ON THE IMPLEMENTATION OF DATA INTENSIVE LOGIC PROGRAMS

Raghunath Ramakrishnan

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-87-21 May 1987

Abstract

The problem of efficiently implementing recursive database queries expressed as logic programs has received much attention. This has been motivated by a desire to extend the query language of a relational database to a general purpose language. This is a necessary step in the evolution of *knowledge bases*, which are sophisticated databases providing support for expert system applications.

Databases typically contain gigabytes of facts and organize this data carefully in order to optimize relational operations such as the *join*. Top-down resolution based strategies (for example, Prolog) do not use this data organization effectively. Further, top-down strategies suffer from the fact that their implementations, for efficiency reasons, are usually *incomplete*, and so they do not realize the declarative semantics of logic programming. Thus, alternative methods which are complete are also of interest in the context of a general purpose logic programming language. Several approaches based on bottom-up evaluation have been proposed to deal with these problems. However, the criticism of bottom-up methods has been that they generally do not succeed in restricting the computation by utilizing information in the query.

We examine these evaluation methods and identify a common factor, *sideways information passing*, that is used to restrict computation. This provides a framework in terms of which various evaluation methods can be compared. We define this notion formally, and use it to generalize a family of bottom-up evaluation methods proposed in the literature. We also treat rules that extend Horn Clause logic programs with *layered* negation and set generation. Our results show that bottom-up methods can implement any sideways information passing strategy, and thus answer the chief criticism of them. The issues of choosing an information passing strategy and an evaluation method for a given query require an understanding of performance measures. While we have no conclusive answers, we make an important first step with a performance analysis that examines several evaluation methods over a range of workloads. Finally, we examine the important issue of *safety*, that is, whether a query has a finite set of answers.

ON THE IMPLEMENTATION OF DATA INTENSIVE LOGIC PROGRAMS

by

Raghunath Ramakrishnan, B.Tech.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 1987

To Appa

Acknowledgements

I owe much to my advisor, Avi Silberschatz, for the wide latitude and constant encouragement that he has given me. Francois Bancilhon's insight, the impeccable condition of his spare tire - no comment on his Buick, though! - and his blackboard wizardry have all been sources of inspiration. Catriel Beeri's rigor and incisive comments have influenced my thinking, and Oded Shmueli has often taken up where Catriel left off. Shamim Naqvi, Ravi Krishnamurthy, Dick Tsur and Carlo Zaniolo have always been willing to waste their time listening to me, and I have cheerfully taken advantage of their soft hearts and hard heads. Many other people at MCC have given freely of their store of advice and bad jokes, and I am grateful to them, and to MCC, for enriching me. Jim Bitner, Hank Korth, Chris Lengauer, and Clement Leung at UT have been generous with their time and comments at various times during my travails as a graduate student. Many graduate students at UT have contributed to my education and entertainment, and in particular, I owe a special debt to Ernie Cohen for simply being Ernie Cohen. On a different note, I'm grateful to my friends Jit Biswas and Avijit Saha for their affectionate tolerance of a harmless nut.

I do not thank my parents, since it would be like thanking myself. I wish my father were here to share this moment with me. His integrity and ideals will always be the standards by which I measure myself, and it is to him that this thesis is dedicated, with love and respect.

My research as a graduate student was partially supported by NSF grants MCS-8104017, MCS-8124613, and DCR-8507224, ONR contract N00014-86-K-1061, a University of Texas Graduate Fellowship, an MCD Fellowship, an IBM Graduate Fellowship, and research grants from MCC.

ON THE IMPLEMENTATION OF DATA INTENSIVE LOGIC PROGRAMS

Publication No. ____________

Raghunath Ramakrishnan, Ph.D.

The University of Texas at Austin, 1987

Supervising Professor: Abraham Silberschatz

The problem of efficiently implementing recursive database queries expressed as logic programs has received much attention. This has been motivated by a desire to extend the query language of a relational database to a general purpose language. This is a necessary step in the evolution of *knowledge bases*, which are sophisticated databases providing support for expert system applications.

Databases typically contain gigabytes of facts and organize this data carefully in order to optimize relational operations such as the *join*. Top-down resolution based strategies (for example, Prolog) do not use this data organization effectively. Further, top-down strategies suffer from the fact that their implementations, for efficiency reasons, are usually *incomplete*, and so they do not realize the declarative semantics of logic programming. Thus, alternative methods which are complete are also of interest in the context of a general purpose logic programming language. Several approaches based on bottom-up evaluation have been proposed to deal with these problems. However, the criticism of bottom-up methods has been that they generally do not succeed in restricting the computation by utilizing information in the query.

We examine these evaluation methods and identify a common factor, *sideways information passing*, that is used to restrict computation. This provides a framework in terms of which various evaluation methods can be compared. We define this notion formally, and use it to generalize a family of bottom-up evaluation methods proposed in the literature. We also treat rules that extend Horn Clause logic programs with *layered* negation and set generation. Our results show that bottom-up methods can implement any sideways information passing strategy, and thus answer the chief criticism of them. The issues of choosing an information passing strategy and an evaluation method for a given query require an understanding of performance measures. While we have no conclusive answers, we make an important first step with a performance analysis that examines several evaluation methods over a range of workloads. Finally, we examine the important issue of *safety*, that is, whether a query has a finite set of answers.

Table of Contents

Chapter 1 - Overture: Logic Programs, Databases, and this Thesis

1. Introduction

Logic programming using Horn Clauses [Van Emden and Kowalski 76, Kowalski 79, 83, Colmerauer 85] has aroused a great deal of interest in the past decade because of its declarative semantics. The logic programming concept is that a set of Horn clauses may be viewed as a program whose semantics is the declarative reading of the clauses as statements in standard logic. A procedure call is a negated literal, and is executed by resolving it with clauses in the program.

Logic has been a traditional tool for knowledge representation and inference, and it is easy to see the potential of logic programming in Artificial Intelligence applications. In fact, logic programming received a strong impetus with its selection as the kernel language for the Japanese Fifth Generation project [Kunifuji and Yokota 82]. Prolog, the best known realization of logic programming, has been used for a variety of applications including symbolic mathematics, compiler aided design, natural language processing, debugging, expert systems and databases [Kowalski 83].

There is a close relationship between logic programming and relational databases since their semantics rely upon first-order logic. The tuples in database relations may be thought of as facts, and each rule in a logic program may be thought of as defining some of the tuples in the relation corresponding to the predicate in the head of the rule. This natural correspondence, added to the power and elegance of logic, has created a great deal of interest in the use of logic as a database query language. We shall return to this point shortly.

An attractive feature of this paradigm is that programs may be understood declaratively without reference to how control flows during execution, and it is this simple and well-defined semantics that has been emphasized by proponents of logic programming. Crucial to this feature is the requirement that the implementation should return exactly the set of values which follow from the interpretation of the program according to standard logic. That is, the answer set should include only values that satisfy the constraints represented by the rules, and further, every value that satisfies these constraints *must* be returned. The latter property is often referred to as *completeness*.

Languages like Prolog [Roussel 75, Clocksin and Mellish 81, Colmerauer 85] and IC-Prolog [Clark et al. 82] have attempted to realize the Horn Clause paradigm, as advocated by Kowalski [Kowalski 79], by retaining the declarative semantics and giving the programmer some control over the execution. These attempts have not been entirely successful since none of these languages preserves logical completeness, thus corrupting the declarative semantics. This has been due to two factors:

i) Their depth-first execution strategy (which raises the possibility of an infinite computation that may not enumerate all answers), and

ii) Annotations such as the *cut*, which prune the search space with nothing to ensure that no valid solutions are pruned. †

To elaborate a little, the depth-first property of their search algorithm raises the possibility that they explore one branch of the search tree forever (of course, such a branch would be infinite). If, for instance, all branches which yield answers are finite, and there are also infinite branches, then one of these methods might well explore one of the infinite branches before it has considered all finite branches. Thus, it could (unsuccessfully) explore one branch forever, while there are still answers to be found. This implies that they may fail to find the set of all answers, even when this set is finite.

Since the late seventies, researchers have recognized that the *complete* semantics of logic programming makes it an attractive high level database query language. Traditionally, relational query languages have been defined over finite structures [Codd 70], and thus they have the completeness property trivially, since infinite branches do not arise in evaluating relational queries. We note that logic programming languages are more powerful than traditional query languages (e.g., they can perform transitive closure) [Aho and Ullman 79]. Unfortunately, implementations

† The logic programming paradigm has been compromised even more by *committed-choice* languages such as Concurrent Prolog [Shapiro 83, Ramakrishnan and Silberschatz 86] which non-deterministically select one of the branches at each OR-node in the search tree and prune the rest.

such as Prolog are unsuitable for database applications because they do not preserve completeness, and further, their implementation strategy, a top-down, depth-first search, is inefficient in the presence of large sets of data containing *duplication* [Bancilhon and Ramakrishnan 86, 87]. (Informally, duplication corresponds to the existence to several derivations of the same fact.)

There has recently been a surge in the development of alternative implementation techniques which ensure completeness and perform efficiently in the presence of large numbers of facts. It must be said at the outset, however, that such techniques will probably not perform optimally under all conditions. Strategies such as those used in the implementation of Prolog will doubtless continue to be more suitable for certain classes of applications.

The focus of these alternative techniques has been the compilation of the clauses into an extended relational algebra which permits us to use efficient techniques for performing selects, projects, and joins. They rely upon bottom-up fixpoint computation, which is logically complete. The major problem is to restrict the computation by only evaluating facts that are relevant to the query. The various methods proposed in the literature primarily address this issue.

2. An Overview of the Thesis

This thesis is concerned with the efficient implementation of Horn clause logic programming as a query language for relational databases. The emphasis is on preserving logical completeness and performing efficiently in the presence of large numbers of facts. (Typically, databases contain gigabytes of facts.) We shall refer to logic programming in this context as *data intensive logic programming*.

This problem has attracted much attention in the past decade, and several methods have been proposed by researchers in the areas of Artificial Intelligence and Databases. While this work has the merit that several people have identified a single problem and presented partial solutions, there has been no uniform presentation of these ideas, and there has been no work done on comparing these methods.

One of the contributions of this thesis is to present a detailed and unified treatment of the methods proposed in the literature. We also present an analytical performance evaluation that compares these methods rigorously, and gives a first insight on their relative merits. We consider this to be important for two reasons. First, it is an attempt to quantitatively compare all the methods proposed in the literature, and is unique in this respect. Second, it is an analytical approach which attempts to develop a model for characterizing *data*. While the resulting model is limited in scope, it is an important first step.

The results of the performance evaluation are surprisingly robust, and succeed in explaining the relative performance of the various methods in terms of a few qualitative factors. This work is described in [Bancilhon and Ramakrishnan 86, 87].

While several methods have been proposed for efficiently implementing data intensive logic programs, their relationship to each other is far from clear. It is difficult to understand what they have in common, and what points of real difference, if any, they possess. In fact, one of the claims we make in presenting a survey of these methods is that some of them are identical.

We consider the notion of *sideways information passing* - the use of information obtained in the solution of some goals to restrict the computation involved in solving other goals - and provide a rigorous definition that is sufficiently general to describe all the methods proposed in the literature (at the level of abstraction of this formalism). We suggest that every method can be viewed as a sideways information passing strategy associated with a *control mechanism* that implements it.

Having defined the class of sideways information passing strategies by means of a graph formalism, we show that bottom-up evaluation is a very general control mechanism. In particular, we show that every sideways information passing strategy can be implemented using bottom-up evaluation by appropriately rewriting the program before evaluating it. This result addresses the main problem associated with bottom-up evaluation, which is (or perhaps we should say ''was'') the inability to restrict the computation by utilizing information present in the query. Thus, for instance, we can ''mimic'' Prolog in the way it restricts its search, while using bottom-up control.

The above result provides partial justification for the claim that information passing and control are two essentially independent components of an evaluation method.

We establish this result by describing several program rewriting algorithms which generalize (to work with arbitrary Horn clause rules) a family of algorithms described in the literature, including Magic Sets [Bancilhon et al. 86], Counting [Bancilhon et al. 86, Sacca and Zaniolo 86] and their variants. This work is described in [Beeri and Ramakrishnan 87].

Horn clause logic programs do not contain negation, but negation is essential for practical programming, and most logic programming languages augment the Horn clause formalism by supporting some form of negation. Negation based on *layering* has been suggested independently by several researchers [Apt et al. 86, Naqvi 86, Van Gelder 86], and has the attractive feature that its semantics is defined by a minimal Herbrand model which is equal to the least fixpoint of the program. The layering approach has been used to define negation and set-generation (the ability to compute sets and assign them as the value of a variable) in the programming language LDL1 [Beeri et al. 87].

We extend the definition of sideways information passing to cover programs containing layered negation and set generation and use this to generalize the rewriting algorithms to cover such programs. This work is presented as part of [Beeri et al. 87].

An important property of a database query expressed as a logic program is whether the set of answers is finite. Since the logic programming formalism allows the use of function symbols to construct terms, it is possible to write programs with an infinite set of answers. We call a query *safe* if it has a finite set of answers.† We present some results on determining whether a query is safe by using a dataflow formalism. This work is described in [Ramakrishnan et al. 87].

3. Organization of the Thesis

We begin by presenting an introduction to Horn clause logic programming and the notation we use in Chapter 2. In Chapter 3, we present a detailed survey of the methods proposed in the literature for evaluating data-intensive logic programs. In

† A similar problem arises in the context of a Horn clause based query language for a non-first normal form data model, and we consider it in [Ramakrishnan and Silberschatz 86a].

Chapter 4, we discuss sideways information passing, and in Chapter 5 we use it to develop a family of program rewriting algorithms. We present some results concerning information passing strategies and rewriting algorithms in Chapter 6. The results of Chapter 5 are extended to cover programs with stratified negation and set-generation in Chapter 7. We present a performance analysis comparing the various methods in Chapter 8. We study safety of queries in Chapter 9, and present conclusions in Chapter 10.

In reading this thesis, the reader who is familiar with logic programs and databases can skip Chapter 2 and refer to it only if needed. The material in Chapter 3 is required in order to understand the performance analysis presented in Chapter 8, but only the sections on Naive and Semi-Naive evaluation, Magic Sets, and Counting are needed for understanding the rest of this thesis. Chapter 4 is a pre-requisite for Chapters 5, 6 and 7. In reading Chapter 5, sections 5.3 through 5.7 may be skipped without loss of continuity. Chapter 6 presents some results characterizing the methods discussed in Chapter 5. Chapter 8 presents a performance analysis of the various methods presented earlier, and requires an understanding of the material in Chapter 3, and sections 5.1 and 5.2 of Chapter 5. Chapters 6 through 9 are independent of each other, and may be read or omitted according to the reader's interests.

Chapter 2 - Introduction, Notation, and Other Preliminaries

In this chapter, we provide an introduction to logic programming and databases, describe the notation we use in this thesis, and define several terms and concepts that are used extensively in subsequent chapters. †

1. Logic Databases: An Example

Let us start by informally discussing an example. Here is an instance of what we call a ''logic database'':

```
parent(cain,adam).
parent(abel,adam).
parent(cain,eve).
parent(abel,eve).
parent(sem,abel).
ancestor(X,Y) :- ancestor(X,Z),ancestor(Z,Y).
ancestor(X,Y) :- parent(X,Y).
generation(adam,1).
generation(X,I) :- generation(Y,J), parent(X,Y),J=I-1.
generation(X,I) :- generation(Y,J), parent(Y,X),J=I+1.
```

In this database, we have a set of predicate or relation names (parent, ancestor and generation), a set of arithmetic predicates (I=J+1, I=J-1), a set of constants (adam, eve, cain, sem and abel), and a set of variables (X,Y and Z). The database consists of a set of sentences ending with a period. ''parent(cain,adam)'' is a fact, while ''ancestor(X,Y) :- parent(X,Y)'' is a rule.

Note that this is a purely syntactic definition. Let us now associate a meaning with the database. We first associate with each constant an object from the real world. Thus, with ''adam'' we associate the individual whose name is ''adam''. Then, we associate with each arithmetic predicate name the corresponding arithmetic operator. Then we can interpret intuitively each fact and each rule. For instance we interpret ''parent(cain,adam)'' by saying that the predicate parent is true for the

† The material in this chapter is from [Bancilhon and Ramakrishnan 86].

couple (cain,adam), and we interpret the rule

ancestor(X,Y) :- ancestor(X,Z), ancestor(Z,Y).

by saying that if there are three objects X, Y and Z such that ancestor(X,Z) is true and ancestor(Z,Y) is true, then ancestor(X,Y) is also true.

This leads to an interpretation which associates with each predicate a set of tuples. For instance with the predicate ancestor we associate the interpretation {(cain,adam), (abel,adam), (cain,eve), (abel,eve), (sem,abel), (sem,adam), (sem,eve)}, and with the predicate generation we associate the interpretation {(adam,1), (eve,1), (cain,2), (abel,2), (sem,3)}

We need to find a method for answering queries posed against the database. For instance, if we want all tuples in the relation ancestor such that the second argument is adam, how do we find the answers {ancestor(cain,adam), ancestor(abel,adam), ancestor(sem,adam)}?

Let us now formalize these notions and define a logic database. We first define it syntactically, then we attach an interpretation to this syntax.

2. Syntax of a Logic Database

We first define four sets of names: *variable* names, *constant* names, *predicate* or *relation* names and *evaluable predicate* names.

We adopt the Prolog convention of denoting variables by strings of characters starting with an upper case letter and constants by strings of characters starting with a lower case letter or integers. For instance X1, Father and Y are variables, while john, salary and 345 are constants. We use identifiers starting with lower case letters for predicates names and relation names (evaluable and non-evaluable). We use the term relation (from database terminology) and predicate (from logic terminology) indistinguishably to represent the same object. We shall, however, interpret them differently: a relation will be interpreted by a set of tuples, and a predicate by a true/false function. There is a fixed arity associated with each relation/predicate.

The set of evaluable predicate names is a subset of the set of predicate names. We will not be concerned with their syntactic recognition; in the examples it will be clear from the name we use. The main examples of evaluable predicate names are arithmetic predicates. For instance, sum, difference and greater-than are examples of evaluable predicates of arity 3, 3 and 2 respectively, while parent and ancestor are non-evaluable predicates of arity 2.

A *literal* is of the form p(t1,t2,...,tn) where p is a predicate name of arity n and each ti is a constant or a variable. For instance father(john,X), ancestor(Y,Z), id(john,25,austin) and sum(X,Y,Z) are literals. An *instantiated* literal is one which does not contain any variables. For instance id(john,doe,25,austin) is an instantiated literal, while father(john,Father) is not.

We allow ourselves to write evaluable literals using functions and equality for the purpose of clarity. For instance, Z = X+Y denotes sum(X,Y,Z), I = J+1 denotes sum(J,1,I), and X > 0 denotes greater-than(X,0).

If p(t1,t2,...,tn) is a literal, we call (t1,t2,...,tn) a *tuple*.

A *rule* is a statement of the form

```
p :- q1,q2,...,qn.
```

where p and the qi's are literals such that the predicate name in p is a non-evaluable predicate. p is called the *head* of the rule, and each of the qi's is called a *goal*. The conjunction of the qi's is the *body* of the rule. We have adopted the Prolog notation of representing implication by ':-' and conjunction by ','. For instance

```
uncle(john,X) :- brother(X,Y), parent(john,Y).
```

is a rule with head ''uncle(john,X)'' and body ''brother(X,Y), parent(john,Y)''.

A *ground clause* is a rule in which the body is empty. A *fact* is a ground clause which contains no variables. For instance

```
loves(X,john).
loves(mary,susan).
```

are ground clauses, but only the second of these is a fact.

A *database* is a set of rules; note that this set is not ordered. Given a database, we can partition it into a set of facts, which we call the *extensional* database, and the set of all other rules, which we call the *intensional* database.

3. Interpretation of a Logic Database

Up to now our definitions have been purely syntactical. Let us now give an interpretation of a database. This will be done by associating with each relation name in the database a set of instantiated tuples. We first assume that with each evaluable predicate p is associated a set natural(p) of instantiated tuples which we call its *natural interpretation*. For instance, with the predicate *sum* is associated an infinite set of all the 3-tuples (x,y,z) of integers such that the sum of x and y is z. In general the natural interpretation of an evaluable predicate is infinite.

Given a database, an *interpretation* of this database is a mapping which associates with each relation name a set of instantiated tuples.

A *model* of a database is an interpretation I such that:

i) For each evaluable predicate p, I(p) = natural(p), and,

ii) For any rule, p(t) :- q1(t1),q2(t2),...,qn(tn), for any instantiation σ of the variables of the rule such that σ(ti) is in the interpretation of qi for all i, σ(t) is also in the interpretation of p.

    This is simply a way of saying that in a model, if the right hand side is true then the left hand side is also true. This implies that for every fact p(x) of the database the tuple x belongs to the interpretation of p.

Of course, for a given database there are many models. The nice property of Horn Clauses is that among all these models there is a *minimal* one (minimal in the sense of set inclusion), which is the one we choose as *the* model of the database [Van Emden and Kowalski 76]. Therefore from now on, when we talk about the model or the interpretation of a database, we mean its minimal model.

Notice that because of the presence of evaluable arithmetic predicates the minimal model is, in general, not finite.

Let p be an n-ary predicate. An *adornment* of p is a sequence *a* of length n of b's and f's [Ullman 85]. For instance bbf is an adornment of a ternary predicate, and fbff is an adornment of predicate of arity 4. An adornment is to be interpreted intuitively as follows: the i-th variable of p is bound (respectively free) if the i-th element of *a* is b (respectively f).

Let p(x1,x2,...,xn) be a literal, an adornment a1a2...an of that literal is an adornment of p such that :

i) If xi is a constant then ai is b,

ii) If xi = xj then ai = aj.

We denote adornments by superscripts. A *query form* is an adorned predicate. Examples of query forms are father^{bf}, id^{bffb}.

A *query* is a query form and an instantiation of the bound variables. We denote it by an adorned literal where all the bound positions are filled with the corresponding constants and the free positions are filled by distinct free variables. Therefore father^{bf}(john,X) and id^{bffb}(john,X,Y,25) are queries. The distinction between queries and query forms are that query forms are actually compiled, and at run-time their parameters will be instantiated. Notice that father(X,X) is not a query form in this formalism.

The *answer* to a query q(t) is the set:

$$\{q(\sigma(t)) \mid \sigma \text{ is an instantiation of t, and } \sigma(t) \text{ is in the interpretation of q}\}$$

4. Structuring and Representing the Database

A predicate which only appears in the intensional database is a *derived* predicate. A predicate which appears only in the extensional database or in the body of a rule is a *base* predicate.

For performance reasons, it is good to decompose the database into a set of pure base predicates (which can then be stored using a standard Database Management System) and a set of pure derived predicates. Fortunately, such a decomposition is always possible, because every database can be rewritten as an ‘‘equivalent’’ database containing only base and derived predicates. By equivalent, we mean that all the predicate names of the original database appear in the modified database and have the same interpretation.

We obtain this equivalent database in the following way: consider any predicate p that is neither base nor derived. By definition, we have a set of facts for p, and p appears on the left of some rules. So we simply introduce a new predicate p_ext and do the following:

i) Replace p by p_ext in each fact of p,

ii) Add a new rule p(X1,X2,...,Xn) :- p_ext(X1,X2,...,Xn), where n = arity of p.

Example:

```
father(a,b).
parent(b,c).
grandfather(b,d).
grandfather(X,Y) :- father(X,Z),parent(Z,Y).
```

becomes:

```
father(a,b).
parent(b,c).
grandfather_ext(b,d).
grandfather(X,Y) :- father(X,Z),parent(Z,Y).
grandfather(X,Y) :- grandfather_ext(X,Y).
```

Most authors have chosen to describe a set of rules through some kind of graph formalism. Predicate Connection Graphs, as presented in [McKay and Shapiro 81], represent the relationship between rules and predicates. Rule/goal graphs, as presented in [Ullman 85], carry more information because predicates and rules are adorned by their variable bindings. We have chosen here to keep the rule/goal graph terminology while using unadorned predicates.

