtained from $r1$ could be:

   $$mr1 : m\_flight(S, D, T) : -\ m\_cheaporshort(S, D), T \leq 240.$$

   The (bottom-up) evaluation of this rule would require computing constraint facts. For instance, given the above query, the fact $m\_flight(madison, seattle, T; T \leq 240)$ would be computed using rule $mr1$. (See Section 4.3 for the notation used for constraint facts in this chapter.) Using such constraint facts in a bottom-up evaluation is likely to be more expensive than using only ground facts in the evaluation.

2. One can compute only ground facts safely in the magic program by *not* making use of the constraints in the body of rule $r1$ to restrict computation of magic facts. For instance, the magic rule obtained from $r1$ in this case, would be:

   $$mr1' : m\_flight(S, D) : -\ m\_cheaporshort(S, D).$$

   The bottom-up evaluation of the magic program which includes rule $mr1'$ would compute many irrelevant facts since not all available constraints are made use of in the magic program. In particular, given the query:

   $$?\ cheaporshort(madison, seattle, Time, Cost).$$

one could compute many *flight* facts with $Cost > 150$ and $Time > 240$; these facts are not relevant to answering the query. □

The rewriting techniques proposed by Balbin et al. [4] and Mumick et al. [56] would not be able to optimize this program. The technique of Balbin et al. treats constraints in a manner similar to "ordinary" literals, and does not make use of semantic properties of constraints. For instance, the technique of Balbin et al. would treat the constraints $C1 > 0, C2 > 0$ and $C = C1+C2$ as three separate literals, and would not be able to infer that the conjunction of these three constraints also entails that $C > 0$. The technique of Mumick et al. does not consider the use of arithmetic functions such as $+, -, *$, etc. (Essentially, the technique of Mumick et al. considers programs with constraints of the form $C < 5$, but not programs with constraints of the form $C < C1 + C2$.)

The rewriting scheme proposed in this chapter propagates the constraints in the bodies of rules $r1$ and $r2$ in the above program into the definition of *flight*, as described in Example 4.5. The bottom-up evaluation of the rewritten program computes only ground facts. Further, given *any* query on *cheaporshort* (i.e., any pattern of bound arguments), the bottom-up evaluation of the rewritten program does not compute any *flight* fact with $Cost > 150$ and $Time > 240$.

Our next example is a program on which the Magic-sets evaluation does not terminate. The techniques we present can propagate constraint information present in this program such that the bottom-up evaluation of the rewritten program always terminates, while computing all the answers to the query (Example 4.6). Unlike the previous example, this program requires the generation of constraint facts at run-time.

Example 4.2 (Computing Backward Fibonacci) Consider the following program P_{fib} to compute the Fibonacci numbers:

$r1 : fib(0, 1).$
$r2 : fib(1, 1).$
$r3 : fib(N, X1 + X2) : - N > 1, fib(N - 1, X1), fib(N - 2, X2).$

This program can be queried with:

$?\ fib(N, 5).$

Iteration	*Derivations made*
0	$\{r6 : m\_fib(N1, 5)\}$
1	$\{r4 : m\_fib(N1, V1; N1 > 0)\}$
2	$\{r2 : fib(1, 1)$, **r4: m_fib(N1,V1; N1>0)** $\}$
3	$\{r5 : m\_fib(0, V2)$, **r5: m_fib(0,4)** $\}$
4	$\{r1 : fib(0, 1)\}$
5	$\{r3 : fib(2, 2)\}$
6	$\{r3 : fib(3, 3)$, **r5: m_fib(1,V2)**, **r5: m_fib(1,3)** $\}$
7	$\{r3 : fib(4, 5)$, **r5: m_fib(2,V2)**, **r5: m_fib(2,2)** $\}$
8	$\{r3 : fib(5, 8)$, **r5: m_fib(3,V2)**, **r5: m_fib(3,0)** $\}$
	⋮

Table 4.17: Derivations in a Bottom-up Evaluation of P_{fib}^{mg}

The Magic-sets transformation would transform P_{fib} (and the above query) to P_{fib}^{mg} below:

$$\begin{array}{ll}
r1 : fib(0,1) & :- m\_fib(0,1). \\
r2 : fib(1,1) & :- m\_fib(1,1). \\
r3 : fib(N, X1 + X2) & :- m\_fib(N, X1 + X2), N > 1, fib(N-1, X1), \\
 & \quad fib(N-2, X2). \\
r4 : m\_fib(N-1, X1) & :- m\_fib(N, X1 + X2), N > 1. \\
r5 : m\_fib(N-2, X2) & :- m\_fib(N, X1 + X2), N > 1, fib(N-1, X1). \\
r6 : m\_fib(N, 5). &
\end{array}$$

A Semi-naive bottom-up evaluation of P_{fib}^{mg} computes facts as shown in Table 4.17. The answer $N = 4$ to the query is computed in the seventh iteration, but the evaluation does not terminate. Note that this evaluation generates constraint facts for the magic predicate, m_fib. Subsumed facts are shown in boldface; these are discarded, and are not used to make new derivations. □

4.2 Outline of Chapter

In this chapter we present a technique that generates and propagates minimum QRP-constraints (if the technique terminates), based on the definition and uses of program predicates (Section 4.5). By propagating minimum QRP-constraints to the original program, we obtain a program that *fully* utilizes the constraint information present in the original program. This technique is based on two algorithms:

1. Gen-Prop-predicate-constraints, which generates and propagates constraints that are satisfied by program predicates based on their *definitions.*

2. Gen-Prop-QRP-constraints, which generates and propagates constraints based on the *uses* of program predicates, using fold/unfold transformations (Tamaki and Sato [88]) and constraint manipulation.

We also show that determining whether (any representation for) the minimum QRP-constraint for a predicate is a finite constraint set is undecidable (Section 4.4). We describe a class of programs for which this problem is decidable (Section 4.6). For this class of programs, our algorithm for computing minimum QRP-constraints always terminates.

The Magic-sets transformation has been widely studied for propagating bindings. An important question is how Magic-sets interacts with the use of Gen-Prop-predicate-constraints and Gen-Prop-QRP-constraints. Our results are as follows:

1. In [82], we present an algorithm based on Magic-sets followed by a finite sequence of fold/unfold transformations that essentially mimics the algorithm of Mumick et al. [56].

   This enables us to view the results of this chapter, and the algorithms in [4] and [56] in a uniform framework; namely, a combination of Magic-sets and (possibly simpler versions of) the algorithms Gen-Prop-predicate-constraints and Gen-Prop-QRP-constraints, in some order.

2. We examine various orderings in which Magic-sets, Gen-Prop-predicate-constraints and Gen-Prop-QRP-constraints can be applied, and show that it is always better to defer the application of Magic-sets (Section 4.8). This is clearly very useful in designing a compiler to deal with CQL programs.

4.3 Preliminaries

For simplicity, we discuss programs without negation or aggregation in this chapter. However, our techniques can be easily generalized to deal with negation and/or aggregation.

Definition 4.1 (Linear Arithmetic Constraint) A *linear arithmetic constraint* is of the form:

$$a_1 X_1 + \dots a_n X_n \;\; op \;\; a_{n+1}$$

where $a_1, \dots, a_{n+1}$ are real-valued coefficients of real-valued variables $X_i, 1 \leq i \leq n$, and the operator *op* is one of $<, >, \leq, \geq$ and $=$. □

In this chapter, we consider only constraint query languages with *linear arithmetic constraints.* However, our techniques extend to programs with other types of constraints as well.

In this chapter, we refer to programs with constraints in rule bodies as CQL programs (following the terminology used by Kanellakis et al. [39]). A constraint fact of the form:

$$p(\overline{X}) : - C.$$

where C is a comma separated sequence of constraints, is also represented as $p(\overline{X}; C)$. It is a finite representation of the (potentially) infinite set of ground facts that are instances of $\overline{X}$ and satisfy the conjunction of constraints denoted by C.

Definition 4.2 (Derivation Tree) Consider a program P with database D. *Derivation trees* in $\langle P, D \rangle$ are defined for ground facts as follows:

- Every ground instance h of a fact in D is a derivation tree for itself, consisting of a single node with label h.
- Let r be a rule:

  $$p(\overline{X}) : - C, p_1(\overline{X_1}), \dots, p_n(\overline{X_n}).$$

  in P, let $d_i, 1 \leq i \leq n$, be facts with derivation trees T_i, and let θ be the *mgu* of $(p_1(\overline{X_1}), \dots, p_n(\overline{X_n}))$ and $(d_1, \dots, d_n)$, such that $\theta[C]$ is satisfiable. Then the following is a derivation tree for each ground instance $p(\overline{a})$ of $\theta[p(\overline{X})]$: the root is a node labeled with $p(\overline{a})$ and r, and each $T_i, 1 \leq i \leq n$, is a child of the root.

Constraints in rules are viewed as conditions that determine whether or not a candidate tree is indeed a derivation tree; constraints are not themselves part of a tree. □

Definition 4.3 (Bottom-up Evaluation of CQL Programs) Bottom-up evaluation of a program in a CQL proceeds by starting with the constraint facts in the database and repeatedly applying all the rules of the program, in iterations, to compute new constraint facts. The evaluation terminates once we have reached a fixpoint. We now intuitively describe a rule application, the basic step in a bottom-up evaluation. Consider a program rule:

$$r : p(\overline{X}) : -\, C, p_1(\overline{X_1}), \ldots, p_n(\overline{X_n}).$$

A derivation of a p fact using rule r consists of two steps:

- First, choose one p_i fact that unifies with literal $p_i(\overline{X_i})$, for each $1 \leq i \leq n$, to obtain a satisfiable conjunction of constraints over the variables present in the body of rule r.
- Next, variables not present in the head of the rule are eliminated using variable (quantifier) elimination techniques to obtain a conjunction of constraints over the variables in the head of the rule.

This newly generated p fact must be compared against previously generated p facts to check whether it is indeed a new fact. An *application* of rule r consists of making all possible derivations that can be made using rule r and the set of facts known at the end of the previous iteration. If no new facts are computed in an iteration, the computation has reached a fixpoint. $\Box$

Note that bottom-up evaluation uses the representation of the constraint facts *directly*, instead of working with the potentially infinite set of ground facts represented by the constraint facts. The equivalence of the constraint facts computed in a bottom-up evaluation of a program P and the meaning of P given by its least model is in terms of the set of ground facts represented by the constraint facts.

Theorem 4.1 *Consider a program P, and database D in a CQL with arithmetic constraints, and let $\mathcal{F}$ be the set of constraint facts computed in a bottom-up evaluation of $\langle P, D\rangle$. Let $\mathcal{M}$ be the meaning of $\langle P, D\rangle$ in terms of its least model. Then,*

Soundness *: Each ground instance f of a constraint fact $F \in \mathcal{F}$ is in $\mathcal{M}$, and*

Completeness *: Each fact f in $\mathcal{M}$ is a ground instance of a constraint fact $F \in \mathcal{F}$.*

Proof: Consider a program P and database D in a CQL with arithmetic constraints. Jaffar and Lassez [36] described a functional semantics for $\langle P, D\rangle$ in terms of an immediate consequence operator $T_{P,D}$, and showed that the least fixpoint of $T_{P,D}$, given by $T^{\omega}_{P,D}$, is equivalent to the meaning of $\langle P, D\rangle$ in terms of its least model. Let $gr(\mathcal{F})$ be the ground facts represented by the set of constraint facts $\mathcal{F}$.

Claim 1: Consider a set of (constraint) facts D_1 for base and derived predicates of a program P. Let $\mathcal{F}_1$ be the set of constraint facts computed from D_1 by a single application of each rule in P. Let $T_P(gr(\mathcal{D}_1))$ be the set of ground facts obtained by a single application of the immediate consequence operator T_P on the set of ground facts, $gr(\mathcal{D}_1)$. Then, (1) each ground instance f_1 of a constraint fact $F_1 \in \mathcal{F}_1$ is in $T_P(gr(\mathcal{D}_1))$, and (2) each fact f_1 in $T_P(gr(\mathcal{D}_1))$ is a ground instance of a constraint fact $F_1 \in \mathcal{F}_1$.

The proof of Claim 1 follows from the decision procedure of Tarski [90] for the theory of real closed fields. All the operations needed for a rule application have straightforward analogues in Tarski's decision procedure; projection (on the variables of the head of a rule) corresponds to quantifier elimination, for instance.

The soundness and completeness results for the program can be shown by induction on the iterations of the bottom-up evaluation of $\langle P, D\rangle$. The database D provides the base case. Claim 1 provides the induction step. □

Consequently, with each constraint fact F computed by a bottom-up evaluation of the program, we can associate the set of derivation trees for each ground instance of F.

Given a program-query pair $\langle P, Q\rangle$, we can treat the query Q as the body of a rule defining a new predicate q, not occurring in P. The arity of q is the same as the number of variables in Q. The predicate q can now be treated as the query predicate, queried with all its arguments free. Whenever a query is given, we assume this transformation has been done and the query is treated as just another program rule.

Definition 4.4 (Constraint Set) A *constraint set* is a disjunction of conjunctions (DNF) of constraints.

A constraint set $C_1(X_1, \ldots, X_n)$ is said to *imply* constraint set $C_2(X_1, \ldots, X_n)$, denoted

$$C_1(X_1, \ldots, X_n) \supset C_2(X_1, \ldots, X_n),$$

if whenever we substitute constant a_i for the variable $X_i, 1 \leq i \leq n$, such that the constraint set $C_1(a_1, \ldots, a_n)$ simplifies to **true**, then so does $C_2(a_1, \ldots, a_n)$.[3] This can be naturally extended to the case when C_1 and C_2 do not contain the same variables. □

[3]Note that the use of the "$\supset$" symbol is different from its traditional use as the superset operator.

For example, the conjunction $(X+Y \leq 4)\&(X \geq 2)$ implies $Y \leq 2$. The techniques described in Srivastava [81] can be used to check for implication of constraint sets of linear arithmetic constraints.

Definition 4.5 (Predicate Constraints) Given a CQL program P, a *predicate constraint* on a predicate p is a constraint set satisfied by each p fact that is derived during the bottom-up evaluation of P, independent of the facts in the EDB predicates. □

Given a program P, a predicate constraint C_p on predicate p is said to be *minimal* if there does not exist a C'_p, such that: (1) C'_p is also a predicate constraint on p, (2) $C'_p \supset C_p$, and (3) $C_p \not\supset C'_p$. The existence of a unique *minimum* predicate constraint is guaranteed as a consequence of the following proposition.

Proposition 4.2 *Given a program P, if constraint sets $C1_p$ and $C2_p$ are predicate constraints on p, then so is $C1_p\&C2_p$ (after conversion to DNF).* □

Definition 4.6 (Constraint Relevance) Given a CQL program P with query predicate q, consider the complete set S of derivation trees that are associated with query answers, for every possible extension of the EDB predicates and every possible query on the query predicate.

A ground program fact $p(\overline{a})$ is said to be *constraint relevant* to the query predicate if it occurs in at least one derivation tree in S. A constraint fact $p(\overline{X}; C)$ is said to be *constraint relevant* to the query predicate if each ground instance of it is constraint relevant to the query predicate. □

Definition 4.7 (Query Relevant Predicate Constraints) Given a CQL program P, a *query relevant predicate constraint* (QRP-constraint) on a predicate p is a constraint set satisfied by each p fact that is derived during the bottom-up evaluation of P, independent of the facts in the EDB predicates, and which is constraint relevant to a query predicate of P. □

A proposition similar to Proposition 4.2 guarantees the existence of minimum QRP-constraints.

In representing constraints on argument positions of a predicate, we use $\$i$ for the i'th argument. Since constraints in rules are in terms of the variables (X, Y, etc.) in the rule, whereas predicate constraints and QRP-constraints are in terms of argument positions ($\$1, \$2$, etc.), we need functions to convert between the two forms. We use $PTOL(p(\overline{X}), C)$ to convert a constraint set over the argument positions of p to an "equivalent" constraint set over the variables in $\overline{X}$.

Definition 4.8 (PTOL()) Consider a predicate p of arity n, and a constraint set C on the argument positions of p. Let $p(\overline{X})$ be a literal, such that $\overline{X}$ is a tuple of n (not necessarily distinct) variables.

We define $PTOL(p(\overline{X}), C)$ as the constraint set obtained from C by replacing each argument position by the "corresponding" variable in that position in $p(\overline{X})$. □

For example, if *flight* is a predicate of arity 4, then $PTOL(flight(S, D, T, C), (\$3 \leq 240) \vee (\$4 \leq 150))$ is given by $(T \leq 240) \vee (C \leq 150)$.

Similarly, we use $LTOP(p(\overline{X}), C)$ to convert a constraint set over the variables in $\overline{X}$ to an "equivalent" constraint set over the argument positions of p.

Definition 4.9 (LTOP()) Consider a literal $p(\overline{X})$, such that $\overline{X}$ is a tuple of n (not necessarily distinct) variables; and a constraint set $C(\overline{X})$ on the variables in $\overline{X}$.

If $\overline{X}$ is a tuple of n distinct variables, we define $LTOP(p(\overline{X}), C(\overline{X}))$ as the constraint set obtained from $C(\overline{X})$ by replacing each variable in $\overline{X}$ by the "corresponding" argument position of the variable in $p(\overline{X})$.

If $\overline{X}$ is not a tuple of n distinct variables, we define $LTOP(p(\overline{X}), C(\overline{X}))$ as equivalent to $LTOP(p(\overline{Y}), \Pi_{\overline{Y}}(C(\overline{X}) \& C_1(\overline{X}, \overline{Y})))$ where $\overline{Y}$ is a tuple of n distinct variables, distinct from the variables in $\overline{X}$, and $C_1(\overline{X}, \overline{Y})$ is a conjunction of equality constraints that equates each variable in $\overline{Y}$ with the variable in the corresponding position in $\overline{X}$. The $\Pi_{\overline{Y}}$ operation is the projection (quantifier elimination) operation, and it guarantees that we focus attention in the constraint set on the variables in $\overline{Y}$. □

For example, $LTOP(flight(S, D, T, C), (T \leq 240) \vee (C \leq 150))$ is given by $(\$3 \leq 240) \vee (\$4 \leq 150)$.

4.4 Propagating Constraints: The Problem

Definition 4.10 (Program Core) Consider a program P in a constraint query language. The *core* of P is the program obtained from P by deleting all constraints in program rules. □

Consider a CQL program P, with query predicate q, and let P_{core} be the core of the program P. In this section, we look at the problem of optimizing P by propagating constraints occurring in P to the bodies of rules in P_{core}. The intuition behind not altering the core of P while propagating constraints is that the core represents the syntactic structure of the program, which encodes the programmer's knowledge about the problem.

More precisely, we address the problem of finding a constraint set for each predicate such that:

1. Adding the constraint set for predicate p to the body of each rule[4] defining p yields a program P' such that P is query equivalent to P' (on all input EDBs).

2. The bottom-up evaluation of P' on an input EDB should compute only program facts such that each of these facts is constraint relevant to the query predicate in P.

A rewritten program P' is said to be completely optimized with respect to the constraints present in the program P, if it satisfies the above two conditions. In terms of the definition of QRP-constraints, a rewritten program P' is said to be completely optimized with respect to the constraints present in P if each ground instance of each p fact computed in a bottom-up fixpoint evaluation of P' on an input EDB satisfies the minimum QRP-constraint on p. We are interested in computing QRP-constraints, since we make the assumption (as is common in optimization of database query languages) that query optimization should be independent of the facts in the EDB predicates.

Theorem 4.3 *Given a CQL program P with linear arithmetic constraints, determining whether any representation for the minimum predicate constraint for a predicate p is a finite constraint set is undecidable.*

Proof: We first show that a variant of the safety (or finiteness) problem for logic programs is undecidable. Consider the Sebelik and Stepanek [79] reduction which showed that every partial recursive function can be expressed as a logic program P_{pr} with one constant symbol and one unary function symbol.

Now consider a program P_1 which has all the rules in P_{pr}, as well as the following rules:

$$\begin{array}{ll} p(a) & :- q(\overline{X}). \\ p(f(X)) & :- p(X). \end{array}$$

[4]If the constraint set has more than one disjunct, this would mean creating copies of the rule, each copy containing one of the disjuncts. This is required since only conjunctions of constraints are allowed in the body of a rule.

The predicate p does not occur in P_{pr}, a is the only constant symbol in P_{pr}, f is the only function symbol in P_{pr}, and $q(\overline{X})$ is a query on P_{pr}.

Clearly, $p(a)$ is in the model for p iff $q(\overline{X})$ is satisfiable. If $p(a)$ is in the model, then so are $p(f(a)), p(f(f(a))), p(f(f(f(a)))), \ldots$, and the model of p in this program is infinite. Hence, the model of p is infinite iff $q(\overline{X})$ is satisfiable in P. Now, satisfiability of a query $q(\overline{X})$ on P_{pr} is undecidable, by reduction of the "halting problem" for partial recursive functions. Hence, it is undecidable whether the model of p is finite in this program.

We now reduce this problem to the problem of deciding whether any representation for the minimum predicate constraint for a predicate is finite. Intuitively, this reduction takes a logic program P and reduces it to a CQL program P' such that there is a unique representation for the minimum predicate constraint for predicate p in P', and the number of disjuncts in the minimum predicate constraint is finite iff the model of p in P is finite.

Consider a logic program P defining a unary predicate p (in addition to defining other (possibly) non-unary predicates). Let there be just one constant, say a, and one unary function symbol, say f, appearing in P. (The program P_1 above satisfies these conditions.) We transform the logic program P into a CQL program P' as follows:

- We replace all occurrences of the constant a by the numeric constant 0 in P'.
- All occurrences of $f(X)$ (appearing in the head or the body of a rule) in P are replaced by Y (a variable not appearing in the rule). The conjunction of constraints $(X \geq 0) \& (Y = X + 2)$ is included in the body of the modified rule in P'.