The *rule/goal* graph has two sets of nodes: square nodes which are associated with predicates, and oval nodes which are associated with rules. If there is a rule

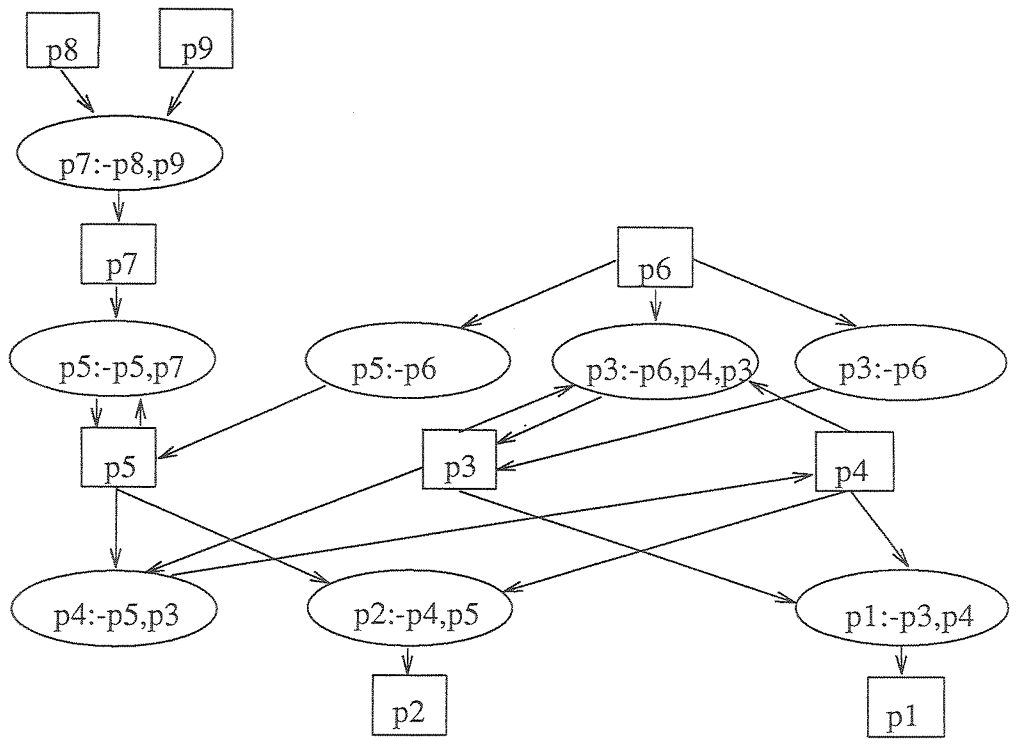
```
r: p:- p1,p2,...,pn
```

in the intensional database, then there is an arc going from node r to node p, and for each predicate pi there is an arc from node pi to node r.

Here is an example of an intensional database. For the sake of simplicity, we omit the variables in the rules:

R1	p1 :- p3,p4
R2	p2 :- p4,p5
R3	p3 :- p6,p4,p3
R4	p4 :- p5,p3
R5	p3 :- p6
R6	p5 :- p5,p7
R7	p5 :- p6
R8	p7 :- p8,p9

The rule/goal graph is:



5. Recursion

Recursion is often discussed in the single rule context. For the purpose of clarity and simplicity, let us first give some temporary definitions in this context. We say that a rule is recursive if it is of the form p(t) :- ...,p(t'),... .

For instance the following rule is recursive:

ancestor(X,Y) :- ancestor(X,Z),parent(Z,Y).

An interesting subcase is that of linear rules. We say that a rule is linear if it is recursive, and the recursive predicate appears once and only once on the right. This property is sometime referred to as regularity [Chang 81].

For instance this rule is linear:

sg(X,Y) :- p(X,XP),p(Y,YP),sg(XP,YP).

This rule is not linear:

ancestor(X,Y) :- ancestor(X,Z),ancestor(Z,Y).

These definitions are fairly simple in the single rule context. They are a little more involved in the context of a set of rules where properties have to be attached to predicates instead of rules. Consider the following database:

p(X,Y) :- b1(X,Z),q(Z,Y).
q(X,Y) :- p(X,Z),b2(Z,Y).

Neither of the rules are recursive according to the above definition, while clearly both predicates p and q are recursive.

We now come to the general definitions of recursion in the multirule context. Let p and q be two predicates. We say that p *derives* q (denoted p → q) if p appears in the body of a rule whose head predicate is q. We define →+ to be the transitive closure of →. A predicate p is said to be *recursive* if p→+p. Two predicates p and q are *mutually recursive* if p→+q and q→+p. It can be easily shown that mutual recursion is an equivalence relation on the set of recursive predicates. Therefore, the set of recursive predicates can be partitioned into disjoint sets of mutually recursive predicates. (We refer to these sets as *blocks*.)

Given a set of rules, we say that the rule

p :- p1,p2,...,pn

is *recursive* if and only if there exists pi in the body of the rule which is mutually recursive to p.

A recursive rule p :- p1,p2,...,pn is *linear* if there is one and only one pi in the body of the rule which is mutually recursive to p. A set of rules is *linear* if every recursive rule in it is linear. For instance, the following system of rules is linear:

```
R1   p(X,Y) :- p1(X,Z),q(Z,Y).
R2   q(X,Y) :- p(X,Z),p2(Z,Y).
R3   p(X,Y) :- b3(X,Y).
R4   p1(X,Y) :- b1(X,Z),p1(Z,Y).
R5   p1(X,Y) :- b4(X,Y).
R6   p2(X,Y) :- b2(X,Z),p2(Z,Y).
R7   p2(X,Y) :- b5(X,Y).
```

The set of recursive predicates is {p,q,p1,p2}, and the set of base predicates is {b1,b2,b3,b4,b5}. The mutually recursive blocks are {[p,q],[p1],[p2]}. The recursive rules are R1, R2, R4 and R6, and the system is linear even though rules R1 and R2 both have two recursive predicates on their right.

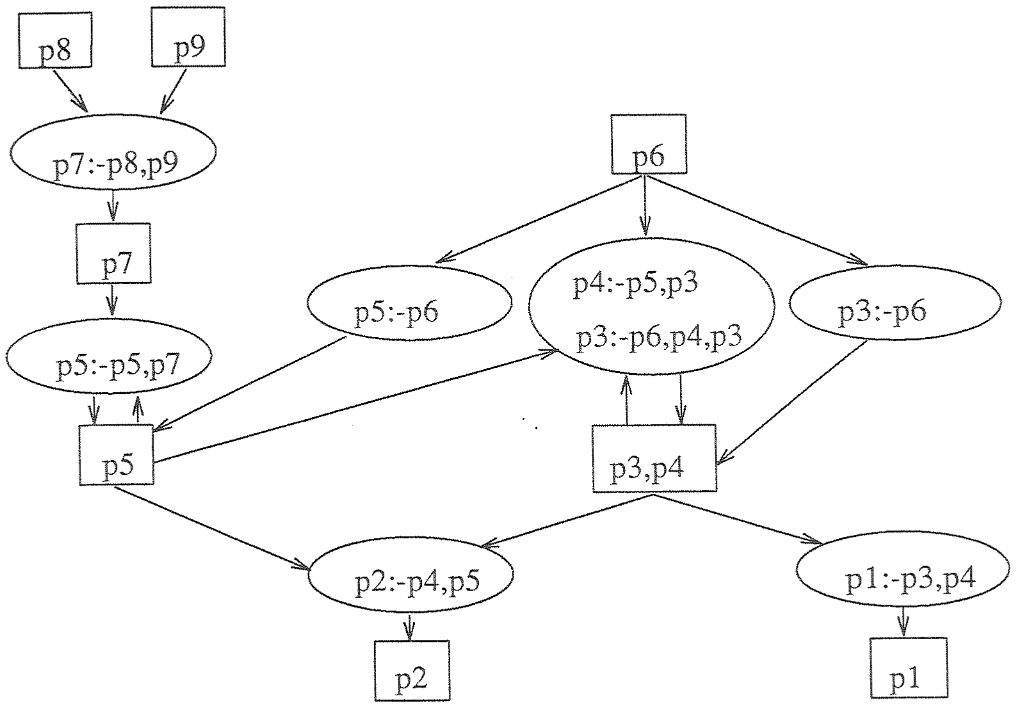
We say that two recursive rules are mutually recursive if and only if the predicates in their heads are mutually recursive. This defines an equivalence relation among the recursive rules.

Thus, mutual recursion defines an equivalence class among recursive predicates and among the recursive rules. Therefore, it groups together all predicates which are mutually recursive to one another, that is, which must be evaluated as a whole. It also groups together all the rules which participate in evaluating those blocks of predicates. Let us now see how this can be represented in the rule/goal graph. We define the *reduced rule/goal graph* as follows:

Square nodes are associated with non-recursive predicates or with blocks of mutually recursive predicates, and oval nodes are associated with non-recursive rules or with blocks of mutually recursive rules. The graph essentially describes the non-recursive part of the database by grouping together all the predicates which are mutually recursive to one another and isolating the recursive parts. For every non-recursive rule of the form r: p:- p1,p2,...,pn, there is an arc going from node r to node p (if p is non-recursive), or to node [p], which is the node representing the block of predicates mutually recursive to p (if p is recursive). For each non-recursive predicate pi, there is an arc from the node pi to the node r, and for each recursive predicate pj there is an arc going from [pj] the node representing the block of predicates mutually recursive to pj.

Finally, each block of recursive rules [r] is uniquely associated to a set of mutually recursive predicates [p], and we draw an arc from [p] to [r] and an arc from [r] to [p]. We also draw an arc from q (if q is non-recursive) or from [q] (if q is recursive) to [r] if there is a rule in [r] which has q in its body. This grouping of predicates in blocks of strongly connected components is presented in [Morris et al. 86].

Here is the representation of the previous database:



6. Safety of Queries

Given a query q in a database D, we say that q is *safe* in D if the answer to q is finite. Obviously unsafe queries are highly undesirable.

Sources of unsafeness are of two kinds:

i) The evaluable arithmetic predicates are interpreted by infinite tables. Therefore they are unsafe by definition. For instance, greater-than(27,X) is unsafe.

ii) Rules with free variables in the head which do not appear in the body are a source of unsafeness in the presence of evaluable arithmetic predicates. (These predicates provide an infinite underlying domain, and the variable from the head which does not appear in the body ranges over that domain).

For instance, in the system

```
good-salary(X) :- X > 100000.
like(X,Y) :- nice(X).
nice(john).
```

the query like(john,X)? is unsafe because in the minimal model of the database like(john,X) is true for every integer X. Note that if the first rule was not there, we could restrict our domain to the (finite) set of constants that appear in our database, and like(john,X)? would be safe and have answer like(john,john).

The problem of safety has received a lot of attention recently [Afrati et al. 86, Ullman 85, Ullman and Van Gelder 85, Ramakrishnan et al. 87, Zaniolo 86]. We shall not survey those results in this section but merely present some simple sufficient syntactic conditions to guarantee safety. A rule is *range restricted* if every variable of the head appears somewhere in the body. Thus in this system:

```
R1    loves(X,Y) :- nice(X).
R2    loves(X,Y) :- nice(X),human(Y).
```

R1, which corresponds to ''nice people love everything'', is not range restricted while R2, which corresponds to ''nice people love all humans'', is. Obviously, every ground rule which is not a fact is not range restricted. For instance

```
loves(john,X).
```

is not range restricted.

A set of rules is range restricted if every rule in this set is range restricted.

It is known [Reiter 78] that if each evaluable predicate has a finite natural interpretation, and if the set of rules is range restricted, then every query defined over this set of rules is safe. This applies obviously to the case where there are no evaluable predicates. However, if there are evaluable predicates with infinite natural interpretations, safety is no longer assured. We now present a simple sufficient condition for safety in the presence of such predicates.

A rule is *strongly safe* if and only if:

i) It is range restricted, and

ii) Each variable in an evaluable predicate also appears in a base predicate.

For example, the rule

```
well-paid(X) :- has-salary(X,Y), Y > 100000.
```

is strongly safe, whereas

```
great-salary(X) :- X > 100000.
```

is not strongly safe.

A set of rules is strongly safe if every rule in this set is strongly safe.

Any query defined over a set of strongly safe rules is safe. However, while this is a sufficient condition, it is not necessary. We can develop better conditions for testing safety, or leave it to the user to ensure that his queries are safe.

7. Effective Computability.

Safety, in general, does not guarantee that the query can be effectively computed. Consider for instance:

```
p1(1,X,Y) :- X≥Y.
p2(X,Y,2) :- X≤Y.
p(X,Y) :- p1(X,Z,Z),p2(Z,Z,Y).
```

The query p(X,Y) is safe (the answer is {p(1,2)}), but there is no safe computation for it. That is, any computation of the answer generates infinite intermediate relations.

However, strongly safe rules are guaranteed to be safe *and* safely computable.

In fact, while we might often be willing to let the user ensure that his queries are safe, it is desirable to ensure that the query can be computed without materializing "infinite" intermediate results. We now present a sufficient condition for ensuring this.

We first need some information about the way (the infinite) arithmetic predicates can propagate bindings. So we characterize each arithmetic predicate by a set of *finiteness dependencies* [Zaniolo 86, Ramakrishnan et al. 87]. A finiteness dependency is of the form $X \rightarrow Y$ where X is a set of attributes and Y is a set of attributes. It is to be interpreted intuitively as follows: Given a fixed set of values for the X attributes, there is a finite number of values of the Y attributes associated with them. Therefore, while their semantics is different from that of functional dependencies, they behave in the same fashion (and have the same axiomatization). Of course, we assume that the natural interpretation of the evaluable predicate satisfies the set of safety dependencies.

For instance, the ternary arithmetic predicate ''sum'' has the dependencies:

$$\{1,2\} \rightarrow \{3\}$$
$$\{1,3\} \rightarrow \{2\}$$
$$\{2,3\} \rightarrow \{1\}$$

while the arithmetic predicate ''greater than'' has only trivial finiteness dependencies (of the form $X \rightarrow X$).

Now consider a rule, and define each variable in the body to be *secure* if it appears in a non-evaluable predicate in the body or if it appears in position i in an evaluable predicate p and there is a subset I of the variables of p which are secure and $I \rightarrow \{i\}$. Note that the definition is recursive.

For example, p(X) :- X = 2*Y, q(Y) is secure since Y appears in the non-evaluable predicate q and $Y \rightarrow X$ in the arithmetic predicate. However, the rule is not secure if we delete q(Y) or replace X = 2*Y by X > Y.

A rule is *bottom-up evaluable* if

i) It is range restricted, and

ii) Every variable in the body is secure.

For instance:

```
p(X,Y) :- Y=X+1, X=Y1+Y2, p(Y1,Y2).
```

is bottom-up evaluable because (i) Y1 and Y2 are secure (they appear in p which is non-evaluable), (ii) in X=Y1+Y2, the safety dependency {Y1,Y2} $\rightarrow$ {X} holds, therefore X is secure, and (iii) in Y=X+1, the safety dependency {X} $\rightarrow$ {Y} holds, therefore Y is secure.

On the contrary

```
p(X,Y) :- X>Y1, q(Y1,Y).
```

is not bottom-up evaluable because X is not secure.

A set of rules is bottom-up evaluable if every rule in this set is bottom-up evaluable.

Any computation using only a set of bottom-up evaluable rules can be carried out without materializing infinite intermediate results. The computation proceeds in a strictly bottom-up manner, using values for the body variables to produce values for the head variables. The bottom-up evaluability criterion ensures that the set of values for body variables is finite at each step. However, there may be an infinite number of steps. For example, if we repeatedly apply the bottom-up evaluable rule given above, at each step we have a finite number of values (in this case, a unique value) for Y1 and Y2, and hence for X and Y. However, we can apply the rule an infinite number of times, producing new values for X and Y at each step.

This completes our brief introduction to logic programs and databases, in which we have defined several important notions and also tried to give an intuitive understanding of some of the important issues. In the next chapter, we present a survey of several proposed evaluation methods for queries presented as logic programs.

Chapter 3 - A Survey of Recursive Query Evaluation Methods

1. Introduction

In the past five years, a large number of methods to deal with the evaluation of Horn rules have been presented in the literature. A method is defined by:

i) An application domain (that is, a class of rules for which it applies), and

ii) An algorithm for replying to queries given such a set of rules.

In studying these methods, we found that the methods were described at different levels of detail and using different formalisms.† They were often difficult to understand, the application domain was not always clearly defined, and no performance evaluation was given for any of the methods, which left the choice of a method completely open when the application domain was the same. Finally, we found that some of the methods were in fact the same.

We think that the methods should be compared according to the following criteria:

i) Size of the application domain, (the larger the better),

ii) Performance of the method, (the faster the better), and

iii) Ease of implementation (the simpler the better).

In this chapter, we give a complete description of our understanding of the methods and of their application domains, and we demonstrate each one of them through an example. As much as possible, we have tried to use the same example, except for some "specialized" methods where we have picked a specific example which exhibits their typical behavior.

2. Characteristics of the Methods

2.1. Query Evaluation vs. Query Optimization

Let us first distinguish between two classes of methods. The first one consists of an actual query evaluation algorithm, that is, a program which, given a query and a database, will produce the answer to the query. Representatives of this class are:

† The material in this chapter is from [Bancilhon and Ramakrishnan 86].

Henschen-Naqvi [Henschen and Naqvi 84], *Query/Subquery* [Vieille 86] or *Extension Tables* [Dietrich and Warren 86], *APEX* [Lozinskii 85], *Prolog* [Roussel 75], *Naive Evaluation* and *Semi-Naive Evaluation.* [Bancilhon 85].

The methods in the second class assume an underlying simple method (which is in fact Naive or Semi-Naive evaluation) and optimize the rules to make their evaluation more efficient. They can all be described as *rule rewriting systems*. These include: *Aho-Ullman* [Aho and Ullman 79], *Counting* [Sacca and Zaniolo 86] and *Reverse Counting* [Bancilhon et al. 86a], *Magic Sets* [Bancilhon et al. 86a] and *Kifer-Lozinskii* [Kifer and Lozinskii 86].

Note that this distinction is somehow arbitrary. Each of the optimization methods could be described as an evaluation algorithm (when adding to it Naive or Semi-Naive evaluation). However, this decomposition has two advantages:

i) It might make sense from an implementation point of view to realize the optimization methods as rule rewriting systems on top of an underlying simpler method such as Naive evaluation, and

ii) From a pedagogical standpoint, they are easier to understand this way, because presenting them as rule rewriting systems captures their essence.

The subsequent characteristics only relate to methods in the first class.

2.2. Interpretation vs. Compilation

A method can be *interpreted* or *compiled*. The notion is somehow fuzzy, and difficult to characterize formally. We say that the method is compiled if it consists of two phases:

i) A compilation phase, which accesses only the intensional database, and which generates an ``object program'' of some form, and

ii) An execution phase, which executes the object program against the facts only.

A second characteristic of compiled methods is that all the database query forms (that is, the query forms on base relations which are directly sent to the DBMS) are generated during the compilation phase. This condition is very important, because it allows the DBMS to precompile the the query forms. Otherwise the database query forms are repetitively compiled by the DBMS during the execution of the query, which is a time consuming operation. If these two conditions do not hold, we say that the method is interpreted. In this case, no object code is produced and there is a fixed program, the ''interpreter'', which runs against the query, the set of rules and the set of facts.

2.3. Iteration vs. Recursion

A rule processing method can be *iterative* or *recursive*. It is iterative if the ''target program'' (in case of a compiled approach) or the ''interpreter'' (in case of the interpreted approach) is iterative. It is recursive if this program is recursive, that is, uses a stack as a control mechanism. Note that in the iterative methods, the data we deal with is statically determined. For instance, if we use temporary relations to store intermediate results, there are a finite number of such temporary relations. On the contrary, in recursive methods the number of temporary relations maintained by the system is unbounded.

2.4. Potentially Relevant Facts

Let D be a database and q be a query. A fact p(a) is *relevant* to the query if there exists a derivation p(a) $\rightarrow^*$ q(b) for some b in the answer set [Lozinskii 85]. If we know all the relevant facts in advance, instead of using the database to reply to the query, we can use the relevant part of the database only, thus cutting down on the set of facts to be examined. A *sufficient set of relevant facts* is a set of facts such that replacing the database by this set of facts gives the same answer to the query. Unfortunately, in general there does not exist a unique minimal sufficient set of facts as the following example shows:

```
suspect(X) :- long-hair(X).
suspect(X) :- alien(X).
long-hair(antoine).
alien(antoine).
```

There are two minimal sufficient sets of relevant facts with respect to the query suspect(X)?, {long-hair(antoine)} and {alien(antoine)}.

The second unfortunate thing about relevant facts is that it is in general impossible to find all the relevant facts in advance without spending as much effort as in replying to the query. Thus, all methods have a way of finding a super-set of relevant facts. We call this set the *set of potentially relevant facts.* A set of potentially relevant facts is *valid* if it contains a sufficient set of relevant facts. An obvious (but not very interesting) valid set is the set of all facts of the database.

2.5. Top Down vs. Bottom Up

Consider the following set of rules and the query:

```
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
ancestor(X,Y) :- parent(X,Y).
query(X) :- ancestor(john,X).
```

We can view each of these rules as productions in a grammar. In this context, the database predicates (parent in this example) appear as terminal symbols, and the derived predicates (ancestor in this example) appear as the non-terminal symbols. Finally, to pursue the analogy, we shall take the distinguished symbol to be query(X). The analogy does not hold totally, for two reasons: (i) the presence of variables and constants in the literals and (ii) the lack of order between the literals of a rule (for instance ''parent(X,Z), ancestor(Z,Y)'' and ''ancestor(Z,Y), parent(X,Z)'' have the same meaning). But we shall ignore these differences, and use the analogy informally.

Let us now consider the language generated by this ''grammar''. It consists of

```
{parent(john,X);
 parent(john,X),parent(X,X1);
 parent(john,X),parent(X,X1),parent(X1,X2);
 ...}
```

This language has two interesting properties: (i) it consists of sentences involving only base predicates, that is, each word of this language can be directly evaluated against the database, and (ii) if we evaluate each word of this language against the database and take the union of all these results, we get the answer to the query.

There is, however, a minor problem with this interpretation. The language is not finite, and we would have to evaluate an infinite number of sentences. To get out of this difficulty, we use termination conditions which tell us when to stop. An example of such a termination condition is: if one word of the language evaluates to the empty set, then all the subsequent words will also evaluate to the empty set, so we can stop generating new words. Another example of a termination condition is: if a word evaluates to a set of tuples, and all these tuples are already in the evaluation of the words preceding it, then no new tuple will ever be produced by the evaluation of any subsequent word, thus we can stop at this point.

All query evaluation methods in fact do the following:

i) Generate the language,

ii) While the language is generated, evaluate all its sentences, and

iii) At each step, check for the termination condition.

Therefore, there are essentially two classes of methods: those which generate the language bottom up, and those which generate the language top-down. The bottom-up methods start from the terminals (the base relations) and keep assembling them to produce non-terminals (derived relations) until they generate the distinguished symbol (the query). The top-down methods start from the distinguished symbol (the query) and keep expanding it by applying the rules to the non-terminals (derived relations). As we shall see, top-down methods are often more efficient because they ''know'' which query is being solved, but they are more complex. Bottom up methods are simpler, but they compute a lot of useless results because they do not know what query they are evaluating.

3. The Methods

We shall use the same example for most of the methods. The intensional database and query are:

R1 ancestor(X,Y) :- parent(X,Z),ancestor(Z,Y).

R2 ancestor(X,Y) :- parent(X,Y).

R3 query(X) :- ancestor(b,X).

The extensional database is:

```
parent(a,b).
parent(a,c).
parent(b,d).
parent(b,e).
parent(d,f).
parent(y,z).
```

3.1. Naive Evaluation

Naive Evaluation is a bottom-up, compiled, iterative method.

Its application domain is the set of bottom-up evaluable rules.