It can be easily seen that there is a one-to-one correspondence between p facts in the model of p in P, and p facts in the model of p in P'. Further, each p fact in the model of p in P' is a ground fact, with an even valued argument ≥ 0. Hence, the minimum predicate constraint for p in P' is the possibly infinite disjunction:

$$\bigvee_{i \geq 0} (\$1 = 2 * i)$$

Again, there is a one-to-one correspondence between p facts in the model of p in P' and the disjuncts in the minimum predicate constraint for p in P'. It can be easily seen that this is a unique representation for the minimum predicate constraint for p in P', given the representations we consider for constraint sets.

Consequently, the number of disjuncts in the minimum predicate constraint for p in P' is finite iff the number of p facts in the model of p in P is finite. The undecidability result follows. □

In general, a constraint set with linear arithmetic constraints can have many different (though equivalent) representations. Even if a specific representation of the minimum predicate constraint is infinite, it *does not* follow that the minimum predicate constraint is infinite, since it could have an equivalent finite representation. The result is hence *independent of* any specific representation of the minimum predicate constraint.

The proof of Theorem 4.3 can also be used to show:

Proposition 4.4 *Given a CQL program P, where the only constraints allowed are of the form $X \leq c$, $X \geq c$, and $X \leq Y + c$, determining whether any representation for the minimum predicate constraint for a predicate p is a finite constraint set is undecidable.* $\square$

Brodsky and Sagiv [16] show that it is undecidable whether a specific procedure for computing minimum predicate constraints computes finite predicate constraints. Proposition 4.4 does not follow from their result since they do not address the issue of multiple representations.

Theorem 4.5 *Given a CQL program P with linear arithmetic constraints, determining whether any representation for the minimum QRP-constraint for a predicate p is a finite constraint set is undecidable.*

Proof: Consider a CQL program P defining predicate p. Let q be a predicate not occurring in P. We add a new rule to P:

$$r : q(\overline{X}) :- p(\overline{X}).$$

The predicate q has the same arity as p, and $\overline{X}$ is a tuple of distinct variables. Let q be the query predicate of the modified program P_1. Since r is the only rule defining q, every p fact is constraint relevant to the query predicate q. Consequently, the minimum QRP-constraint for p in P_1 is the same as the minimum predicate constraint for p in P.

From Theorem 4.3, we know that determining whether the minimum predicate constraint for p in P is a finite constraint set is undecidable. Hence, determining whether the minimum QRP-constraint for predicate p in P_1 is a finite constraint set is also undecidable. $\square$

As a corollary, we have:

Proposition 4.6 *Given a CQL program P, where the only constraints allowed are of the form $X \leq c$, $X \geq c$, and $X \leq Y + c$, determining whether any representation for the minimum QRP-constraint for a predicate p is a finite constraint set is undecidable.* $\square$

In Section 4.6, we describe a class of constraint query languages for which it is decidable whether the minimum QRP-constraint for a predicate in a program can be represented as a finite constraint set.

Independent of its use in establishing Theorem 4.5, Theorem 4.3 is of interest since our technique for generating and propagating minimum QRP-constraints first generates and propagates minimum predicate constraints (see Section 4.5.4).

4.5 The Transformation: Propagating Constraints

In this section, we describe a rewriting technique for propagating constraints in a CQL program. Our technique has two components to it:

1. For each derived predicate p of a program P, it generates QRP-constraints on p, using semantic properties of constraints.

2. It then uses the fold/unfold transformations [88] to propagate the QRP-constraint on p into the program rules defining p.

Procedure Gen-Prop-QRP-constraints in Appendix A.2 describes this technique algorithmically. If the rewriting technique terminates, it propagates QRP-constraints for each derived predicate in the program, while preserving the core of the program. However, the rewriting technique does not terminate in general.

A key feature of our rewriting technique is that it makes essential use of semantic properties of constraints, unlike previous techniques that had a similar objective [4, 56]. As a consequence, we are able to optimize a larger class of programs than previous techniques.

4.5.1 An Example

First, we give a simple example of how the fold/unfold transformations can be used along with semantic properties of constraints to propagate constraint selections in a program. The fold/unfold transformations used are described in Appendix B.

Example 4.3 Consider the following program P with query predicate q:

$$\begin{array}{ll} r1 : q(X) & :- \; p_1(X,Y), p_2(Y), X+Y \leq 6, X \geq 2. \\ r2 : p_1(X,Y) & :- \; b_1(X,Y). \\ r3 : p_2(X) & :- \; b_2(X). \end{array}$$

First, two new rules are created:

$$\begin{aligned} r4 : p'_1(X,Y) &: - X+Y \leq 6, X \geq 2, p_1(X,Y). \\ r5 : p'_2(Y) &: - Y \leq 4, p_2(Y). \end{aligned}$$

The body of each rule includes a derived literal from the body of rule $r1$, and the projection of the constraints in the body of $r1$ onto the variables of the derived literal. (Such a projection can be performed using the techniques described in [47], for instance.) Thus, although $Y \leq 4$ is not present in the body of $r1$, it is implied by the conjunction of constraints $(X+Y \leq 6)\&(X \geq 2)$.

Next, the definitions of p_1 and p_2 are unfolded into the definitions of p'_1 and p'_2 obtaining:

$$\begin{aligned} r4' : p'_1(X,Y) &: - X+Y \leq 6, X \geq 2, b_1(X,Y). \\ r5' : p'_2(Y) &: - Y \leq 4, b_2(Y). \end{aligned}$$

Finally, rules $r4$ and $r5$ are folded into rule $r1$ obtaining

$$r1' : q(X) : - p'_1(X,Y), p'_2(Y), X+Y \leq 6, X \geq 2.$$

The transformed program obtained (after deleting rules not reachable from the query predicate q) is P', shown below:

$$\begin{aligned} r1' : q(X) &: - p'_1(X,Y), p'_2(Y), X+Y \leq 6, X \geq 2. \\ r4' : p'_1(X,Y) &: - X+Y \leq 6, X \geq 2, b_1(X,Y). \\ r5' : p'_2(Y) &: - Y \leq 4, b_2(Y). \end{aligned}$$

Note that P' is equivalent to P on the query predicate; the bottom-up evaluation of P' computes only ground facts, if the bottom-up evaluation of P does so; and the bottom-up evaluation of P' computes, in general, fewer facts than the bottom-up evaluation of P. Further, this program can now be rewritten using Magic-sets to take advantage of any constants in the actual query; if the bound-if-ground rule is used (i.e., an argument is treated as bound only if it is bound to a ground term), the Magic-sets transformed program also computes only ground facts.

Note that the *LTOP* of the conjunction of constraints in the body of $r4'$, viz. $(\$1 + \$2 \leq 6)\&(\$1 \geq 2)$, is the minimum QRP-constraint for p_1 in the original program P. Similarly, the *LTOP* of the constraint in the body of rule $r5'$, viz. $\$1 \leq 4$, is the minimum QRP-constraint for p_2 in the original program P. We have thus generated and propagated the minimum QRP-constraints for the various derived predicates in the original program.

Neither the C-transformation of [4], nor the GMT-transformation of [56] would be able to propagate all the constraints in this example. Since the C-transformation of [4] treats constraints as any other literal, it would not be able to propagate any constraints into the definition of p_2; the problem is that in rule $r1$, there is no explicit constraining literal on Y. Our technique utilizes the fact that $((X + Y \leq 6) \& (X \geq 2)) \supset (Y \leq 4)$, a semantic property of the conjunction of linear arithmetic constraints in the body of rule $r1$, to propagate constraints into the definition of p_2, and hence restrict the potentially relevant p_2 facts computed.

As described in [56], the GMT-transformation does not handle constraints with arithmetic function symbols such as $+$. Consequently, it would not be able to propagate constraints either. □

4.5.2 Generation of QRP-constraints

Our algorithm, Gen-QRP-constraints, described in Appendix A.2, for generating QRP-constraints works iteratively. In each iteration, given "approximate" QRP-constraints on each predicate defined in program P, the algorithm computes a new approximation for the QRP-constraints for each predicate in the program. The algorithm terminates when a "fixpoint" is reached. Theorem 4.8 shows that, in the limit, Gen-QRP-constraints does compute QRP-constraints for each predicate in the program.

Non-recursive Inference

Consider a rule r of the form:

$$r : p(\overline{X}) :- C_r(\overline{Y}), p_1(\overline{X_1}), \ldots, p_n(\overline{X_n}).$$

where $C_r(\overline{Y})$ is the conjunction of constraints in the body of rule r. Given a constraint set C_p on the arguments of the head predicate p of rule r, a *literal constraint* on $p_i(\overline{X_i})$ in the body of r is a constraint set (on the variables in $\overline{X_i}$) that needs to be satisfied by each p_i fact that can be used (in literal $p_i(\overline{X_i})$) to derive a p fact (using rule r) that satisfies C_p.

Proposition 4.7 *Consider a CQL program P, with linear arithmetic constraints. Given a rule r of the form:*

$$r : p(\overline{X}) :- C_r(\overline{Y}), p_1(\overline{X_1}), \ldots, p_n(\overline{X_n}).$$

where $C_r(\overline{Y})$ is the conjunction of constraints in the body of rule r, if C_p is the desired constraint set on head predicate p,

$$C_{p_i(\overline{X_i})}(\overline{X_i}) = \Pi_{\overline{X_i}}(PTOL(p(\overline{X}), C_p) \& C_r(\overline{Y}))$$

is a literal constraint on $p_i(\overline{X_i})$ in the body of rule r. $\square$

Recursive Inference

Since each fact in the query predicate q is constraint relevant to an answer to some query, Gen-QRP-constraints initially assumes the constraint **true** as the "approximate" QRP-constraint for predicate q, and the constraint **false** as the "approximate" QRP-constraint for every other predicate defined in P. Gen-QRP-constraints works iteratively.

In each iteration, given the "approximate" QRP-constraints on each predicate p as the desired constraint set on the head of each rule defining p, the algorithm computes literal constraints $C_{p_i(\overline{X_i})}$ for each derived literal in the body of each rule defining p. Let $p(\overline{X_1}), \ldots, p(\overline{X_k})$ be all the occurrences of p in the bodies of the rules in P, and $C_{p(\overline{X_1})}, \ldots, C_{p(\overline{X_k})}$ the corresponding literal constraints inferred. Let $C1_p$ be the "approximate" QRP-constraint on p before the iteration and let $C2_p$ be $\bigvee_{i=1}^{k} LTOP(p(\overline{X_i}), C_{p(\overline{X_i})})$, i.e., the disjunction of the $LTOP$s of the various literal constraints inferred on p literals in this iteration. For each predicate p, the new "approximate" QRP-constraint is given by $C1_p \vee C2_p$. (Before adding disjuncts to the "approximate" QRP-constraint, we can eliminate redundant disjuncts.) If, for each predicate p defined in P, $C1_p \equiv C1_p \vee C2_p$, Gen-QRP-constraints has reached a fixpoint, and it terminates. Else, the algorithm continues iterating with the new "approximate" QRP-constraints.

Theorem 4.8 *Given a CQL program P with linear arithmetic constraints, the constraint set generated for each derived predicate p in P by Gen-QRP-constraints is a QRP-constraint for p.*

Proof: Consider a CQL program P with linear arithmetic constraints. For each derived predicate p in P, let C_p be the constraint set generated by Gen-QRP-constraints. We prove the result by contradiction.

Let us suppose that there exists some query Q, some database D, and some ground fact $p(\overline{a})$ in the least model of $\langle P, D \rangle$ that *does not* satisfy C_p, and which occurs in a derivation tree for an answer to the query Q.

We associate a number with each program fact as follows: Consider the set of derivation trees $\mathcal{T}_Q$ for answers to query Q, and the set of all facts $p(\overline{a})$ that occur in at least one derivation tree in $\mathcal{T}_Q$ such that p is defined in P.

- If $p(\overline{a})$ is the root of a derivation tree T in $\mathcal{T}_Q$, then $p(\overline{a})$ is given the number 0.
- Else, let $p(\overline{a})$ have a parent $p_1(\overline{a_1})$ in a derivation tree T in $\mathcal{T}_Q$. Then $p(\overline{a})$ is given the number $j+1$, where j is the lowest numbered parent of $p(\overline{a})$.

Choose any fact $p(\overline{a})$ that occurs in a derivation tree T in $\mathcal{T}_Q$, such that $p(\overline{a})$ does not satisfy C_p, and let its number be k.

Let $C_p^0, C_p^1, \ldots$ be the sequence of constraint sets generated by Gen-QRP-constraints as "approximate" QRP-constraints for each predicate p. We now show by induction that $p_2(\overline{a_2})$ facts with number j satisfy $C_{p_2}^j$.

For the base case, a fact $p_2(\overline{a_2})$ with number 0 is the root of a derivation tree T in $\mathcal{T}_Q$. It is an answer to the query and trivially satisfies $C_{p_2}^0$, which is **true**. For the induction step, assume that all p_2 facts with number $j \geq 0$ satisfy $C_{p_2}^j$, and consider fact $p_2(\overline{a_2})$ with number $j+1$. This fact has as a parent $p_1(\overline{a_1})$ (and rule r), where $p_1(\overline{a_1})$ has number j. Now consider the rule r (with head $p_1(\overline{X_1})$), and the literal $p_2(\overline{X_2})$ where the fact $p_2(\overline{a_2})$ is used to compute $p_1(\overline{a_1})$. Given the constraint set $C_{p_1}^j$ (satisfied by $p_1(\overline{a_1})$ by hypothesis) on the head of rule r, fact $p_2(\overline{a_2})$ has to satisfy the *LTOP* of the literal constraint on $p_2(\overline{X_2})$, and hence $C_{p_2}^{j+1}$. This is because "projection" (or quantifier elimination) of linear arithmetic constraint sets can be done exactly using the algorithm described in [47], for instance. This concludes the induction. Hence, fact $p(\overline{a})$ satisfies C_p^k (since it's number is k), and hence C_p. This contradicts the original assumption that $p(\overline{a})$ does not satisfy C_p, and concludes the proof of the theorem. □

Gen-QRP-constraints may not terminate, in general. Termination could be guaranteed by various modifications of our algorithm. For example, instead of iterating until the QRP-constraints "stabilize", we could iterate for a (pre-determined) fixed number of iterations. If after the fixed number of iterations, the "approximate" QRP-constraints have not stabilized, our algorithm can return **true** (which is trivially a QRP-constraint, though not the minimum possible) as the QRP-constraint for program predicates. Clearly, this does not affect the correctness of our algorithm. The larger the number of iterations chosen, the larger the class of programs for which we will be able to infer non-trivial QRP-constraints. Clearly, there is a tradeoff here: how large a class of programs we wish to optimize versus the cost we are willing to incur in optimizing such programs. What bound to choose depends on the relative costs, and is outside the scope of this thesis.

4.5.3 Propagating QRP-constraints

If Gen-QRP-constraints terminates (with the QRP-constraint for p having m disjuncts), we can use the fold/unfold transformations, described in Appendix B, to propagate this QRP-constraint into rules defining p. This propagation consists of three steps:

1. Perform a a *definition* step creating m rules with head $p'(\overline{X})$ and the sole body literal being $p(\overline{X})$. Further, each of the m rules contains one of the m disjuncts of the QRP-constraint generated for p.
2. Unfold the definition of p into each of the rules defining p'.
3. Fold the original definition of p' into each rule containing an occurrence of p.

With this, the QRP-constraints generated for the predicate p have been propagated into the rules defining the predicate p. Procedure Gen-Prop-QRP-constraints in Appendix A.2 describes this algorithmically. Example 4.3 illustrates this algorithm for a simple program.

The correctness of the fold, unfold, and definition steps ensures the following:

Theorem 4.9 *Given a CQL program P with linear arithmetic constraints, if Gen-Prop-QRP-constraints terminates, the rewritten program is equivalent to the original program with respect to the query predicates, on all input EDBs.* □

The following result indicates that if the original program could be evaluated "efficiently", so can the rewritten program.

Theorem 4.10 *Consider a CQL program P with linear arithmetic constraints, such that the bottom-up evaluation of a program P on database D computes only ground facts. Let P' be the program obtained from P using Gen-Prop-QRP-constraints. Then,*

- *the bottom-up evaluation of P' on D also computes only ground facts,*
- *the bottom-up evaluation of P' on D computes a subset of the facts computed by the bottom-up evaluation of P on database D, and*
- *if the QRP-constraint C_p generated for each derived predicate p were such that the intersection of no two disjuncts of C_p is satisfiable, the bottom-up evaluation of P' on D makes a subset of the derivations made by the bottom-up evaluation of P on D.*

Proof: Consider a program P such that the bottom-up evaluation of P on database D computes only ground facts. We prove the results by considering the sequence of definition, unfold, and fold steps performed by Gen-Prop-QRP-constraints.

First, consider the definition step. Each new rule is of the form:

$$r : p'(\overline{X}) :- C, p(\overline{X}).$$

where $\overline{X}$ is a tuple of distinct variables, and C is a conjunction of constraints involving only the variables in $\overline{X}$. (It is the $PTOL$ of a disjunct in the QRP-constraint C_p generated for predicate p.) Since each rule defining p computes only ground facts, adding an additional conjunction of constraints C only prevents the computation of certain facts; it does not affect the property that all p' facts computed are ground. Also, the set of facts computed for p' is a subset of the set of facts computed for p because of the additional conjunction of constraints C in the body of rule r. Further, if C_p were such that the intersection of no two disjuncts is satisfiable, each of the rules defining p' would compute distinct facts. Hence, the set of derivations made for p' is a subset of the set of derivations made for p.

Next, consider the unfold step. The definition of p is unfolded into the rules defining p'. This does not affect the set of p' facts computed, nor the derivations made.

Finally, consider the fold step. The fold step essentially replaces occurrences of p in rule bodies by p'. Since the set of facts computed for p' is a subset of the set of facts computed for p, by replacing occurrences of p by p' in the body of a rule defining p_1, no new facts are computed for p_1 by this rule. Since the facts computed for p_1 prior to the fold step were ground, so are facts computed for p_1 subsequent to this fold step. If the derivations made by each rule defining p' were distinct, the derivations made for p_1 subsequent to the fold step does not increase. This completes the proof of the theorem. □

4.5.4 Using Inferred Predicate Constraints

The QRP-constraints generated and propagated by Gen-Prop-QRP-constraints need not be the *minimum* QRP-constraints, in general. However, if the program P is of a certain form, then we *can* guarantee that Gen-Prop-QRP-constraints does generate and propagate the minimum QRP-constraints (if it terminates). Theorem 4.13 below provides conditions on the form of such programs. We first describe a program where the minimum QRP-constraint is not generated using Gen-QRP-constraints.

Example 4.4 Consider the following program P:

$r1 : q(X,Y) : -\ a(X,Y), X \leq 10.$
$r2 : a(X,Y) : -\ p(X,Y), Y \leq X.$
$r3 : a(X,Y) : -\ a(X,Z), a(Z,Y).$

where q is the query predicate. Assuming **true** as the QRP-constraint for q, we would generate the literal constraint $X \leq 10$ for the occurrence of $a(X,Y)$ in the body of rule $r1$. Now, assuming the constraint set $\$1 \leq 10$ as the "approximate" constraint set for a, we would generate the following literal constraints: for $a(X,Z)$ in rule $r3$, we would get $X \leq 10$; for $a(Z,Y)$ in rule $r3$, we would get **true**. Gen-QRP-constraints would infer **true** (or, unconstrained) as the QRP-constraint for a and the algorithm would terminate.

Note, however, the following. Each a fact that is derived using rule $r2$ of program P has the constraint $\$2 \leq \$1$. These facts will be used in the recursive rule $r3$ to derive more a facts. If each of the facts used in the body of $r3$ satisfies $\$2 \leq \$1$, then so does the head fact derived using these facts and rule $r3$. Thus, each a fact derived using rules $r2$ and $r3$ in the above program P satisfies the predicate constraint $\$2 \leq \$1$. (Similarly, each q fact derived using rule $r1$ in the above program P satisfies the predicate constraint $(\$2 \leq \$1) \& (\$1 \leq 10)$.) If, for each a literal in the program, the $PTOL$ of the predicate constraint $\$2 \leq \$1$ were explicitly introduced, we would get the following program $P1$:

$r1' : q(X,Y) : -\ a(X,Y), X \leq 10, Y \leq X.$
$r2 :\ a(X,Y) : -\ p(X,Y), Y \leq X.$
$r3' : a(X,Y) : -\ a(X,Z), Z \leq X, a(Z,Y), Y \leq Z.$

Now, introducing these constraints does not change the facts computed by the program $P1$. However, with program $P1$ we can use Gen-QRP-constraints to obtain the minimum QRP-constraint $((\$1 \leq 10) \& (\$2 \leq \$1))$ for a. This constraint can now be propagated using the fold/unfold transformations to reduce the number of facts computed by the rewritten program P'. □

Given a CQL program P with linear arithmetic constraints, there is a procedure to enumerate minimum predicate constraints for program predicates. Intuitively, given constraints on base predicates, the procedure works by iteratively computing the constraints that hold on derived predicates bottom-up. Procedure Gen-predicate-constraints in Appendix A.2 algorithmically describes this. In general, Gen-predicate-constraints may not terminate.

Theorem 4.11 *Given a CQL program P with linear arithmetic constraints, procedure Gen-predicate-constraints generates the minimum predicate constraint for each derived predicate of the program P.*

Proof: Consider a CQL program P. For each derived predicate p in P, let C_p be the constraint set generated by Gen-predicate-constraints.
Claim 1: C_p is a predicate constraint for p.
Proof of Claim 1: We prove this by contradiction. Let us suppose that there exists some database D, and some ground fact $p(\overline{a})$ that *does not* satisfy C_p, and which occurs in a derivation tree.

We associate a number with each program fact as follows: Consider the set of derivation trees $\mathcal{T}$, and the set of all facts $p(\overline{a})$ that occur in at least one derivation tree in $\mathcal{T}$.

- Let $p(\overline{a})$ be a leaf node in some derivation tree T in $\mathcal{T}$. Then $p(\overline{a})$ is given the number 0.
- Else, let $p(\overline{a})$ be such that in some derivation tree T in $\mathcal{T}$, a child of $p(\overline{a})$ is of the form $p_1(\overline{a_1})$. Then $p(\overline{a})$ is given the number $j+1$, where j is the highest numbered child of $p(\overline{a})$.

Choose any fact $p(\overline{a})$ such that $p(\overline{a})$ does not satisfy C_p, and let its number be k.

Let $C_p^0, C_p^1, \ldots$ be the sequence generated by Gen-predicate-constraints as "approximate" predicate constraints for predicate p. We now show by induction that $p_2(\overline{a_2})$ facts with number j satisfy $C_{p_2}^j$.

For the base case, a fact $p_2(\overline{a_2})$ with number 0 is a database fact. It trivially satisfies C_p^0, the predicate constraint given on the database predicate. For the induction step, assume that all facts with number $j \geq 0$ satisfy $C_{p_2}^j$, and consider fact $p_2(\overline{a_2})$ with number $j+1$. There is a derivation tree and a node in the derivation tree with label $p_2(\overline{a_2})$ (and rule r) such that each child of it has a number $\leq j$. Now consider the rule r (with head $p_2(\overline{X_2})$) which is used to compute $p_2(\overline{a_2})$. Given the constraint set $C_{p_1}^j$ on each p_1 literal in the body of rule r, fact $p_2(\overline{a_2})$ has to satisfy the *LTOP* of the inferred head constraint on $p_2(\overline{X_2})$, and hence $C_{p_2}^{j+1}$. This is because quantifier elimination of linear arithmetic constraint sets can be done exactly. This concludes the induction. Hence, fact $p(\overline{a})$ satisfies C_p^k (since it's number is k), and hence C_p. This contradicts the original assumption that $p(\overline{a})$ does not satisfy C_p, and concludes the proof of Claim 1.

Claim 2: C_p is a *minimum* predicate constraint for p.

Proof of Claim 2: We prove this by contradiction. Let $C_p^0, C_p^1, \ldots$ be the sequence of constraint sets generated by Gen-predicate-constraints as "approximate" predicate constraints for predicate p. Consider the lowest numbered C_p^k such that there exists a fact $p(\overline{a})$ that satisfies C_p^k and is not in the least model of $\langle P, D\rangle$, i.e., each fact that is an instance of $C_p^m, m < k$ is in the least model of $\langle P, D\rangle$. The number k cannot be 0, since C_p^0 is **false** for all derived predicate p, and no p fact satisfies this constraint; C_p^0 is not **false** only for database predicates p, and for these predicates, we have assumed that the minimum predicate constraints were made available as input.

Since ground fact $p(\overline{a})$ satisfies C_p^k, it satisfies some disjunct of C_p^k which is not present in C_p^{k-1}. From Gen-predicate-constraints, we know that this disjunct is generated in iteration k of Gen-predicate-constraints using some rule r. Consider this rule, and the literals in the body of r. By hypothesis, each fact that satisfies $C_{p_1}^{k-1}$ is present in the least model and can be used in body literal $p_1(\overline{X_1})$. Since quantifier elimination of linear arithmetic constraint sets can be done exactly, each ground fact that satisfies the inferred head constraint C_p^k can also be derived, and must be present in the least model. This contradicts the original assumption that $p(\overline{a})$ is not present in the least model, and concludes the proof of Claim 2, as well as the proof of the theorem. $\square$

Brodsky and Sagiv [16] study this problem of generating predicate constraints for a special case, where the only constraints allowed are of the form $\$i \leq \$j + c$. Van Gelder [95] also studies a similar but more restricted problem; the techniques of [95] can be used only to derive a single conjunction of constraints on the arguments of a predicate; general constraint sets (which are disjunctions of conjunctions) are not inferred.

Using Gen-predicate-constraints for generating predicate constraints, one can infer that $\$2 \leq \$1$ is a predicate constraint for a in program P of Example 4.4. One can also infer that, on the $flight$ predicate in Program P of Example 4.1, $\$4 > 0$ (that is, the cost of each flight is > 0) as well as $\$3 > 0$ (that is, the time taken by each flight is > 0). For each of these examples, our algorithm for generating predicate constraints terminates.

If Gen-predicate-constraints terminates, we can associate the $PTOL$ of the predicate constraint for p with body occurrences of p. Procedure Gen-Prop-predicate-constraints in Appendix A.2 describes this algorithmically.

Theorem 4.12 *Consider a CQL program P with linear arithmetic constraints. If Gen-Prop-predicate-constraints terminates (resulting in program P'), then*

- *The rewritten program P' is equivalent to the original program with respect to all derived predicates, on all input EDBs that satisfy the predicate constraints for the database predicates.*
- *If the bottom-up evaluation of P on database D computes only ground facts, the bottom-up evaluation of P' on database D computes only ground facts.*
- *If the bottom-up evaluation of P on database D computes only ground facts, and the predicate constraint $C1_p$ generated for each derived predicate p were such that the intersection of no two disjuncts of $C1_p$ is satisfiable, then the bottom-up evaluation of P' on database D makes the same set of derivations for each predicate p as the bottom-up evaluation of P on database D.* $\Box$

The proof of the above theorem follows easily from the form of the rules in the rewritten program obtained using Gen-Prop-predicate-constraints, and the fact that Gen-predicate-constraints generates minimum predicate constraints.

One of the main results of this chapter is the following result about when Gen-Prop-QRP-constraints generates and propagates minimum QRP-constraints.

Theorem 4.13 *Consider a CQL program P with linear arithmetic constraints. Let the minimum predicate constraint for each predicate p_i be a finite constraint set, C'_{p_i}. Further, let the constraints in the body of each rule in P imply the constraint set $PTOL(p_i(\overline{X_i}), C'_{p_i})$ for each literal $p_i(\overline{X_i})$ in the body of the rule. Then, if it terminates, Gen-Prop-QRP-constraints generates and propagates the minimum QRP-constraints for each derived predicate of the program.*

Proof: Consider a CQL program P, and let the constraints in the body of each rule in P imply the constraint set $PTOL(p_i(\overline{X_i}), C'_{p_i})$ for each literal $p_i(\overline{X_i})$ in the body of the rule, where C'_{p_i} is the minimum predicate constraint for predicate p_i. For each derived predicate p in P, let C_p be the constraint set generated by Gen-Prop-QRP-constraints. From Theorem 4.8, we already know that it is a QRP-constraint for p. We only need to show that it is the *minimum* QRP-constraint for p.

We prove the result by contradiction. Since the constraints in the body of each rule in P imply the constraint set $PTOL(p_i(\overline{X_i}), C'_{p_i})$ for each literal $p_i(\overline{X_i})$ in the body of the rule, each ground fact $p(\overline{a})$ that satisfies C_p is present in the least model of $\langle P, D\rangle$. Hence, we only need to show that each such fact is constraint relevant to an answer to the query Q.

Let $C_p^0, C_p^1, \ldots$ be the sequence of constraint sets generated by Gen-QRP-constraints as "approximate" QRP-constraints for predicate p. Consider the lowest numbered C_p^k such that there exists a fact $p(\overline{a})$ that satisfies C_p^k and is not constraint relevant to an answer to the query Q. The number k cannot be equal to 0 since C_p^0 is **false** for all non-query predicates, and no p fact can satisfy this constraint; C_p^0 is **true** for the query predicate, and every p fact has to satisfy this constraint.

Since ground fact $p(\overline{a})$ satisfies $C_p^k, k > 0$, it satisfies some disjunct of C_p^k which is not present in C_p^{k-1}. From Gen-QRP-constraints, we know that this disjunct is generated in iteration k of Gen-QRP-constraints using some rule r. Now consider the rule r, and the literal $p(\overline{X})$, which is used to generate this disjunct. By hypothesis, the "approximate" QRP-constraint $C_{p_1}^{k-1}$ for the head $p_1(\overline{X_1})$ is such that each ground fact that satisfies this constraint set is constraint relevant to an answer to query Q.

Since quantifier elimination of linear arithmetic constraint sets can be done exactly, and ground fact $p(\overline{a})$ is in the least model, there is a derivation tree where $p(\overline{a})$ is a child of some $p_1(\overline{a_1})$ (with rule r), which satisfies C_{p_1} and is constraint relevant to an answer to query Q. Consequently, $p(\overline{a})$ is also constraint relevant to the same answer to query Q as $p_1(\overline{a_1})$ is. This contradicts the assumption that $p(\overline{a})$ is not constraint relevant to an answer to query Q, and completes the proof of the theorem. □

4.5.5 Putting it all Together

Given a CQL program P with linear arithmetic constraints, and a query predicate q, our technique for generating and propagating minimum QRP-constraints has two components to it:

1. First, we use Gen-Prop-predicate-constraints for generating and propagating minimum predicate constraints for each predicate in P.

2. Next, we use Gen-Prop-QRP-constraints to generate and propagate the QRP-constraints for each derived predicate in P.

Procedure Constraint-rewrite in Appendix A.2 describes this algorithmically.

The following result follows from Theorems 4.11, 4.12 and 4.13.

Theorem 4.14 *Given a CQL program P with linear arithmetic constraints, if Constraint-rewrite terminates, the QRP-constraints generated and propagated are minimum QRP-constraints for each derived predicate in P.* □

Let us illustrate the result of applying this technique to our original motivating program from Example 4.1.

Example 4.5 (Computing Flights: Rewritten and Optimized) Consider the program P in Example 4.1.

$$\begin{array}{ll} r1: cheaporshort(S,D,T,C) & :- flight(S,D,T,C), T \leq 240. \\ r2: cheaporshort(S,D,T,C) & :- flight(S,D,T,C), C \leq 150. \\ r3: flight(Src,Dst,Time,Cost) & :- leg(Src,Dst,Time,Cost), \\ & \quad Cost > 0, Time > 0. \\ r4: flight(S,D,T,C) & :- flight(S,D1,T1,C1), flight(D1,D,T2,C2), \\ & \quad T = T1 + T2 + 30, C = C1 + C2. \end{array}$$

The query predicate is *cheaporshort*. We first add a rule:

$$r0: q_1(S,D,T,C) :- cheaporshort(S,D,T,C).$$

The new query predicate is q_1. Our algorithm for generating minimum predicate constraints terminates on this example, and concludes that the predicate *flight* has the minimum predicate constraint $(\$3 > 0)\&(\$4 > 0)$. It also concludes that the predicate *cheaporshort* has the minimum predicate constraint $((\$3 > 0)\&(\$3 \leq 240)\&(\$4 > 0)) \vee ((\$3 > 0)\&(\$4 > 0)\&(\$4 \leq 150))$. For each body predicate occurrence of *flight* and *cheaporshort*, we introduce the *PTOL* of the predicate constraints into the rule body.

Now, Gen-Prop-QRP-constraints can be applied to obtain the disjunctive constraint $((\$3 > 0)\&(\$3 \leq 240)\&(\$4 > 0)) \vee ((\$3 > 0)\&(\$4 > 0)\&(\$4 \leq 150))$ as the minimum

QRP-constraint on *flight* as well as *cheaporshort*. Propagating these (minimum) QRP-constraints, and deleting the rules defining q_1 results in the following program P'.

$$
\begin{array}{ll}
r1: cheaporshort(S,D,T,C) & :-\ flight'(S,D,T,C), T>0, T\leq 240, C>0. \\
r2: cheaporshort(S,D,T,C) & :-\ flight'(S,D,T,C), T>0, C>0, C\leq 150. \\
r3: cheaporshort(S,D,T,C) & :-\ flight'(S,D,T,C), T>0, T\leq 240, \\
 & \quad C>0, C\leq 150. \\
r4: flight'(Src,Dst,Time,Cost) & :-\ Time>0, Time\leq 240, \\
 & \quad leg(Src,Dst,Time,Cost), Cost>0. \\
r5: flight'(S,D,T,C) & :-\ T>0, T\leq 240, C>0, \\
 & \quad flight'(S,D1,T1,C1), \\
 & \quad flight'(D1,D,T2,C2), \\
 & \quad T1>0, T2>0, T=T1+T2+30, \\
 & \quad C1>0, C2>0, C=C1+C2. \\
r6: flight'(Src,Dst,Time,Cost) & :-\ Time>0, Cost\leq 150, \\
 & \quad leg(Src,Dst,Time,Cost), Cost>0. \\
r7: flight'(S,D,T,C) & :-\ T>0, C>0, C\leq 150, \\
 & \quad flight'(S,D1,T1,C1), \\
 & \quad flight'(D1,D,T2,C2), \\
 & \quad T1>0, T2>0, T=T1+T2+30, \\
 & \quad C1>0, C2>0, C=C1+C2.
\end{array}
$$

Note that the bottom-up evaluation of P' does not compute any *flight'* fact which is not constraint relevant to the query predicate; in particular, no *flight'* fact with $Time > 240$ and $Cost > 150$ is computed. Further, each of the facts computed during a bottom-up evaluation of P' is a ground fact.

Consider the query:

$$?\ cheaporshort(madison, seattle, Time, Cost).$$

One could now rewrite this program using Magic-sets, and the bound-if-ground rule, to take advantage of the pattern of constants in the actual query. This is *not* something that our optimization (based on generating minimum QRP-constraints and propagating these into the bodies of rules) is designed to do. The magic rewritten program is now able to utilize the various constraints in the program rules, as well as the constants in the query, without computing constraint facts. □

Iteration	*Derivations made*
0	$\{r6 : m\_fib(N1, 5)\}$
1	$\{r4 : m\_fib(N1, V1; N1 > 0, V1 \geq 1, V1 \leq 4)\}$
2	$\{r2 : fib(1, 1)$, **r4: m_fib(N1,V1; N1>0, V1 $\geq$ 1, V1 $\leq$ 3)** $\}$
3	$\{r5 : m\_fib(0, V2; V2 \geq 1, V2 \leq 3), r5 : m\_fib(0, 4)\}$
4	$\{r1 : fib(0, 1)\}$
5	$\{r3 : fib(2, 2)\}$
6	$\{r3 : fib(3, 3)$, **r5: m_fib(1,3), r5: m_fib(1,V2; V2 $\geq$ 1, V2 $\leq$ 2)** $\}$
7	$\{r3 : fib(4, 5)$, **r5: m_fib(2,2), r5: m_fib(2,1)** $\}$
8	$\{\}$

Table 4.18: Derivations in a Bottom-up Evaluation of $P^{mg}_{fib\_1}$

We next show how propagating predicate constraints can make the difference between non-termination and termination in answering queries on a CQL program, even when the evaluation computes constraint facts.

Example 4.6 (Computing Backward Fibonacci: Rewritten and Optimized) Consider the program P_{fib} from Example 4.2.

$r1 : fib(0, 1).$
$r2 : fib(1, 1).$
$r3 : fib(N, X1 + X2) : - N > 1, fib(N - 1, X1), fib(N - 2, X2).$

Note that $\$2 \geq 1$ is a predicate constraint (though not the minimum) for fib. We can associate the *PTOL* of this constraint with each body occurrence of fib in rule $r3$ of P_{fib}. By introducing these constraints, we get the following program P_{fib_1}:

$r1 : fib(0, 1).$
$r2 : fib(1, 1).$
$r3 : fib(N, X1 + X2) : - N > 1, fib(N - 1, X1), X1 \geq 1,$
$\qquad fib(N - 2, X2), X2 \geq 1.$

The Magic-sets transformation of the resultant program is $P^{mg}_{fib\_1}$ shown below:

$$\begin{array}{ll}
r1 : fib(0,1) & :- m\_fib(0,1). \\
r2 : fib(1,1) & :- m\_fib(1,1). \\
r3 : fib(N, X1+X2) & :- m\_fib(N, X1+X2), N > 1, fib(N-1, X1), \\
 & \quad X1 \geq 1, fib(N-2, X2), X2 \geq 1. \\
r4 : m\_fib(N-1, X1) & :- m\_fib(N, X1+X2), N > 1, X1 \geq 1, X2 \geq 1. \\
r5 : m\_fib(N-2, X2) & :- m\_fib(N, X1+X2), N > 1, fib(N-1, X1), \\
 & \quad X1 \geq 1, X2 \geq 1. \\
r6 : m\_fib(N, 5). &
\end{array}$$

A Semi-naive bottom-up evaluation of $P^{mg}_{fib\_1}$ computes facts as shown in Table 4.18. Note that the answer to the query is computed in the seventh iteration, and the evaluation terminates after the eighth iteration, since no new derivations are made during the eighth iteration. The additional constraints in the bodies of rules $r3, r4$ and $r5$ of $P^{mg}_{fib\_1}$ prevent subsequent derivations.

In a similar manner, given the query:

$$?\ fib(N, 6).$$

a Semi-naive bottom-up evaluation of $P^{mg}_{fib\_1}$ (with $r6$ replaced by $m\_fib(N,6)$) terminates, and answers "no" since there is no N whose Fibonacci number is 6. The bottom-up evaluation of P^{mg}_{fib} would not terminate. □

4.5.6 Discussion

Given a CQL program P, Constraint-rewrite generates and propagates minimum QRP-constraints for each derived predicate in the program. Propagating QRP-constraints into rule bodies has several advantages. For instance, in the bottom-up evaluation of program P' of Example 4.5:

1. Fewer $flight'$ facts need be computed (since the constraints in the rules defining $cheaporshort$ are used earlier). In particular, no $flight'$ fact with $Time > 240$ and $Cost > 150$ is computed. Since there could be an arbitrary number of $flight$ facts (in P) with $Time > 240$ and $Cost > 150$, considerable savings (in terms of the number of facts derived) are achieved.

2. These constraints can be used for effective indexing of relations. In particular, the constraints $Cost \leq 150$ and $Time \leq 240$ could be used to efficiently retrieve (via B-trees, etc.) $leg(\_,\_,Time,Cost)$ tuples satisfying this constraint. This can improve the efficiency of rule application.

With disjunctive constraints, however, using fold/unfold transformations may lead to an increase in the number of derivations of each fact, though the number of facts computed may decrease. For instance, if the *leg* relation contained $leg(madison, chicago, 50, 100)$, the original program P in Example 4.5 would derive $flight(madison, chicago, 50, 100)$ just once using the non-recursive rule, whereas $flight'(madison, chicago, 50, 100)$ would be derived twice using the non-recursive rules in P'. Since the number of disjuncts in the minimum QRP-constraint, though finite, is unbounded, the number of derivations could increase considerably, in general.

Notice that this problem of multiple derivations of $flight'$ facts arises because the minimum QRP-constraint for $flight$ is a non-trivial disjunction, and the two disjuncts $((\$3 > 0)\&(\$3 \leq 240)\&(\$4 > 0))$ and $((\$3 > 0)\&(\$4 > 0)\&(\$4 \leq 150))$ "overlap" in that their intersection is satisfiable. There are two possible solutions to this problem:

- First, one can represent the minimum QRP-constraint in such a way that the intersection of no two disjuncts is satisfiable. The algorithms described in [81] can be used to obtain such non-overlapping disjuncts.

  Thus, for instance, the minimum QRP-constraint for $flight$ can be represented as $((\$3 > 0)\&(\$3 \leq 240)\&(\$4 > 0)\&(\$4 \leq 150)) \vee ((\$3 > 0)\&(\$3 \leq 240)\&(\$4 > 0)\&(\$4 > 150)) \vee ((\$3 > 0)\&(\$3 > 240)\&(\$4 > 0)\&(\$4 > 150))$.

  If this representation of the minimum QRP-constraint for $flight'$ is propagated, the rewritten program does not make any more derivations than the original program. However, the number of rules in the rewritten program could increase exponentially, because representing a given constraint set C as a constraint set C' with non-overlapping disjuncts could result in an exponential increase in the number of disjuncts.

- Another possible solution is to bound the number of disjuncts (say, to 1) by simplification of the QRP-constraint, using constraint manipulation techniques. This is always possible, though the result of such a simplification would be a non-minimal QRP-constraint, in general.

Thus, for instance, in Program P in Example 4.5, bounding the number of disjuncts in the QRP-constraint to one, results in obtaining the QRP-constraint as $(\$3 > 0)\&(\$4 > 0)$. Propagating this would not result in any reduction in the number of $flight'$ facts computed.

In practical terms, there is a tradeoff between the number of potentially relevant facts computed, the number of derivations of potentially relevant facts, and the overheads of applying a large number of rules. What choice to make depends on estimates of the relative costs of evaluation of the alternative rewritten programs. We do not discuss this issue further, as it is outside the scope of this thesis.

4.6 Sufficient Conditions for Termination

Given a CQL program P with linear arithmetic constraints, determining whether (any representation for) the minimum QRP-constraint for a predicate p is a finite constraint set is undecidable, according to Theorem 4.5. However, one might be able to obtain decidability results, if we restrict the classes of constraints that we consider in the CQL. In this section, we show that for a restricted class of constraint query languages we can obtain decidability results, and that our technique for generating and propagating the minimum QRP-constraints always terminates.

Example 4.7 Consider program $P1$ of Example 4.4.

$r1:\ q(X,Y) :- a(X,Y), X \leq 10, Y \leq X.$
$r2:\ a(X,Y) :- p(X,Y), Y \leq X.$
$r3':\ a(X,Y) :- a(X,Z), Z \leq X, a(Z,Y), Y \leq Z.$

It is easy to see that, since there are no arithmetic function symbols in the program, and the only arithmetic predicate used is $\leq$, the only "simple" constraints that can be part of the QRP-constraint for a, generated by Gen-QRP-constraints, are: $\$1 \leq 10, \$2 \leq 10, 10 \leq \$1, 10 \leq \$2, \$1 \leq \$1, \$1 \leq \$2, \$2 \leq \$2$, and $\$2 \leq \$1$. No constant other than 10 can be part of the QRP-constraint generated for a, since that would require the use of an arithmetic function symbol to create the new constant.