In a first phase, the rules which derive the query are compiled into an iterative program. The compilation process uses the reduced rule/goal graph. It first selects all the rules which derive the query. A temporary relation is assigned to each derived predicate in this set of rules. A statement which computes the value of the output predicate from the value of the input predicates is associated with each rule node in the graph. With each set of mutually recursive rules, there is associated a loop which applies the rules in that set until no new tuple is generated. Each temporary relation is initialized to the empty set. Then computation proceeds from the base predicates, capturing the nodes of the graph.

In this example, the rules which derive the query are {R1, R2, R3}, and there are two temporary relations: ancestor and query. The method consists of applying R2 to parent, producing a new value for ancestor, then applying R1 to ancestor until no new tuple is generated, then applying R3.

The object program is:

```
begin
initialize ancestor to the empty set;
evaluate (ancestor(X,Y) :- parent(X,Y));
insert the result in ancestor;
while ‘‘new tuples are generated’’ do
  begin
  evaluate (ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y))
  using the current value of ancestor;
  insert the result in ancestor
  end;
evaluate (query(X) :- ancestor(b,X));
insert the result in query
end.
```

The execution of the program against the data goes as follows:

Step 1: Apply R2.

The resulting state is:

ancestor = {(a,b), (a,c), (b,d), (b,e), (d,f), (y,z)}

query = { }

Step 2: Apply R1.

The following new tuples are generated:

ancestor: {(a,d), (a,e), (b,f)}

And the resulting state is:

{(a,b), (a,c), (b,d), (b,e), (d,f), (y,z), (a,d), (a,e), (b,f)}

query = { }

New tuples have been generated so we continue:

Step 3: Apply R1.

The following tuples are generated:

ancestor: {(a,d), (a,e), (b,f), (a,f)}

The new state is:

{(a,b), (a,c), (b,d), (b,e), (d,f), (y,z), (a,d), (a,e), (b,f), (a,f)}

query = { }

Because (a,f) is new, we continue:

Step 4: Apply R1.

The following tuples are generated:

ancestor: {(a,d), (a,e), (b,f), (a,f)}

Because there are no new tuples, the state does not change and we move to R3.

Step 5: Apply R3.

The following tuples are produced:

query: {(b,d), (b,f)}

The new state is:

{(a,b), (a,c), (b,d), (b,e), (d,f), (y,z), (a,d), (a,e), (b,f), (a,f)}

query = {(b,d), (b,f)}.

The algorithm terminates.

In this example, we note the following inefficiencies:

i) The entire relation is evaluated, that is, the set of potentially relevant facts is the set of facts of the base predicates which derive the query, and

ii) Step 3 completely duplicates step 2.

Naive evaluation is the most widely described method in the literature. It has been presented in a number of papers under different forms. The inference engine of SNIP, presented in [Shapiro and McKay 80, Shapiro et al. 82, McKay and Shapiro 81], is in fact an interpreted version of Naive evaluation. The method described in [Chang 81], while based on a very interesting language paradigm and restricted to linear systems, is a compiled version of Naive evaluation based on relational algebra. The method in [Marque-Pucheu 83, Marque-Pucheu et al. 84] is a compiled version of Naive evaluation using a different algebra of relations. The method in [Bayer 85] is another description of Naive evaluation. The framework presented in [Delobel 86] also uses Naive evaluation as its inference method. SNIP is, to our knowledge, the only existing implementation in the general case.

3.2. Semi-Naive Evaluation

Semi-Naive evaluation is a bottom-up, compiled and iterative method.

Its application range is the set of bottom-up evaluable rules.

This method uses the same approach as Naive evaluation, but tries to cut down on the number of duplications. It behaves exactly as Naive evaluation, except for the loop mechanism where it tries to avoid redundancy.

Let us first try to give an idea of the method as an extension of Naive evaluation. Let p be a recursive predicate. Consider a recursive rule having p as a head predicate and let us write this rule:

$$p \text{ :- } \phi(p1,p2,\ldots,pn,q1,q2,\ldots,qm).$$

where p1,p2,...,pn are mutually recursive to p, and q1,q2,..,qm are base or derived predicates, which are not mutually recursive to p.

In the Naive evaluation method, all the qi's are fully evaluated when we start computing p and the pi's. On the other hand p and the pi's are all evaluated inside the same loop (together with the rest of predicates mutually recursive to p).

Let pj(i) be the value of the predicate pj at the i-th iteration of the loop. At this iteration, we compute

$$\phi(p1(i),p2(i),\ldots,pn(i),q1,q2,\ldots,qm).$$

During that same iteration each pj receives a set of new tuples. Let us call this new set dpj(i). Thus the value of pj at the beginning of step (i+1) is pj(i) + dpj(i) (where + denotes union).

At step (i+1) we evaluate

$$\phi((p1(i)+dp1(i)),\ldots,(pn(i)+dpn(i)),q1,\ldots,qm),$$

which, of course, recomputes the previous expression (because ϕ contains no negation and is thus monotonic).

Ideally however, we wish to compute only the *new* tuples, that is, the expression:

$$d\phi(p1(i),dp1(i),\ldots,pn(i),dpn(i),q1,\ldots,qm) =$$
$$\phi((p1(i)+dp1(i)),\ldots,(pn(i)+dpn(i)),q1,\ldots,qm) - \phi(p1(i),\ldots,pn(i),q1,\ldots,qm)$$

The basic principle of the Semi-Naive method is the evaluation of the differential of ϕ instead of the entire ϕ at each step. The problem is to come up with a first order expression for $d\phi$, which does not contain any difference operator. Let us assume there is such an expression, and describe the algorithm. With each recursive predicate p are associated four temporary relations p.before, p.after, dp.before and dp.after. The object program for a loop is as follows:

```
while ``the state changes'' do
  begin
  for all mutually recursive predicates p do
    begin
```

```
   initialize dp.after to the empty set;
   initialize p.after to p.before;
   end
 for each mutually recursive rule do
   begin
   evaluate $d\phi$(p1,dp1,...,pn,dpn,q1,...,qn) using the current values of
   pi.before for pi and of dpi.before for dpi;
   add the resulting tuples to dp.after;
   add the resulting tuples to p.after
   end
 end.
```

All we have to do now is provide a way to generate $d\phi$ from ϕ. The problem is not solved in its entirety and only a number of transformations are known. In [Bancilhon 85], some of them are given in terms of relational algebra.

It should be noted however, that for the method to work, the only property we have to guarantee is that:

$$\phi(p1+dp1,...) - \phi(p1,...) \subseteq d\phi(p1,dp1,...) \subseteq \phi(p1+dp1,...)$$

Clearly, the closer $d\phi(p1,dp1,...)$ is to $(\phi(p1+dp1,...) - \phi(p1,...))$, the better the optimization is. In the worse case, where we use ϕ for $d\phi$, Semi-Naive evaluation behaves as Naive evaluation. Here are some simple examples of rewrite rules:

if $\phi(p,q) = p(X,Y),q(Y,Z)$, **then** $d\phi(p,dp,q) = dp(X,Y),q(Y,Z)$

More generally when ϕ is linear in p, the expression for $d\phi$ is obtained by replacing p by dp.

if $\phi(p1,p2) = p1(X,Y),p2(Y,Z)$,
then $d\phi(p,dp) = p1(X,Y),dp2(Y,Z)+dp1(X,Y),p2(Y,Z)+dp1(X,Y),dp2(Y,Z)$

Note that this is not an exact differential but a reasonable approximation.

The idea of Semi-Naive evaluation underlies many papers. A complete description of the method based on relational algebra is given in [Bancilhon 85]. The idea is also present in [Bayer 85].

It should also be pointed out that, in the particular case of linear rules, because the differential of $\phi(p)$ is simply $\phi(dp)$, it is sufficient to have an inference engine which only uses the new tuples. Therefore many methods which are restricted to linear rules do indeed use Semi-Naive evaluation. Note also that when the rules are not linear, applying Naive evaluation only to the ''new tuples'' is an incorrect method (in the sense that it does not produce the whole answer to the query). This can be easily checked on the recursive rule:

ancestor(X,Y) :- ancestor(X,Z),ancestor(Z,Y).

In this case, if we only feed the new tuples at the next stage, the relation which we compute consists of the ancestors whose distance to one another is a power of two.

We illustrate the method for the case of linear rules by showing the execution of our running example.

Step 1: Apply R2.
The resulting state is:
ancestor = {(a,b), (a,c), (b,d), (b,e), (d,f), (y,z)}
query = {}
Step 2: Apply R1.
The following new tuples are generated:
ancestor: {(a,d), (a,e), (b,f)}
And the resulting state is:
{(a,b), (a,c), (b,d), (b,e), (d,f), (y,z), (a,d), (a,e), (b,f)}
query = {}
New tuples have been generated so we continue:
Step 3: Apply R1.
The following tuples are generated:
ancestor: {(a,f)}
The new state is:
{(a,b), (a,c), (b,d), (b,e), (d,f), (y,z), (a,d), (a,e), (b,f), (a,f)}
query = {}
Because (a,f) is new, we continue:
Step 4: Apply R1.

No tuples are generated, and so we move to R3.

Step 5: Apply R3.

The following tuples are produced:

query: {(b,d), (b,f)}

The new state is:

{(a,b), (a,c), (b,d), (b,e), (d,f), (y,z), (a,d), (a,e), (b,f), (a,f)}

query = {(b,d), (b,f)}.

The algorithm terminates.

A comparison with the Naive method shows that the method improves by not duplicating step 2 in step 3. However, the entire relation is still computed.

To our knowledge, outside of the special case of linear rules, the method as a whole has not been implemented.

3.3. Iterative Query/Subquery

Iterative Query/Subquery (QSQI) is an interpreted, top-down method.

Its application domain is the set of range restricted rules without evaluable predicates.

The method associates a temporary relation with every relation which derives the query, but the computation of the predicates deriving the query is done at run time. QSQI also stores a set of queries which are currently being evaluated. When several queries correspond to the same query form, QSQI stores and executes them as a single object. For instance, if we have the queries p(a,X) and query p(b,X), we can view this as query p({a,b},X). We call such an object a *generalized query*. The state memorized by the algorithm is a couple <Q,R>, where Q is a set of generalized queries, and R is a set of derived relations, together with their current values.

The iterative interpreter is as follows:

```
Initial state is <{query(X)},{}>
while the state changes do
 for all generalized queries in Q do
   for all rules whose head matches the generalized query do
      begin
```

```
    unify rule with the generalized query; (that is, propagate
    the constants. this generates new generalized queries for each
    derived predicate in the body by looking up the base relations.)
    generate new tuples;
    (by replacing each base predicate on the right by its value and every
    derived predicate by its current temporary value.)
    add these new tuples to R;
    add these new generalized queries to Q
    end
```

Let us now run this interpreter against our example logic database:

The initial state is: <{query(X)},{}>

Step 1

We try to solve query(X). Only rule R3 applies. The unification produces the generalized query ancestor({b},X). This generates temporary relations for query and ancestor with empty set values. Attempts at generating tuples for this generalized query fail.

The new state vector is:
<{query(X),ancestor(b,X)}, {ancestor={},query={}}>

Step 2

A new generalized query has been generated, so we go on. We try to evaluate each of the generalized queries: query(X) does not give anything new, so we try ancestor({b},X).
Using rule R2, and unifying, we get parent(b,X). This is a base relation, so we can produce a set of tuples. Thus we generate a value for ancestor which contains all the tuples of parent(b,X) and the new state vector is:
<{query(X),ancestor(b,X)}, {ancestor={(b,d),(b,e)},query={}}>

We now solve ancestor(b,X) using R1. Unification produces the expression : parent(b,Z),ancestor(Z,Y).

We try to generate new tuples from this expansion and the current ancestor value but get no tuples. We also generate new generalized queries by looking up parent

and instantiating Z. This produces the new expression:
parent(b,{d,e}),ancestor({d,e},Z).

This creates two new queries which are added to the generalized query and the new state is:
<{query(X),ancestor({b,d,e},X)}, {ancestor={(b,d),(b,e)},query={}}>

Step 3

New generalized queries and new tuples have been generated so we continue. We first solve query(X) using R3 and get the value {(b,d), (b,e)} for query. The resulting new state is:
<{query(X),ancestor({b,d,e},X)},
{ancestor={(b,d),(b,e)}, query={(b,d),(b,e)}}>

We now try to solve ancestor({b,d,e},X). Using R2, we get parent({b,d,e),X) which is a base relation and generates the following tuples in ancestor: {(b,d),(b,e),(d,f)}. This produces the new state:
<{query(X),ancestor({b,d,e},X)}, {ancestor={(b,d),(b,e),(d,f)},
query={(b,d),(b,e)}}>

We now solve ancestor({b,d,e},X)} using R1 and we get: p({b,d,e},Z),ancestor(Z,Y). We bind Z by going to the parent relation, and we get: p({b,d,e},{d,e,f}), ancestor({d,e,f},Y). This generates the new generalized query ancestor({d,e,f},Y) and the new state:
<{query(X),ancestor({b,d,e,f},X)}, {ancestor={(b,d),(b,e),(d,f)},
query={(b,d),(b,e)}}>

Step 4

A new generalized query has been generated, so we continue. Solving the ancestor queries using R2 will not produce any new tuples, and solving it with R3 will not produce any new generalized query nor any tuples. The algorithm terminates.

Concerning the performance of the method, one can note that (i) the set of potentially relevant facts is better than for Naive (in this example it is optimal), and (ii) QSQI has the same duplication problem as Naive evaluation: each step entirely duplicates the previous method.

Iterative Query/Subquery is presented in [Vieille 85 and 86]. To our knowledge it has not been implemented.

3.4. Recursive Query/Subquery or Extension Tables

Recursive Query/Subquery (QSQR) is a top-down interpreted recursive method.

The application domain is the set of range restricted rules without evaluable predicates.

It is of course a recursive version of the previous method. As before, we maintain temporary values of derived relations and a set of generalized queries. The state memorized by the algorithm is still a couple <Q,R>, where Q is a set of generalized queries and R is a set of derived relations together with their current values. However, besides this explicit state, the recursion mechanism stores at each level in the stack the tuples returned by the evaluation of the query, but this seems to have been solved reasonably in the existing implementation. The algorithm uses a selection function which, given a rule, can choose the first and the next derived predicate in the body to be ''solved''.

The recursive interpreter is as follows:

```
procedure evaluate(q)   (* q is a generalized query *)
begin
while ''new tuples are generated'' do
   for all rules whose head matches the generalized query do
     begin
     unify the rule with the generalized query; (propagate the constants)
     until there are no more derived predicates on the right do
       begin
       choose the first/next derived predicate according to the selection function;
       generate the corresponding generalized query;
       (This is done by replacing in the rule each base predicate by its value
       and each previously solved derived predicate by its current value).
       eliminate from that generalized query the queries that are already in Q;
       this produces a new generalized query q';
       add q' to Q;
```

```
      evaluate(q')
      end;
    replace each evaluated predicate by its value and evaluate the generalized query q;
    (This can be done in some order without waiting for all predicates to be evaluated.)
    add the results in R;
    return the results
    end
end.
Initial state is <{query(X)},{}>
evaluate(query(X)).
```

It is important to note that this version of QSQ is very similar to Prolog. It solves goals in a top-down fashion using recursion, and it considers the literals ordered in the rule (the order is defined by the selection function). The important differences with respect to Prolog are:

i) The method is set-at-a-time instead of tuple-at-a-time, through the generalized query concept, and

ii) As pointed out in [Dietrich and Warren 86], the method uses a dynamic programming approach of storing the intermediate results and re-using them when needed.

This dynamic programming feature also solves the problem of cycles in the facts. While Prolog will run in an infinite loop in the presence of such cycles, QSQR will detect them and stop the computation when no new tuple is generated. Thus, QSQR is complete over its application domain whereas Prolog is not.

Here is the ancestor example:

```
evaluate(query(X))
    use rule R3
    query(X) :- ancestor(b,X)
    this generates the query ancestor({b},X)
    new state is: <{ancestor({b},X), query(X)},{}>
    evaluate(ancestor({b},X)
      Step 1 of the iteration
```

```
use rule R1
ancestor({b},Y) :- parent({b},Z), ancestor(Z,Y).
by looking up parent we get the bindings {d,e} for Z.
  this generates the query ancestor({d,e},X)
new state is: <{ancestor({b,d,e},X), query(X)},{}>
evaluate (ancestor({d,e},X))
(this is a recursive call)
    Step 1.1
    use R1
    ancestor({d,e},Y) :- parent({d,e},Z),ancestor(Z,Y).
    by looking up parent we get the binding {f} for Z
    new state is: <{ancestor({b,d,e,f},X), query(X)},{}>
    evaluate(ancestor({f},X))
    (this is a recursive call)
        Step 1.1.1
        use R1
        ancestor({f},Y) :- parent({f},Z),ancestor(Z,Y).
        by looking up parent we get no binding for Z
        use R2
        ancestor({f},Y) :- parent({f},Y)
        this fails to return any tuple
        end of evaluate(ancestor({f},X))
        Step 1.1.2
        nothing new is produced
        end of evaluate(ancestor({f},Y))
     use R2
     ancestor({d,e},Y) :- parent({d,e},Y)
     this returns the tuple ancestor(d,f)
     new state is: <{ancestor({b,d,e,f},X),
     query(X)}, {ancestor={(d,f)}}>
     Step 1.2
     same as Step 1, nothing new produced
```

```
        *end of* **evaluate** (ancestor({d,e},X))
      use rule R2
      ancestor({b},X) :- parent({b},Y)
      returns the tuples ancestor(b,d) and ancestor(b,e)
      new state is: <{ancestor({b,d,e,f},X), query(X)},
                    {ancestor={(d,f), (b,d),(b,e)}}>
      *Step* 2
      nothing new produced
      *end of* **evaluate**({b},X)
    generate tuples from R3
    new state is: <{ancestor({b,d,e,f},X), query(X)},{ancestor={(d,f),
                    (b,d),(b,e)},query=(d,f), (b,d),(b,e)}}>
    *end of* **evaluate**(query(X))
```

Recursive Query/Subquery is described in [Vieille 85 and 86]. A compiled version has been implemented on top of the INGRES relational system [Vieille 86]. In [Dietrich and Warren 86], along with a good survey of some of these methods, a method called ‘‘extension tables’’ is presented. It is, up to a few details, the same method.

3.5. Henschen-Naqvi

Henschen-Naqvi is a top-down, compiled and iterative method.

The application domain is that of linear range restricted rules.

The method has a compilation phase which generates an iterative program. That iterative program is then run against the data base. The general method is fairly complex to understand, and we shall restrict ourselves to describing it in the ‘‘typical case’’ which is:

```
p(X,Y) :- up(X,XU),p(XU,YU),down(YU,Y).
p(X,Y) :- flat(X,Y).
query(X) :- p(a,X).
```

Note that the relation names *up* and *down* are not to be confused with the notions ‘‘top-down’’ or ‘‘bottom-up’’, which are characteristics of evaluation methods.

Let us introduce some simple notation, which will make reading the algorithm much simpler. Since we are only dealing with binary relations, we can view these as set-to-set mappings. Thus, the relation r maps each set A to a set A.r, where:

$$A.r = \{ y \mid r(x,y) \text{ and } x \in A \}$$

Clearly,

$$A.(r.s) = (A.r).s$$

This approach is similar to the formalism described in [Gardarin and Maindreville 86]. We shall denote the composition of relation r n times with itself r^n. Finally we shall denote set union by '+'. It is now easy to see that the answer to the query is

$$\{a\}.\text{flat} + \{a\}.\text{up}.\text{flat}.\text{down} + \ \ldots\ + \{a\}.\text{up}^n.\text{flat}.\text{down}^n + \ldots$$

The state memorized by the algorithm is a couple <V,E>, where V is a the value of a unary relation and E is an expression. At each step, using V and E, we compute some new tuples and compute the new values of V and E.

The iterative program is as follows:

```
V :={a};
E := λ;        /* the empty string */
while ``new tuples are generated in V'' do
   begin
   /* produce some answer tuples */
   answer := answer + V.flat.E;
   /* compute the new value */
   V := V.up ;
   /* compute the new expression */
   E := E | .down;
  end.
```

Note that E is an *expression*, and is augmented each time around the loop by concatenating ``.down'' to it through the ``cons'' operator. As can be seen from this program, at step i, the value V represents $\{a\}.\text{up}^i$ and the expression E represents down^i. Therefore the produced tuples are:

{a}.up^i.flat.$down^i$.

This is not meant to be a complete description of the method, but a description of its behavior in the typical case.

The Henschen-Naqvi method is described in [Henschen and Naqvi 84]. The method has been implemented in the case described here. This implementation can be found in [Laskowski 84]. An equivalent method is described using a different formalism in [Gardarin and Maindreville 86]. The performance of the method is compared to Semi-Naive evaluation in [Han and Lu 86].

3.6. Prolog

Prolog is a top-down, interpreted and recursive method.

The application domain of Prolog is difficult to state precisely. It is data dependent in the sense that the facts have to be acyclic for the interpreter to terminate, and there is no simple syntactic characterization of a terminating Prolog program. The job of characterizing the ''good'' rules is left to the programmer.

We consider its execution model to be well known and will not describe it here. In fact, Prolog is a programming language and not a general method to evaluate Horn clauses. We essentially mention Prolog for the sake of completeness and because it is interesting to compare its performance with the other methods.

3.7. APEX

APEX is a method which is difficult to categorize. It is partly compiled in the sense that a graph similar to the predicate connection graph is produced from the rules, which takes care of some of the preprocessing needed for interpretation. It is not fully compiled in the sense that the program which runs against the database is still unique (but driven by the graph). It is, however, clearly recursive, because the interpreter program is recursive. Finally, it is partly top-down and partly bottom-up as will be seen in the interpreter.

The application domain of APEX is the set of range restricted rules which contain no constants and no evaluable predicates.

The interpreter takes the form of a recursive procedure, which, given a query, produces a set of tuples for this query. It is as follows:

```
procedure solve(q, answer)
begin
answer := {};
if query q is on a base relation
then evaluate q against the date base
else
  begin
   select the relevant facts for q in the base predicates;
   put them in relevant;
   while new tuples are generated do
      begin
      for each rule do (this can be done in parallel)
         begin
         instantiate the right predicates with the relevant facts
         and produce tuples for the left predicate;
         add these tuples to the set of relevant facts;
         initialize the set of useful facts to the set of relevant facts;
         for each literal on the right do (this can be done in parallel)
            begin
            for each matching relevant fact do
               begin
               plug the fact in the rule and propagate the constants;
               this generates a new rule and a new set of queries;
               for all these new queries q' do
                  begin
                  solve(q',answer(q')) (this is the recursion step)
                  add answer(q') to the useful facts
                  end
               end
            instantiate the right predicates with the useful facts;
            produce tuples for the left predicate;
```

```
                add these to the relevant facts;
                extract the answer to q from the relevant facts
                **end**
            **end**
        **end**
    **end**.
**end**;
solve(query(X),answer).
```

Let us now run this program against our ancestor example. We cannot have a constant in the rules and we must modify our rule set and solve directly the query ancestor(b,X):

```
solve (ancestor(b,X), answer)
we first select the relevant base facts.  relevant = {parent(b,d),parent(b,e)};
we now start the main iteration:
Step 1
rule R1
``ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y)''
    we cannot produce any new tuple from this rule because ancestor
    does not yet have any relevant fact
    useful = {parent(b,d),parent(b,e)};
    process parent(X,Z)
        use parent(b,d)
            the new rule is ``parent(b,d),ancestor(d,Y)''
            solve(ancestor(d,Y),answer1)
            ... (this call is not described)
            this returns {ancestor(d,f)}, which we add to useful
            useful = {parent(b,d),parent(b,e),ancestor(d,f)};
        use parent(b,e)
            the new rule is ``parent(b,e),ancestor(e,Y)''
            solve(ancestor(e,Y),answer2)
            ... (this call is not described)
            this returns nothing
```

process ancestor(Z,Y)
we instantiate parent and ancestor with the useful facts.
this produces ancestor(b,f) which we add to the relevant facts:
relevant = {parent(b,d),parent(b,e), ancestor(b,f)};

rule R2 ''ancestor(X,Y) :- parent(X,Y)''
using the relevant facts we produce {ancestor(b,d),ancestor(b,e)}
we add these to relevant:
relevant = {parent(b,d),parent(b,e), ancestor(b,d), ancestor(b,e), ancestor(b,f)};
this rule does not produce any subquery

Step 2
will not produce anything new, and so the algorithm stops.