Each disjunct in a constraint set can contain any or all of these "simple" constraints. Since there are 8 "simple" constraints, there can be at most $2^8 = 256$ disjuncts in the QRP-constraint generated for a. Each iteration of our algorithm to generate QRP-constraints, checks whether the "new" constraints are subsumed by the "approximate"

QRP-constraint, and adds at least one "new" disjunct in each iteration (else it terminates). Since there are only a bounded number (256) of disjuncts *possible*, our algorithm can iterate only 256 times, before it *must* terminate.

In the case of $P1$, our algorithm, Gen-Prop-QRP-constraints, terminates in just two iterations. □

Theorem 4.15 *Consider a constraint query language with linear arithmetic constraints of the form X op Y and X op c, where op is one of $\{\leq, \geq, <, >\}$ and c is a constant. Given a CQL program P with these constraints, there is a terminating algorithm to compute minimum QRP-constraints.*

Proof: Consider a CQL with linear arithmetic constraints of the form $X \ op \ Y$ and $X \ op \ c$, where *op* is one of $\{\leq, \geq, <, >\}$ and c is a constant. In such a constraint query language, no n-ary ($n > 0$) function symbols (such as $+$ or $*$) are permitted. A "simple" constraint (on the arguments of a predicate) in a program in such a CQL can either be of the form $\$i \leq c, \$i < c, c \leq \$i, c < \$i, \$i \leq \$j$, or $\$i < \$j$. (A simple constraint involving $\geq$ or $>$ can be treated as an equivalent constraint using $\leq$ or $<$.) If predicate p has arity k, there can be at most $2k^2 + 4k$ "simple" constraints that can be part of the predicate constraint or the QRP-constraint generated for p. (There can be k^2 constraints each of the forms $\$i \leq \$j$ and $\$i < \$j$, and k constraints each[5] of the forms $\$i \leq c, \$i < c, c \leq \$i$ and $c < \$i$.) Consequently, there can be at most 2^{2k^2+4k} disjuncts in the predicate constraint or the QRP-constraint for p.

If program P contains n predicates, with arity at most k, it is easy to see that at most $n * 2^{2k^2+4k}$ disjuncts are possible in the predicate or the QRP-constraints generated for predicates in P. Since each iteration of Gen-predicate-constraints and Gen-QRP-constraints adds at least one "new" disjunct, each of these algorithms terminates in at most $n * 2^{2k^2+4k}$ iterations.[6] Consequently, Constraint-rewrite terminates on this class of programs, having generated and propagated minimum QRP-constraints. This also gives us the decidability result. □

This result can be easily generalized to constraint query languages with no n-ary ($n > 0$) function symbols, and only a finite number of constraint predicate symbols.

[5]Even if there are, say two constants c_1 and c_2 in the program, in each disjunct there can only be one of $\$i \leq c_1$ and $\$i \leq c_2$ present. The other constraint is redundant.

[6]This is a combinatorial upper-bound for the number of iterations taken. For most programs, we expect the bound to be considerably lower.

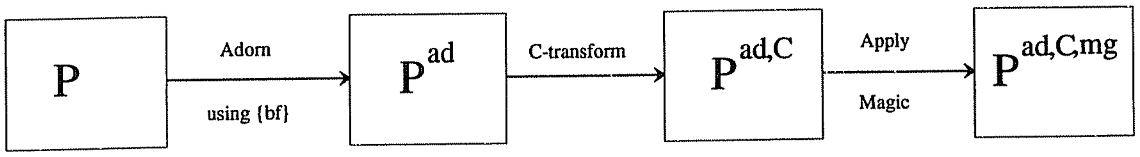

Figure 4.9: The Transformation of Balbin et al.

4.7 Understanding Previous Techniques

Balbin et al. [4] and Mumick et al. [56] consider the problem of propagating constraints such as $X > 10$ in a range-restricted, function-free CQL program P.[7] Both [4] and [56] use a combination of constraint propagation and Magic-sets.

A fundamental limitation of each of these techniques is that they do not utilize semantic properties of constraints. The techniques presented in this chapter make essential use of such properties, and are hence able to optimize programs that could not be handled by previous techniques.

4.7.1 Balbin et al.'s C-transformation

The approach taken by Balbin et al. [4] is to try to propagate constraints using (a more limited version than we consider of) fold/unfold and then apply the Magic-sets transformation. Given a program P, their technique is depicted in Figure 4.9. It can be split into three phases:

1. First, they have an adornment phase, which uses the bf adornment, where b stands for bound, and f for free.

   An argument is treated as bound only if it is bound to a ground term. Let the adorned program obtained be P^{ad}.

2. Second, they perform a C-transformation on the adorned program. This is expressed as a sequence of fold, unfold, and definition steps using the fold/unfold transformations of Tamaki and Sato [88].

   This step propagates constraints into the recursive rules, while obtaining a query-equivalent program. A constraint in a rule body is treated as any other rule body

[7]Range-restrictedness is a sufficient syntactic condition for all the facts computed during the bottom-up evaluation of a program to be ground facts.

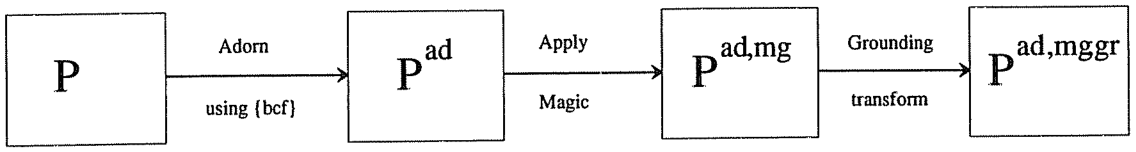

Figure 4.10: The Transformation of Mumick et al.

literal for the purposes of the transformation.[8] Let the C-transformed program be $P^{ad,C}$.

3. In the third phase, they perform the Magic-sets rewriting on the C-transformed program. The resultant program is denoted $P^{ad,C,mg}$.

Thus, the approach of [4] is to C-transform the (adorned) program before the Magic-sets rewriting is applied. Further, given a program P which has only range-restricted rules, the transformation of [4] has the property that each of P^{ad}, $P^{ad,C}$ and $P^{ad,C,mg}$ has only range-restricted rules. Consequently, all the facts computed during the bottom-up evaluation of the C-transformed program as well as the magic rewritten program are ground facts.

Our algorithms for generating and propagating minimum QRP-constraints can be straightforwardly used to replace the constraint propagation phase of [4]'s technique. The resulting technique can optimize a larger class of programs than [4].

4.7.2 Mumick et al.'s GMT-transformation

The approach taken by Mumick et al. [56] (the Ground Magic-sets transformation, or GMT) directly extends the Magic-sets rewriting of [64] to support propagation of arithmetic constraints, without leading to computation of constraint facts. Although, [56] presents the GMT-transformation as a single algorithm that combines Magic-sets with the propagation of constraints, to understand the GMT-transformation, it is best to think of GMT as a three step transformation, as shown in Figure 4.10:

1. Given a program P, the first step is an adornment phase.

   They generalize the class of bound (b) and free (f) adornments to include a condition (c) adornment that describes selections involving arithmetic inequalities. They

[8][4] refers to constraints in P as *constraining predicates*; for instance, $>$ is a constraining predicate and $X > 3$ is a constraining literal.

describe how sideways information passing strategies (sips) can be modified to allow conditions, in addition to bindings, to be passed sideways. Let the adorned program obtained be P^{ad}.

2. In the second step, they take a *bcf* adorned program, and apply the Magic-sets transformation of [64] to get a program that may have non-range restricted magic rules. Let the Magic-sets transformed program be $P^{ad,mg}$.

3. Finally, they *ground* the magic rules in $P^{ad,mg}$ to get a range restricted program $P^{ad,mg,gr}$.

We show in [82] that this final (and quite complicated) grounding step can be understood as a sequence of fold/unfold transformations using the system of Tamaki and Sato [88]. Thus, the technique of [56] can also be decomposed into a combination of the Magic-sets rewriting and the fold/unfold transformation, each of which is well-understood.

Semantic properties of constraints can also be used to enhance the adornment phase in the GMT-algorithm of [56], and permit a larger class of programs to be optimized.

In contrast to [4], the approach of [56] is to magic transform the (adorned) program *before* applying the fold/unfold transformations. This intermediate (magic) program $P^{ad,mg}$ could compute constraint facts; however, the final program obtained $P^{ad,mg,gr}$ has only range restricted rules, and hence computes only ground facts.

4.8 Combining Constraint Propagation with Magic Rewriting

Consider a CQL program P, with linear arithmetic constraints. We described two algorithms, Gen-Prop-predicate-constraints and Gen-Prop-QRP-constraints, that preserved the core of P while propagating constraints that occur in the program P. (A combination of these algorithms generated and propagated minimum QRP-constraints.) The rewritten programs obtained using these two algorithms preserve equivalence with respect to every possible query on the query predicate. An advantage of these techniques is that if the evaluation of the original program computed only ground facts, so do the evaluations of the rewritten programs. A shortcoming of these techniques for propagating constraints is that they are not able to take advantage of the pattern of constants in the actual query, known only at run-time, or the actual set of facts in the database predicates.

Magic-sets [64] is a rewriting strategy that *is* able to take advantage of constants in the actual query, as well as the actual set of facts in the database predicates, thereby restricting the computation to facts that are potentially relevant to answering a given query. However, a shortcoming of this technique is that the bottom-up evaluation of the Magic-sets transformed program could compute constraint facts, even if the bottom-up evaluation of the original program computed only ground facts. There are two cases in which the Magic-sets transformation computes only ground facts:

- First, when we use the bound-if-ground rule (equivalently, the class of bf adornments), where an argument of a subgoal is treated as bound only if it is ground; it is free otherwise.
- Second, when we use Mumick et al's [56] class of bcf adornments for groundable programs, with grounding sips, in conjunction with their GMT algorithm.

An important question is how Magic-sets interacts with the use of Gen-Prop-predicate-constraints and Gen-Prop-QRP-constraints in these cases.

In the rest of this section, we describe this interaction in detail for the case when an argument of a subgoal is treated as bound only it is ground, i.e., the Magic-sets transformation uses the class of bf adornments. In Section 4.8.7, we briefly consider how the interaction of Magic and Gen-Prop-predicate-constraints and Gen-Prop-QRP-constraints can be explored further for the case of bcf adornments.

We use the following notation. Let P^{pred} be the program obtained from P using Gen-Prop-predicate-constraints, P^{qrp} be the program obtained from P using Gen-Prop-QRP-constraints, and P^{mg} be the program obtained from program P using Magic-sets rewriting. (Thus, $P^{pred,qrp}$ is the program obtained by first applying Gen-Prop-predicate-constraints to P and then applying Gen-Prop-QRP-constraints on the resultant program P^{pred}; etc.)

4.8.1 Problems Addressed

Consider a CQL program P, with linear arithmetic constraints, such that the bottom-up evaluation of P computes only ground facts. The program P can be successively optimized using a sequence of applications of Gen-Prop-predicate-constraints (to generate and propagate minimum predicate constraints), Gen-Prop-QRP-constraints (to generate and propagate QRP-constraints), and the Magic-sets rewriting (to propagate binding information). An important question concerns their interaction. Our main result is that:

Given a CQL program P with linear arithmetic constraints, the rewritten program obtained by first applying Gen-Prop-predicate-constraints, followed by applying Gen-Prop-QRP-constraints, and finally applying the Magic-sets rewriting, that is $P^{pred,qrp,mg}$, is optimal among *all* rewritten programs obtained from P by using a sequence of applications of the three rewritings, where the Magic-sets rewriting can be applied exactly once, and the other two rewritings can be applied any number of times in the sequence.

We also show that:

- Given a CQL program P with linear arithmetic constraints, Magic-sets rewriting and Gen-Prop-QRP-constraints are not confluent, i.e., the order in which these two rewritings are applied on a program affects the final resultant program.

  For some programs P, the program $P^{qrp,mg}$ computes fewer facts (for all EDBs and queries) than the program $P^{mg,qrp}$; for other programs, the program $P^{mg,qrp}$ computes fewer facts than $P^{qrp,mg}$. We also identify conditions when it is more advantageous to apply Gen-Prop-QRP-constraints first.

- Given a CQL program P with linear arithmetic constraints, Constraint-rewrite (which is a sequence of applications of procedures Gen-Prop-predicate-constraints and Gen-Prop-QRP-constraints) and the Magic-sets rewriting are not confluent either.

  However, as shown by our main result, the program $P^{pred,qrp,mg}$ (obtained by applying Constraint-rewrite first, and then applying the Magic-sets rewriting) always computes fewer facts than the program $P^{mg,pred,qrp}$.

The Magic-sets rewriting that we consider for our results is one in which all the constraint information present in a rule r in program P is also present in each magic rule generated from rule r. We refer to this as *constraint magic rewriting*, described next.

4.8.2 Constraint Magic Rewriting

Consider a CQL program P. The Magic-sets rewriting of P to P^{mg} is said to be a *constraint magic rewriting* iff the following condition is satisfied. Let r be any rule in P of the form:

$$r : p(\overline{X}) : - C_r, p_1(\overline{X_1}), \ldots, p_n(\overline{X_n}).$$

where C_r is the comma separated list of constraints in the body of rule r. Let mr_i be a magic rule in P^{mg}, obtained from body literal $p_i(\overline{X_i})$ in rule r, of the form:

$$mr_i : m\_p_i(\overline{X_i}^1) : -\ m\_p(\overline{X}^1), C_{mr_i}, \ldots.$$

C_{mr_i} is the conjunction of constraints in the body of rule mr_i. For each such mr_i in P^{mg}, let $\overline{Y_i}$ be the variables appearing in the (head and body of the) rule. Then, for each rule r in P and each such mr_i in P^{mg}, it should be the case that:

$$\Pi_{\overline{Y_i}}(C_r) \equiv \Pi_{\overline{Y_i}}(C_{mr_i})$$

The intuition is that all the constraints in rule r that are "relevant" to the magic rule mr_i should be present in the body of rule mr_i. We can guarantee that a Magic-sets rewriting is a constraint magic rewriting if the constraints in the body of a rule are to the left of every other body literal.

Proposition 4.16 *Consider a CQL program P with linear arithmetic constraints, such that the bottom-up evaluation of P computes only ground facts. Let P^{mg} be the result of the constraint magic rewriting of P. Then, the bottom-up evaluation of P^{mg} computes only ground facts, and is query equivalent to P for all input databases.* □

4.8.3 Combining Propagation of QRP-constraints with Constraint Magic Rewriting

Each of the rewritings, Gen-Prop-QRP-constraints and constraint magic rewriting, propagates information available on the head of a rule to predicates occurring in the body of the rule. Gen-Prop-QRP-constraints propagates constraint information, while constraint magic rewriting propagates information about the pattern of constants in the actual query, and the facts in the actual database. Consequently, it is of interest to determine whether these two rewritings are confluent, i.e., does the order in which these two rewritings are applied on a program affect the final resultant program?

We first describe example programs to show that the two rewritings are *not* confluent.

Example 4.8 Consider the following program P:

$$\begin{aligned} r1 &: q(X,Y) \ : -\ a1(X,Y), X \leq 4. \\ r2 &: a1(X,Y) : -\ b1(X,Z), a2(Z,Y). \\ r3 &: a2(X,Y) : -\ b2(X,Y). \\ r4 &: a2(X,Y) : -\ b2(X,Z), a2(Z,Y). \end{aligned}$$

where q is the query predicate, queried with all its arguments free. For this program P, it is preferable to apply Gen-Prop-QRP-constraints followed by the constraint magic rewriting, independent of the facts in the database. The programs obtained by applying the rewritings in different orders are discussed in Srivastava and Ramakrishnan [82]. □

The above example also illustrates that Constraint-rewrite and constraint magic rewriting are *not* confluent either.

Example 4.9 Consider the following program P:

$$
\begin{aligned}
&r1 : q(X,Y) \ \ :- a1(X,Y).\\
&r2 : a1(X,Y) :- b1(X,Z), X \leq 4, a2(Z,Y).\\
&r3 : a2(X,Y) :- b2(X,Y).\\
&r4 : a2(X,Y) :- b2(X,Z), a2(Z,Y).
\end{aligned}
$$

where q is the query predicate, queried with its first argument bound to a constant and the second argument free. For this program P, it is preferable to apply constraint magic rewriting followed by Gen-Prop-QRP-constraints, independent of the facts in the database, and pattern of constants in the actual query. The programs obtained by applying the rewritings in different orders are discussed in Srivastava and Ramakrishnan [82]. □

Examples 4.8 and 4.9 show that, in general, no ordering of Gen-Prop-QRP-constraints and constraint magic rewriting is always superior to the other. However, for a restricted class of CQL programs, we can show that applying Gen-Prop-QRP-constraints followed by constraint magic rewriting is superior to applying constraint magic rewriting followed by applying Gen-Prop-QRP-constraints. Theorem 4.17 below identifies conditions on the form of such programs.

Theorem 4.17 *Consider a CQL program P, such that the bottom-up evaluation of P computes only ground facts. Also, let r be a rule in P of the form:*

$$r : p(\overline{X}) :- C_r, p_1(\overline{X_1}), \ldots, p_n(\overline{X_n}).$$

Let C_r imply $PTOL(p_i(\overline{X_i}), C'_{p_i}), 1 \leq i \leq n$, where C'_{p_i} is the minimum predicate constraint for predicate p_i.

Then, for all databases D and pattern of constants in the actual query, the bottom-up evaluation of $P^{qrp,mg}$ on database D computes a subset of the set of facts computed by the bottom-up evaluation of $P^{mg,qrp}$ on database D.

Proof: Consider a CQL program P satisfying the conditions of the theorem. Consider any rule r in P:

$$r : p(\overline{X}) :- C_r, p_1(\overline{X_1}), \ldots, p_n(\overline{X_n}).$$

In the magic program P^{mg}, the modified rule and the magic rules generated from rule r in P are of the form:

$$\begin{array}{lll} r : & p(\overline{X}) & :- C_r, m\_p(\overline{X}^1), p_1(\overline{X_1}), \ldots, p_n(\overline{X_n}). \\ mr_1 : & m\_p_1(\overline{X_1}^1) & :- C_r, m\_p(\overline{X}^1). \\ \vdots & & \\ mr_n : & m\_p_n(\overline{X_n}^1) & :- C_r, m\_p(\overline{X}^1), \ldots. \end{array}$$

Again, consider rule r in P, and the propagation of QRP-constraints into the body of rule r using Gen-Prop-QRP-constraints. Each of the resulting rules[9] in P^{qrp} is of the form:

$$r : p(\overline{X}) :- C_p, C_r, p_1(\overline{X_1}), \ldots, p_n(\overline{X_n}).$$

where C_p is a disjunct in the QRP-constraint propagated for p. Since the rules in P^{qrp} have at least as many constraints as the corresponding rule r in P, constraint magic rewriting guarantees that each rule in $P^{qrp,mg}$ obtained from rule r in P^{qrp} has at least as many constraints as the corresponding rules in P^{mg}.

Hence, to prove the theorem we only need prove that the QRP-constraints generated and propagated by Gen-Prop-QRP-constraints for each (magic and non-magic) derived predicate p in P^{mg} are implied by the conjunction of constraints present in the body of each rule defining p in $P^{qrp,mg}$.

First consider a non-magic derived predicate p in P^{mg} and $P^{qrp,mg}$. The constraints present in the body of each rule defining p in $P^{qrp,mg}$ include, in addition to the constraints present in the corresponding rule in P^{mg}, a disjunct from the QRP-constraint generated for p in P (and propagated in P^{qrp}).

Claim 1: If $C1_p$ is the QRP-constraint generated for predicate p in P, and $C2_p$ is the QRP-constraint generated for predicate p in P^{mg}, then $C1_p \supset C2_p$.

Proof of Claim 1: The QRP-constraint obtained for p in P is based on occurrences of p literals in the body of rules in P, and corresponding literal constraints. In P^{mg}, each of these occurrences is present (in modified original rules); in addition, there are some

[9]There could be many such rules if the QRP-constraint propagated is a non-trivial disjunction.

more occurrences of p literals in the bodies of magic rules. Using induction, it can be shown that at the end of each iteration i of Gen-QRP-constraints, the "approximate" QRP-constraint $C1_p^i$ (the disjunction of $C1_p^{i-1}$ and the literal constraints on occurrences of p in the i'th iteration) implies the "approximate" QRP-constraint $C2_p^i$. The claim follows from the monotonic nature of the non-recursive inference in the generation of QRP-constraints.

Next, consider a magic predicate m_p in P^{mg} and $P^{qrp,mg}$.

Claim 2: If $C1_{m\_p}$ is the conjunction of constraints in the body of a rule defining m_p in $P^{qrp,mg}$, and $C2_{m\_p}$ is the QRP-constraint generated for predicate m_p in P^{mg}, then $C1_{m\_p} \supset C2_{m\_p}$.

Proof of Claim 2: (Sketch) The QRP-constraint generated for predicate m_p in P^{mg} is obtained from rules containing body occurrences of m_p. These are the (modified original) rules defining p in P^{mg} and the magic rules obtained from the rules defining p in P. Using simultaneous induction on the iterations of Gen-QRP-constraints and Gen-predicate-constraints, it can be shown that $C2_{m\_p}$, the QRP-constraint generated for predicate m_p in P^{mg} is implied by the predicate constraint C_p' for predicate p in P. By hypothesis, the conjunction of constraints in the body of each rule containing a body occurrence of p implies the $PTOL$ of C_p' on this literal. Magic rules defining m_p in $P^{qrp,mg}$ are obtained from such body occurrences of p. The claim follows from the property of constraint magic rewriting (the relationship between the constraints present in a magic rule and the rule from which it is generated) in obtaining $P^{qrp,mg}$ from P^{qrp}.