The APEX method is described in [Lozinskii 85 and 85a]. The method has been implemented.

3.8. Aho-Ullman

Aho and Ullman [Aho and Ullman 79] present an algorithm for optimizing recursive queries by commuting selections with the *least fixpoint operator* (LFP). The operator is applied to a recursive definition of the form r = f(R), and it denotes the computation of the least fixpoint of this definition, that is, the least value of r such that this equation holds. The input to the Aho-Ullman algorithm is an expression

$$\sigma_F(LFP(r=f(r))$$

where f(r) is a relational algebra expression not containing set difference and with at most one occurrence of r. The output is an equivalent expression where the selection has been pushed through as far as possible.

We introduce their notation and ideas through an example. Consider:

```
a(X,Y) :- a(X,Z), p(Z,Y).
a(X,Y) :- p(X,Y).
q(X) :- a(john,X).
```

Aho-Ullman write this as:

$\sigma_{a_1=john}(LFP(a = a.p \cup p))$

In this definition, *a* is a relation which is defined by a *fixpoint* equation in relational algebra, and *p* is a base relation. If we start with *a* empty and repeatedly compute *a* using the rule $a = a.p \cup p$, at some iteration, there is no change (since the relation *p* is finite). Because the f does not contain set difference, it is *monotonic*, that is, if relation r_1 contains r_2, then $f(r_1)$ contains $f(r_2)$. By a result of [Tarski 55], the relation *a* that we compute by the above method (repeated use of the rule a = a.p) is the *least fixpoint* of the rule. It is the smallest relation *a* which satisfies the equation, that is, contains every tuple which can be generated by using the fixpoint rule, and no tuple which cannot. The query is simply the selection a_1=john applied to this relation. Thus, the query is a selection applied to the transitive closure of p.

We now describe how the Aho-Ullman algorithm optimizes the above query. We use '.' to denote composition, which is a join followed by projecting out the join attributes. We begin with the expression

$\sigma_{a_1=john}(a)$

and by replacing a by f(a) we generate

$\sigma_{a_1=john}(a.p \cup p))$

By distributing the selection across the join, we get

$\sigma_{a_1=john}(a.p) \cup \sigma_{a_1=john}(p)$.

Since the selection in the first subexpression only involves the first attribute of a, we can rewrite it as

$\sigma_{a_1=john}(a) . p$

We observe that this contains the subexpression

$\sigma_{a_1=john}(a)$

which was the first expression in the series. If we denote this by E, the desired optimized expression is then

$LFP(E = E.p \cup \sigma_{a_1=john}(p))$

This is equivalent to the Horn Clause query:

```
a(john,Y) :- a(john,Z), p(Z,Y).
a(john,Y) :- p(john,Y).
q(X) :- a(john,X).
```

The essence of the method is to construct a series of equivalent expressions starting with the expression $\sigma_F(r)$ and repeatedly replacing the single occurrence of r by the expression f(r). Note that each of these expressions contains just one occurrence of R. In each of these expressions, we push the selection as far inside as possible. Selection distributes across union, commutes with another selection and can be pushed ahead of a projection. However, it distributes across a Cartesian product $Y \times Z$ only if the selection applies to components from just one of the two arguments Y and Z. The algorithm fails to commute the selection with the LFP operator if the (single) occurrence of r is in one of the arguments of a Cartesian product across which we cannot distribute the selection. We stop when this happens or when we find an expression of the form $h(g(\sigma_F(r)))$ and one of the previous expressions in the series is of the form $h(\sigma_F(r))$. In the latter case, the equivalent expression that we are looking for is h(LFP(s=g(s))), and we have succeeded in pushing the selection ahead of the LFP operator.

We note in conclusion that the expression f(r) must contain no more than one occurrence of *r*. For instance, the algorithm does not apply in this case:

$\sigma_{a_1=john}(LFP(a = a.p \cup p))$

Aho and Ullman also present a similar method for commuting projections with the LFP operator, but we do not discuss it here.

3.9. Kifer-Lozinskii

The Kifer-Lozinskii algorithm is an extension of the Aho-Ullman algorithm described above. However, rules are represented as rule/goal graphs rather than as relational algebra expressions, and the method is described in terms of *filters* which are applied to the arcs of the graph. It is convenient to think of the data as flowing through the graph along the arcs. A *filter* on an arc is a selection which can be applied to the tuples flowing through that arc, and is used to reduce the number of tuples that are generated. Transforming a given rule/goal graph into an equivalent graph with (additional) filters on some arcs is equivalent to rewriting the corresponding set of rules.

The execution of a query starts with the nodes corresponding to the base relations sending all their tuples through all arcs that leave them. Each axiom node that receives tuples generates tuples for its head predicate and passes them on through all its outgoing arcs. A relation node saves all new tuples that it receives and passes them on through its outgoing arcs. Computation stops (with the set of tuples in the query node as the answer) when there is no more change in the tuples stored at the various nodes at some iteration. We note that this is just Semi-Naive evaluation.

Given filters on all the arcs leaving a node, we can 'push' them through the node as follows. If the node is a relation node, we simply place the disjunction of the filters on each incoming arc. If the node is an axiom node, we place on each incoming arc the strongest consequence of the disjunction that can be expressed purely in terms of the variables of the literal corresponding to this arc.

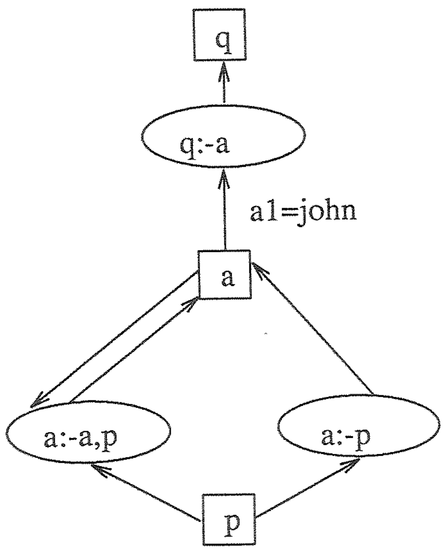
The objective of the optimization algorithm is to place the "strongest" possible filters on each arc. Starting with the filter which represents the constant in the query, it repeatedly pushes filters through the nodes at which the corresponding arcs are incident. Since the number of possible filters is finite, this algorithm terminates. It stops when further pushing of filters does not change the graph, and the graph at this point is equivalent to the original graph (although the graph at intermediate steps may not). Note that since the disjunction of 'true' with any predicate is 'true', if any arc in a loop is assigned the filter 'true', all arcs in the loop are subsequently assigned the filter 'true'.

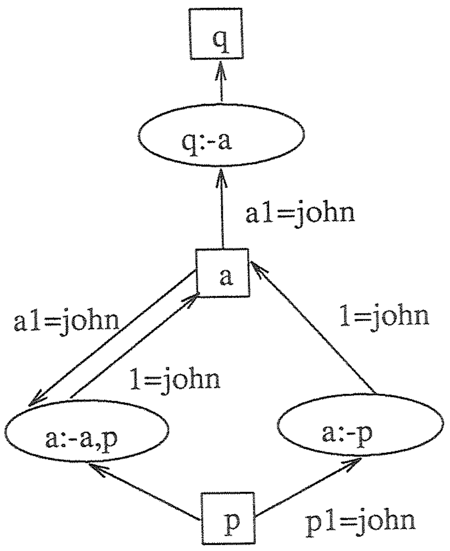
Consider the transitive closure example that we optimized using the Aho-Ullman algorithm. We would represent it by the following axioms:

R1 a(X,Y) :- a(X,Z), p(Z,Y).
R2 a(X,Y) :- p(X,Y).
R3 q(X) :- a(john,X).

Given below is the corresponding system graph, before and after optimization (We have omitted the variables in the axioms for clarity):

Before: *After*:





We begin the optimization by pushing the selection through the relation node a. Thus the arcs from R1 to *a* and from R2 to *a* both get the filter '1=john' (We have simplified the conventions for keeping track of variables - '1' refers to the first attribute of the corresponding head predicate). We then push these filters through the corresponding axiom nodes, R1 and R2. Pushing '1=john' through node R2 puts the filter 'p_1=john' on the arc from *p* to R2. Pushing '1=john' through node R1 puts the filter 'a_1=john' on the arc from *a* to R1. Note that it does not put anything on the arc from *p* to R1 (empty filters are equivalent to 'true'). There are no arcs entering *p*, and the filter on the arc from *a* to R1 does not change the disjunction of the filters on arcs leaving *a* (which is still 'a_1=john'). So the algorithm terminates here.

The analogy with the Aho-Ullman algorithm is easily seen when we recognize that a filter is a selection, pushing through a relation node is distribution across a $\cup$ and pushing through an axiom node is distribution across a Cartesian product. In general, the optimizations achieved by the two algorithms are identical. However, the Kifer-Lozinskii algorithm is more general in that it successfully optimizes some expressions containing more than one occurrence of the defined predicate. An example is the expression

$$\sigma_{a_1=john}(LFP(a = (a.p \cup a.q \cup p)))$$

The Aho-Ullman algorithm does not apply in this case because there are two occurrences of R in f(R). The Kifer-Lozinskii algorithm optimizes this to

$$LFP((\sigma_{a_1=john}(a) \, . \, p) \cup (\sigma_{a_1=john}(a) \, . \, q) \cup (\sigma_{a_1=john}(p)))$$

Essentially, it improves upon the Aho-Ullman algorithm in that it is able to distribute selection across some unions where both arguments contain r.

Further, the algorithm can work directly upon certain mutually recursive rules, for example

```
R1   r(X,Y) :- b(X), s(X,Y).
R2   s(X,Y) :- c(X), r(X,Y).
R3   q(X) :- r(X,john).
```

Before applying the Aho-Ullman algorithm, these rules must be rewritten as:

```
R1   r(X,Y) :- b(X), c(X), r(X,Y).
R2   q(X) :- r(X,john).
```

Note that the Kifer-Lozinskii algorithm fails to optimize both

$\sigma_{a_1=john}(LFP(a = a.a \cup p))$, and

$\sigma_{a_1=john}(LFP(a = a.p \cup p.a \cup p))$

In this description, we have treated only the ''static'' filtering approach of Kifer and Lozinskii [Kifer and Lozinskii 86]. In [Kifer and Lozinskii 86a], they have also proposed ''dynamic'' filters, which are not determined at compile time but are computed at run time, and this approach is similar to Generalized Magic Sets, discussed later in this section.

3.10. Magic Sets

The idea of the Magic Sets optimization is to simulate the sideways passing of bindings a la Prolog by the introduction of new rules. This cuts down on the number of potentially relevant facts.

The application domain is the set of bottom-up evaluable rules.

We shall describe the method in detail, using as an example a modified version of the same-generation rule set:

```
sg(X,Y) :- p(X,XP),p(Y,YP),sg(YP,XP).
sg(X,X).
query(X) :- sg(a,X).
```

Note that in this version the two variables XP and YP have been permuted. Note also that the second rule is not range restricted. The first step of the magic set transformation is the generation of adorned rules.

Given a system of rules, to obtain the *adorned rule system* [Ullman 85]:
For each rule r and for each adornment *a* of the predicate on the left, generate an adorned rule. Define recursively an argument of a predicate in the rule r to be *distinguished* [Henschen and Naqvi 84] if either it is bound in the adornment *a*, or it is a constant, or it appears in a base predicate occurrence that has a distinguished variable. Thus, the sources of bindings are (i) the constants and (ii) the bindings in the head of the rule. These bindings are propagated through the base predicates. If we consider each distinguished argument to be bound, this defines an adornment for each derived literal on the right. The adorned rule is obtained by replacing each derived literal by its adorned version.

If we consider the rule

```
sg(X,Y) :- p(X,XP),p(Y,YP),sg(YP,XP).
```

with adornment bf on the head predicate, then X is distinguished because it is bound in sg(X,Y), XP is distinguished because X is distinguished and p(X,XP) is a base predicate and these are the only distinguished variables. Thus the new adorned rule is

sg^{bf}(X,Y) :- p(X,XP),p(Y,YP),sg^{fb}(YP,XP).

If we consider a set of rules, this process generates a set of adorned rules. The set of adorned rules has size K.R where R is the size of the original set of rules and K is a factor exponential in the number of attributes per derived predicate. So, for instance, if every predicate has three attributes, then the adorned system is eight times larger than the original system. However, we do not need the entire adorned system and we only keep the adorned rules which derive the query. In our example the reachable adorned system is:

sg^{bf}(X,Y) :- p(X,XP),p(Y,YP),sg^{fb}(YP,XP).
sg^{fb}(X,Y) :- p(X,XP),p(Y,YP),sg^{bf}(YP,XP).
sg^{bf}(X,X).
sg^{fb}(X,X).
$query^{f}$(X) :- sg^{bf}(a,X).

Clearly, this new set of rules is equivalent to the original set in the sense that it will generate the same answer to the query.

The magic set optimization consists of generating from the given set of rules a new set of rules, which are equivalent to the original set with respect to the query, and such that their bottom-up evaluation is potentially more efficient. This transformation is done as follows:

i) For each occurrence of a derived predicate on the right of an adorned rule, we generate a magic rule.

ii) For each adorned rule we generate a modified rule.

Here is how the magic rules are generated:

i) Choose an adorned literal predicate p on the right of the adorned rule r,

ii) Erase all the other derived literals on the right,

iii) In the derived predicate occurrence replace the name of the predicate by magic.p^{a} where a is the literal adornment, and erase the non distinguished variables,

iv) Erase all the non distinguished base predicates,

v) In the left hand side, erase all the non distinguished variables and replace the name of the predicate by *magic.p* $1^{a'}$, where p1 is the predicate on the left, and a' is the adornment of the predicate p1, and finally

vi) Exchange the two magic predicates.

For instance the adorned rule:

sg^{bf}(X,Y) :- p(X,XP),p(Y,YP),sg^{fb}(YP,XP).

generates the magic rule:

$magic^{fb}$(XP) :- p(X,XP), $magic^{bf}$(X).

Note that the magic rules simulate the passing of bound arguments through backward chaining. (We have dropped the suffix "sg" in naming the magic predicates since it is clear from the context.)

Here is how we generate the modified rule: For each rule whose head is p.a, add on the right hand side the predicate magic.p.a(X) where X is the list of distinguished variables in that occurrence of p. For instance the adorned rule:

sg^{bf}(X,Y) :- p(X,XP),p(Y,YP),sg^{fb}(YP,XP).

generates the modified rule:

sg^{bf}(X,Y) :- p(X,XP),p(Y,YP),$magic^{bf}$(X), sg^{fb}(YP,XP).

Finally the complete modified set of rules for our example is:

$magic^{fb}$(XP) :- p(X,XP), $magic^{bf}$(X).
$magic^{bf}$(YP) :- p(Y,YP),$magic^{fb}$(Y).
$magic^{bf}$(a).
sg^{bf}(X,Y) :- p(X,XP),p(Y,YP),$magic^{bf}$(X),sg^{fb}(YP,XP).
sg^{fb}(X,Y) :- p(X,XP),p(Y,YP),$magic^{fb}$(Y),sg^{bf}(YP,XP).
sg^{bf}(X,X) :- $magic^{bf}$(X).
sg^{fb}(X,X) :- $magic^{bf}$(X).
query.f(X) :- sg^{bf}(a,X).

The idea of the magic set method was presented in [Bancilhon et al 86] and the precise algorithm is described in [Bancilhon et al 86a]. To our knowledge, the method is not implemented.

3.11. Counting and Reverse Counting.

Counting and Reverse Counting are derived from the magic set method.

They apply under two conditions:

i) The data is acyclic, and

ii) There is at most one recursive rule for each predicate, and it is linear.

We first describe counting using the ''typical'' single linear rule system:

```
p(X,Y) :- flat(X,Y).
p(X,Y) :- up(X,XU),p(XU,YU),down(YU,Y).
query(Y) :- p(a,Y).
```

The idea consists in introducing magic sets (called *counting* sets) in which elements are numbered by their distance to the element a. Remember that the magic set essentially marks all the *up* ancestors of a and then applies the rules in a bottom-up fashion to only the marked ancestors. In the counting method, at the same time we mark the ancestors of john, we number them by their distance from a. Then we can ''augment'' the p predicate by numbering its tuples and generate them by levels as follows:

```
counting(a,0).
counting(X,I) :-counting(Y,J),up(Y,X),I=J+1.
p'(X,Y,I) :- counting(X,I),flat(X,Y).
p'(X,Y,I) :- counting(X,I),up(X,XU), p'(XU,YU,J),down(YU,Y),I=J-1.
query(X) :- p'(a,X,0).
```

Thus at each step, instead of using the entire magic set, we only use the tuples of the correct level, thus minimizing the set of relevant tuples. But in fact, it is useless to compute the first attribute of the p predicate. Thus the system can be further optimized into:

```
counting(a,0).
counting(X,I) :-counting(Y,J),up(Y,X),I=J+1.
p''(Y,I) :- counting(X,I),flat(X,Y).
p''(Y,I) :- p''(YU,J),down(YU,Y),I=J-1,J>0.
query(X) :- p''(Y,0).
```

It is interesting to notice that this new set of rules is in fact simulating a stack.

Reverse counting is another variation around the same idea. It works as follows:

i) First compute the magic set, then

ii) For each element b in the magic set number all its *down* descendants and its *up* descendants and add to the answer all the *down* descendants having same number as a (because a is in the *up* descendants).

This gives the following equivalent system:

```
magic(a).
magic(Y) :- magic(X),up(X,Y).
des.up(X,X,0) :- magic(X).
des.down(X',Y,0) :- magic(X'),flat(X',Y).
des.up(X',X,I) :- des.up(X',Y,J),up(X,Y),I=J+1.
des.down(X',X,I) :- des.down(X',Y,J),down(Y,X),I=J+1.
query(Y) :- des.up(X',a,Y),des.down(X',Y,I).
```

This can be slightly optimized by limiting ourselves to the b's which will join with *flat* and restricting the *down* des's to be in the magic set. This generates the following system:

```
magic(a).
magic(Y) :- magic(X),up(X,Y).
des.up(X,X,0) :- magic(X),flat(X,Y).
des.down(X',Y,0) :- magic(X'),flat(X',Y).
des.up(X',X,I) :- magic(X),des.up(X',Y,J),up(X,Y),I=J+1.
des.down(X',X,I) :- des.down(X',Y,J),down(Y,X),I=J+1.
sg(a,Y) :- des.up(X',a,Y),des.down(X',Y,I).
```

Note that we still have the problem of a ''late termination'' on *down* because we number *all* the descendants in *down*, even those of a lower generation than a.

The idea of counting was presented in [Bancilhon et al 86] and a formal description of counting and of an extension (not covered here) called ''magic counting'' was presented in the single rule case in [Sacca and Zaniolo 86]. An extension to the fully general case of Horn Clauses with function symbols is described in [Sacca and Zaniolo 86a]. Reverse counting is described in [Bancilhon et al. 86]. They have not been implemented.

3.12. Generalized Magic Sets

This is a generalization of the Magic Sets method and is described in [Beeri and Ramakrishnan 87]. The intuition is that the Magic Sets method works essentially by passing bindings obtained by solving body predicates ''sideways'' in the rule to restrict the computation of other body predicates. The notion of *sideways information passing* is formalized in terms of labeled graphs. A sideways information passing graph is associated with each rule, and these graphs are used to define the Magic Sets transformation. (In general, many such graphs exist for each rule, each reflecting one way of solving the predicates in the body of the rule; and we may choose any one of these to associate with the rule.)

There are examples, such as transitive closure defined using double recursion, in which the original Magic Sets transformation achieves no improvement over Semi-Naive evaluation. Intuitively, this is because the only form of sideways information passing that it implements consists of using base predicates to bind variables. Thus, in the same generation example discussed earlier, the predicate p is used to bind the variable XP. The method, however, fails to pass information through derived predicates, and so it fails with transitive closure expressed using double recursion (since the recursive rule contains no base predicates in the body). The generalized version of the method succeeds in passing information through derived predicates as well.

As with the original Magic Sets method, a set of *adorned rules* is first obtained from the given rules, and these adorned rules are then used to produce the optimized set of rules. Both these steps are now directed, however, by the notion of sideways information passing graphs. We describe this method in Chapters 5, 6 and 7.

The ''Alexandre'' method described in [Rohmer and Lescoeur 85] is essentially a variant of the Generalized Magic Sets method.

The dynamic filtering approach of Kifer and Lozinskii is similar to the Generalized Magic Sets method, although it cannot implement some sideways information passing graphs. The dynamic filters essentially perform as magic sets, but this is a run-time method, and the overhead of computing and applying the filters falls outside our framework. We do not discuss dynamic filtering, and it is understood that any comments about the Kifer-Lozinksii method refer to the static version.

4. Summary of Method Characteristics.

A summary of the characteristics of each method is presented in Table 1.

5. Conclusions

In this chapter, we have given a description of the major methods for processing logic queries.

We have tried to identify the exact application domain for each method. We have also tried to describe the methods in a uniform manner. Unfortunately, we have only been partially successful at that. We have identified a set of major characteristics of the methods: method vs. rewriting method, top-down vs. bottom-up, recursive vs. iterative and compiled vs. interpreted. But some of these characteristics are somewhat arbitrary: for the same method it is sometimes possible to have a compiled or interpreted version. For instance, we have presented a compiled version of Naive evaluation, while SNIP is an interpreted version of it. It seems also reasonable to design a compiled version of iterative QSQ. We also argued that the distinction between rewriting method and method was mainly of pedagogical interest. However, the top-down vs. bottom-up and recursive vs. iterative distinctions seem to capture intrinsic properties of the methods.

Table 1: Summary of Method Characteristics

Method	*Application Range*	*Top down vs. Bottom Up*	*Compiled vs. Interpreted*	*Iterative vs. Recursive*
Naive Evaluation	Bottom-up Evaluable	Bottom Up	Compiled	Iterative
Semi-Naive Evaluation	Bottom-up Evaluable	Bottom Up	Compiled	Iterative
Query/Subquery	Range Restricted No Arithmetic	Top Down	Interpreted	Iterative
Query/Subquery	Range Restricted No Arithmetic	Top Down	Interpreted	Recursive
APEX	Range Restricted No Arithmetic Constant Free	Mixed	Mixed	Recursive
Prolog	User responsible	Top Down	Interpreted	Recursive
Henschen-Naqvi	Linear	Top Down	Compiled	Iterative
Aho-Ullman	Strongly Linear	Bottom Up	Compiled	Iterative
Kifer-Lozinskii	Range Restricted No Arithmetic	Bottom Up	Compiled	Iterative
Counting	Strongly Linear	Bottom Up	Compiled	Iterative
Magic Sets	Bottom-up evaluable	Bottom Up	Compiled	Iterative

Chapter 4 - Sideways Information Passing

In this chapter, we discuss and formally define the notion of *sideways information passing*, which is a component of any query evaluation method. We suggest that this component is largely independent of the *control* strategy associated with the method. This leads to a separation of concerns that has important consequences, and the work presented in this and the subsequent three chapters is a step towards understanding and taking advantage of them. †

1. Introduction

Consider the following program:

$$anc(X, Y) :- par(X, Y)$$
$$anc(X, Y) :- par(X, Z), anc(Z, Y)$$

and let the query be

$$anc(john, Y) ?$$

Assume that a database contains a relation *par*. Then the program defines a derived relation describing ancestors, and the query asks for the ancestors of *john*. Consider Semi-Naive evaluation. While the strategy is reasonably efficient when the query does not contain instantiated variables, this example clearly shows that it is very inefficient when bindings for some variables are given in the query. The reason is that it computes the complete *anc* relation and then applies selection to it. Thus, all ancestors are computed, even though only the ancestors of john are needed. A *top−down* strategy (as used, for example, by Prolog), may do much better by computing only the ancestors of john. The first rule computes the nodes reachable from *john* in one step. Then the second rule generates the same query for these nodes, and the first rule is used again to find the nodes reachable in two steps, and so on. At each step, the computation is restricted to nodes identified at previous steps. This is achieved by using bindings generated earlier to restrict subgoals, that is, by passing binding information across subgoal invocations.