This concludes the proof of the theorem. □

4.8.4 Combining Predicate and QRP-constraint Propagation

Consider a CQL program P, with linear arithmetic constraints, such that the bottom-up evaluation of P computes only ground facts. In this section, we show several results about the rewritten program obtained by combining Gen-Prop-predicate-constraints and Gen-Prop-QRP-constraints on program P.

Theorem 4.18 *Consider a CQL program P, with linear arithmetic constraints, such that the bottom-up evaluation of P computes only ground facts. Then, the bottom-up evaluation of $P^{pred,qrp}$ computes a subset of the facts computed by the bottom-up evaluation of $P^{qrp,pred}$.*

Proof: Consider a CQL program P satisfying the conditions of the theorem. From Theorem 4.14, it follows that the evaluation of $P^{pred,qrp}$ computes a subset of the facts computed by the evaluation of P^{qrp}. From Theorem 4.12, it follows that the evaluation of P^{qrp} computes the same set of facts as the evaluation of $P^{qrp,pred}$. Combining these two results, we have a proof of the theorem. □

The following result indicates that consecutive applications of Gen-Prop-predicate-constraints on a program are redundant.

Theorem 4.19 *Consider a CQL program P, with linear arithmetic constraints. Then, for each predicate p, the minimum predicate constraint $C1_p$ on predicate p in P^{pred} is equivalent to C_p, the minimum predicate constraint on predicate p in P.*

If the bottom-up evaluation of P computes only ground facts, the bottom-up evaluation of $P^{pred,pred}$ computes the same set of facts for each program predicate as the bottom-up evaluation of P^{pred}. □

The first half of the result follows from the definition of minimum predicate constraints. The second half of the result is a corollary of Theorem 4.12. The following result indicates that consecutive applications of Gen-Prop-QRP-constraints on a program are redundant.

Theorem 4.20 *Consider a CQL program P, with linear arithmetic constraints. Then, for each predicate p, the QRP-constraint $C1_p$ generated by Gen-QRP-constraints on predicate p in P^{qrp} is equivalent to C_p, the QRP-constraint generated by Gen-QRP-constraints on predicate p in P.*

If the bottom-up evaluation of P computes only ground facts, the bottom-up evaluation of $P^{qrp,qrp}$ computes the same set of facts for each program predicate as the bottom-up evaluation of P^{qrp}. □

The result can be shown by induction on the iterations of Gen-QRP-constraints on P and P^{qrp}. From Theorems 4.19 and 4.20, it follows that in rewriting a program using Gen-Prop-predicate-constraints and Gen-Prop-QRP-constraints, one only needs to alternate between the two rewritings; consecutive applications of the same rewriting are redundant.

From Theorem 4.14, we know that $P^{pred,qrp}$ is the rewritten program obtained by generating and propagating the minimum QRP-constraints for predicates in P. The next result shows that more than one alternation of Gen-Prop-predicate-constraints and Gen-Prop-QRP-constraints is redundant.

Theorem 4.21 *Consider a CQL program P, with linear arithmetic constraints. Then,*

- *The minimum predicate constraint $C1_p$ on predicate p in $P^{pred,qrp}$ is equivalent to C_p, the minimum QRP-constraint on predicate p in P.*
- *The minimum QRP-constraint $C2_p$ on predicate p in $P^{pred,qrp}$ is equivalent to C_p, the minimum QRP-constraint on predicate p in P.* □

As a corollary to the second part of the above theorem, we have:

Corollary 4.22 *Consider a CQL program P, with linear arithmetic constraints. If the bottom-up evaluation of P computes only ground facts, P^{S1}, where $S1$ is the sequence $\{pred, qrp, pred, qrp\}$, computes the same set of facts for each program predicate as $P^{pred,qrp}$.* □

4.8.5 Adding Constraint Magic Rewriting

Consider a CQL program P with linear arithmetic constraints. In this section, we discuss properties about a sequence of applications of Gen-Prop-predicate-constraints, Gen-Prop-QRP-constraints, and constraint magic rewriting on P. Example 4.8 shows that constraint magic rewriting and the sequence of rewritings, Gen-Prop-predicate-constraints and Gen-Prop-QRP-constraints are not confluent, that is, $P^{pred,qrp,mg}$ does not compute the same set of facts as $P^{mg,pred,qrp}$. In the example, $P^{pred,qrp,mg}$ computes fewer facts than $P^{mg,pred,qrp}$ for all input databases. We now show that this is true, in general.

Theorem 4.23 *Consider a CQL program P with linear arithmetic constraints, such that the bottom-up evaluation of P computes only ground facts. Then, for all databases D and pattern of constants in the actual query, the bottom-up evaluation of $P^{pred,qrp,mg}$ on database D computes a subset of the set of facts computed by the bottom-up evaluation of $P^{mg,pred,qrp}$ on database D.*

Proof: Consider a CQL program P satisfying the conditions of the theorem. Using arguments similar to the proof of Theorem 4.17, it can be seen that each rule in $P^{pred,qrp,mg}$ has at least as many constraints as the corresponding rules in P^{mg}.

Hence, to prove the theorem we only need prove that the minimum QRP-constraints generated and propagated for each (magic and non-magic) derived predicate p in P^{mg} are implied by the conjunction of constraints present in the body of each rule defining p in $P^{pred,qrp,mg}$. First, we prove certain results about the application of Gen-Prop-predicate-constraints on P^{mg}.

Claim 1: The minimum predicate constraints generated for each non-magic predicate p and the corresponding magic predicate m_p in P^{mg} by Gen-predicate-constraints are implied by the minimum QRP-constraints generated for predicate p in P.

Proof of Claim 1: (Sketch) The claim is proved by induction on the SCC structure of P^{mg}.

For the base case, consider the magic predicate m_q corresponding to the query predicate q in P^{mg} and the database predicates in P^{mg}. The minimum predicate constraint on the magic predicate m_q is given to be **true**, since every query is possible. Since the minimum predicate constraints on the database predicates in P are the same as the minimum predicate constraints on the corresponding predicates in P^{mg}, the minimum QRP-constraints on the database predicates in P trivially imply the minimum predicate constraints on the database predicates in P^{mg}.

Now consider the induction step. Consider SCC S_i of P^{mg} and predicate m_p in S_i. The minimum predicate constraint obtained for m_p in P^{mg} is based on the predicate occurrences in the body of the rules defining m_p, and the minimum predicate constraints for those predicates in P^{mg}. For each rule mr_i in P^{mg} defining m_p, consider the rule r in P from which it is generated. The body of rule r in P contains occurrences of all the body predicates and constraints occurring in the body of mr_i. In the process of computing the minimum QRP-constraint for p in P, Gen-Prop-predicate-constraints propagated the minimum predicate constraint associated with each literal in the body of r in P. Theorem 4.21 guarantees that the minimum QRP-constraint for p in P would not be different had the minimum QRP-constraint constraint been associated with each literal in the body of r in P. By the induction hypothesis, this minimum QRP-constraint for a predicate p_1 in the body of r implies the minimum predicate constraint for the corresponding predicate p_1 in the body of mr_i, if p_1 is defined in a lower SCC of P^{mg}. If p_1 in the body of mr_i is not defined in a lower SCC of P^{mg}, it has to be defined in S_i. The proof now requires an additional induction on the iterations performed by Gen-predicate-constraints on P^{mg}.

From this it follows that the minimum predicate constraint on m_p in P^{mg} is implied by the minimum QRP-constraint on p in P.

Consider now predicate p defined in S_i of P^{mg}. Again, the minimum predicate constraint for p depends on the minimum predicate constraints associated with predicates in the bodies of rules defining p. The rules defining p in P^{mg} are similar to rules defining p in P; the only additional literal is an occurrence of m_p. Again, an application of Theorem 4.21 and the form of the minimum predicate constraints for m_p ensures that the minimum predicate constraint on p in P^{mg} is implied by the minimum QRP-constraint on p in P. This concludes the induction step, and the proof of the claim.

Claim 2: If $C1_p$ is the minimum QRP-constraint generated for predicate p in P, and $C2_p$ is the minimum QRP-constraint generated for predicate p in P^{mg}, then $C1_p \supset C2_p$.

Proof of Claim 2: This is very similar to the proof of Claim 1 in the proof of Theorem 4.17, except that it additionally requires an application of Theorem 4.21 along with the above Claim 1.

Claim 3: If $C1_{m\_p}$ is the conjunction of constraints in the body of a rule defining m_p in $P^{pred,qrp,mg}$, and $C2_{m\_p}$ is the minimum QRP-constraint generated for predicate m_p in P^{mg}, then $C1_{m\_p} \supset C2_{m\_p}$.

Proof of Claim 3: This is very similar to the proof of Claim 2 in the proof of Theorem 4.17, except that it additionally requires an application of Theorem 4.21 along with the above Claim 1.

By combining Claims 2 and 3, we have a proof of the theorem. □

The following theorem shows that generating and propagating predicate constraints on $P^{pred,qrp}$ prior to constraint magic rewriting is redundant.

Theorem 4.24 *Consider a CQL program P with linear arithmetic constraints, such that the bottom-up evaluation of P computes only ground facts. Then, the bottom-up evaluation of P^{S1}, where $S1$ is the sequence $\{pred, qrp, pred, mg\}$ computes the same set of facts for each predicate as the bottom-up evaluation of P^{S2}, where $S2$ is the sequence $\{pred, qrp, mg\}$.*

Proof: Consider a CQL program P satisfying the conditions of the theorem.

Claim 1: Consider a rule r in $P^{pred,qrp}$ of the form:

$$r : p(\overline{X}) :- C_r, p_1(\overline{X_1}), \ldots, p_n(\overline{X_n}).$$

where C_r is the conjunction of constraints in the body of the rule. Let C_{p_i} be the minimum QRP-constraint on predicate p_i in P. Then,

$$LTOP(p_i(\overline{X_i}), \Pi_{\overline{X_i}}(C_r)) \supset C_{p_i}.$$

Proof of Claim 1: The proof is a direct consequence of the fact that the conjunction of constraints in the body of each rule in $P^{pred,qrp}$ is stronger than the conjunction of constraints in the body of the corresponding rule in P.

Claim 2: Let $p_i(\overline{X_1}), \ldots, p_i(\overline{X_m})$ be all the body occurrences of p_i in $P^{pred,qrp}$. Let C_{p_i} be the minimum QRP-constraint on predicate p_i in P. Then,

$$\bigvee_{j=1}^{m} LTOP(p_i(\overline{X_j}), \Pi_{\overline{X_j}}(C_{r,j})) \equiv C_{p_i}$$

where $C_{r,j}$ is the conjunction of constraints in the body of the rule containing the occurrence $p_i(\overline{X_j})$.

Proof of Claim 2: The conjunction of constraints $LTOP(p_i(\overline{X_j}), \Pi_{\overline{X_j}}(C_{r,j}))$ restricts the set of p_i facts that can be used in literal $p_i(\overline{X_j})$. Since C_{p_i} is the *minimum* QRP-constraint on predicate p_i, each ground p_i fact that satisfies C_{p_i} must satisfy at least one of the $LTOP$'s of the various p_i literals, else that fact would be a witness for the non-minimality of C_{p_i}. Consequently, $C_{p_i} \supset$ the disjunction of the $LTOP$'s. From Claim 1, we have that the disjunction of the $LTOP$'s $\supset C_{p_i}$. Combining the two, we have a proof of the claim.

From Theorem 4.21, we know that the minimum predicate constraint $C1_p$ on predicate p in $P^{pred,qrp}$ is equivalent to C_p, the minimum QRP-constraint on predicate p in P. From the above two claims, it follows that propagating $C1_p$ (using Gen-Prop-predicate-constraints) would not change the constraints associated with any rule. Consequently, applying constraint magic rewriting on $P^{pred,qrp}$ and $P^{pred,qrp,pred}$ would result in equivalent corresponding (non-magic and magic) rules. This proves the desired result. □

4.8.6 An Optimal Sequence of Transformations

We now show that $P^{pred,qrp,mg}$ is optimal among a class of transformation sequences on program P.

Theorem 4.25 *Consider a CQL program P with linear arithmetic constraints, such that the bottom-up evaluation of P computes only ground facts. Consider the class of all programs obtained from P using a sequence of applications of Gen-Prop-predicate-constraints, Gen-Prop-QRP-constraints, and constraint magic rewriting, such that constraint magic rewriting is applied only once. Among all such programs, $P^{pred,qrp,mg}$ is optimal in that it computes a subset of the facts computed by any other program from this class, for all input databases.*

Proof: Consider a CQL program P satisfying the conditions of the theorem, and consider the class of programs obtained from P using a sequence of applications of Gen-Prop-predicate-constraints, Gen-Prop-QRP-constraints and constraint magic rewriting, such that constraint magic rewriting is applied only once.

Such programs can be denoted by P^{S1}, where $S1$ is the sequence of transformations:

$$\{pred^{i_1}, qrp^{j_1}, \ldots, pred^{i_k}, qrp^{j_k}, mg, pred^{i_{k+1}}, qrp^{j_{k+1}}, \ldots, pred^{i_n}, qrp^{j_n}\}$$

From Theorems 4.19, 4.20 and 4.21, it follows that any such program computes the same set of facts for each program predicate as the program P^{S2}, where $S2$ is the sequence of transformations $\{pred, qrp, pred, mg, pred, qrp\}$. From Theorem 4.23, it follows that the program P^{S3}, where $S3$ is the sequence $\{pred, qrp, pred, pred, qrp, mg\}$ computes a subset of the set of facts computed by P^{S2}, for all input databases. Using another application of Theorems 4.19, 4.21, and 4.24, we have that P^{S3} is equivalent to $P^{pred,qrp,mg}$ for all input databases. This gives us the desired optimality result. □

The importance of Theorem 4.25 is in prescribing an optimal order in which to apply Gen-Prop-predicate-constraints, Gen-Prop-QRP-constraints and the constraint magic rewriting on program P, if we desire a rewritten program P' such that the bottom-up evaluation of P':

- utilizes constraint information present in the program P,
- utilizes the actual facts present in the database and the pattern of constants in the actual query, and
- computes only ground facts.

To summarize, the optimal order for rewriting a CQL program P with linear arithmetic constraints, is as follows:

1. Apply Gen-Prop-predicate-constraints on P to generate and propagate minimum predicate constraints; let the resultant program be P^{pred}.
2. Apply Gen-Prop-QRP-constraints on P^{pred} to generate and propagate minimum QRP-constraints; let the resultant program be $P^{pred,qrp}$.
3. Apply constraint magic transformation on $P^{pred,qrp}$ to obtain $P^{pred,qrp,mg}$.

The program $P^{pred,qrp,mg}$ is the resultant optimized program.

4.8.7 Other Classes of Adornments

A natural question to ask is how do Gen-Prop-predicate-constraints, Gen-Prop-QRP-constraints and constraint magic rewriting interact when given a program P adorned using a class of adornments different from bf (or, the bound-if-ground rule for arguments of subgoals). One such possibility is the class of bcf adornments of Mumick et al. [56], where the 'c' adornment denotes an argument that is independently constrained.

Given a bcf adorned CQL program P with linear arithmetic constraints, such that the bottom-up evaluation of P computes only ground facts, it can be seen easily that the rewritten program obtained by applying either of Gen-Prop-predicate-constraints or Gen-Prop-QRP-constraints also computes only ground facts. Further, each of these transformations also preserves the semantics of the 'c' adornment. Intuitively, this is because these transformations only add constraints to rule bodies without altering the core of the program.

The main problem is that for the class of bcf adornments, constraint magic rewriting alone does not guarantee that the evaluation of the rewritten program P^{mg} computes only ground facts, even if the evaluation of the original program P computed only ground facts. However, for groundable programs [56], one could replace the constraint magic rewriting described in Section 4.8.2 by (a suitably enhanced version of) the GMT algorithm, to obtain a magic rewriting that computes only ground facts.

We conjecture that our results for the class of bf adornments can be extended to the class of groundable programs with bcf adornments as well. The intuition behind this conjecture is that each of the three transformations for the class of groundable programs with bcf adornments preserves the semantics of the 'c' adornment. Further, none of the proofs of the various results relied on the actual adornment class used by the program.

4.9 Discussion

We formally defined the problem of propagating constraints occurring in a program without altering the syntactic program structure, using the notion of minimum query relevant predicate constraints. If these constraints are propagated into a program, the rewritten program is query equivalent to the original program (on all input EDBs), and computes only facts that are constraint relevant to the program on an input EDB. We showed that the problem of determining whether any representation for the minimum QRP-constraints for program predicates is finite is undecidable in general, for linear arithmetic constraints. We presented two algorithms:

1. for generating and propagating minimum predicate constraints based on the definition of predicates, and
2. for generating and propagating query relevant predicate constraints based on the uses of predicates.

We showed that a combination of these algorithms generates and propagates minimum QRP-constraints, if it terminates. It is the generation aspect that is non-terminating; once we have a finite minimum QRP-constraint, the propagation uses fold/unfold transformations, and always terminates. We also identified a class of programs for which our technique terminates.

We described a uniform framework—namely, a combination of Magic-sets and (possibly simpler versions of) our algorithms for generating and propagating minimum predicate constraints and QRP-constraints—for the results of this chapter, and for related work in the literature for propagating constraint selections. By considering semantic manipulation of constraints, the techniques presented here significantly extend earlier work.

We studied the interaction of our algorithms (for generating and propagating minimum predicate constraints and QRP-constraints) with the Magic-sets transformation for a class of programs where the Magic-sets evaluation computes only ground facts if the original program computed only ground facts. We showed the optimality (among a class of transformation sequences where the Magic-sets rewriting is applied exactly once) of a transformation sequence that applies the Magic-sets transformation *after* generating and propagating minimum QRP-constraints.

There are several directions in which this work can be extended. Our technique for generating and propagating minimum QRP-constraints does not terminate, in general. A practical direction of research is to provide a terminating algorithm that generates and propagates QRP-constraints. (These constraints have to be non-minimal, because of our undecidability results.) Kemp and Stuckey [44] recently proposed an algorithm that modifies our technique for generating QRP-constraints, by using abstract interpretation to guarantee termination.

An important direction is to identify classes of programs for which there is a terminating procedure to compute minimum predicate constraints and minimum QRP-constraints. A promising candidate is the class of programs called "strongly unique" programs by Brodsky and Sagiv [16].

By first propagating predicate constraints, it may be possible for a Magic-sets evaluation to terminate, whereas the evaluation may not have terminated otherwise. Another interesting problem is to study this interaction between our algorithm for computing minimum predicate constraints and the Magic-sets transformation when the evaluation computes constraint facts.

Chapter 5

Coral++

5.1 Background

In recent years considerable research has been done in extending relational database languages, such as SQL, which have proven inadequate for a variety of emerging applications. Two main directions of research in database programming languages have been object-oriented database languages and deductive database languages, and the issue of combining the two paradigms has received attention recently ([3, 15, 18, 24, 28, 32, 37, 38, 45, 51, 57, 60, 77, 100], among others).

Object-oriented database languages, such as *E* [72], O++ [1] and O_2 [25], among others, enhance the relational data model by providing support for abstract data types, encapsulation, object identifiers, methods, inheritance and polymorphism. Such sophisticated features are very useful for data modeling in many scientific, engineering, and multimedia applications. Deductive database languages, such as LDL [59], Coral [65] and Glue-Nail! [62], among others, enhance the declarative query language by providing a facility for generalized recursive view definition, which is of considerable practical importance. However, data models for deductive databases are typically structural, and do not have the richness of object-oriented data models.

In this chapter, we demonstrate that the advantages of object-oriented database languages and deductive database languages can indeed be combined in a clean and practical manner. Our proposal, Coral++, has the following objectives:

- To combine an object-oriented data model with a deductive query language.

This permits the programmer to take advantage of the features of both object-oriented database languages and deductive database languages in developing applications. The Coral++ query language is significantly more expressive than the object-oriented extensions of SQL ([24, 28], for instance). A non-operational semantics is however maintained, and this makes Coral++ more amenable to automatic query optimization than imperative languages for object-oriented databases ([1, 25], for instance) that have similar expressive power.

- To cleanly integrate a declarative language with an imperative language.

  Such an integration is extremely useful since several operations such as updating the database in response to changes in the real world, input/output, etc., are imperative notions, whereas one can easily express many complex queries declaratively. A clean integration allows the programmer to do tasks in either the declarative style or the imperative style, whichever is appropriate to the task, and mix and match the two programming styles with minimal impedance mismatch.

- To keep object storage and retrieval orthogonal to the rest of the design, so that techniques developed for implementing object stores can be used freely in conjunction with other optimizations in the declarative query language.

5.1.1 Overview of the Coral++ Design

The central observation is as follows: Object-oriented features such as abstract data types, encapsulation, inheritance and object-identity are essentially extensions of the data model. We can achieve a clean integration of these features into a deductive query language by allowing the deductive language to draw values from a richer set of domains, and by allowing the use of the facilities of the deductive language to maintain, manipulate and query collections of objects of a given type.

In relational query languages such as SQL, values in fields of tables have been restricted to be atomic constants (e.g., integers or strings). In logic programs, values can be Herbrand terms, which are essentially structured values. In Coral++, values can additionally be of any class definable in C++ [84]. (We chose C++ since it provides a well-understood and widely used object-oriented type system.)

Coral++ provides support for maintaining extents, or collections of objects of a given type, either in a simple manner that reflects the inclusions associated with traditional IS-A hierarchies, or in a more sophisticated way through the use of declarative rules. The idea is to automatically invoke code that handles extent maintenance whenever objects are created or destroyed, and provide constructs to use these extents in Coral++. We also provide support for creating and manipulating various types of collections: sets, multisets, lists and arrays, whereas traditionally only sets and multisets are provided.