† The work described in Chapters 4, 5 and 6 is from [Beeri and Ramakrishnan 87].

There are two modes of *information passing* in the evaluation of a query. The first is unification. Given a constant in a goal, unification with a rule head will cause some of the variables in the head to be bound to that constant. These bindings are valid in the rule's body as well. (This is normally seen as part of top-down evaluation.) The second mode is *sideways information passing*. Given bindings for some variables of a predicate, we can solve the predicate with these bindings and thus obtain bindings for some of its other variables. These new bindings can be "passed" to another predicate in the same rule to restrict the computation for that predicate. In the example above, in the top-down evaluation, unification with the query binds X in the second rule to *john*; these bindings are passed from the rule's head to the base predicate *par*. The values obtained by evaluating *par* with these bindings are then passed to *anc*, to generate yet another subgoal.

Several strategies have been proposed for evaluating recursive queries expressed using sets of Horn Clauses (rules), as was seen in the previous chapter. The main thrust of these strategies is to improve efficiency by restricting the computation to tuples that are related to the query. They all use information passing in some form, but they also use other ideas, intended, for example, to avoid repeatedly computing the same fact, using the same derivation, several times. However, so far there is no uniform framework in terms of which these strategies may be described and compared, and the basic ideas that are common to these strategies remain unclear.

It is our thesis that each of these strategies has two distinct components - a *sideways information passing strategy* (henceforth abbreviated to *sip*) for each rule (or even several such strategies per rule) and a control strategy. A sip represents a decision on how information gained about tuples in some predicates in a rule is to be used in evaluating other predicates in the rule. The control strategy implements this decision, possibly using additional optimization techniques. Thus, a given sip collection may be implemented by several control strategies, and a given control strategy may be used to implement distinct sip collections. In particular, we show that simple bottom-up evaluation may be used to implement a wide class of sip collections by first rewriting the given set of Horn Clauses and then evaluating the rewritten set.

To what extent is it justified to claim that the collection of sips and the control strategy are distinct, possibly independent, components of a query evaluation strategy? The research reported here can be seen as a step towards answering this question. We provide a formal definition of a sip. Then (in Chapters 5 and 7) we present several program transformation strategies that can be applied to an arbitrary (Horn Clause) program and a query to produce a program that is equivalent to the given program with respect to the query, and which uses the bindings in the query to direct the computation, and hence is usually more efficient. Each of these transformations uses a collection of sips that are attached to the rules of the given program. In a sense, the sip collection serves as a definition of what facts are *relevant* to the given query. The transformation produces a program that computes only these facts. The fact that the transformed programs use information passing, yet can be computed bottom-up, shows that there is no inherent relationship of information passing to top-down evaluation, thus supporting the claim that sips and control are independent. The transformations are generalizations of strategies that have been proposed in earlier work, namely the *Magic Sets* and the *Counting* strategies [Bancilhon et al. 86], but which, as presented there, were of quite limited applicability. We note that the notion of a sideway information passing graph has been introduced previously, although it is less general than our definition ([Kifer and Lozinskii 86]). The work reported in [Van Gelder 86a] can be viewed as supporting our claim that information passing and control are to a large extent independent, although this claim is not explicit there.

2. Sideways Information Passing

A *sideways information passing strategy*, henceforth referred to as a *sip*, is an inherent component of any query evaluation strategy. Informally, for a rule of a program, a sip represents a decision about the order in which the predicates of the rule will be evaluated, and how values for variables are passed from predicates to other predicates during evaluation. Sip strategies for the various rules of a program are not enough to specify an evaluation strategy. A control component that specifies, for example, whether to obtain all solutions for a goal when it is first called, or whether to obtain them one at a time (e.g., Prolog) is required. Control is a separate, often independent, component; here, we only discuss sips.

Intuitively, a sip describes how bindings passed to a rule's head by unification are used to evaluate the predicates in the rule's body. Thus, a sip describes how we evaluate a rule when a given set of head arguments are bound to constants. Consider, for example, the program presented in Section 1.

$$anc(X, Y) :- par(X, Y)$$
$$anc(X, Y) :- par(X, Z), anc(Z, Y)$$
$$anc(john, Y)\ ?$$

In the query, the first argument is bound to *john*, and by unification, the variable X in the second rule is bound to *john*. We can evaluate *par* using this binding, and obtain a set of bindings for Z. These are passed to *anc* to generate subgoals, which in this case have the same binding pattern. Generalizing from this example, we may say that the basic step of sideways information passing is the evaluation of a set of predicates (possibly with some arguments bound to constants), and using the results to bind variables appearing in another predicate. This leads to the definition of a sip as a labeled graph, below.

Let r be a rule, with head predicate $p(\theta)$, and let p_h be a special predicate, denoting the head predicate restricted to its bound arguments. (If no bindings are given, then p_h is a 0-ary predicate, the constant *false*. In such a case, we may consider it not to exist at all, as far as the following discussion is concerned.) If a predicate appears in r's body more then once, we number its occurrences, starting from 0. (The numbering is just to identify the positions in the rule. It is irrelevant when unification with heads of other rules is considered.) Let $P(r)$ denote the set of predicate occurrences in the body. A sip for r is a labeled graph that satisfies the following conditions:

1. Each node is either a subset or a member of $P(r) \cup \{p_h\}$.
2. Each arc is of the form $N \rightarrow q$, with label χ, where N is a subset of $P(r) \cup \{p_h\}$, q is a member of $P(r)$, and χ is a set of variables, such that
   (i) Each variable of χ appears in N.
   (ii) Each member of N is connected to a variable in χ.

(iii) For some argument of q, all its variables appear in χ. Further, each variable of χ appears in an argument of q that satisfies this condition.

These two conditions define the nature of nodes and arcs of a sip. They are explained below. The following condition provides a consistency restriction on a sip. For a graph with nodes and arcs as above, define a precedence relation on the members of $P(r) \cup \{p_h\}$ as follows:

(i) p_h precedes all members of $P(r)$.

(ii) A predicate that does not appear in the graph, follows every predicate that appears in it.

(iii) If $N \rightarrow q$ is an arc, and $q' \in N$, then q' precedes q.

We can now state the last condition defining a sip:

3. The precedence relation defined by the sip is acyclic, that is, its transitive closure is a partial order.

Consider the program discussed earlier:

$$anc(X, Y) :- par(X, Y)$$
$$anc(X, Y) :- par(X, Z), anc(Z, Y)$$
$$anc(john, Y)\ ?$$

In solving the second rule, if X is bound to *john* by unification, and *par* is solved with this binding to generate a binding for Z (thus creating another query on *anc*, which is solved similarly), we can represent this strategy by the following sip:

$$\{anc_h\} \rightarrow_X par$$
$$\{anc_h, par\} \rightarrow_Z anc$$

We explain the meaning of such a graph by first explaining how the computation of a rule uses an arc, and then discussing the complete computation of the rule. Assume we want to use the rule r, with some arguments of the head predicate bound to constants. The special node p_h may be thought of as a base relation whose attributes are the variables appearing in bound arguments of the head predicate. An arc labeled χ from a set of predicates N to a predicate q means that by evaluating the join of the predicates in N (with some arguments possibly bound to constants), values for the variables in χ are obtained, and these values are passed to the predicate q, and used to restrict its computation. Thus, for each such arc, the variables in its label must be bound when the goals corresponding to the predicates in N are solved, and any control strategy which implements the sip must ensure this.

As stated above, we separate the issue of control from the sip. Thus, we allow predicates to be computed (for given bindings) all at once, or in stages. We can imagine a box associated with each predicate in which its tuples are collected. For an arc $N \rightarrow q$, with label χ, attach a filter that performs a join of the tuples in the boxes of the predicates of N, and for each qualifying tuple, its projection on χ is sent along the arc. (Note that if arguments are complex terms with function symbols, then the arguments are evaluated, and these are converted into values for the variables before the join is performed. See [Ullman 85] for details.) Obviously, a predicate in N that is not connected to a variable of χ does not serve any useful role in the join; such predicates are excluded by condition 2(ii).

In general, there may exist several arcs entering a predicate q. The tuples arriving along these arcs are joined, and the resulting tuples passed as bindings for the evaluation of q. A binding for q is useful, however, only if it is a binding for an argument of q. That is, all the methods we present treat an argument as free, if one or more of its variables are not bound, even if some other variables in it are bound. (In this decision, we follow [Ullman 85].) This explains condition 2(iii).

The evaluation of a rule proceeds as follows. Each node with no arcs entering it is evaluated with all arguments free. (An exception is the special predicate p_h; it is treated as a base predicate and the tuples in it are those supplied for θ^b by unification.) A predicate with arcs entering it is evaluated only for values supplied through arcs. Finally, when all predicates have been evaluated, they are joined, and the result is projected on the variables of the head predicate p, to be returned as a result. (This join, like the evaluation of the predicates, can be performed in stages, as the tuples are generated, or all at once, when all tuples become available.) The third condition ensures that the sip denotes a consistent strategy for passing information through the predicates in the body. Thus, we disallow sips according to which two goals make a cyclic assumption about a variable being bound, that is, each assumes that the variable is bound by the other.

We emphasize that the above discussion of the interpretation of a sip is to be understood as an abstraction which conveys *what* is done rather than *how* it is done. For example, Prolog does not explicitly create special predicates p_h to store bound head arguments, nor does it explicitly evaluate the joins we mentioned. These operations are, however, implicit in the way Prolog maintains variable bindings through unification and backtracking.

We now illustrate and discuss these ideas by means of an example. This is also used in later chapters as a running example.

Example 4.2.1: Consider the following rules:

```
sg(X,Y) :-  flat(X,Y)
sg(X,Y) :-  up(X,Z1),sg.1(Z1,Z2),flat(Z2,Z3),sg.2(Z3,Z4),down(Z4,Y)
sg(john,?)
```

This is a non-linear version of the same-generation example. We have numbered the sg occurrences in the second rule for convenience.

Given the query, the natural way to use the second rule seems to be to solve the predicates in the indicated order, using bindings from each predicate to solve the next predicate. This information passing strategy may be represented by the following sip:

$\{sg_h\} \rightarrow_X up;$ (I)

$\{sg_b, up\} \rightarrow_{Z1} sg.1$

$\{sg_h, up, sg.1\} \rightarrow_{Z2} flat;$

$\{sg_h, up, sg.1, flat\} \rightarrow_{Z3} sg.2$

Observe that in this sip, each set at the tail of an arc contains also predicates appearing in previous sets. Another sip that seems to represent the same order of information passing steps is:

$\{sg_h\} \rightarrow_X up$; (II)
$\{up\} \rightarrow_{Z1} sg.1$;
$\{sg.1\} \rightarrow_{Z2} flat$;
$\{flat\} \rightarrow_{Z3} sg.2$

There is no implied order of evaluation for the nodes of a sip, although that provides good intuition in the case of sip (I). In sip (II), for instance, any tuple in *flat* can be used to provide values for Z3 in computing *sg.2*. This is in spite of the sip arc entering *flat*, which indicates that only a subset of *flat* needs to be used. This is because there is no implied order of evaluation between the predicates *flat* and *sg.2* - any such order must be explicitly imposed, as in sip (I), by using the partial ordering imposed by a sip. []

To see the difference between the two sips, observe that in the first, as we proceed from left to right, we carry along the bindings for all variables that have been computed so far. (This is implied by the contents of the arcs' tails and labels.) In the second, "past" information is not used. For the example as given, this does not seem to matter, provided we assume the same order of evaluation. In general, even this assumption is not enough to eliminate the difference. This can be illustrated as follows. Assume that in the example a variable W is added to the predicates *up* and $flat$. The first sip can be changed by adding W to the label of the arc entering $flat$. This is possible since a predicate containing this variable appears at the tail of the arc. For the second sip we can not make such a change to the arc label, unless we add the predicate *up* to the tail. † If the second sip is not changed, we can actually compute $flat$ with $Z2$ bound, and obtain some W values that are not compatible with the W values produced by the evaluation of *up*. The incompatible values will be dropped only by the final join of all body predicates.

We have considered arcs entering both base and derived predicates. The information passed along an arc is indeed important for both types, but for somewhat different reasons. A base predicate is always directly evaluable. Binding information

† If we choose to always pass all available information, then we can omit the labels altogether, since they can be deduced from the heads and tails of the arcs. Then, the change in the sets of attributes would not change the first sip at all.

is used as a selection condition (which may have a considerable influence on the method, as well as on the size of the result). Bindings passed to derived predicates influence the computation by restricting the subqueries that are generated. Here we are interested in binding propagation and how it can be used to improve the efficiency of evaluation in the presence of recursion. Our transformations make no use of bindings passed to base predicates. We therefore restrict our attention to sips in which only arcs entering derived predicates are represented.

Our definitions open the way to consider relationships between sips. For example, when can one sip be considered to be ''better'' than another? In particular, we can distinguish between *full* and *partial* sips. A *partial* sip is one which does not always propagate all available information.

Example 4.2.2: Consider the sip for the second rule in the previous example. It is a full sip, but it becomes partial if we modify it as follows:

$$\{sg_h\ , up\} \rightarrow_{Z1} sg.1$$
$$\{flat\} \rightarrow_{Z3}\ sg.2$$

The tail of the second arc does not contain some predecessors of $sg.2$. This arc does not depend on the bindings for the head predicate. It uses values from a base predicate, $flat$ to restrict the computation of $sg.2$, but these values are independent of the bindings known for the head predicate. []

In Chapter 6, we present some results concerning the relative merits of different sips for a given rule. However, in order to fully answer the question of how to choose a sip for a rule, we must first understand the issue of performance, which we investigate in Chapter 8.

In this chapter, we have defined information passing independently of control. A given control strategy can implement several information passing strategies, and vice-versa, and it is important to understand the relationship between these two components of a query evaluation method. In subsequent chapters, we establish the important result that bottom-up evaluation suffices to implement any information passing strategy.

Chapter 5 - Bottom-up Implementation of Sips

We consider how a given information passing strategy can be implemented using bottom-up control. We show that this can be done by rewriting a program according to the given sips and evaluating the rewritten program using Semi-Naive or Naive evaluation. In this chapter, we describe several rewriting algorithms and prove their equivalence to the original program. In the next chapter, we consider in detail how these rewriting algorithms in fact implement the associated sips.

All the rewriting algorithms first produce an intermediate program, called the *adorned program*, from the given program and sips. This adorned program is then further transformed to produce the final rewritten program. We begin by describing the first step in the transformation, which is the production of the adorned program.

1. The Adorned Rule Set

An *adornment* for an n-ary predicate p is a string a of length n on the alphabet {b,f}, where b stands for *bound* and f stands for *free*. We assume a fixed order of the arguments of the predicate. Intuitively, an adorned occurrence of the predicate, p^a, corresponds to a computation of the predicate with some arguments bound to constants, and the other arguments free, where the bound arguments are those that are so indicated by the adornment. For example, p^{bbf} corresponds to computing p with the first two arguments bound and the last argument free. If $p(X,Y,Z)$ appears in the head of a rule, then we expect the rule to be invoked with X and Y bound to constants. If $p(X,f(X,Z),W)$ is the head of a rule, then the rule will be invoked with X and $f(X,Z)$ bound to constants. Note that since bindings refer to arguments (positions) of p, if X is bound but Z is not, then $f(X,Z)$ is considered to be free.

Let a program P and a query $q(\overline{c},\overline{X})$ be given, where $\overline{c}$ is the vector of bound arguments and $\overline{X}$ is the vector of free arguments. q is called the *query predicate*. We construct a new, adorned version of the program, denoted by P^{ad}. In the construction we replace derived predicates of the program by adorned versions, where for some predicates we may obtain several adorned versions. For each adorned predicate p^a, and for each rule with p as its head, we choose a sip and use it to generate an adorned version of the rule (the details are presented below). Since the head of a rule may appear with several adornments, it follows that we may attach several distinct sips to versions of the same rule, one to each version.

The process starts from the given query whose bindings determine an adorned version in which precisely the positions bound in the query are designated as bound, say q^e. Initially, this is the only adorned predicate. In general, if p^a is an adorned predicate, then for each rule that has p in its head, we generate an adorned version for the rule and add it to P^{ad}. Having done this, we mark the adorned predicate p^a. In generating the adorned version of a rule, we may generate new adorned versions of predicates in the body of the rule. The process terminates when no unmarked adorned predicates are left. Termination is guaranteed, since the number of adorned versions of predicates for any given program is bound.

Let r be a rule in P with head p. We generate an adorned version, corresponding to an adorned predicate p^a, as follows. The new rule has p^a as a head. Choose a sip s_r for the rule, that matches the binding a; that is, the special predicate p_h is the head p restricted to arguments that are designated as bound in the adornment a. Next, we replace each derived predicate in the body of the rule by an adorned version (and if this version is new, we add it to our collection). We obtain the adorned version of a derived predicate in the body of the rule as follows. For each occurrence p_i of such a predicate in the rule let χ_i be the union of the labels of all arcs coming into p_i. (If there is no arc coming into p_i, let χ_i denote the empty label.) We replace p_i by the adorned occurrence $p_i^{a_i}$, where an argument of p_i is bound in a_i only if all the variables appearing in it are in χ_i. (For a predicate occurrence with no incoming arc, the adornment contains only f's. For our purposes here, we do not distinguish between a predicate with such an adornment and an unadorned predicate.) The arguments of the predicates in the new rule are the same as in the original rule. Since the adornments attached to a rule's predicates are determined by the sip that was chosen, the sip is attached to the rule.

Example 5.1.1: The following is the adorned rule set corresponding to the non-linear same generation example, for the sip of Example 4.2.1, considering only arcs entering derived predicates:

1. sg^{bf}(X,Y) :- flat(X,Y)
2. sg^{bf}(X,Y) :- up(X,Z1), sg^{bf}(Z1,Z2), flat(Z2,Z3), sg^{bf}(Z3,Z4), down(Z4,Y)
Query: sg^{bf}(john,Y)?

We will use these adorned rules to illustrate the rule rewrite algorithms presented later. Note that if we use the partial sip of Example 4.2.2 instead, we obtain the same adorned program. The difference between the sips will only become significant in the next stage of the transformation. (It is not the case, however, that all sips for the same rule generate the same adorned version.) []

Note that a single adorned version of a rule is chosen for each adorned version of the head predicate. Thus, all goals which match the (adorned) head predicate are solved using the same adorned version of the rule, chosen at compile time. This does not cover dynamic strategies which choose a sip for a goal at run time, that is, allow two goals which match the same (adorned) head predicate to be solved with different adorned versions of the rule.

Given an unadorned program, an adorned predicate p^a can be viewed as a *query form*. It represents queries of the form $p(\chi)$, in which all arguments corresponding to b's in adornment a are assigned constants. The same view holds for an adorned program, except that now p^a is both a predicate name and a query form that represents a class of queries on itself. Keeping these slightly different viewpoints in mind, we can now consider the equivalence of adorned and unadorned programs.

For programs P_1 and P_2 (where each may be adorned or unadorned), and a query form p^a, we say that (P_1, p^a) and (P_2, p^a) are *equivalent*, if for any assignment of constants to the arguments of p (or p^a for an adorned program) that are bound in a, the two programs produce the same answer for the resulting queries on p.

Theorem 5.1.1: For each p^a that appears in P^{ad}, (P, p^a) and (P^{ad}, p^a) are equivalent.

Proof: First, we note that for each rule of P^{ad}, if the adornments are dropped, we obtain a rule of P. It easily follows that if a rule of P^{ad} is applied to some facts to produce a new fact, then the unadorned version of the rule can be applied to the unadorned versions of those facts to generate the new unadorned fact. By induction, if an adorned version of a fact is generated in a bottom-up computation of P^{ad}, then the unadorned version of the same fact is generated in a bottom-up computation of P. It follows by the completeness of bottom-up evaluation that the answer set for (P^{ad}, p^{a}) is contained in the answer set for (P, p^{a}).

For the other direction, we observe that if a fact $p(c)$ is generated by a computation of P, then there exists a *derivation tree* for it. The fact $p(c)$ is at the root of the tree, the leaves are base facts, and each internal node is labeled by a fact, and by a rule which generates this fact from the facts labeling its children. We prove by induction on the height of trees, that given a derivation tree for p^{a} in P, if p^{a} is in P^{ad} then there is a derivation tree in P^{ad} for every node in the given derivation tree.

For the basis, a tree of height one is simply a base fact, hence it is also a tree for P^{ad}. Assume the claim holds for all trees of height less then n, and consider a tree of height n for a node $q(c)$ which appears in the given derivation tree for p^{a}. The derivation of the root node from its children corresponds to the use of an instance of a rule of the form:

$$q(c) :- q_1(c_1), \cdots q_k(c_k)$$

where each $q_i(c_i)$ is a fact. Now, consider the adorned image, q^{a1}, of the node $q(c)$. By Lemma 3.1, it appears in P^{ad}, and thus, by the construction of P^{ad}, there exists an adorned version of the same rule in P^{ad}, with q^{a1} as its head. Let its instance corresponding to the instance above be

$$q^{a1}(c) :- q_1^{a_1}(c_1), \cdots q_k^{a_k}(c_k)$$

Now, for $i = 1, ..., k$, the derivation trees for $q_i(c_i)$ (in P) are of height less than n. Further, by the construction of P^{ad}, the adorned predicates $q_i^{a_i}$ also appear in it. By the induction hypothesis, there exist derivation trees for $q_i^{a_i}(c_i)$, for each such i, in P^{ad}. It follows that a derivation tree for $q^{a1}(c)$ also exists. []

Corollary 5.1.2: (P, q^{e}) and (P^{ad}, q^{e}) are equivalent. []

We can now state the main result of this section, namely that our transformation preserves equivalence.

Corollary 5.1.3: $(P, q(\overline{c}, \overline{X}))$ and $(P^{ad}, q^{e}(\overline{c}, \overline{X}))$ are equivalent. []

2. Generalized Magic Sets

Henceforth, we only consider the adorned set of rules, P^{ad}. The next stage in the proposed transformation is to define additional predicates which compute the values that are passed from one predicate to another in the original rules, according to the sip strategy chosen for each rule. Each of the original rules is then modified so that it fires only when values for these additional predicates are available. These auxiliary predicates are called *magic predicates* and the sets of values that they compute are called *magic sets*. The intention is that the bottom-up evaluation of the modified set of rules simulate the sip we have chosen for each adorned rule, thus restricting the search space.

The transformation consists of the following.

1. We create a new predicate $magic\_p^a$ for each p^a in P^{ad}. (Thus, we create magic predicates only for derived predicates, possibly only for some of them.) The arity of the new predicate is the number of occurrences of b in the adornment a, and its arguments correspond to the bound arguments of p^a.

2. For each rule r in P^{ad}, and for each occurrence of an adorned predicate p^a in its body, we generate a *magic rule* defining $magic\_p^a$ (see below). (Note that an adorned predicate may have several occurrences, even in one rule, so several rules that define $magic\_p^a$ may be generated from a single adorned rule.)

3. Each rule is *modified* by the addition of magic predicates to its body.

4. We create a *seed* for the magic predicates, in the form of a fact, obtained from the query. The seed provides an initial value for the magic predicates. Using our notation above, the seed is $magic\_q^e(\overline{c})$.

We now explain the second step in more detail. We use the following notation. Greek letters (possibly numbered using subscripts) are used to denote argument lists. If χ denotes the argument list of a predicate p^a, then χ^f (respectively χ^b) denotes χ with all arguments that are bound (resp. free) in adornment 'a' deleted. Consider the adorned rule:

$$r: \quad p^a(\chi) :- \ q_1^{a_1}(\theta_1), q_2^{a_2}(\theta_2), \ldots, q_n^{a_n}(\theta_n)$$

Let s_r be the sip associated with this rule. Assume that the predicates in the body are ordered according to the sip, that is, those that participate in the sip precede those that do not, and the predicates in the tail of an arc precede the predicate at the head of the arc.

Consider q_i. Let $N \rightarrow q_i$ be the only arc entering q_i in the sip. We generate a magic rule defining $magic\_q_i^{a_i}$ as follows. The head of the magic rule is $magic\_q_i^{a_i}(\theta_i^b)$. If q_j, $j<i$, is in N, we add $q_j^{a_j}(\theta_j)$ to the body of the magic rule. If q_j is a derived predicate and the adornment a_j contains at least one b, we also add $magic\_q_j^{a_j}(\theta_j^b)$ to the body. If the special predicate denoting the bound arguments of the head is in N, we add $magic\_p^{\,a}(\chi^b)$ to the body of the magic rule.

If there are several arcs entering q_i, we define the magic rule defining $magic\_q_i^{a_i}$ in two steps. First, for each arc $N_j \rightarrow q_i$ with label χ_j, we define a rule with head $label\_q_i\_^{\,j}(\chi_j)$. The body of the rule is the same as the body of the magic rule in the case where there is a single arc entering q_i (described above). Then the magic rule is defined as follows. The head is $magic\_q_i^{a_i}(\theta_i^b)$. The body contains $label\_q_i\_^{\,j}(\chi_j)$ for all j (that is, for all arcs entering q_i).

In the third step, we modify the original rule by inserting occurrences of the magic predicates corresponding to the derived predicates of the body and to the head predicate, into the rule body. In principle, the magic predicates can be inserted anywhere in the rule, but it helps to understand how they interact with the other predicate occurrences by considering specific positions for the insertions. The position for the insertion of $magic\_p^{\,a}$ is at the left side of the rule's body, before all other predicates. The position for insertion of $magic\_q_i^{a_i}$ is just before the occurrence of $q_i^{a_i}$. Intuitively, $magic\_q_i^{a_i}$ computes the values that may be passed to $q_i^{a_i}$ from the left during evaluation of the rule. Insertion of $magic\_q_i^{a_i}$ serves as a guard. In a bottom-up evaluation, the rule does not fire, unless the appropriate values are first computed in $magic\_q_i^{a_i}$. The occurrence of $magic\_p^{\,a}$ serves the same purpose for the rule's head.

Example 5.2.1: Using the sips presented earlier, the Generalized Magic Sets strategy rewrites the adorned rule set corresponding to the non-linear same generation example into the following set of rules. (For each rule, we indicate how it was generated in square brackets. We do this by indicating the rule in the adorned set from which it was generated, and in the case of magic rules, which literal in this adorned rule. Further, we simplify the rules by discarding some unnecessary occurrences of magic predicates from the rules. Lemma 5.2.2 describes when this can be done.)

$magic\_sg^{bf}$(john)   [From the query rule]
$magic\_sg^{bf}$(Z1) :- $magic\_sg^{bf}$(X), up(X,Z1)
[From rule 2, 2nd body literal]
$magic\_sg^{bf}$(Z3) :- $magic\_sg^{bf}$(X), up(X,Z1), sg^{bf}(Z1,Z2), flat(Z2,Z3)
[From rule 2, 4th body literal]
sg^{bf}(X,Y) :- $magic\_sg^{bf}$(X), flat(X,Y)   [Modified rule 1]
sg^{bf}(X,Y) :- $magic\_sg^{bf}$(X), up(X,Z1), sg^{bf}(Z1,Z2),
flat(Z2,Z3), sg^{bf}(Z3,Z4), down(Z4,Y)   [Modified rule 2]   []

Let P^{mg} denote a program obtained from P^{ad} by the transformation above. We now consider the correctness of the transformation. There is a factor we need to consider first. For the given query, we have a seed definition for the magic sets. If we choose a different query with the same query form, the same magic predicates, magic predicate definitions and modified rules will result, but the seed will be specific to the query. Therefore, let us consider the seed not to be a part of P^{mg}. We say that (P^{ad},p^{a}) and (P^{mg}, p^{a}) are equivalent if the two programs produce the same results for every instance of the query form p^{a}, if the corresponding seed is added to P^{mg}.

Theorem 5.2.1: Let P^{ad}, P^{mg} be as above, and let p^{a} be a predicate that appears in P^{ad}. Then (P^{ad}, p^{a}) is equivalent to (P^{mg}, p^{a}).

Proof: One direction is obvious. Each rule of P^{mg} that is derived from a rule of P^{ad}, is more restrictive than that rule, since it has additional predicates in its body. It follows that any answer produced by P^{mg} for a query can also be produced by P^{ad}.

The other direction is proved by induction on the height of derivation trees of facts in P^{ad}. The basis of the induction is the set of derivation trees of height zero.

These are simply base facts, and they are also derivation trees for P^{mg}. Consider now a derivation tree of height n, and assume that the rule used to derive its root is the following

$$r:\quad p^{a}(\chi) \text{ :- } q_1^{a_1}(\theta_1), q_2^{a_2}(\theta_2), \ldots, q_n^{a_n}(\theta_n)$$

Let the instance of the rule be

$$p^{a}(c) \text{ :- } q_1^{a_1}(c_1), q_2^{a_2}(c_2), \ldots, q_n^{a_n}(c_n)$$

By the induction hypothesis, there exist derivation trees for the derived predicate occurrences in the body of the rule, since each of them is the root of a tree of height less then n. Note however, that for each such derivation tree the (new) seed should correspond to the fact being derived. For the fact $q_i^{a_i}(c_i)$, the seed is $magic\_q_i^{a_i}(c_i^b)$. Since these seeds are not known *a priori*, to rely on the induction hypothesis, we have to show that they can be computed in P^{mg} augmented with the seed for the original query.

The proof is by induction on the position of the predicate occurrence in the rule body. For $magic\_p^{a}$, the fact that is to be derived is $magic\_p^{a}(c^b)$. However, note that $p^{a}(c)$ is an answer for a query that is an instance of p^{a}. The seed for that query is, by assumption, given to us. It is precisely the desired fact, $magic\_p^{a}(c^b)$. It is easy to see now that each of the other magic facts that we need, that is, $magic\_q_i^{a_i}(c_i^b)$ (for each i such that q_i is derived), is derivable. Indeed, for $i = 1$, if q_1 is derived, $magic\_q_1^{a_1}(c_1^b) = magic\_p^{a}(c^b)$; for $i = 2$, (if q_i is derived), we use the fact $magic\_p^{a}(c^b)$ and the induction hypothesis to show the existence of a derivation tree for $q_1^{a_1}(c_1)$, and thus derive the fact $magic\_q_2^{a_2}(c_2)$. If q_2 is a base relation, then we evaluate it, and pass the bindings to the right, and so on. []

In constructing the magic rules and the modified rules, we added a number of magic predicates to the body. We now prove an important lemma which shows that in each rule, some of these magic predicates may be dropped without loss of information, that is, the sets of values computed by the magic predicates remain unchanged, and the number of firings of the modified rules remains unchanged.

Consider an adorned rule and a sip for it. Let us define $p \Rightarrow q$ as follows. If the sip contains an arc $N \rightarrow q$, and N contains p, then $p \Rightarrow q$. If $p \Rightarrow l$ and $l \Rightarrow q$, then $p \Rightarrow q$. (Clearly, we cannot have $p \Rightarrow q$ and $q \Rightarrow p$ simultaneously because the sip induces a total ordering.) Let us define the *order* of q to be the length of the longest chain $q_i \Rightarrow q_{i+1} \Rightarrow \ldots \Rightarrow q$. The order is 0 if there is no such chain.

Consider the set P^{mg_opt} which is obtained from P^{mg} by repeated applications of the following transformation:
Let r be an adorned rule and let r' be a (magic or modified) rule generated from r. If the body of r' contains occurrences of both $magic\_p_i^{a_i}$ and $magic\_p_j^{a_j}$, and $p_i \Rightarrow p_j$, then we may delete the occurrence of $magic\_p_j^{a_j}$.

We have the following lemma.

Lemma 5.2.2: Let P^{mg} and P^{mg_opt} be as above, and let p^a be a predicate that appears in P^{mg}. Then, (P^{mg}, p^a) is equivalent to (P^{mg_opt}, p^a).

Proof: One direction is obvious. Since each rule in P^{mg_opt} is obtained by deleting some predicates from the body of a rule in P^{mg}, it follows that a fact produced by P^{mg} can also be produced by P^{mg_opt}.

We prove the other direction by induction on the height of derivation trees. If a fact p^a has a derivation tree in P^{mg_opt}, it also has a derivation tree in P^{mg}. The basis for this claim is the set of base facts which are simply derivation trees of height 0, and are unaffected by the deletion of magic predicates from rules.

As the induction hypothesis, let the claim hold for derivation trees of height less than n. Consider a fact $p^a(c)$ which has a derivation tree of height n in P^{mg_opt}. Let the root be derived from the following rule:

$$r: \quad p^a(\chi) :- \ q_1^{a_1}(\theta_1), q_2^{a_2}(\theta_2), \ldots, q_n^{a_n}(\theta_n)$$

Note that some of the q_i's may be magic predicates. Let the instance of the rule be:

$$p^a(c) :- \ q_1^{a_1}(c_1), q_2^{a_2}(c_2), \ldots, q_n^{a_n}(c_n)$$

There exists a corresponding rule, say r', in P^{mg} with additional magic predicates in the body. Consider the instance of this rule which corresponds to the above instance of r. To prove the existence of a derivation tree for $p^a(c)$ in P^{mg}, we need only show the existence of derivation trees for magic facts which have been deleted from the instance of r' to obtain the instance of r. We prove this by induction on the position of magic facts in the instance of r'.

Without loss of generality, let the body of r' be ordered in accordance with the total ordering induced by the sip. As the basis of our induction, it is true that the first magic fact, say $magic\_q(t)$, is not deleted, since there is no predicate q' such that q' => q. Assume that there is a derivation tree for each magic fact (in the instance of r') to the left of the i'th fact in the body. Consider a magic fact $magic\_q^{a1}(c1)$ which is the i'th (or after the i'th) body fact. Consider the magic rule defining $magic\_q^{a1}$ in P^{mg} which was generated from the given occurrence of q^{a1}. The body of this magic rule only contains predicates to the left of $magic\_q^{a1}$ in r'. Replacing each of these predicates in the magic rule by its ground instance in the ground instance of r', we have a derivation tree in P^{mg} for $magic\_q^{a1}(c1)$, since by the induction hypothesis, we have derivation trees for each of these ground instances (including instances of magic predicates which are missing in r). This completes our proof of the existence of derivation trees in P^{mg} for all magic facts in the instance of r' that are missing in the instance of r, and thus also completes our proof of the existence of a derivation tree in P^{mg} for $p^a(c)$. []

The previous lemma tells us that we may drop some occurrences of magic predicates while passing only the information indicated by the sip. In particular, if h^b => p for each p in the sip, then all magic predicates can be dropped, except for the one corresponding to the head of the rule. This is a very common case. (The only exception, as we remarked earlier, is when a variable is passed to an occurrence of predicate p, but appears only in arguments which also contain variables that are not passed to p. In this case, the fact that adornments only indicate bound and free *arguments* implies that no use is made of bindings that are passed for such variables.) We may drop other occurrences of magic predicates if we wish, without affecting the equivalence of P^{mg} to P^{ad}. In fact, if we drop *all* occurrences of magic predicates, P^{mg} reduces to P^{ad}. However, by dropping additional occurrences of magic predicates from rule bodies, we compute larger sets of values in the magic predicates, and thus increase the number of firings of the modified rules.

3. Generalized Supplementary Magic Sets

The Generalized Magic Sets algorithm for rewriting a set of adorned rules succeeds in implementing a given set of sips, but it suffers from the drawback that many facts are evaluated repeatedly. If $p \Rightarrow q$ in some adorned rule, then the evaluation of *magic_q* repeats much of the work done in evaluating *magic_p*; and further, the evaluation of the modified rules repeats the work done in evaluating the magic sets.

We now present another algorithm for rewriting a set of adorned rules. This algorithm is motivated by the drawback of the previous algorithm, and by the observation that much of the duplicate work can be eliminated if we store intermediate results that are potentially useful later. We store these results in special predicates called *supplementary magic predicates*, following Sacca and Zaniolo [Sacca and Zaniolo 86], who used essentially the same idea in generalizing the versions of the Magic Sets and Counting algorithms presented in [Bancilhon et al. 86].

The algorithm is as follows. We first order the predicates in the body of each adorned rule according to the total ordering induced by the sip associated with that rule.

For each adorned rule:

1. We introduce a number of *supplementary magic predicates*, $supmagic_i^r$, associated with this rule, and define them using *supplementary magic rules*.
2. For each occurrence of an adorned predicate p^a in the body of the adorned rule, we generate a *magic rule* defining the *magic* predicate $magic\_p^a$ if the sip associated with r contains an arc $N \rightarrow p$.
3. We generate a *modified* rule from the adorned rule by replacing some of the predicates in the body by a supplementary magic predicate.

Finally, we create a *seed* for the magic predicates, in the form of a fact, from the query.

Consider the adorned rule:

$$r:\ p^a(\chi) :-\ q_1^{a_1}(\theta_1), q_2^{a_2}(\theta_2), \ldots, q_n^{a_n}(\theta_n)$$

Let the body predicates be ordered in accordance with the sip ordering, and let q_m be the last body predicate which has an incoming arc in the sip. We generate m supplementary magic rules. The first supplementary magic rule is:

$supmagic^r_1(\phi_1)$:- $magic\_p^a(\chi^b)$

where ϕ_1 is the set of variables which appear in arguments of χ^b.

Supplementary magic rule i, i = 2 to m is:

$supmagic^r_i(\phi_i)$:- $supmagic^r_{i-1}(\phi_{i-1}), q^{a_{i-1}}_{i-1}(\theta_{i-1})$

where ϕ_i is the set of variables which appear in arguments of ϕ_{i-1} or θ_{i-1}.

In generating the supplementary magic rules, the following simple optimizations may be applied. We may discard from ϕ_i, i = 1 to n, all variables which do not appear in any arguments of χ or θ_j, j = i to n. Also, the supplementary magic rule defining $supmagic^r_1(\phi_1)$ may be deleted if we replace every occurrence of this literal in a rule body by $magic\_p^a(\chi^b)$, where $p^a(\chi)$ is the head of the adorned rule r.

We generate a magic rule defining $magic\_q_i^{a_i}$ if the sip s_r contains an arc $N \rightarrow q_i$. (In generating the predicate name $magic\_q_i^{a_i}$, we exclude any subscripts of q_i introduced for the purpose of distinguishing different occurrences of the same predicate.) This magic rule is:

$magic\_q_i^{a_i}(\theta_i^b)$:- $supmagic^r_i(\phi_i)$

The modified rule corresponding to r is:

$p^a(\chi)$:- $supmagic^r_m(\phi_m), q^{a_{m+1}}_{m+1}(\theta_{m+1}), \ldots, q^{a_n}_n(\theta_n)$

Finally, if the query is $q^a(\eta)$, we also add a magic rule corresponding to it, to act as the seed:

$magic\_q^a(\eta^b)$

Example 5.3.1: Continuing with the same generation example, the Generalized Supplementary Magic Sets algorithm produces the following rules by rewriting the adorned rules according to the given sip. (The rule numbers refer to the adorned rules.)

$magic\_sg^{bf}$(john)   [From the query rule]
$supmagic_2^2$(X,Z1) :- $magic\_sg^{bf}$(X), up(X,Z1)   [From rule 2]
$supmagic_3^2$(X,Z2) :- $supmagic_2^2$(X,Z1), sg^{bf}(Z1,Z2)   [From rule 2]
$supmagic_4^2$(X,Z3) :- $supmagic_3^2$(X,Z2), flat(Z2,Z3) [From rule 2]
sg^{bf}(X,Y) :- $magic\_sg^{bf}$(X), flat(X,Y)   [Modified rule 1]
sg^{bf}(X,Y) :- $supmagic_4^2$(X,Z3), sg^{bf}(Z3,Z4), down(Z4,Y)
[Modified rule 2]
$magic\_sg^{bf}$(Z1) :- $supmagic_2^2$(X,Z1)   [From rule 2, 2nd body literal]
$magic\_sg^{bf}$(Z3) :- $supmagic_4^2$(X,Z3)   [From rule 2, 4th body literal] []

Let us denote by P^{sup-mg} any program obtained by this transformation.

Theorem 5.3.1: Let P^{ad}, P^{sup-mg} be as above, and let p^a be a predicate in P^{ad}. Then (P^{ad}, p^a) is equivalent to (P^{sup-mg}, p^a).

Proof: As in the previous case, the proof in one direction is obvious. The other direction is proved by induction on the height of derivation trees of facts in P^{ad}. The proof is similar to the previous proof. It suffices to note that the seeds have the same form as in the previous case, and that the seed for $q_i^{a_i}(c_i)$ can be obtained by first generating $supmagic_i^r(d)$, where d is a vector of constants, and then using the auxiliary rule $magic\_q_i^{a_i}(..) :- supmagic_i^r(..)$ to generate the required seed $magic\_q_i^{a_i}$. []

The Alexander strategy, described in [Rohmer and Lescoeur 85], is essentially the Generalized Supplementary Magic Sets strategy, although they only consider Datalog.

4. Generalized Counting

Counting is a further elaboration on the theme of restricting the search by auxiliary predicates. Using magic predicates, we were able to restrict the invocation of a predicate to values that were reachable from those given in the query. The new idea here is to keep track of which rules and which predicate occurrence in each rule were used to reach a vector of values that is now used in the invocation of a predicate. For example, in the same generation program, with the first argument bound to a constant, the magic construction restricts the search to the ancestors of that constant. However, when we compute the "nephews" of such an ancestor, we do not know if any of them qualifies for the answer. What we need for that is first, the distance of that ancestor from the constant given in the query and second, the distance of that nephew from that ancestor. If these are equal, then we have an instance of the answer.

We now describe the Counting transformation. We first replace each adorned derived predicate p^a in the adorned rules by an *extended version* of this predicate, $\overline{p}^a$, which has three new arguments. These arguments are used for constructing indices, and we assume that they are the first three arguments. Note that the adornment a refers only to the non-index fields. For each rule r in P^{ad}:

1. For each occurrence of an adorned predicate p^a in the body of the adorned rule, we generate a *counting rule* defining the *counting* predicate cnt_p^a if the sip associated with r contains an arc $N \rightarrow p$.
2. The rule is *modified* by the addition of counting predicates to its body.

Finally, we create a *seed* for the counting predicates, in the form of a fact, from the query.

Before each of these steps is explained, let us consider how the rules and predicates used in a derivation may be encoded in a predicate, together with the derived value(s). We call a rule a *exit rule* if all the predicates in its body are base predicates. We need to encode only non-exit rules. Assume we have m such rules, numbered r_0 to r_{m-1}. Then the sequence of rules used in a derivation can be represented by a sequence of numbers, each in the range $[0 .. m-1]$. Any standard encoding can be used to represent such sequences by numbers. We will use the encoding that represents the sequence $i_0, i_1, ..., i_k$ by the value $(...((i_0 \times m) + i_1) \times m + ...) \times m + i_k$. In other words, given a number that represents a sequence, to concatenate an element to the sequence we multiply by m and add the element. The last element is the remainder modulo m, and previous elements can be obtained by repeated use of the modulus operation. A similar encoding is used for predicate occurrences. Assuming that there are at most t occurrences of adorned predicates in any rule's body, we use t as the base for the encoding. (Note that only derived predicates are adorned; we need only to keep track of invocation of derived predicates.) These two encodings occupy the second and third positions of the counting predicates. The first position is used to record the number of rules that were applied so far (starting from the seed). (Note that this number can be computed from the encodings; it is convenient to have an explicit representation for it.)

We now describe each step of the Counting algorithm in detail. We use '_' to denote a *don't-care* variable, that is, a variable whose value is irrelevant to the success or failure of the rule in which it occurs. Further, it is understood in the following that the index fields are omitted from arguments of base predicates.

Consider the adorned rule:

$$r_i : p^{a}(\chi) :\text{-}\ q_1^{a_1}(\theta_1), q_2^{a_2}(\theta_2), \ldots, q_n^{a_n}(\theta_n)$$

We generate a counting rule defining $cnt\_\overline{q}_j^{a_j}$ if the sip associated with r_i contains an arc[†] $N \rightarrow q_j$. The head of the counting rule is $cnt\_\overline{q}_j^{a_j}(I+1, K\times m+i, H\times t+j, \theta_j^b)$. If q_k is in N, we add $\overline{q}_k^{a_k}(I+1, K\times m+i, H\times t+k, \theta_k)$ to the body of the counting rule. If q_k is a derived predicate and the adornment a_k contains at least one b, we also add $cnt\_\overline{q}_k^{a_k}(I+1, K\times m+i, H\times t+k, \theta_k^b)$ to the body. If the special predicate denoting the bound arguments of the head is in N, we add $cnt\_\overline{p}^{a}(I, K, H, \chi^b)$ to the body of the counting rule.

The modified rule corresponding to adorned rule r_i is:

$$\overline{p}^{a}(I, k, h/t, \chi) :\text{-}\ cnt\_\overline{p}^{a}(I, k, h/t, \chi^b), \overline{q}_1^{a_1}(I+1, k\times m+i, h+1, \theta_1)$$
$$, \ldots, \overline{q}_n^{a_n}(I+1, k\times m+i, h+n, \theta_n)$$

If there is no arc entering q_j in the sip, the three index fields in the modified rule given above are replaced by '_'. Further, we could add the counting predicates corresponding to the predicates $q_j^{a_j}$ to the body of the modified rule, but we have a

† We assume that there is at most one such arc. The generalization to the case where there are several such arcs is similar to that in the Generalized Magic Sets strategy.

lemma (see below) which tells us that they are unnecessary, and so we have omitted them altogether for simplicity.

Finally, if the query is $q^a(\eta)$, we add the fact $cnt\_\overline{q}^a(0,0,0,\eta^b)$ to serve as the seed for the counting predicates.

Example 5.4.1: Continuing our running example, we present below the rewritten rules produced by the Generalized Counting method.

$cnt\_\overline{sg}^{bf}$ (I+1, k*2+2, h*5+2, Z1) :- $cnt\_\overline{sg}^{bf}$ (I, k, h, X), up(X,Z1)
[From rule 2, second body literal]

$cnt\_\overline{sg}^{bf}$ (I+1, k*2+2, h*5+4, Z3) :- $cnt\_\overline{sg}^{bf}$ (I, k, h, X), up(X,Z1),
$\overline{sg}^{bf}$ (I+1, k*2+2, h*5+2, Z1,Z2), flat(Z2,Z3)
[From rule 2, fourth body literal]

$\overline{sg}^{bf}$ (I, k, h/5, X,Y) :- $cnt\_\overline{sg}^{bf}$ (I, k, h/5, X), flat(X,Y) [Modified rule 1]

$\overline{sg}^{bf}$ (I, k, h/5, X,Y) :- $cnt\_\overline{sg}^{bf}$ (I, k, h/5, X), up(X,Z1),
$\overline{sg}^{bf}$ (I+1, k*2+2, h*5+2, Z1,Z2), flat(Z2,Z3),
$\overline{sg}^{bf}$ (I+1, k*2+2, h*5+4, Z3,Z4), down(Z4,Y) [Modified rule 2]

$cnt\_\overline{sg}^{bf}$ (0, 0, 0, john) [From the query rule] []

Let us denote a program obtained from P^{ad} by the transformation above by P^{count}. We now need a convention for comparison of the queries in P^{ad} and P^{count}. We will say that (P^{ad}, p^a) is equivalent to $(P^{count}, \overline{p}^a)$ are equivalent if for any vector θ of appropriate arity, with constants in the positions bound by a, the program-query $(P^{ad}, p^a(\theta)$ has the same set of answers as the program-query pair $(P^{count}, \overline{p}^a(I, K, H, \theta))$, for any values of I, K, H. (Note that we are not restricting attention to queries with a triple of 0's. We need arbitrary triples for the induction hypothesis of the theorem below. Intuitively, it should not matter which numerical values are supplied with the query, so this is not a real generalization.) We will also use the same conventions as in the previous sections, regarding seeds. We now have the following theorem.