Coral++ separates the querying of objects from the creation, updating and deletion of objects, and provides separate sub-languages for these two purposes. This methodology stems from the view that querying is possible in a declarative language, whereas creation, updates and deletion should be performed only using an imperative language. We discuss querying in this chapter and refer the interested reader to Srivastava et al. [83] for a description of the imperative features of Coral++.

In summary, the proposal is simple, combines features of C++ and Coral—two existing languages—with minimal changes to either, and yields a powerful combination of the object-oriented and deductive paradigms. The essential aspects of the integration are not specific to our choice of C++ and Coral, and the ideas can readily be used to combine other object-oriented and deductive systems in a similar manner.

5.1.2 Motivating Examples

Example 5.1 (A University Database) A university database maintains information about various departments as well as information about students and employees. The Coral++ class declarations below specify both structural and behavioral properties of objects of these types.[1] (These class declarations are not complete, and are given only to illustrate some features of the language.)

```
class employee ;     /* forward declaration */
class date ;
class department {
public:
     employee   *head ;
     int   budget( ) ;
} ;
class person {
private:
     date   *date_of_birth ;
```

[1]Coral++ class declarations have the same syntax as C++ class declarations.

```
public:
     char    *name ;
     int    age ( ) ;
} ;
class employee : public person {
public:
     int    salary ;
     department    *dept ;
     employee    *supervisor ;
} ;
class student : public person {
public:
     department    *dept ;
} ;
```

The above class declarations have the following meaning. Each object of type person has two "stored" attributes called name and date_of_birth, and a "computed" attribute called age, which may be computed using the date of birth of the person and the current date. Each object of type student is also of type person and hence inherits the stored and computed attributes of type person. In addition, each student has a certain major department. Similarly, each object of type employee is also of type person and hence inherits the attributes of type person. In addition, each employee has a salary, is affiliated to a department, and reports to a supervisor. Each object of type department has an attribute called head, and an attribute called budget which may be computed by adding the funding the department receives from several sources.

Assume that for each of these classes, the class extent (i.e., the collection of all objects that are instances of the class) is maintained as a relation, and the relation name is the same as the class name. One can now ask several queries of interest against this database. For instance, the following query might be of interest for accounting purposes: "Find all departments where the sum of the salaries of the employees has exceeded the department budget." The corresponding Coral++ program/query is:

$$\begin{aligned}
budget\_exceeded(D) \quad & :-\ sum\_sals(D, S), \\
& \quad S > ((department*)D) \rightarrow budget(). \\
sum\_sals(D, sum(< S >)) & :-\ employee(E), D = ((employee*)E) \rightarrow dept, \\
& \quad S = ((employee*)E) \rightarrow salary.
\end{aligned}$$

Coral++ programs use a rule-based syntax, similar to Coral and logic programming languages. A difference is that Coral++ allows the use of C++ expressions (for instance, $((employee*)E) \rightarrow dept)$ in rules to access attributes and invoke methods. Each Coral++ rule can be read as an "if-then" statement in logic. For instance, the meaning of the first rule is "if the sum of salaries in a given department D is S and S is greater than the budget of department D, then the budget of department D has been exceeded"; the second rule gives a way of computing the sum of the salaries of the employees in a given department.

The Coral++ program corresponding to the query "Find all students in departments where the head of the department is named John" is:

$$\begin{array}{ll} jds(D, < S >) & :- j\_dept(D), student(S), D = ((student*)S) \rightarrow dept. \\ j\_dept(D) & :- department(D), \\ & \quad ((department*)D) \rightarrow head \rightarrow name = "John". \end{array}$$

The second rule in the above program is used to find out all departments where the head of the department is named *John*, and the first rule collects all the students in such departments.

These two queries can also be expressed in object-oriented extensions of SQL, and are provided here to primarily illustrate the difference in program specification styles. □

Example 5.2 (An Engineering Application) An engineering database for a manufacturing company stores information about the various parts manufactured, along with information about composition of parts. Some Coral++ class declarations for this database are given below.

```
class part {
private:
     int   functionality_test ;
     int   connection_test ;
public:
     char   *part_type ;
     int   tested ( );
} ;
class connection {
public:
     part   *from ;
```

```
        part    *to ;
        char    *ctype ;
    };
```

The above class declarations have the following meaning. Each object of type part has a certain part type (for instance, "wrench," "numerically controlled machine," etc.), and one can check if a part has been tested to be working. Parts may be connected together in several different ways, and each object of type connection indicates a binary relationship between two parts. Connections can also be of many different types (for instance, "subpart," "electrical connection," etc.), and this is stored in the attribute ctype.

The following Coral++ program can be used to express the bill-of-materials problem: "Find all subparts of a given part". This problem is of considerable practical importance in inventory control, and other applications.

$$\begin{array}{ll}
subpart(P1, P2) & :-\ connection(C), P1 = ((connection*)C) \rightarrow from, \\
 & \quad P2 = ((connection*)C) \rightarrow to, \\
 & \quad ((connection*)C) \rightarrow ctype = "subpart". \\
subpart(P1, P3) & :-\ connection(C), P1 = ((connection*)C) \rightarrow from, \\
 & \quad P2 = ((connection*)C) \rightarrow to, \\
 & \quad ((connection*)C) \rightarrow ctype = "subpart", \\
 & \quad subpart(P2, P3). \\
all\_subparts(P1, < P2 >) & :-\ subpart(P1, P2).
\end{array}$$

Consider the following query from [74]: "Find if a given part is working, where a part is known to be working either if it has been (successfully) tested or if it is constructed from smaller parts, and all the smaller parts are known to be working." This can be expressed in Coral++ as follows:

$$\begin{array}{ll}
working(P) & :-\ part(P), ((part*)P) \rightarrow tested() = 1. \\
working(P) & :-\ connection(C), P = ((connection*)C) \rightarrow from, \\
 & \quad ((connection*)C) \rightarrow ctype = "subpart", \\
 & \quad not\ has\_suspect\_part(P). \\
has\_suspect\_part(P) & :-\ connection(C), P = ((connection*)C) \rightarrow from, \\
 & \quad P1 = ((connection*)C) \rightarrow to, \\
 & \quad ((connection*)C) \rightarrow ctype = "subpart", \\
 & \quad not\ working(P1).
\end{array}$$

Neither of these two queries can be expressed in SQL or its object-oriented extensions because of the recursive definitions of *subpart* and *working*. □

5.2 Outline of Chapter

The rest of this chapter is structured as follows.

We describe the Coral++ object-oriented data model in Section 5.3. The Coral++ data model combines the Coral data model, which is presented in Section 5.3.1, with the C++ type system, which is briefly described in Section 5.3.2. In addition, the Coral++ data model includes new types of relations, and these are discussed in Section 5.3.3.

The Coral++ declarative query language is presented in Section 5.4. The Coral++ query language combines the Coral query language, which is briefly described in Section 5.4.1, with the use of C++ class extents and C++ expressions in rules. We discuss the incorporation of these features into the query language in Sections 5.4.3 and 5.4.4.

We discuss in detail how Coral++ is implemented using the existing Coral run-time system in Section 5.5. In Section 5.5, we also describe how queries are evaluated in the Coral++ system, and the design decisions made to enable ease of implementation.

Finally, in Section 5.6, we compare our proposal with some of the related proposals in the literature.

5.3 Coral++: Data Model

In database languages such as SQL, values in fields of relational tables are unstructured, i.e., restricted to be of a basic type supported by the system (e.g. integers or strings). In deductive database languages such as LDL [59] and Coral [65], values can be Herbrand terms, which are essentially structured values. However, data modeling in many scientific and engineering applications require support for more sophisticated features such as abstract data types, encapsulation, methods and inheritance. To support the data modeling needs of such applications, the Coral++ data model enhances the untyped Coral data model [65] with the C++ class facility. Values in Coral++ can additionally be of any class definable in C++, which can be manipulated using only the corresponding methods, supporting encapsulation. This allows a programmer to effectively use a combination of C++ and Coral.

One of the goals of Coral++ was to integrate Coral with an *existing* object data model, instead of inventing yet another object data model. By using the C++ type system as an object model, our approach is able to benefit from the support of data abstraction, inheritance, parameterized types, and polymorphism already available in C++. The choice of C++ was based on practical implementation considerations (Coral is implemented in C++), but we believe that our approach can also be applied to extending Coral with an alternative object-oriented data model.

5.3.1 Overview of the Coral Data Model

We informally describe features of the Coral data model using examples. The following facts could be interpreted as follows: the first fact indicates that John is an employee in the "Toys for Tots" department who has been with the company for 3 years and makes \$35000 annually. The second fact indicates that Joan has worked for the same department for 2 years and makes \$30000 annually.

$$works\_for(john, "Toys\ for\ Tots", 3, 35000).$$
$$works\_for(joan, "Toys\ for\ Tots", 2, 30000).$$

In order to express structured data, complex terms are required. In Coral, function symbols are used as record constructors, and such terms can be arbitrarily nested. The following fact can be interpreted as: John lives in Madison, and has a street address with a zip of 53606.

$$address(john, residence("Madison", street\_add("Oak\ Lane", 3202), 53606)).$$

Sets and multisets are allowed as values in Coral; $\{peter, mary\}$ is an example of a set representing the children of John, $\{60000, 35000, 35000, 30000\}$ is an example of a multiset representing the salaries of employees in the "Toys for Tots" department.

Coral permits variables within facts. A fact with a variable in it represents a possibly infinite set of ground facts. Such facts are often useful in knowledge representation, natural language processing and could be particularly useful in a database that stores (and possibly manipulates) rules. There is another, possibly more important, use of variables — namely to specify constraint facts [39, 64].

However, the Coral data model does not allow values of (arbitrary) user-defined types in facts. These are extremely useful in several applications, especially when the user-defined types have behavioral components.

5.3.2 Overview of the C++ Type System

C++ allows the specification of user-defined types using the class definition facility. An implementation of a C++ class is a combination of the attributes that specify the "structure" of the class along with the implementation of the methods that specify the "behavior" of the class. Attributes and methods of a class may be specified as either "public," "private," or "protected," providing different levels of encapsulation. Classes can be organized in an inheritance hierarchy in C++, and a class can have more than one subtype as well as more than one supertype (i.e., C++ supports multiple inheritance). The class declarations of Examples 5.1 and 5.2 illustrate some of these features.

By integrating the C++ object model with Coral, the Coral++ user benefits from having sophisticated data modeling and manipulation capabilities.

5.3.3 Relations in Coral++

Coral supports database relations that are multisets (i.e., unordered collections) of tuples. Typically, current database systems support only multisets of tuples, and the utility of these collections can be seen from the variety of applications written in SQL. However, for many applications ([71, 76]) involving sequence data and spatial data, for example, ordered collections of list-type and array-type are more natural. Hence, Coral++ also supports list-relations and array-relations, in addition to multiset-relations.

Each of these relation types supports the operation of iterating through the elements in the collection. The difference is the *primary* mode of access to elements in the collection. Multisets support unordered access, lists support ordered access in the total order of the list elements, and arrays support access in the array index order. In addition, each collection type can have value-based indexed modes of access, where the index can be on specified attributes or patterns.

Example 5.3 (Stock Market Data) A stock market database maintains daily information about the stocks traded for each company. A small fragment of such type information is shown below:

```
class DailyStockInfo {
public:
    double   low ;
    double   high ;
    double   average ( ) ;
```

```
        int    volume_traded ;
    } ;
```

Stock market information for individual companies can be naturally represented as array relations, which results in extremely efficient querying and manipulation of such information, as is demonstrated in the Mimsy system [76]. □

5.4 Coral++: Query Language

The Coral++ query language is modular, declarative and provides support for generalized recursive view definition. It is based on the Coral query language [65] which supports general Horn clauses with complex terms, multiset-grouping, aggregation and negation. Coral++ extends the Coral query language by allowing C++ expressions for accessing attributes and invoking (side-effect free) methods of classes in program rules. Declarative Coral++ programs can be largely understood in terms of standard Horn clause logic with C++ attribute accesses and method invocations treated as external functions.

Coral++ incorporates several important design decisions in the way the data model interacts with the declarative query language:

- The notion of a class is kept *orthogonal* to the related notion of class extents. This is achieved by providing the Coral++ programmer considerable flexibility in explicitly defining and maintaining collections of objects of a given class. We describe this in more detail in Section 5.4.3.

- The C++ expression truth semantics is kept distinct from the Coral++ predicate truth semantics. This is achieved by allowing C++ expressions to appear *only* in the argument positions of predicates (including evaluable predicates such as $=, \leq$, etc.). This is discussed in more detail in Section 5.4.4.

- Declarative rules in Coral++ do not create new objects that are instances of user-defined C++ classes, although they can create facts describing relationships between existing objects. The rationale for this decision is discussed in Section 5.4.5.

One of the major advantages of our proposal is that evaluation of Coral++ programs is based on the existing Coral run-time system, which facilitates implementation considerably. More generally, it suggests that optimization techniques developed for deductive and for object-oriented database languages can be combined cleanly. In this section, we first give an overview of the Coral query language and the evaluation of Coral queries, and then discuss the Coral++ design decisions.

5.4.1 Overview of the Coral Query Language

Coral supports and efficiently evaluates a class of programs with negation that properly contains the class of non-floundering left-to-right modularly stratified programs (see Chapter 3). Intuitively, this class is one in which the subgoals and answers generated during program evaluation involve no cycles through negation. There are two ways in which sets and multisets can be created using Coral rules, namely, multiset-enumeration ({ }) and multiset-grouping (<>). The following examples illustrate the use of these constructs:

$$children(john, \{mary, peter, paul\}).$$
$$p(X, <Y>) :- q(X, Y, Z).$$

The second rule uses facts for q to generate a multiset S of instantiations for the variables X, Y, and Z. For each value x for X in this set it creates a fact $p(x, \pi_Y \sigma_{X=x} S)$, where π_Y is a multiset projection (i.e., it does not do duplicate elimination). Thus, with facts $q(1,2,3), q(1,2,5)$ and $q(1,3,4)$, we get the fact $p(1, \{2,2,3\})$. The use of the multiset-grouping construct in Coral is similar to the grouping construct in LDL, except that in LDL the grouping construct creates a set (as opposed to a multiset).

Coral requires the use of the multiset-grouping operator to be left-to-right modularly-stratified (in the same way as negation). This ensures that all derivable q facts with a given value x for X can be computed before a fact $p(x, \_)$ is created.

Coral provides several standard operations on sets and multisets as system-defined predicates. These include $member, union, intersection, difference, subset, cardinality$, $multisetunion$ and $makeset$. It also allows several aggregate operations on sets and multisets: these include $count, min, max, sum, product, average$ and any. Some of the aggregate operations can be combined directly with the multiset-generation operations for increased efficiency (see Ramakrishnan et al. [65] for further details).

A Coral declarative program can be organized as a collection of interacting modules. Modules provide a way, as the name suggests, to modularize Coral code. In developing large applications, incremental program development and testing is critical, and modules in Coral provide the basis for this kind of programming. A module in Coral consists of a collection of rules defining a collection of predicates. A subset of these defined predicates are named as exported predicates, and other modules can pose queries over these predicates. The query forms permitted for each exported predicate are also indicated in the export declarations. Non-exported predicates are not visible outside this module and this provides a way of encapsulating the definition of Coral predicates.

5.4.2 Evaluating Coral Queries

The evaluation of a Coral module, given a query on an exported predicate of a module, is determined by the control annotations in the module, and the expert user can control the evaluation in several ways. We refer the interested reader to [65] for a discussion of these annotations and their effect on module evaluation. In the absence of any user-specified annotations, the Coral system chooses from among a set of default evaluation strategies.

For declarative modules, Coral evaluation, using these default strategies, is guaranteed to be sound, i.e., if the system returns a fact as an answer to a query, that fact indeed follows from the semantics of the declarative program. The evaluation is also "complete" in a limited sense — as long as the execution terminates, all answers to a query are actually generated. It is possible however, to write programs that do not terminate; in some such cases (e.g., programs without negation, set-grouping or aggregation) Coral is still complete in that it enumerates all answers in the limit.

5.4.3 Class Extents in Coral++

Coral++ keeps the notion of a class (as an encapsulation of data and methods) orthogonal to the related notion of class extents (i.e., the collection of all objects of the given class). Although maintaining class extents is necessary for iterating over all objects of a given class (as in Example 5.2), Coral++ does not automatically maintain class extents since doing so is very expensive, and one does not always need to iterate over *all* objects of a given class.

Coral++ provides the programmer considerable flexibility in explicitly defining and maintaining collections of objects of a given class. Collections of objects can be maintained either in a simple manner that reflects the inclusions associated with traditional IS-A hierarchies, or in a more sophisticated way through the use of declarative rules. Coral++ provides functions that handle extent maintenance, and these functions can be explicitly invoked from class constructors and destructors. Such class extents are maintained as Coral++ relations, and can be used as literals in the bodies of Coral++ rules.

The Coral++ extent maintenance functions take the name of the relation as a parameter, and hence the user can choose the name of extent relations for user-defined classes; in this chapter, we use the class name as the name of the class extent for simplicity. For instance, the following literal can be used to iterate over the extent of class *part*, in the body of a Coral++ rule:

$part(P)$.

The logic variable P is successively bound to (pointers to) objects in the extent. (See Example 5.2 for further uses of class extents.)

5.4.4 C++ Expressions in Rules

By integrating the C++ object model with the Coral data model, Coral++ allows facts to contain objects of C++ classes. Such objects can be manipulated only using the corresponding methods, supporting encapsulation. This is achieved by allowing C++ expressions in Coral++ rules to access attributes and invoke methods of objects. For example, one can iterate over all tested parts using:

$part(P), ((part*)P) \rightarrow tested() = 1$.

All variables in Coral++ rules are logic variables, and hence Coral++ requires variables participating in C++ expressions to be type cast to their intended C++ type. For instance, in the above example, the variable P has to be cast to the type $(part*)$ before the method *tested*() of class *part* can be invoked on the object to which P refers. The expression $P \rightarrow tested()$ would result in a compile-time error, since the method *tested*() is not defined for logic variables. By supporting type declarations for variables in rules, the syntax for attribute accesses and method invocations can be simplified; we do not discuss this further in the thesis.

C++ expressions can appear *only* in the argument positions of predicates (including evaluable predicates such as $=, \leq$, etc.) in Coral++ rules. This ensures that the Coral++ predicate truth semantics is kept distinct from the C++ expression truth semantics. (The C++ type system does not include a boolean type; any arithmetic expression is considered false if its value is zero, and true otherwise.) For instance, the following are not legal Coral++ literals, although the C++ conditional expressions can each be used in the C++ if-statement:

$((part*)P) \rightarrow tested()$.
$((department*)D) \rightarrow head \rightarrow name$.

5.4.5 Creating Objects in Coral++

Objects can be created using constructor methods (specified along with the class definition), and deleted using destructor methods (also specified along with the class definition). Coral++ requires that objects that are instances of C++ classes be explicitly created *only* using C++; the database can be populated with such objects only from C++. However, the Coral++ declarative language can create facts describing relationships between existing objects in the database.

Rules in Coral++ are deliberately restricted to avoid creating new objects, since this is an issue that is *not* yet well-understood despite work by Maier [52], Kifer et al. [45], and others. A number of issues, notably the resolution of conflicts when rules generate distinct objects with the same object identifier, remain unclear, especially in the presence of partially specified objects (e.g. some fields are variables, in the Coral++ context).

Similarly, updating and deleting objects that are instances of C++ classes should be performed only using the imperative language. This methodology stems from the view that the query language has to be *declarative*, whereas creating, updating and deleting objects are *operational* notions.

5.5 Implementing Coral++

One of the fundamental design decisions of our proposal is to use the run-time system of the Coral implementation [68] as much as possible in the implementation of Coral++. Several design decisions are a practical consequence of this:

- The notation for class definitions in Coral++ is the same as in C++. This allows the Coral++ class definitions to be handled by the C++ compiler directly.
- All variables in Coral++ rules have to be cast to the appropriate type before invoking a method or accessing an attribute. This permits Coral++ to avoid inferencing types at compile-time.

As a consequence of these design decisions, the compilation and evaluation of a Coral++ program possibly augmented with class definitions proceeds as follows. (Figure 5.11 depicts the Coral++ program compilation process pictorially.) First, the user compiles the class definitions and the method definitions (if any) along with the basic Coral++ system to create an enhanced Coral++ system that "knows" about these new

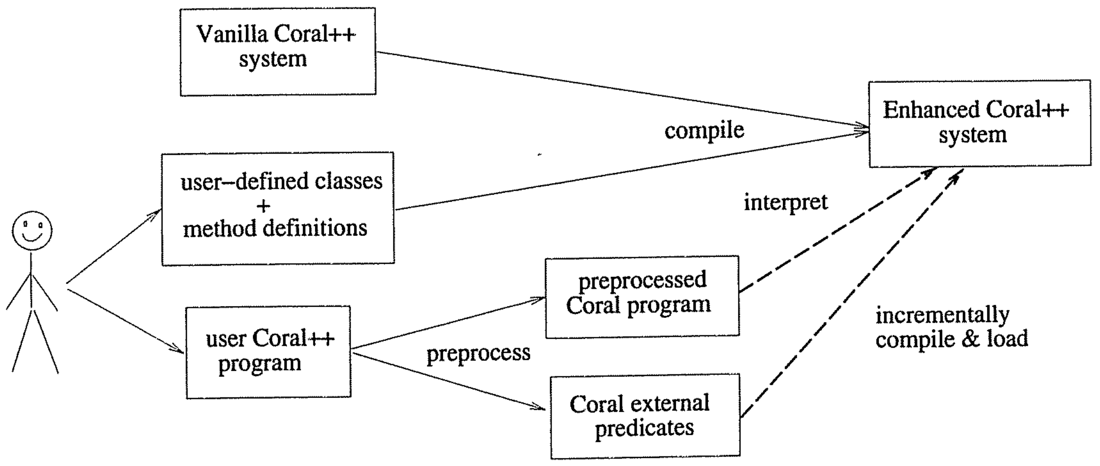
Vanilla Coral++ system
Enhanced Coral++ system
compile
user–defined classes + method definitions
interpret
preprocessed Coral program
user Coral++ program
preprocess
incrementally compile & load
Coral external predicates

Figure 5.11: Coral++ Program Compilation

classes; Section 5.5.1 describes this process in more detail. Second, Coral++ program modules go through a translation phase (at run-time) for handling attribute accesses and method invocations; Section 5.5.2 describes this process in more detail. (These two processes are indicated using solid arrows in Figure 5.11.) Finally, the translated programs are directly evaluated using the Coral++ interpreter. (This process is indicated using dashed arrows in Figure 5.11.)