Theorem 5.4.1: Let P^{ad} and P^{count} be as above and let p^a be a query form. Then $(P^{ad}, p^)$ and $(P^{count}, \overline{p}^a)$ are equivalent.

Proof: Again, one direction is obvious, since each rule of P^{count} is more restricted then the corresponding rule of P^{ad}. The other direction is proved by induction on the height of derivation trees of facts in P^{ad}.

Derivation trees of height zero are base facts, and are also derivation trees of P^{count}. For a derivation tree of height n, assume the rule used in P^{ad} to derive its root is the rule

$$r_i: \qquad p^a(\chi) \text{ :- } q_1^{a_1}(\theta_1), q_2^{a_2}(\theta_2), \ldots, q_n^{a_n}(\theta_n)$$

and the instance of the rule that is used is

$$\overline{p}^a(c) :- \overline{q}_1^{a_1}(c_1), \ldots, \overline{q}_n^{a_n}(c_n)$$

By the induction hypothesis, there exist derivation trees in P^{count} for the extended versions of the derived predicate occurrences that appear in the body, for any values of the numerical triples that may be added. Here again, we have to show, however, that appropriate seeds can be generated first. Let us consider the case where $\overline{p}^a(I, K, H, c)$ is the instance of the root for which we want to show the existence of a derivation tree. The seed, $cnt\_\overline{p}^a(I, K, H, c^b)$ is by assumption given to us. We now prove by induction on the position of a predicate occurrence in the body that the appropriate seed for each predicate, which is an instance of the corresponding counting predicate, can be generated. The details follow the line of the previous proof and are omitted. []

Let P^{cnt_opt} be defined similarly to P^{mg_opt}. We have the following lemma which allows us to delete unnecessary occurrences of counting predicates from rule bodies.

Lemma 5.4.2: Let P^{cnt} and P^{cnt_opt} be as above, and let p^a be a predicate that appears in P^{cnt}. Then, (P^{cnt}, p^a) is equivalent to (P^{cnt_opt}, p^a).

Proof: The proof is similar to the proof of Lemma 5.2.2 and is omitted. []

5. Generalized Supplementary Counting

The Generalized Counting algorithm suffers from duplicate work just like the Generalized Magic Sets algorithm. We use the same idea - of eliminating this duplication by storing potentially useful intermediate results - to define the Generalized Supplementary Counting strategy.

The algorithm is as follows. We first order the predicates in the body of each adorned rule according to the total ordering induced by the sip associated with that rule. We also replace each adorned derived predicate p^a in the adorned rules by an *extended version* of this predicate, $\overline{p}^a$, which has three new arguments. These arguments are used for indices, and we assume that they are the first three arguments. Now, for each adorned rule:

1. We introduce a number of *supplementary counting predicates*, $supcnt_i^r$, associated with this rule, and define them using *supplementary counting rules*.
2. For each occurrence of an adorned predicate p^a in the body of the adorned rule, we generate a *counting rule* defining the *counting* predicate cnt_p^a if the sip associated with r contains an arc $N \rightarrow p$.
3. We generate a *modified* rule from the adorned rule by replacing some of the predicates in the body by a supplementary counting predicate and appropriately indexing the remaining predicates.

Finally, we create a *seed* for the counting predicates, in the form of a fact, from the query.

We now explain each of the above steps in detail. It is understood in the following that the three index fields are omitted from arguments of base predicates.

Consider the adorned rule:

$$r:\ p^a(\chi) :\text{-}\ q_1^{a_1}(\theta_1), q_2^{a_2}(\theta_2), \ldots, q_n^{a_n}(\theta_n)$$

Let the body predicates be ordered in accordance with the sip ordering, and let q_m be the last body predicate which has an incoming arc in the sip. We generate m supplementary counting rules. The first supplementary counting rule is:

$$supcnt_1^r(I, K, H, \phi_1) :\text{-}\ cnt\_\overline{p}^a(I, K, H, \chi^b)$$

where ϕ_1 is the set of variables which appear in arguments of χ^b, and I, K and H are running indices.

Supplementary counting rule j, j = 2 to m is:

$$supcnt_j^r(I, K, H, \phi_j) :\text{-}\ supcnt_{j-1}^r(I, K, H/t, \phi_{j-1}), \overline{q}_{j-1}^{a_{j-1}}(I+1, K \times m+i, H \times t+j, \theta_{j-1})$$

where ϕ_j is the set of variables which appear in arguments of ϕ_{j-1} or θ_{j-1}.

In generating the supplementary counting rules, the following simple optimizations may be applied. We may discard from ϕ_j, j = 1 to m, all variables which do not appear in any arguments of χ or θ_k, k = j to m. Also, the supplementary counting rule defining $supcnt_1^r(I,K,H,\phi_1)$ may be deleted if we replace every occurrence of this literal in a rule body by $cnt\_\overline{p}^a(I,K,H,\chi^b)$, where $p^a(\chi)$ is the head of the adorned rule r.

We generate a counting rule defining $cnt\_\overline{q}_j^{a_j}$ if the sip associated with the adorned rule contains an arc $N \rightarrow q_j$. (In generating the predicate name $cnt\_\overline{q}_j^{a_j}$, we exclude any subscripts of q_j introduced for the purpose of distinguishing different occurrences of the same predicate.) This counting rule is:

$$cnt\_\overline{q}_j^{a_j}(I+1, K\times m+i, H\times t+j, \theta_j^b) \text{ :- } supcnt_j^r(I,K,H,\phi_j)$$

The modified rule corresponding to r_i is:

$$\overline{p}^a(I,K,H/t,\chi) \text{ :- } supcnt_m^r(I,K,H,\phi_m), \overline{q}_{m+1}^{a_{m+1}}(\_,\_,\_,\theta_{m+1}), \ldots, \overline{q}_n^{a_n}(\_,\_,\_,\theta_n)$$

Finally, if the query is $q^a(\eta)$, we also add a counting rule corresponding to it, to act as the seed:

$$cnt\_\overline{q}^a(0,0,0,\eta^b)$$

Example 5.5.1: The same non-linear same generation query is rewritten as follows by the Generalized Supplementary Counting algorithm. In this example, we consider the literal down(Z4,Y) in the second adorned rule to be non-distinguished.

$supcnt_2^2$(I, k, h, X,Z1) :- $cnt\_\overline{sg}^{bf}$(I, k, h, X), up(X,Z1) [From rule 2]
$supcnt_3^2$(I, k, h, X,Z2) :- $supcnt_2^2$(I, k, h, X,Z1),
$\overline{sg}^{bf}$(I+1, k*2+2, h*5+2, Z1,Z2) [From rule 2]
$supcnt_4^2$(I, k, h, X,Z3) :- $supcnt_3^2$(I, k, h, X,Z2), flat(Z2,Z3) [From rule 2]
$\overline{sg}^{bf}$(I, k, h/5, X,Y) :- $cnt\_\overline{sg}^{bf}$(I, k, h/5, X), flat(X,Y) [Modified rule 1]
$\overline{sg}^{bf}$(I, k, h/5, X,Y) :- $supcnt_4^2$(I, k, h/5, X,Z3),
$\overline{sg}^{bf}$(I+1, k*2+2, h*5+4, Z3,Z4), down(Z4,Y) [Modified rule 2]
$cnt\_\overline{sg}^{bf}$(I+1, k*2+2, h*5+2, Z1) :- $supcnt_2^2$(I, k, h, X,Z1)
[From rule 2, second body literal]

$cnt\_\overline{sg}^{bf}$ (I+1, k*2+2, h*5+4, Z3) :- $supcnt_4^2$ (I, k, h, X,Z3)
[From rule 2, fourth body literal]
$cnt\_\overline{sg}^{bf}$ (0, 0, 0, john) [From the query rule]

Let us denote by $P^{sup-cnt}$ any program obtained by this transformation.

Theorem 5.5.1: Let P^{ad}, $P^{sup-cnt}$ be as above, and let p^a be a predicate in P^{ad}. Then (P^{ad}, p^a) is equivalent to $(P^{sup-cnt}, p^a)$.

Proof: As in the previous case, the proof in one direction is obvious. The other direction is proved by induction on the height of derivation trees of facts in P^{ad}. The proof is similar to the previous proof. []

6. Further Optimizations

We now present some optimizations which may sometimes be used in further rewriting rules produced by the Generalized Counting or Generalized Supplementary Counting Strategies. They do not apply to the Magic Sets strategies since they rely on the indices generated by the Counting strategies.

Lemma 5.6.1: Consider a rule r in the given program, with an arc $N \rightarrow q_k$ in the corresponding sip. Consider the modified rule produced from r by the (Supplementary) Counting method, or the (supplementary) counting rule generated from this sip arc. Denote this rule r_1.

If no variable appearing in a predicate in N (or an associated counting predicate) appears outside N in r_1, except possibly in bound arguments of $\overline{q}_k^{a_k}$, then all predicates in N (and their counting predicates) may be deleted from the given rule.

Proof: The predicates in N (and their counting predicates) represent a join with the bound arguments of $\overline{q}_k^{a_k}$. By construction of the counting predicates, the projection of N that participates in this join is a subset of the counting predicate for $\overline{q}_k^{a_k}$. The indices identify the subset which belongs in this projection. Since only facts that agree with the tuples in this counting predicate are computed in $\overline{q}_k^{a_k}$, the join with the predicates in N is satisfied for every tuple in $\overline{q}_k^{a_k}$, with the appropriate index values. We may therefore delete the predicates in N from the rule. []

recursive with the head of the rule, we may not be able to drop the bound arguments from the body literal because some of the variables in them also appear in bound arguments of the head. In this case, it may still be possible to drop the bound arguments from the body predicate if, intuitively, the bound arguments in the head are not really needed, and may also be dropped. (In this case, it may also be possible to drop some literals which share variables only with the bound arguments in the head (which have been dropped), by applying Lemma 5.6.1.)

Theorem 5.6.3: (The Semijoin Optimization) Consider a block B of mutually recursive (adorned) predicates in the rewritten set of rules produced by the Counting or Supplementary Counting method. Suppose the following conditions hold for every predicate p in B, in every rule defining a predicate in block B:

1. No variable in a bound argument of (a body literal) p appears anywhere else in the rule (that is, other than in bound arguments of p) except possibly in bound arguments of the head or in arguments of predicates in N, where there is an arc $N \rightarrow p$ in the corresponding sip.
2. If we uniformly drop the bound arguments of all predicates in the block B in all occurrences (in rule heads and bodies), for each arc $N \rightarrow p$ encountered in 1), Lemma 5.6.1. can be applied to drop all literals in N and their counting predicates.

Then, both steps 1) and 2) can indeed be applied.

Proof: The proof for step 1) is similar to the corresponding proof in Lemma 5.6.2. The only additional observation is the following. Consider the case where variables in bound arguments of a body predicates cannot be dropped because they occur in bound arguments of the head. Dropping these arguments in the body predicates thus prevents us from computing certain arguments for the head predicate. But clearly, these arguments are not needed if for every predicate in the mutually recursive block, this is the only reason the bound arguments cannot be dropped by the previous lemma. Thus, the justification for this optimization can be seen by viewing it in two steps. First, for each rule defining a predicate in block B, the bound arguments are dropped from the head by conjecturing that they are not needed. Next, this conjecture is verified by checking that variables in these arguthe semijoin optimization and one without, but this is a refinement that we will not describe further.

7. Discussion

We have presented the following rule rewriting strategies:

1. Generalized Magic Sets (GMS)
2. Generalized Supplementary Magic Sets (GSMS)
3. Generalized Counting (GC)
4. Generalized Supplementary Counting (GSC)

We have also presented some important optimizations.

In this section, we discuss their relative merits informally. The main point we make is that for each of these strategies, with or without semijoin optimizations, there is some set of rules and data such that it is the best strategy. Therefore, we need to consider all of them in deciding on a rule rewrite strategy.

In the following discussion, we refer to the strategies by their acronyms. GMS suffers from the fact that it duplicates the work it does in computing the magic sets when computing the corresponding predicates (that is, when firing the modified rules). GC suffers from the same drawback.

This problem is solved in GSMS and GSC by storing all results that are potentially useful later on. Thus, they tradeoff additional memory (and possibly, increased lookup times) for the time gained in avoiding some duplicate firings of rules.

GC and GSC refine the notion of a *relevant fact* by essentially numbering the magic sets. This means that they avoid many unnecessary firings by starting at the query node and working outwards. They do this at the cost of maintaining a system of indices (and of course, are applicable only for a restricted set of data and rules).

The semijoin optimization offers two benefits. It reduces the number of joins (by deleting some literals) in the optimized rule, and reduces the width (number of arguments) of the optimized predicate. It is a powerful optimization which could significantly improve performance. If the semijoin optimization is not applicable for an occurrence of an adorned predicate, it might become applicable if we consider some of the bound arguments to be free. (For example, if there is a variable in

Chapter 6 - On the Power of Magic

In this chapter, we present some results characterizing the Generalized Magic Sets rule rewriting algorithm with respect to other methods for implementing a sip. The discussion in this chapter is limited to methods that treat arguments as bound or free, that is, which only consider adornments that specify bound and free *arguments*, and thus do not utilize the bindings present in partially bound arguments. All methods trivially obey this restriction for Datalog programs.

1. Optimality of a Method

We begin this discussion by describing and motivating a syntactic restriction that we place upon the rules, for the purposes of this chapter only. Two predicate occurrences in a rule are said to be *connected* if a variable X appears in some argument of each occurrence. A rule is *connected* if every predicate occurrence in the body is connected to the head predicate. A *connected component* in a rule is a maximal set of connected predicates. Thus, a connected rule has exactly one connected component. We observe that a sip only contains arcs $N \rightarrow p$ such that p and all predicates in N belong to the same connected component. So a sip describes information passing within a connected component. Further, if a rule contains a component that is not connected to the head predicate, the conjunction of predicates in this component represent an *existential* query, that is, we are only interested in determining whether this conjunction of predicates can be satisfied.

For simplicity's sake, we only consider connected rules. This is, effectively, an assumption that we begin each computation by first evaluating, for each rule, the components that are not connected to the head. If the component can be satisfied, we delete it from the rule body, else we discard the rule. This seems to be a reasonable assumption about how an intelligent method would work.

Our main result concerns the optimality of the Generalized Magic Sets method, in the sense that it implements a given sip by computing a minimal number of facts. We make no claims about the number of times a fact is (re)computed, and in fact, it is in this area that the approach must be refined. The other variants of the Magic Sets and Counting methods presented in this paper address this issue.

the given query. In a top-down method, for every rule head matching the query, we invoke the rule by setting up (sub)queries for each predicate occurrence in the body. This is done in accordance with the sip associated with the rule. Thus, we define adornments as described earlier, and create a query using the adorned versions of the predicates, with the bound arguments set to values provided by the sip arcs entering the given predicate. On the other hand, in a bottom-up method such as Magic Sets, these subqueries are not created explicitly, but are implicit in the values computed in the "magic sets".

In the above definition, we carefully avoid specifying that the set of associated subqueries should actually be created - the only requirement is with respect to the set of facts that are computed.

By our definition of adornments, an argument is considered free if any variable in it is free. This restriction essentially limits us to the class of methods which make no use of partially instantiated arguments. (Prolog is an example of a method which does not belong in this class.) However, if we consider only Datalog programs, this distinction does not arise, and our definition of a method includes all methods which infer facts solely from the logical implications of the rules.

An *optimal* method is defined to be a method which generates only the facts required by the above definition.

We have the following theorem.

Theorem 6.1.1: Consider a query over a set of connected rules P, with a sip being associated with each rule. Let P^{mg} be the set of rewritten rules produced by the Generalized Magic Sets method. The bottom-up evaluation of P^{mg} is optimal (upto the overhead of generating the magic sets).

Proof: For each adorned predicate p^a, only facts $p^a(\overline{c},\overline{d})$ are produced where the values in (the bound arguments) $\overline{c}$ agree with the corresponding values in some magic fact ($magic\_p^a(\overline{c})$). By construction of the magic rules, it is easy to see that the set of associated subqueries contains the query $p^a(\overline{c},\overline{X})?$. Thus, the fact $p^a(\overline{c},\overline{d})$ must be computed by any valid method. []

Thus, the Magic Sets method implements a sip optimally in that it generates no unnecessary facts. This claim is modulo the overhead involved in computing the

the head, $s_1 > s_2$. We have the following lemma.

Lemma 6.1.3: Let s_1 and s_2 be two sips for a given connected rule. If $s_1 > s_2$, then the set of facts computed by an optimal method for s_1 is contained in the set of facts computed by an optimal method for s_2.

Proof: If $s_1 > s_2$, every subquery generated according to s_1 is also generated according to s_2. The lemma follows immediately from the definition of a method.
[]

If the Magic Set method is chosen to implement the sips, then the above lemma holds even if we consider the magic facts too, since every magic fact corresponds to a subquery.

Corollary 6.1.4: Every fact generated by an optimal method for a full sip is also generated by any other method for that sip.

Proof: By definition of a full sip. []

We observe that our metric in the above discussion has been the number of distinct facts generated. This ignores several important factors, including the overhead involved in implementing a control method (e.g. for the Magic Set method, this is the number of magic facts, and the overhead of implementing the bottom-up evaluator) and the number of times a given fact is generated. The other three methods presented in this paper address the second issue, and it is possible to devise other schemes as well.

2. Safety

We now consider the issue of safety, that is, does the bottom up evaluation of the rewritten rules terminate after computing all answers? In the previous section, we observed that this is indeed the case if there exists any safe implementation of the program according to the given sips. Thus, the rewriting algorithms we discussed are really orthogonal to the issue of safety. The problem of safety can thus be stated at the level of sip collections, and this provides a more general approach to safety. However, this does not tell us whether such a program exists. In this section, we present some sufficient conditions for recognizing that the bottom up evaluation of a rewritten set of rules is safe.

Theorem 6.2.2: The Magic Sets methods are safe for Datalog programs.

Proof: Follows immediately from the fact that Naive bottom-up evaluation is safe for Datalog. (Intuitively, the set of constants is contained in the (finite) database relations, and new terms cannot be constructed since there are no function symbols. Since the constants can only be combined to form tuples (of finite arity) in a finite number of ways, a fixpoint is reached in a finite number of steps.) []

The above theorem does not hold for the Counting methods. It is well known that the Counting methods may not terminate if the data is cyclic, since the same value may be computed periodically at several levels (of indexing).

There are also some Datalog rules for which the Counting methods will not terminate, regardless of the data. Consider a Datalog program. Construct the binding graph for the query. Construct an *argument* graph from the binding graph as follows.

Consider an arc $p_1^{a_1} \rightarrow p_2^{a_2}$ in the binding graph, with label $[r_i, j]$. If a variable X appears in the mth argument of p_1 and the nth argument of p_2 and these are the only bound arguments in these predicates, then add the arc $p_1^{a_1} \rightarrow p_2^{a_2}$ to the argument graph. The adorned query predicate is the root.

We have the following theorem.

Theorem 6.2.3: The Generalized Counting and Generalized Supplementary Counting methods will not terminate for Datalog programs with cyclic reachable argument graphs.

Proof: If there is a cycle in the reachable argument graph, clearly the counting predicates associated with nodes on the cycle compute the same fact an infinite number of times, incrementing the index each time. []

Theorem 6.2.3 can be strengthened by considering cases in which there is more than one bound argument. The intuition is the same - we try to detect whether there is a cycle in which the set of bound arguments of some predicate occurrence is mapped onto itself (and thus appears with an infinite number of indices in the corresponding counting set). The formulation of the theorem is more involved however, and we do not discuss it.

Consider the program containing the single rule:

```
p(<X>) :- q(X).
```

Two possible models are M_1 = {q(1), q(2), p({1,2})} and M_2 = {q(2), q(3), p({2,3})}. We note that the intersection of these two models is not a model because it does not contain p({2}). The unrestricted use of <> can lead to semantic difficulties. The following rule, for instance, introduces Russell's paradox:

```
p(<X>) :- p(X).
```

Thus, this construct must be used with care. []

In addition to set grouping, programs may contain negated literals in the body.

Example 7.1.2:

Consider

```
p(X) :- q(X), ¬ r(X).
```

This is a rule using negation, and it simply defines p to be $q - r$, which is a familiar relational algebra operation. However, the introduction of negation in the context of recursive rules may mean that there is no least model. For example, consider:

```
p :- ¬ q.
q :- ¬ p.
```

This program has two minimal models, {p, ¬ q} and {q, ¬ p}. []

In order to deal with the problems introduced by set generation and negation, we consider a restricted class of programs for which there is a unique minimal model.

Let us first define a relation (>, ≥) over the set of predicates in a program P. For any two predicates p and q, if there is a rule in P with head p, define

$p \geq q$ if there is no <> in the head, and there is a positive occurrence of q in the body,

$p > q$ if there is <> in the head, and an occurrence of q in the body, and

$p > q$ if there is a negated occurrence of q in the body.

1. Each node is either a subset or a member of $P(r) \cup \{p_h\}$.

2. Each arc is of the form $N \rightarrow q$, with label χ, where N is a subset of $P(r) \cup \{p_h\}$, q is a member of $P(r)$, and χ is a set of variables, such that

i. Each variable of χ appears in q and in an argument, not a head argument of the form <X>, of a positive member of N.

ii. Each member of N is connected to a variable in χ.

iii. For some argument of q, all its variables appear in χ. Further, each variable of χ appears in an argument of q that satisfies this condition.

These two conditions define the nature of nodes and arcs of a sip. There is a third condition which provides a consistency restriction on a sip. For a graph with nodes and arcs as above, define a precedence relation on the members of $P(r) \cup \{p_h\}$ as follows:

i. p_h precedes all members of $P(r)$.

ii. A predicate that does not appear in the graph, follows every predicate that appears in it.

iii. If $N \rightarrow q$ is an arc, and $q' \in N$, then q' precedes q.

We can now state the last condition defining a sip:

3. The precedence relation defined by the sip is acyclic, that is, its transitive closure is a partial order.

We interpret the graph as follows: Assume we want to use the rule r, with some arguments of the head predicate bound to constants. The special node p_h may be thought of as a base relation whose attributes are the variables appearing in bound arguments of the head predicate. An arc labeled χ from a set of predicates N to a predicate q means that by evaluating the join of the predicates in N (with some arguments possibly bound to constants), values for the variables in χ are obtained, and these values are passed to the predicate q, and used to restrict its computation. Thus, for each arc from N to p, the arc label must contain only variables that are bound when the goals corresponding to the predicates in N are solved, and any control strategy which implements the sip must ensure this. Clearly, if there is no arc entering a node q, it is evaluated with all arguments free, that is, all tuples in q

N, and X is in χ, it appears at first sight that we can restrict X to the values that appear in the set denoted by <X>. This is incorrect because of the definition of set generation using <>.[†] <X> denotes the set of all values for X for which the body is satisfied. By passing (for X) only the values contained in the bound argument corresponding to <X>, we can test whether the body is satisfied for each of these values. However, since we thereby restrict the computation of the body to those values of X, we have no way of determining whether the body is satisfied for any *other* values of X. Thus, we cannot determine whether the set bound to <X> is indeed the set of all values for X for which the body succeeds, or whether it is a subset thereof. It follows that we cannot consider X bound if it only appears in a bound argument of the form <X>; rather, we must compute the body with X free, and then check the set of all values for X for which the body succeeds with the set bound to <X>.

Example 7.2.1: Consider the rule:

p(X) :- q1(X), ¬ q2(X,Y),q3(Y),q4(X)

The following sip is legal:

$$\{q1\} \rightarrow_X \neg q2$$
$$\{q1, \neg q2, q3\} \rightarrow_X q4$$

For every tuple in $q1$, we can compute all tuples in $q2$ which contain that value in the first argument. For every tuple in the join[†] of the positive predicates $q1$ and $q3$, which has this value for X, we can check whether it is a tuple in $q2$. If so, we throw it away. Otherwise, we pass the X value to $q4$.