5.5.1 Implementing Classes and Extents

We briefly describe the Coral run-time system with a view to describing the implementation of Coral++. The Coral system is implemented using C++ and all Coral data types are represented as C++ classes. The root of all data types is the virtual class CoralArg; specific types such as complex terms and multisets are all subclasses of the class CoralArg. The class CoralArg defines virtual methods that must be defined for each Coral data type; this includes methods such as the method "print", which is used to display the value to the Coral user.

Our approach for a practical implementation of Coral++ classes is summarized as follows:

- User-defined classes in Coral++ have the same syntax as C++ classes. All class definitions including method definitions are completely handled using the C++ compiler. Subtyping (including multiple inheritance) in Coral++ is automatically implemented using the inheritance mechanism of C++.

- All user-defined Coral++ classes should be subclasses of the root class CoralArg. Because all values used in Coral++ rules at run-time are of a type derived from CoralArg, Coral++ does not have to perform any dynamic type inferencing and type conversion to determine the methods that need to be invoked during rule evaluation.

- We provide C++ functions that can be used to maintain class extents. The user has to explicitly insert these functions into the definitions of the constructor and destructor methods of each class whose extent has to be maintained.

5.5.2 Program Evaluation in Coral++

Program evaluation in Coral++ requires modifying the existing program evaluation strategy in Coral to access named attributes and invoke methods of objects, instead of simply accessing relation field values using position notation. In a Coral++ program these requirements can be satisfied as follows:

- First, for each attribute access and method invocation in a Coral++ rule, the Coral++ preprocessor generates external (C++) predicates that perform the appropriate attribute access or method invocation at run-time. This code can be separately compiled and incrementally loaded.

  This approach relegates the task of binding the method name with the actual code to invoke the method to the C++ compiler. The alternative approach of invoking the C++ methods directly from the Coral++ interpreter would involve duplicating some of the tasks of the C++ compiler including maintaining symbol tables and virtual function tables, which would be quite impractical.

- Second, the program is translated to replace all occurrences of method invocations and attribute accesses by the appropriate external predicates.

  Appropriate indexes are also created at this time for providing associative access to relations containing objects.

- Finally, the translated program is evaluated using the Coral interpreter for evaluating rules, modules and programs.

The evaluation of Coral++ modules can use the query-directed rewriting optimizations as well as the various optimizations of the existing Coral run-time system.

The decision to relegate the task of determining which code is to be evaluated at method invocation to the C++ compiler results in the following practical design decision for methods invoked in Coral++ rules: All uses of rule variables should be cast to the appropriate type before accessing an attribute or invoking a method. This is done to avoid type inferencing in Coral++.

The current Coral++ implementation does not parse C++ expressions. Consequently, C++ expressions in declarative rules must be set apart syntactically for recognition by the Coral++ preprocessor. This is done by using the reserved symbol "#" to surround each C++ expression within a declarative Coral++ rule. Logic variables within these C++ expressions are also distinguished syntactically by preceding each variable with the "$" symbol.

Thus, the query "Find all departments where the sum of the salaries of the employees has exceeded the department budget," from Example 5.1, would be written in the current Coral++ implementation as the following program:

$$\begin{array}{ll} budget\_exceeded(D) & :-\ sum\_sals(D, S), \\ & \quad S > \#((department*)\$D) \rightarrow budget()\#. \\ sum\_sals(D, sum(<S>)) & :-\ employee(E), D = \#((employee*)\$E) \rightarrow dept\#, \\ & \quad S = \#((employee*)\$E) \rightarrow salary\#. \end{array}$$

5.6 Related Work

There are many proposals in the literature ([3, 15, 18, 24, 28, 32, 37, 38, 45, 51, 57, 60, 77, 100], among others) for integrating object-oriented databases and declarative query languages. Typically, these proposals support features such as complex objects, data abstraction, inheritance and polymorphism in their data model, and the ability to pose queries on collections of objects using a suitable query language. The Coral++ data model and query language have been influenced by many of these proposals. Some of the important aspects of our design, and how related proposals differ are as follows:

- Coral++ uses the type system of an existing object-oriented programming language (i.e., C++) as an object data model rather than inventing yet another object data model which is the approach taken by, for instance, [3, 28, 32, 37, 45, 57, 77]. This approach benefits from the support for data abstraction, inheritance, polymorphism and parametrized types already available in C++.

Other query languages that use the C++ type system include CQL++ [24], ObjectStore [60] and ZQL[C++] [15].

- The Coral++ declarative query language supports the combination of Coral rules with C++ expressions in a clean fashion. This approach can effectively utilize the Coral implementation and the C++ compiler.

  SWORD [57] and ObjectStore [60], for instance, take the alternative approach of inventing new syntax to query an object data model.

- Coral++ is more expressive than most of the other proposals [24, 28, 32, 77]. In particular, it provides a facility for generalized recursive view definition in the query language. It also supports unordered relations (i.e., sets and multisets) and ordered relations (lists and arrays), which are useful in applications involving sequence data [76].

- The Coral++ query language can be largely understood in terms of standard Horn clause logic (with C++ method invocations treated as external functions), unlike Noodle [57] which is based on HiLog [20] and XSQL [45] which is based on F-logic [46]. Bottom-up evaluation of HiLog and F-logic programs is not as well understood as the evaluation of Horn clause programs and is likely to be more expensive.

- Our proposal includes a detailed implementation design that clearly demonstrates the practicality of extending Coral (an existing deductive system) with object-oriented features of C++ (a widely-used object-oriented type system). An initial implementation based on the run-time system of the Coral implementation [68] is underway.

We now examine some of the closely related proposals in more detail.

5.6.1 Proposals Based on C++

ZQL[C++] [15] and CQL++ [24] are the proposals most closely related to Coral++ since they are also based on the C++ object model.

The Coral++ query language is more expressive than CQL++ or ZQL[C++], which are based on SQL. However, each of these proposals is integrated with a computationally complete imperative language: CQL++ with O++ [1], and Coral++ and ZQL[C++] with C++.

CQL++ has a syntax similar to SQL syntax for class definition. These classes do not have any facility for data abstraction (i.e., all class members are *public*). Further, accessing an attribute or invoking a method in a CQL++ query uses the "dot notation" of SQL, i.e., the system has to dereference pointers automatically. In Coral++ and ZQL[C++], class definitions can use all the features of C++ including data abstraction, and C++ expressions can be used for accessing attributes and invoking methods in a query.

In Coral++ and CQL++, path expressions are treated as values that can be arguments to boolean-valued predicates. ZQL[C++], on the other hand, allows C++ expressions to serve directly as predicates. Since ZQL[C++] also allows SQL subqueries to appear as predicates, it does not distinguish between the predicate truth semantics and the C++ expression truth semantics, unlike Coral++ and CQL++.

5.6.2 Proposals Based on Deductive Languages

The COMPLEX data model [32] is a structural, typed data model that adds features such as object identity, object sharing and inheritance to the relational model. It does not support abstract data types, encapsulation, or methods; consequently, the data model is not as rich as the Coral++ data model. The query language of COMPLEX is C-Datalog which can be automatically translated to Datalog, and evaluated using an engine for evaluating Datalog programs. This translation is possible because of the lack of behavioral features and polymorphism in the data model. It is not clear how the translation approach generalizes once we introduce behavioral features in the model.

LDL++ [3] is a deductive database system whose type system extends that of LDL [59] with an abstract data type facility that supports inheritance and predicate-valued methods. However, it does not support object sharing or ADT extents, and its support of encapsulation and object identity is limited. Consequently, the data model is not as rich as the Coral++ data model. Further, LDL++ methods can be defined only using LDL++ rules; however, this can be done more naturally than in Coral++.

5.6.3 Proposals Based on Non-Horn Logics

XSQL [45] extends SQL by adding path expressions that may have variables that range over classes, attributes and methods. This facilitates querying schema information as well as instance-level information in object-oriented databases, using a single declarative query language. Noodle [57, 58] is a declarative query language for the SWORD declarative object-oriented database. Unlike Coral++ and XSQL, Noodle does not use path expressions to access attributes and invoke methods on objects. Instead, Noodle uses a syntax reminiscent of HiLog [20] for this purpose. Noodle also has a number of built-in classes to facilitate schema querying. Orlog [37] combines the modeling capabilities of object-oriented and semantic data models, and is similar to Noodle in that its logic-based language for querying and implementing methods uses a higher-order syntax with first order semantics.

In Coral++, methods and other aspects of data abstraction borrowed from C++ are viewed as being outside the scope of the deductive machinery, notably the unification mechanism. A more comprehensive treatment of features like path expressions (e.g., as in XSQL [45]) may well enable more efficient (i.e., set-oriented) processing of certain queries. We make no attempt to give these features a logical semantics; we simply borrow the C++ semantics, in order to enable ease of implementation.

The semantic foundations of XSQL, Noodle and Orlog (i.e., F-logic [46] and HiLog), have features that are difficult to support efficiently, at least in a bottom-up implementation. In particular, variables can get bound to predicate names only at run-time, and this causes problems with analysis of strongly connected components (SCCs) and can make Semi-naive evaluation inefficient. In contrast, one of the design motivations of Coral++ was to have a language that is rich in expressive power *and* can be efficiently evaluated within the framework of existing evaluation techniques.

There are several other interesting proposals for combining semantically rich data models with deductive databases that are less closely related to Coral++. ConceptBase [38] and Quixote [100] are two such systems. ConceptBase is based on the Telos knowledge representation language, and allows the specification of methods using deductive rules and integrity constraints. Quixote is a knowledge representation language that allows subsumption constraints, knowledge classification and inheritance and query processing for partial information databases.

5.7 Discussion

We described Coral++, an object-oriented extension of Coral. The Coral++ data model extends the structural data model of Coral by integrating it with the C++ type system. The Coral++ query language extends Coral by allowing C++ expressions for accessing attributes and invoking methods of objects. The Coral++ query language is much more expressive than object-oriented extensions proposed for SQL, while remaining declarative at the same time. Consequently, a variety of rewriting and evaluation-time optimizations can be performed to improve efficiency; in particular, the optimizations performed for Coral programs are applicable to Coral++ programs as well. The Coral++ imperative language can be used to create, update and remove objects from the database. It is cleanly integrated with C++, providing the user the ability to program in a combination of programming styles, with minimal impedance mismatch.

We proposed an implementation strategy for Coral++ that effectively uses the existing Coral run-time system [68] and the C++ compiler to implement object-oriented features of the data model and the query language. This, in our view, is one of the strong points of our proposal, and distinguishes it from many proposals in the literature describing query languages for object-oriented databases. The implementation strategy is orthogonal to issues such as object clustering, caching, indexing, storage management, etc. Although we described the implementation using the C++ class hierarchy, Coral++ does not depend on C++ implementation techniques for classes and class instances; it could also be implemented on top of a typed, persistent object store. We believe that Coral++ is a realistic and useful proposal for engineering, scientific and multi-media applications that can benefit from object-oriented data models and high-level data access and manipulation capabilities.

Chapter 6

Conclusions

This thesis contributes to our goal of making deductive database technology practical and usable in two main areas: optimization techniques for efficiently answering queries in deductive databases, and a combination of an object-oriented data model with a deductive query language for the natural modeling and expressive querying of complex information.

We presented a bottom-up evaluation technique, Ordered Search, that can be used to efficiently evaluate programs with left-to-right modularly stratified negation, multiset-grouping and aggregation. We provided theoretical results to demonstrate that Ordered Search is more efficient than other bottom-up techniques in the literature. We also provided performance results from the implementation of Ordered Search in the Coral deductive database system to show the practicality of this evaluation technique. While considerable research still needs to be done for efficiently evaluating arbitrary programs, we believe that Ordered Search provides a very satisfactory solution for evaluating the class of left-to-right modularly stratified programs.

We proposed a program transformation technique, Constraint-rewrite, that propagates constraints specified in a program and a query, such that the evaluation of the transformed program fully utilizes the constraints present in the original program. We described a uniform framework for our results, and for earlier related work in the literature on propagating constraint selections. By considering semantic manipulation of constraints, the techniques presented significantly extend earlier work. The Constraint-rewrite transformation can be combined with the Magic-sets transformation to propagate constant binding information in addition to propagating constraint information. We believe that our techniques provide some important insight into the optimization of constraint query languages. However, there is considerable research to be done on the optimization and evaluation of constraint query languages before this technology become practical; our work is an initial step in this direction.

Finally, we presented a deductive, object-oriented language, Coral++, that combines the data modeling features of C++ and the querying capability of Coral—two existing languages—with minimal changes to either, and yields a powerful combination of the object-oriented and deductive paradigms. We described an implementation strategy for Coral++ that effectively uses the existing Coral run-time system and the C++ compiler to implement object-oriented features of the data model and the query language. Coral++ is being implemented using this strategy. We believe that the Coral++ approach of combining an existing object data model with a deductive query language is a very practical avenue to providing users the ability to represent and query complex information.

Bibliography

[1] R. Agrawal and N. H. Gehani. Ode (Object Database and Environment): The language and the data model. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, Portland, Oregon, June 1989.

[2] 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.

[3] N. Arni, K. Ong, S. Tsur, and C. Zaniolo. The LDL++ system: Rationale, technology and applications. (Submitted), 1993.

[4] I. Balbin, D. B. Kemp, K. Meenakshi, and K. Ramamohanarao. Propagating constraints in recursive deductive databases. In *Proceedings of the North American Conference on Logic Programming*, pages 16–20, Oct. 1989.

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

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

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

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

[9] M. Baudinet, M. Niezette, and P. Wolper. On the representation of infinite temporal data and queries. In *Proceedings of the Tenth ACM Symposium on Principles of Database Systems*, pages 280–290, Denver, Colorado, May 1991.

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

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

[12] 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.

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

[14] N. Bidoit and P. Legay. WELL! An evaluation procedure for all logic programs. In *Proceedings of the International Conference on Database Theory*, pages 335–348, Paris, France, December 1990.

[15] J. A. Blakeley. ZQL[C++]: Integrating the C++ language and an object query capability. In *Proceedings of the Workshop on Combining Declarative and Object-Oriented Databases*, pages 138–144, Washington, D.C., May 1993.

[16] A. Brodsky and Y. Sagiv. Inference of inequality constraints in logic programs. In *Proceedings of the Tenth ACM Symposium on Principles of Database Systems*, pages 227–240, Denver, Colorado, May 1991.

[17] F. Bry. Query evaluation in recursive databases: Bottom-up and top-down reconciled. *IEEE Transactions on Knowledge and Data Engineering*, 5:289–312, 1990.

[18] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. Integrating object-oriented data modeling with a rule-based programming paradigm. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 225–236, Atlantic City, New Jersey, May 1990.

[19] M. Carey, D. DeWitt, J. Richardson, and E. Shekita. Object and file management in the EXODUS extensible database system. In *Proceedings of the International Conference on Very Large Databases*, Aug. 1986.

[20] W. Chen, M. Kifer, and D. 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.

[21] W. Chen and D. S. Warren. A goal-oriented approach to computing the well founded semantics. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, 1992.

[22] D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, S. Tsur, and C. Zaniolo. The LDL system prototype. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):76–90, 1990.

[23] J. Chomicki. Polynomial time query processing in temporal deductive databases. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 379–391, Nashville, Tennessee, Apr. 1990.

[24] S. Dar, N. H. Gehani, and H. V. Jagadish. CQL++: An SQL for a C++ based object-oriented DBMS. In *Proceedings of the International Conference on Extending Database Technology*, Vienna, Austria, Mar. 1992. (A full version is available as AT&T Bell Labs Technical Memorandum 11252-910219-26).

[25] O. Deux. The O_2 database programming language. *Communications of the ACM*, Sept. 1991.

[26] S. W. Dietrich. Extension tables: Memo relations in logic programming. In *Proceedings of the Symposium on Logic Programming*, pages 264–272, 1987.

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

[28] L. J. Gallagher. Object SQL: Language extensions for object data management. In *Proceedings of the ISMM First International Conference on Information and Knowledge Management*, pages 17–26, Baltimore, Maryland, Nov. 1992.

[29] S. Ganguly, S. Greco, and C. Zaniolo. Minimum and maximum predicates in logic programming. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1991.

[30] P. Gardner and J. Shepherdson. Unfold/Fold transformations of logic programs. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic*. The MIT Press, 1991.

[31] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. Fifth International Conference and Symposium on Logic Programming*, 1988.

[32] S. Greco, N. Leone, and P. Rullo. COMPLEX: An object-oriented logic programming system. *IEEE Transactions on Knowledge and Data Engineering*, 4(4):344–359, Aug. 1992.

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

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

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

[36] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM POPL*, pages 111–119, Munich, Jan. 1987.

[37] M. H. Jamil and L. V. S. Lakshmanan. ORLOG: A logic for semantic object-oriented models. In *Proceedings of the ISMM First International Conference on Information and Knowledge Management*, pages 584–592, Baltimore, Maryland, Nov. 1992.

[38] M. Jarke, S. Eherer, R. Gallersdoerfer, M. Jeusfeld, and M. Staudt. ConceptBase – a deductive object base manager. Submitted, 1993.

[39] P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz. Constraint query languages. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 299–313, Nashville, Tennessee, Apr. 1990.

[40] D. Kemp, D. Srivastava, and P. 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., Oct. 1991.

[41] D. Kemp, D. Srivastava, and P. Stuckey. Query restricted bottom-up evaluation of normal logic programs. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, 1992.

[42] D. Kemp and P. Stuckey. Semantics of logic programs with aggregates. In *Proceedings of the International Logic Programming Symposium*, pages 387–401, San Diego, CA, U.S.A., Oct. 1991.

[43] D. B. Kemp. *On the Foundations of Query Evaluation in Deductive Databases.* PhD thesis, University of Melbourne, Nov. 1992. Report No. CITRI/TR-92-70.

[44] D. B. Kemp and P. J. Stuckey. Analysis based constraint query optimization. In *Proceedings of the International Conference on Logic Programming*, Budapest, Hungary, June 1993.

[45] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 393–402, San Diego, California, 1992.

[46] M. Kifer and G. Lausen. F-logic, a higher-order language for reasoning about objects, inheritance and schemes. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1989.

[47] J.-L. Lassez and M. J. Maher. On Fourier's algorithm for linear arithmetic constraints. *Journal of Automated Reasoning*, 9:373–379, 1992.

[48] A. Lefebvre. Towards an efficient evaluation of recursive aggregates in deductive databases. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, June 1992.

[49] N. Leone and P. Rullo. Safe computation of the well-founded semantics of Datalog queries. *Information Systems*, 17(1):17–31, 1992.

[50] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

[51] Y. Lou and Z. M. Ozsoyoglu. LLO: An object-oriented deductive language with methods and method inheritance. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 198–207, Denver, Colorado, May 1991.

[52] D. Maier. A logic for objects. Technical Report Technical report CS/E-86-012, Oregon Graduate Center, Beaverton Oregon 97006-1999, November 1986.

[53] S. Morishita. An alternating fixpoint tailored to magic programs. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1993.

[54] K. Morris, J. D. Ullman, and A. Van Gelder. Design overview of the NAIL! system. In *Proceedings of the Third International Conference on Logic Programming*, 1986.

[55] I. S. Mumick. *Query Optimization in Deductive and Relational Databases.* PhD thesis, Stanford University, Dec. 1991. Report No. STAN-CS-91-1400.

[56] I. S. Mumick, S. J. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic conditions. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 314–330, Nashville, Tennessee, Apr. 1990.

[57] I. S. Mumick and K. A. Ross. An architecture for declarative object-oriented databases. In *Proceedings of the JICSLP-92 Workshop on Deductive Databases*, pages 21–30, Washington, D.C., Nov. 1992.

[58] I. S. Mumick and K. A. Ross. The influence of class hierarchy choice on query language design. In *Proceedings of the Workshop on Combining Declarative and Object-Oriented Databases*, pages 152–154, Washington, D.C., May 1993.

[59] S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases.* Principles of Computer Science. Computer Science Press, New York, 1989.

[60] J. Orenstein, S. Haradhvala, B. Margulies, and D. Sakahara. Query processing in the ObjectStore database system. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 403–412, San Diego, California, 1992.

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

[62] G. Phipps, M. A. Derr, and K. A. Ross. Glue-NAIL!: A deductive database system. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 308–317, 1991.

[63] T. Przymusinski. On the declarative semantics of stratified deductive databases. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216, 1988.

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

[65] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL: Control, Relations and Logic. In *Proceedings of the International Conference on Very Large Databases*, 1992.

[66] R. Ramakrishnan, D. 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.

[67] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Rule ordering in bottom-up fixpoint evaluation of logic programs. *IEEE Transactions on Knowledge and Data Engineering*, 1993. To appear. (A shorter version appeared in VLDB, 1990).

[68] R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. Implementation of the CORAL deductive database system. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1993.

[69] R. Ramakrishnan and S. Sudarshan. Top-Down vs. Bottom-Up Revisited. In *Proceedings of the International Logic Programming Symposium*, 1991.

[70] P. Z. Revesz. A closed form for Datalog queries with integer order. In *International Conference on Database Theory*, pages 187–201, France, Dec. 1990.

[71] J. Richardson. Supporting lists in a data model (a timely approach). In *Proceedings of the International Conference on Very Large Databases*, pages 127–138, Vancouver, Canada, 1992.

[72] J. E. Richardson, M. J. Carey, and D. T. Schuh. The design of the E programming language. *ACM Trans. Prog. Lang. Syst.*, 15(3):494–534, July 1993.

[73] 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.

[74] K. 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.

[75] K. A. Ross. *The Semantics of Deductive Databases*. PhD thesis, Stanford University, Aug. 1991. Report No. STAN-CS-91-1386.

[76] W. G. Roth. Mimsy: A system for analyzing time series data in the stock market domain. Technical Report To appear, University of Wisconsin at Madison, 1993.

[77] L. A. Rowe and M. R. Stonebraker. The POSTGRES data model. In *Proceedings of the Thirteenth International Conference on Very Large Databases*, pages 83–96, Brighton, England, Sept. 1987.

[78] 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.

[79] J. Sebelik and P. Stepanek. Horn clause programs for recursive functions. In K. Clark and S.-A. Tarnlund, editors, *Logic Programming*. Academic Press, 1982.

[80] H. Seki. On the power of Alexander templates. In *Proc. of the ACM Symposium on Principles of Database Systems*, pages 150–159, 1989.

[81] D. Srivastava. Subsumption and indexing in constraint query languages with linear arithmetic constraints. *Annals of Mathematics and Artificial Intelligence*, 8(3-4), 1993. To appear.

[82] D. Srivastava and R. Ramakrishnan. Pushing constraint selections. *Journal of Logic Programming*, 1993. To appear. (A shorter version appeared in the Proceedings of the ACM Symposium on the Principles of Database Systems, 1992).

[83] D. Srivastava, R. Ramakrishnan, S. Sudarshan, and P. Seshadri. Coral++: Adding object-orientation to a logic database language. In *Proceedings of the International Conference on Very Large Databases*, 1993.

[84] B. Stroustrup. *The C++ Programming Language (2nd Edition).* Addison-Wesley, Reading, Massachusetts, 1991.

[85] S. Sudarshan and R. Ramakrishnan. Aggregation and relevance in deductive databases. In *Proceedings of the Seventeenth International Conference on Very Large Databases*, Sept. 1991.

[86] S. Sudarshan and R. Ramakrishnan. Optimizations of bottom-up evaluation with non-ground terms. In *Proceedings of the International Logic Programming Symposium*, 1993.

[87] S. Sudarshan, D. Srivastava, R. Ramakrishnan, and C. Beeri. Extending the well-founded and valid semantics for aggregation. In *Proceedings of the International Logic Programming Symposium*, 1993.

[88] H. Tamaki and T. Sato. Unfold/fold transformations of logic programs. In *Proceedings of the Second International Conference on Logic Programming*, pages 127–138, Uppsala, Sweden, July 1984.

[89] R. E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, 1983.

[90] A. Tarski. *A Decision Method for Elementary Algebra and Geometry.* University of California Press, Berkeley, California, 1951.

[91] J. 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.

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

[93] J. Vaghani, K. Ramamohanarao, D. B. Kemp, Z. Somogyi, and P. J. Stuckey. Design overview of the Aditi deductive database system. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 240–247, Apr. 1991.

[94] A. Van Gelder. Negation as failure using tight derivations for general logic programs. In *Proceedings of the Symposium on Logic Programming*, pages 127–139, 1986.

[95] A. Van Gelder. Deriving constraints among argument sizes in logic programs. *Annals of Mathematics and Artificial Intelligence*, 3:361–392, 1991.

[96] A. Van Gelder. The well-founded semantics of aggregation. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 127–138, 1992.

[97] A. Van Gelder, K. Ross, and J. S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[98] L. 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.

[99] L. Vieille. From QSQ towards QoSaQ: Global optimizations of recursive queries. In *Proc. 2nd International Conference on Expert Database Systems*, Apr. 1988.

[100] K. Yokota, H. Tsuda, and Y. Morita. Specific features of a deductive object-oriented database language QUIXOTE. In *Proceedings of the Workshop on Combining Declarative and Object-Oriented Databases*, pages 89–99, Washington, D.C., May 1993.

Appendix A

Algorithms

A.1 Ordered Search

Ordered-Search below is applicable to programs with function symbols that compute non-ground facts.

```
Ordered-Search($P^{mg}$, $Q^{mg}$)
{
    Let $Context$ be a list of sets of (magic and supplementary) facts,
            initially containing $Q^{mg}$.
    repeat
        repeat
            Initialize-SN-Relations($P^{mg}$, $Context$).
            Apply-Rules($P^{mg}$).     /* Compute new facts */
            for each newly computed pair of magic/supplementary facts $\langle Q_1, Q_2 \rangle$
                Insert-in-Context($\langle Q_1, Q_2 \rangle$, $Context$)
            end for
        until (no new facts computed)
        Remove-Marked-Subgoals($P^{mg}$, $Context$)
    until ($Context$ is empty)
    return answers to $Q^{mg}$.
}
```

Initialize-SN-Relations below is used to maintain the differential relations that are needed to ensure that derivations are not repeated.

```
Initialize-SN-Relations($P^{mg}$, $Context$)
{
    Let $p_1, \ldots, p_n$ be the original program predicates defined in $P^{mg}$.
    Let $m\_p_1, \ldots, m\_p_n$ be the corresponding magic predicates.
    Let $sup_1, \ldots, sup_k$ be the supplementary predicates in $P^{mg}$.
    for each predicate $p_i, 1 \leq i \leq n$,    SN-Update($p_i$).
    for each predicate $done\_m\_p_i, 1 \leq i \leq n$,    SN-Update($done\_m\_p_i$).
    for each predicate $m\_p_i, 1 \leq i \leq n$
        $m\_p_i^{old} := m\_p_i^{old} \sqcup \delta m\_p_i^{old}$.    $\delta m\_p_i^{old} := \phi$.    $\delta m\_p_i^{new} := \phi$.
    end for
    for each predicate $sup_i, 1 \leq i \leq k$
        $sup_i^{old} : -sup_i^{old} \sqcup \delta sup_i^{old}$.    $\delta sup_i^{old} := \phi$.    $\delta sup_i^{new} := \phi$.
    end for
    Let $Q_1$ be an unmarked fact in the head of $Context$; mark $Q_1$.
    if (head of $Context$ is unmarked)
        Mark head of $Context$.
        Link head of $Context$ with previous marked node.
        Set $number$ of head of $Context$ to $number$ of previous marked node +1.
    end if
    Let $l$ be the predicate of $Q_1$.
    $\delta l^{old} := Q_1$.
    for each predicate $m\_p_i, 1 \leq i \leq n$,    $m\_p_i := m\_p_i^{old} \sqcup \delta m\_p_i^{old}$.
    for each predicate $sup_i, 1 \leq i \leq k$,    $sup_i := sup_i^{old} \sqcup \delta sup_i^{old}$.
}
```

Apply-Rules below is used to make new derivations. It also records which magic or supplementary fact was used to derive a newly generated magic/supplementary fact. This is used to maintain dependency information within the *Context*.

```
Apply-Rules($P^{mg}$)
{
    Let $\mathcal{R}$ be the rules of $P^{mg}$, and let $\mathcal{R}_{SN}$ be the Semi-naive rules
                obtained from $\mathcal{R}$.
    Apply each rule in $\mathcal{R}_{SN}$ once; when computing magic facts,
        compute pairs $\langle Q_1, Q_2 \rangle$, where each of $Q_1$ and $Q_2$ is either a magic fact
        or a supplementary fact, and $Q_2$ was used to compute $Q_1$.
}
```

Insert-in-Context below is used to insert newly generated magic and supplementary facts into the *Context*, which maintains dependency information between magic facts.

```
Insert-in-Context($\langle Q_1, Q_2 \rangle, Context$)
{
    /* $Q_2$ was used to compute $Q_1$. */
    Let $p_i$ be the predicate of $Q_1$.
    (a) if ($Q_1$ is subsumed by a fact in the $done\_m\_p_i$ relation)
        return     /* completely solved */
    (b) else
        (i)  Add-After($Q_1, Q_2, Context$).
        (ii) if ($Q_1$ is subsumed by some $Q_3$ that occurs after $Q_1$ in $Context$)
                 Delete $Q_1$ from $Context$.
        (iii) else
                 Delete occurrences $Q_3$ earlier than $Q_1$ in $Context$ that are
                     not marked and are subsumed by $Q_1$.
                 Let $Q_4$ be the earliest fact in $Context$ subsumed by $Q_1$.
                     /* This fact is either marked, or is the fact $Q_1$ itself. */
                 Collapse-Nodes($Q_4, Q_1, Q_2, Context$).
        (iv) end if
    (c) end if
}
```

Add-After below is used to insert Q_1 in *Context*, as the last unmarked fact generated from Q_2.

```
Add-After($Q_1, Q_2, Context$)
{
    (a)  if ($Q_2$ is in the last marked node in $Context$ )
         (i)  Add a new node containing $Q_1$ to head of $Context$.
    (b)  else
         (i)  Let $N$ be the earliest marked node after $Q_2$ in $Context$.
         (ii) Add a new node containing $Q_1$ immediately before $N$ in $Context$.
    (c)  end if
```

```
        (d) Set the $parent$ of $Q_1$ to be $Q_2$.
    }
```

Collapse-Nodes below ensures that all magic facts which cyclically depend on each other are kept together in a node in *Context*; these magic facts will be declared fully evaluated, and moved into the corresponding *done_m_p* relations together.

```
    Collapse-Nodes($Q_4, Q_1, Q_2, Context$)
    {
        if ($Q_4$ is marked)      /* $Q_4$ is in a different node from $Q_1$. */
            (a) (i)  Remove all marked nodes in $Context$ marked between
                         (not including) the node containing $Q_4$ and
                         (including) the node containing $Q_2$.
                (ii) Set the $Subgoals$ set of the node of $Q_4$ to the union
                         of its $Subgoals$ set and the $Subgoals$ sets
                         of all the removed nodes.
                (iii) Set the $unmarkedSubgoals$ set of the node of $Q_4$ to
                         the union of its $unmarkedSubgoals$ set and the
                         $unmarkedSubgoals$ sets of all the removed nodes.
            (b) Delete the (newly inserted) node containing $Q_1$ from $Context$.
            (c) if ($Q_1$ is not subsumed by $Q_4$)
                    Add $Q_1$ to the $Subgoals$ set as well as the
                        $unmarkedSubgoals$ set of the resultant node.
            (d) end if
        end if
    }
```

Remove-Marked-Subgoals below is used to remove facts from *Context* and insert magic facts into the *done_m_p* relations.

```
    Remove-Marked-Subgoals($P^{mg}, Context$)
    {
        if (all facts in head of $Context$ are marked)
            Add all magic facts in head of $Context$ to the corresponding
                        $\delta done\_m\_p^{new}$ relations.
            Remove head of $Context$.
        end if
    }
```

A.2 Pushing Constraint Selections

The following procedure generates and propagates QRP-constraints, though not the minimum possible, for each derived predicate of a program P, if it terminates. Procedure Constraint-rewrite, described later, generates and propagates minimum QRP-constraints.

```
Gen-Prop-QRP-constraints ($P$)
{
    let $p_1, \ldots, p_m$ be the predicates defined in the program $P$,
                and let $q$ be the query predicate.
    Gen-QRP-constraints ($P$).
    let $C_{p_1}, \ldots, C_{p_m}$ be the QRP-constraints obtained.
    let $p'_1, \ldots, p'_m$ be new predicates, not occurring in the program.
    for $j = 1$ to $m$ do
        let $C_{p_j}$ have $k_j$ disjuncts.
        perform a *definition* step creating $k_j$ rules with head
            $p'_j(\overline{X})$, and the sole body literal $p_j(\overline{X})$.
        each rule has $PTOL(p_j(\overline{X}), C)$ as the conjunction of constraints
            in the body, where $C$ is one of the $k_j$ disjuncts.
    end for
    for $j = 1$ to $m$ do
        *unfold* the definition of $p_j$ in $P$ into each of the rules defining $p'_j$.
    end for
    for $j = 1$ to $m$ do
        *fold* the original definition of $p'_j$ into rules in $P$ containing
                    body occurrences of $p_j$.
    end for
    the resultant program $P'$ has all the QRP-constraints propagated.
}

Gen-QRP-constraints ($P$)
{
    let $p_1, \ldots, p_m$ be the predicates defined in the program $P$.
    let $q$ (one of the $p_i$'s) be the query predicate.
    for $i = 1$ to $m$ do
        $C1_{p_i}$ = **false**.
    end for
    $C1_q = \mathbf{true}$.
    repeat
        assuming $C1_{p_i}$ as a QRP-constraint for each $p_i$, obtain literal
            constraints $C_{p_k(\overline{X_l})}$ for each literal in each rule in $P$.
        for $i = 1$ to $m$ do
            $C2_{p_i} = \bigvee$ each $LTOP(p_i(\overline{X_l}), C_{p_i(\overline{X_l})})$ for $p_i(\overline{X_l})$ in a rule in $P$.
        end for
        for $i = 1$ to $m$ do
            if $(C2_{p_i} \supset C1_{p_i})$ then
                'mark' $p_i$.
            else
                'unmark' $p_i$.
                $C1_{p_i} = C1_{p_i} \vee C2_{p_i}$.
            end if
        end for
    until (all predicates are 'marked')
    $C1_{p_i}$ is the QRP-constraint obtained for each $p_i$.
}
```

The following procedure generates minimum predicate constraints for each derived predicate of a program P.

```
Gen-Prop-predicate-constraints ($P$)
{
    let $p_1, \ldots, p_m$ be the predicates defined in the program $P$.
    let $b_1, \ldots, b_n$ be the database predicates.
    let $C1_{b_1}, \ldots, C1_{b_n}$ be the minimum predicate constraints for database
        predicates.    /* These predicate constraints are part of the input. */
    Gen-predicate-constraints $(P, C1_{b_1}, \ldots, C1_{b_n})$.
    let $C1_{p_1}, \ldots, C1_{p_m}$ be the predicate constraints obtained.
    for each rule in $P$ of the form:   $r_i : p(\overline{X})\ :-\ C_{r_i}, p_{i1}(\overline{X_{i1}}), \ldots, p_{ik}(\overline{X_{ik}})$ do
        let each $C1_{p_{ij}}, 1 \le j \le k$ have $c_{ij}$ disjuncts.
        create $c_{i1} * \cdots * c_{ik}$ new rules of the form:
            $r_{i,h} : p(\overline{X})\ :-\ C_{r_i}, C3'_{p_{i1}}, p_{i1}(\overline{X_{i1}}), \ldots, C3'_{p_{ik}}, p_{ik}(\overline{X_{ik}})$
            where each $C3'_{p_{ij}}$ is a disjunct of $PTOL(p_{ij}(\overline{X_{ij}}), C1_{p_{ij}})$
    end for
    the resultant program is composed of the new set of rules.
}

Gen-predicate-constraints $(P, C1_{b_1}, \ldots, C1_{b_n})$
{
    let $p_1, \ldots, p_m$ be the predicates defined in the program $P$.
    let $b_1, \ldots, b_n$ be the database predicates, and
        $C1_{b_i}, 1 \leq i \leq n$ the corresponding minimum predicate constraints.
    let $\mathcal{R}$ be the rules in $P$ defining $p_i, 1 \leq i \leq m$.
    for $i = 1$ to $m$ do
        $C1_{p_i} =$ **false**.
    end for
    repeat
        Single-step $(\mathcal{R}, C1_{p_1}, \ldots, C1_{p_m}, C1_{b_1}, \ldots, C1_{b_n})$.
        for $i = 1$ to $m$ do
            if $(C2_{p_i} \supset C1_{p_i})$ then 'mark' $p_i$.
            else
                'unmark' $p_i$.
                $C1_{p_i} = C1_{p_i} \vee C2_{p_i}$.
            end if
        end for
    until (all predicates are 'marked')
    $C1_{p_i}$ is the minimum predicate constraint obtained for each $p_i$.
}

Single-step $(\mathcal{R}, C1_{p_1}, \ldots, C1_{p_m}, C1_{b_1}, \ldots, C1_{b_n})$
{
    for $i = 1$ to $m$ do
        $C2_{p_i} =$ **false**.
    end for
    let the rules in $\mathcal{R}$ be $r_1, \ldots, r_k$.
    for $i = 1$ to $k$ do
        let rule $r_i$ be of the form: $r_i : p_j(\overline{X_j}) \ :- \ C_i(\overline{X}), p_{i1}(\overline{X_{i1}}), \ldots, p_{is}(\overline{X_{is}})$.
        $C2_{p_j} = C2_{p_j} \vee LTOP(p_j(\overline{X_j}), \Pi_{\overline{X_j}}(C_i(\overline{X}) \& \wedge_{h=1}^{s} PTOL(p_{ih}(\overline{X_{ih}}), C1'_{p_{ih}})))$
```

```
                for each choice $C1'_{p_{ih}}$ of a disjunct from $C1_{p_{ih}}$.
                /* The disjunction of the $LTOP$s is the inferred
                        head constraint for rule $r_i$. */
        end for
        return $C2_{p_1}, \ldots C2_{p_m}$.
    }
```

The following procedure generates and propagates minimum QRP-constraints for each derived predicate of a program P, if it terminates. It combines Procedures Gen-Prop-predicate-constraints and Gen-Prop-QRP-constraints.

```
    Constraint-rewrite ($P$)
    {
        let $q$ be the query predicate.
        define a new predicate $q_1$ with the same arity as $q$,
                    and let the only rule defining $q_1$ be
            $q_1(\overline{X_1})\ :-\ q(\overline{X_1})$.
            where $\overline{X_1}$ is a tuple of distinct variables.
        add this rule to $P$, and call the resultant program $P1$.
        the predicate $q_1$ is the new query predicate.
        Gen-Prop-predicate-constraints ($P1$).
        call the resultant program $P2$.
        Gen-Prop-QRP-constraints ($P2$)
        delete rules defining $q_1$ from the resultant program.
        the resultant program $P3$ is the rewritten program obtained by
            generating and propagating minimum QRP-constraints.
    }
```

Appendix B

Fold/Unfold Transformations

In this chapter, we formally define the *fold, unfold,* and *definition* steps for programs in a constraint query language, restricted to the transformations required for our purposes.

The set $P_i, i \geq 0$, is the set of rules in the program obtained after applying i definition, fold, or unfold steps. The set $N_i, i \geq 0$, is the set of rules defining new predicates after applying i definition, fold, or unfold steps. At any step, a definition, fold, or unfold step may be applied to produce P_i and N_i from P_{i-1} and N_{i-1}, for $i > 0$. The set P_0 is the set of rules in the initial program P, and the set of rules defining new predicates, N_0, is initially $\emptyset$.

Definition Step

1. Let $r_1, \ldots, r_m$ be m rules of the form:

   $$\begin{aligned} &r_1 : p'(\overline{X}) :- C_1(\overline{X}), p(\overline{X}). \\ &\vdots \\ &r_m : p'(\overline{X}) :- C_m(\overline{X}), p(\overline{X}). \end{aligned}$$

   where the variables in $\overline{X}$ are distinct variables; each $C_i(\overline{X}), 1 \leq i \leq m$ is a conjunction of constraints; p' is a predicate not appearing in P_{i-1}, N_{i-1}, and p is a predicate appearing in P_0.

2. The updated set of program rules is given by $P_i = P_{i-1} \cup \{r_1, \ldots, r_m\}$. The updated set of rules defining new predicates is given by $N_i = N_{i-1} \cup \{r_1, \ldots, r_m\}$.

Note that since the variables occurring in the head of the rule are all and only the variables occurring in the body of the rule, the problems described in [30] do not apply.

Unfolding Step

1. Let r be a rule in P_{i-1}, $p(\overline{X})$ a body literal occurring in r, and $r_1, \ldots, r_n$ be all the rules in P_{i-1} whose head literals are unifiable with $p(\overline{X})$.

2. Let $r'_j, 1 \leq j \leq n$, be the result of resolving r with r_j upon $p(\overline{X})$.

3. The updated set of program rules is given by $P_i = (P_{i-1} - \{r\}) \cup \{r'_1, \ldots, r'_n\}$. The set of rules defining new predicates remains unchanged, i.e. $N_i = N_{i-1}$.

Folding Step

1. Let r be a rule in P_{i-1} of the form:

   $$r : p_0(\overline{X_0}) : -\ C_r(\overline{Y}), C_1(\overline{X_1}), p_1(\overline{X_1}), \ldots, C_n(\overline{X_n}), p_n(\overline{X_n}).$$

   where each $C_i(\overline{X_i}), 1 \leq i \leq n$, is a conjunction of constraints on the variables in $p_i(\overline{X_i})$, and $C_r(\overline{Y})$ is also a conjunction of constraints. Let $r1$ be a rule in N_{i-1} of the form:

   $$r1 : p'(\overline{X}) : -\ C(\overline{X}), p_i(\overline{X}).$$

   where $\overline{X}$ is a tuple of distinct variables and $C(\overline{X})$ is a conjunction of constraints.

2. Let there be a substitution θ and a body literal $p_i(\overline{X_i})$ in r such that: $p_i(\overline{X_i}) = \theta[p_i(\overline{X})]$ and $C_i(\overline{X_i}) \supset \theta[C(\overline{X})]$.

3. Let r' be a rule obtained from r by deleting the literal $p_i(\overline{X_i})$ from the body of the rule and adding $\theta[p(\overline{X})]$ to the body.

4. The folding step is described by $P_i = (P_{i-1} - \{r\}) \cup \{r'\}; N_i = N_{i-1}$.

Note that we do not mark rules as "foldable" or not, as is done in [4]. The algorithm that uses these steps to rewrite a program in a constraint query language ensures that no undesirable folds (like a rule being folded by itself) occur.