For example, suppose we compute the tuples {(5), (6)} for $q1$, and {(7)} for $q3$. Their cartesian product is {(5,7), (6,7)}. We also compute $q2$ fully with the bindings X=5 and X=6 (according to the first arc in the sip). Suppose the only tuples in $q2$ with these values for X are {(5,1), (5,6), (5,7), (6,5)}. The tuple (5,7) in the cartesian product of $q1$ and $q3$ is also present in $q2$, and so we discard it. The

† This is why the second condition in the definition of a sip states that the label of a sip arc can only contain variables that appear in some argument *not of the form <X>*.

† Actually, cartesian product, since they have no common variables.

p(Y, <X>) :- q(X,Y,Z).

Suppose p is called with both arguments bound. The following sip is illegal:

$\{p_h\} \rightarrow_X q$

In order to check whether the given tuple is in p, we must first find all tuples in q with the given values for Y. We must then group with respect to X, and check to see if the given tuple (the query tuple) is indeed in p. Suppose the given goal is $p(1, \{2,3\})$?. Using set unification, we get the bindings Y=1, <X> = {2,3}. This tuple is in p, that is, the goal succeeds, if there are exactly two tuples in q with Y=1, and they have X=2 and X=3 respectively. []

We have not discussed how to choose sips for a given set of rules. In general, several sips are possible, and the choice must be made on the basis of which sips result in the most efficient evaluation strategy. Thus, we must be able to estimate the cost of various strategies resulting from the choice of a given set of sips. As an extreme case, some choices of sips may result in non-terminating computations.

3. Derivation Trees

We now define the notion of a *derivation tree*, which we use extensively in proofs of subsequent theorems. We first observe that the adorned program P^{ad} is generated from a given program P and a set of sips exactly as in Chapter 5. In the following, we denote by $p^a(c)$ a ground tuple for the adorned predicate p^a. The vector of (ground) arguments is denoted as c. We denote by $\{p^a(c)\}$ the set of all ground tuples $p^a(c)$ (for a given set of values for the bound arguments of c). A *ground instance* of a rule is defined as follows: Each positive occurrence of a predicate p^a is replaced by a ground tuple $p^a(c)$, and each negative occurrence of a predicate p^a is replaced by a set $\{p^a(c)\}$. Further, if each positive ground tuple in the body is held to be a fact, and for each negated predicate occurrence p^a, the set $\{p^a(c)\}$ associated with it is taken to be the set of all facts p^a (that is, for the given set of bound arguments in c), then, the (ground) body logically implies the (ground) head according to the given rule.

Derivation trees are defined as follows:

For programs P_1 and P_2, and a query of the form $p^a(\theta)$, we say that (P_1, p^a) and (P_2, p^a) are equivalent, if for any assignment of constants to the arguments of p that are bound in a, the two programs produce the same answer for the resulting query on p.

Theorem 7.3.1: For each p^a that appears in P^{ad}, (P, p^a) and (P^{ad}, p^a) are equivalent.

Proof: First, we note that for each rule of P^{ad}, if the adornments are dropped, we obtain a rule of P. We show that if a rule of P^{ad} is applied to some facts to produce a new fact, then the unadorned version of the rule can be applied to the unadorned versions of those facts to generate the new unadorned fact. Let a derivation tree in P^{ad} for a fact $p^a(t)$ be given. If the adornments of the predicates are dropped, this is a derivation tree for $p^a(t)$ in P, except that for nodes $\{q(c)\}$ corresponding to negated literals, subtrees for some facts $q(c)$ (for which derivation trees exist in P) are missing. If we look at the corresponding node $\{q^a(c)\}$ in the given derivation tree in P^{ad}, these missing facts are precisely those facts which do not agree with the values of the bound arguments at this node. Adding these facts does not affect the success or failure of the rule, and thus adding these facts gives us a valid derivation tree for $p^a(t)$ in P.

For the other direction, we prove by induction on the height of trees, that given a derivation tree for a fact $p^a(t)$ in P, there exists a derivation tree in P^{ad} for every fact, in particular $p^a(t)$, which occurs in this tree.

For the basis, a tree of height one is simply a base fact, hence it is also a tree in P^{ad}. Assume the claim holds for all trees of height less than n, and consider a tree of height n.

Let the root be a node $p(c)$ and let the rule used to expand it be

$$r:\ p(\theta) :- p_1(\theta_1)\ , \ldots ,\ p_k(\theta_k).$$

The child of node $p(c)$ is a node $r(c)$. Consider the children of $p(c)$. The subtree with these children as the leaves and node $p(c)$ as the root is a *derivation step* corresponding to an application of rule r.

By the construction of P^{ad}, there exists an adorned version of the same rule in P^{ad}:

transformation itself remains the same; there is no change in any of the steps.

Let P^{mg} denote a program obtained from P^{ad} by the Generalized Magic Sets transformation. We say that (P^{ad}, p^a) and (P^{mg}, p^a) are equivalent if the two programs produce the same results for every instance of the query form p^a, if the corresponding seed is added to P^{mg}.

In evaluating P^{mg} bottom-up, the constraints imposed by the semantics of negation and set grouping must be obeyed †. Consider a rule whose head uses grouping. For each value of the bound arguments in the head, the body must be fully evaluated before we can group and apply the rule to produce a tuple for the head. Since the values for bound arguments are stored in the magic predicate corresponding to the head, this implies that for each tuple in this magic predicate, the other body predicates must be fully evaluated (that is, all tuples which agree with the given tuple in the magic head predicate must be computed) before the rule is applied. Similarly, if a rule contains a negated literal $\neg p$, we must evaluate p fully for each value of the bound arguments. Since values for the bound arguments are stored in the magic predicate corresponding to p, this means that for each tuple in this magic predicate, we must compute all p tuples that agree with it.

In both cases (sets and negation), because the original rules are stratified, the predicate occurrences that must be fully evaluated do not depend on the head predicate of the rule in which they occur. However, these predicate occurrences could depend on the magic predicate corresponding to the head, and this magic predicate could in turn depend on them. This cyclicity does not pose a problem since we only need to evaluate these predicate occurrences fully for a *given* tuple in the magic head predicate: Consider one such predicate occurrence. The corresponding magic predicate might depend on the magic head predicate, but since there is no grouping in the head of a magic rule, we are not forced to evaluate the magic head predicate fully. Thus, given a tuple for the magic head predicate, we obtain all corresponding tuples for the magic predicate associated with the predicate occurrence under consideration. Using this set of tuples in the rules defining the predicate occurrence, we solve the rule fully. Thus, we have all tuples for this

† Henceforth, when we refer to bottom-up evaluation, this is implicit.

If the sip contains an arc into the first body literal, the tail can only contain the head, and thus for the first body literal, the bound arguments (if any) appear in the bound arguments of the head ($q^{a1}(c)$) of the rule. Thus, the fact corresponding to the first body literal is in the tree for P^{ad}. For the second body literal, the values in the corresponding magic literal are derived from the join of the previous magic literal and body literal. Since we just showed that the fact corresponding to the previous body literal was in P^{ad}, it follows that the bound arguments of the second body literal (the argument of the corresponding magic literal) are also imported into the second body literal according to the sip, and thus the fact corresponding to the second body literal is also in the tree for P^{ad}. Similarly, we can show that the facts corresponding to each body literal are present in the corresponding tree for P^{ad}, thus completing our proof.

The other direction is proved by induction on the height of derivation trees of facts in P^{ad}. The basis of the induction is the set of derivation trees of height zero. These are simply base facts, and they are also derivation trees for P^{mg}. Consider now a derivation tree of height n.

Let the root be a node $p^a(c)$ and let the corresponding rule be

$$r:\ p^a(\theta):-p_1^{a_1}(\theta_1),\ ,\ \ldots\ ,\ p_k^{a_k}(\theta_k)$$

Without loss of generality, let us assume that the predicates are ordered according to the ordering induced by the sip for this rule. Consider the children of $p^a(c)$. The subtree with these children as the leaves and node $p^a(c)$ as the root is a *derivation step* corresponding to an application of rule r.

By the induction hypothesis, there exist derivation trees for the nodes corresponding to these children since each of them is the root of a tree of height less than n. Note however, that for each such derivation tree the (new) seed should correspond to the fact(s) being derived. For the derivation of the fact $p_i^{a_i}(c_i)$, the seed is $magic\_p_i^{a_i}(c_i^b)$. Since these seeds are not known *a priori*, to rely on the induction hypothesis, we have to show that they can be computed in P^{mg} augmented with the seed for the original query. We claim that this is possible.

The proof of this claim is by induction on the position of the predicate occurrence in the rule body. For $magic\_p^a$, the fact that is to be derived is $magic\_p^a(c^b)$.

produce tuples for P^{bff} and so on. []

In constructing the magic rules and the modified rules, we added a number of magic predicates to the body. We now state an important lemma which shows that in each rule, some of these magic predicates may be dropped without loss of information, that is, the sets of values computed by the magic predicates remain unchanged, and the number of successful executions of the modified rules remains unchanged. †

Consider an adorned rule and a sip for it. Let us define $p \Rightarrow q$ as follows, where p and q are literal occurrences. If the sip contains an arc $N \rightarrow q$, and N contains p and p is not negated, then $p \Rightarrow q$. If $p \Rightarrow l$ and $l \Rightarrow q$, then $p \Rightarrow q$. (Clearly, we cannot have $p \Rightarrow q$ and $q \Rightarrow p$ simultaneously because the sip induces a ordering.) Let us define the *order* of q to be the length of the longest chain $q_i \Rightarrow q_{i+1} \Rightarrow \ldots \Rightarrow q$. The order is 0 if there is no such chain.

Consider the rule set P^{mg_opt} which is obtained from P^{mg} by repeated applications of the following transformation:

Let r be an adorned rule and let r' be a (magic or modified) rule generated from r. If the body of r' contains occurrences of both $magic\_p_i^{a_i}$ and $magic\_p_j^{a_j}$, and $p_i \Rightarrow p_j$, then we may delete the occurrence of $magic\_p_j^{a_j}$.

We have the following lemma.

Lemma 7.4.2: Let P^{mg} and P^{mg_opt} be as above, and let p^a be a predicate that appears in P^{mg}. Then (P^{mg}, p^a) is equivalent to (P^{mg_opt}, p^a). []

We omit the proof of the above lemma since it is identical to the proof of Lemma 5.2.2. We conclude this section with an illustrative example of the Generalized Magic Sets strategy in the presence of both negated literals and the set generation operator.

† We may drop other occurrences of magic predicates, if we wish to, without affecting the equivalence of P^{mg} to P^{ad}. In fact, if we drop *all* occurrences of magic predicates, P^{mg} reduces to P^{ad}. However, by dropping additional occurrences of magic predicates from rule bodies, we compute larger sets of values in the magic predicates, and thus increase the number of firings of the modified rules, where a *firing* is the execution of a rule to produce a head tuple.

1'. $magic\_a^{bf}$ (X) :- $magic\_a^{bf}$ (X) [From rule 2; may be deleted]
2'. $magic\_a^{bf}$ (Z) :- $magic\_a^{bf}$ (X), a^{bf} (X,Z) [From rule 2]
3'. $magic\_a^{bf}$ (X) :- $magic\_old^{bf}$ (X) [From rule 5]
4'. $magic\_sg^{bf}$ (Z1) :- $magic\_sg^{bf}$ (X), up(X,Z1) [From rule 4]
5'. $magic\_sg^{bf}$ (X) :- $magic\_old^{bf}$ (X), $\neg$ a^{bf} (X,Z) [From rule 5]
6'. a^{bf} (X,Y) :- $magic\_a^{bf}$ (X), p(X,Y) [Modified rule 1]
7'. a^{bf} (X,Y) :- $magic\_a^{bf}$ (X), a^{bf} (X,Z), a^{bf} (Z,Y)
[Modified rule 2]
8'. sg^{bf} (X,Y) :- $magic\_sg^{bf}$ (X), flat(X,Y) [Modified rule 3]
9'. sg^{bf} (X,Y) :- $magic\_sg^{bf}$ (X), up(X,Z1), sg^{bf} (Z1,Z2),
down(Z2,Y) [Modified rule 4]
10'. old^{bf} (X, <Y>) :- $magic\_old^{bf}$ (X), $\neg$ a^{bf} (X,Z),
sg^{bf} (X,Y) [Modified rule 5]
11'. $magic\_old^{bf}$ (john) [From the query rule]

In evaluating the rewritten set of rules, a certain ordering is imposed in our (strictly bottom-up) execution of the rules. We begin by executing the magic rule derived from the query, 11'. When we subsequently execute 3' we obtain a magic value for a^{bf}. By executing the rules defining a^{bf} and $magic\_a^{bf}$, we may compute (possibly many) a^{bf} tuples. These may then be used in 5' to produce magic values for sg^{bf}. From the semantics of negation, however, before we execute 5', the entire subset of the a^{bf} relation with the magic value for X must be computed. Having done this and executed 5' to produce a magic value for sg^{bf}, we may execute the rules defining sg^{bf} and $magic\_sg^{bf}$ to produce sg^{bf} tuples. These tuples may then be used in rule 10' to produce answer tuples. From the semantics of the set generation operator <>, however, before we execute 10', we must compute the entire subset of the sg^{bf} relation corresponding to the magic values of X in rule 10'. The ordering discussed above may be summarized as follows:

(11' + 2' + 3' + 6' + 7')* . (4' + 5' + 8' + 9')* . 10'

This indicates that we execute rules 11', 2', 3', 6' and 7' until no more tuples are produced; then execute rules 4', 5', 8' and 9' until no more tuples are produced, and finally execute rule 10'. This ordering is actually a little more liberal than that cate $body\_r^{bb}(\overline{c},X)$, defined as follows:

$body\_r^{bb}(\overline{c},X)$:- $magic\_body\_r^{bb}(\overline{c},X)$, body of rule r
$magic\_body\_r^{bb}(\overline{c},X)$:- $magic\_p^{a}(\overline{c},S)$, $X \in S$.

The rule defining $body\_r^{b}$ is simply the body of r with an additional magic predicate that binds X. Finally, for each predicate q in the body of r, if the body of the rule defining $magic\_q$ contains $magic\_p^{a}(\overline{c},S)$, add the literal $\neg\ bad\_r(\overline{c},S)$ also to the body. This acts as a further guard on the values $(\overline{c},S)$, indicating that we are only interested in those values for which $p^{a}(\theta)$ does not fail immediately. Thus, we only compute q entirely when the query for p^{a} does not fail immediately. The price we pay is that in the event of success, we compute the body of r twice - once in evaluating $body\_r^{bb}$ and once in evaluating rule r - for every value $X \in S$.

```
Query 1    a(X,Y) :- p(X,Y).
           a(X,Y) :- p(X,Z),a(Z,Y).
           query(X) :- a(john,X).
```

Because most strategies are representation dependent, we have studied the same example with the second attribute bound instead of the first. This will allow us to determine which strategies can solve both cases.

```
Query 2    a(X,Y) :- p(X,Y).
           a(X,Y) :- p(X,Z),a(Z,Y).
           query(X) :- a(X,john).
```

The third version of the ancestor example specifies ancestor using double recursion. This enables us to see how the strategies react to the non linear case. This example being fully symmetric, it is sufficient to test it with its first attribute bound.

```
Query 3    a(X,Y) :- p(X,Y).
           a(X,Y) :- a(X,Z),a(Z,Y).
           query(X) :- a(john,X).
```

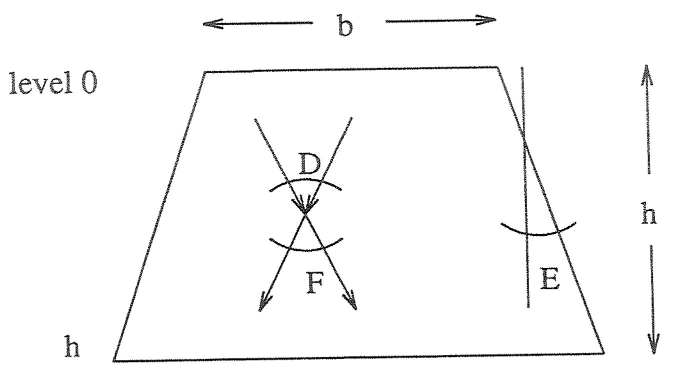
Finally, to study something more complex than transitive closure, we have chosen a generalized version of the same generation example, bound on its first attribute.

```
Query 4    p(X,Y) :- flat(X,Y).
           p(X,Y) :- up(X,XU),p(XU,YU), down(YU,Y).
           query(X) :- p(john,X).
```

1.2. Characterizing Data: Sample Extensional Databases

Because we decided on an analytical approach, we had to obtain tractable formulae for the cost of each strategy against each query. Therefore, each relation must be characterized by a *small* set of parameters. Fortunately, because of the choice of our workload, we can restrict our attention to binary relations.

We represent every binary relation by a directed graph and view tuples as edges and domain elements as nodes. Nodes are arranged in layers and each edge goes from a node in one layer to a node in the next. Note that in these graphs each node



This "parametrized structure" is fairly general and can represent a number of typical configurations:

A binary balanced tree of height k is defined by:

F=2; D=1; h=k; b=1

The same binary tree upside down is defined by:

F=1; D=2; h=k, b=2^k

A list of length k is defined by:

F=1; D=1; h=k; b=1

A set of n lists of length k is defined by:

F=1; D=1; h=k; b=n

A parent relation, where each person has two children and each child has two parents is defined by:

F=2; D=2; h=number of generations; b=people of unknown parentage

We emphasize that we assume the data to be *random*, with a uniform distribution. Thus, the values F and D are average values. Our characterization of a binary tree, for instance, describes a random (but layered) data structure in which the average values of F and D are 2 and 1 respectively. An actual binary tree has a regular pattern (*each* internal node has exactly one incoming and two outgoing edges incident on it) and this is not captured by our characterization.

Our assumption that the duplication factor is independent of the size is a very crude approximation. For instance it implies that if you start from one node you still generate some duplicates. Obviously the duplication factor increases with the size of the start set. Therefore, our approximation overestimates the number of duplicates. However, it becomes reasonable as the size of the start set becomes large. It is also arguments (each tuple present in both argument is going to fire twice); and the measure of complexity of projection is the size of the argument. Readers familiar with performance evaluation of relational queries might be surprised by these measures.

Our concern, however, is primarily with recursive queries. In particular, all but one of our queries (ancestor using double recursion) are *linear*, that is, the body of each recursive rule contains exactly one occurrence of the recursive predicate. We justify our measurement of only successful inferences by the observation that the number of successful inferences (for the recursive predicate) at one step constitutes the operand at the next step. We justify the approximation in estimating the cost of a join in terms of the size of just one of the operands as follows. The join represented by the predicates in the body of a rule may be thought of as a fixed ''operator'' which is repeatedly applied to the relation corresponding to the recursive predicate. It is reasonable to assume that the cost of each such application is proportional to the size of this relation (the operand). By measuring the size of this intermediate relation over all steps, we obtain a cost that is proportional to the actual cost.

In essence, our cost is a measure of one important factor in the performance of a query evaluation system, the number of successful inferences, rather than a measure of the actual run-time performance. This cost model is studied further in [Bancilhon 85].

2. Notation and Preliminary Derivations

In this section, we explain the notation and terminology used in analytically deriving the cost functions. We also derive some expressions that are used in the analysis of some of the strategies. The derivations of these expressions are of some interest in their own right, since they are good examples of the techniques we use in subsequent analyzes.

We denote multiplication by simply juxtaposing the operands. Where there is ambiguity, parentheses are used to clarify the expression, or we use * to denote multiplication.

age we can reach E distinct nodes by a path of length 1, E^2 distinct nodes by a path of length 2, and so on. The number of arcs of length k going from level i to level (i+k) is thus:

$$n(i)E^k = n(i+k)$$

Of course, if D is not one, this is an approximation which depends on our assumption of random data. In particular, it breaks down for regular data, such as an actual inverted tree. The intuition is as follows. The parameters F and D are used to estimate the number of *arcs* of length k, as opposed to the number of *paths* of length k. Several paths may generate the same arc (that is, they have the same end points). Thus, we use the parameters F and D to estimate this ''duplication'' of arcs. This approximation depends upon the randomness of data - in an inverted tree, for instance, the number of paths is exactly the number of arcs because there is a unique path between any two points. The inverted tree is one instance of a family of data structures with given values of F (=1) and D (=2), and in this particular instance, due to the regular pattern in the data, the above approximation breaks down. In general, however, for such a structure the number of paths is *not* equal to the number of arcs; and if the data is randomly (and uniformly) distributed, our approximation is accurate.

We denote by $a_{R^*}(k)$ the number of arcs of length exactly k in R*. Where the context is clear, we write a(k).

a(k) is obtained by summing all the arcs of length k that enter level i for i = k to h. Thus:

$$\begin{aligned} a(k) &= n(k) + n(k+1) + \ldots + n(h) \\ &= n(k)\ \mathrm{gsum}(E,h-k) \end{aligned}$$

Finally, given a relation R(A,B), its transpose R^T(B,A) is defined to be such that R^T(B,A) holds if and only if R(A,B) holds, for all pairs (A,B). We have the following relationships:

assume that we perform duplicate elimination at the end of each step, a given arc is computed D times at each step. To see this, consider an arc which is extended in a given step. The step essentially extends this arc by adding an edge to one end point. The number of new arcs generated is the number of new nodes reached, and there is a duplication factor D in this set of nodes.

Finally, we fire R3 once to produce the answer. The number of successful firings of R3 is the size of the answer set. This is equal to the number of nodes in the subtree rooted at 'john', and is thus:

$$= E + E^2 + \ldots + E^{h'}$$
$$= (E)\text{gsum}(E,h'-1)$$

Thus the total number of successful firings is:

$$D\sum_{i=1}^{h}(h-i+1)a(i) + (E)\text{gsum}(E,h'-1).$$

3.1.2. Semi-Naive Evaluation

The performance of the Semi-Naive strategy is similar to Naive Evaluation, with the difference that we don't recompute arcs. At each step, we use for relation *a* only the tuples that were computed for it in the previous step. Thus, at step i we generate all arcs of length exactly i, with a duplication rate of D.

The number of successful firings is therefore:

$$D\sum_{i=1}^{h}a(i) + (E)\text{gsum}(E,h'-1).$$

3.1.3. QSQ, Iterative

At step 1, we compute all arcs in *p* (that is, arcs of length 1 in the transitive closure of *p*) leaving 'john'. At step 2, we compute all arcs leaving nodes that are reachable from 'john' by an arc of length 1, and also recompute all arcs of length 1 leaving 'john'. At step 3, we compute all arcs leaving nodes that are reachable from 'john' by an arc of length 2. We also compute all arcs in the transitive closure of the tree of height 2 rooted at 'john' (and in doing so, duplicate all the work done in step 1). At step j, we compute all arcs leaving nodes that are reachable from 'john' by an

As before, we also assume a duplication factor of D. Finally, rule R3 is fired to produce the answer, the cost being the size of the answer set.

The total cost is therefore:

$$(E)\mathrm{gsum}(E,h'-1) + D\sum_{i=1}^{h'} E^{i}\mathrm{gsum}(E,h'-i)$$

3.1.5. Henschen-Naqvi

The Henschen-Naqvi method evaluates the sequence of expressions:

p(john,X)
p(john,X1),p(X1,X2)
p(john,X1),p(X1,X2),p(X2,X3)
etc.

At each step, the previous expression is extended by one join. The cost of doing this (assuming duplicate elimination) is:

$$\begin{aligned}&ED + E^{2}D + \ldots + E^{h'}D\\ &= (DE)\mathrm{gsum}(E,h'-1)\\ &= (F)\mathrm{gsum}(E,h'-1)\end{aligned}$$

We also fire R3 to produce the answer. Thus the total cost is:

$$(F+E)\mathrm{gsum}(E,h'-1)$$

3.1.6. Prolog

In this case, Prolog performs very much like QSQR, setting up the same subqueries. But since Prolog does not perform duplicate elimination, the subquery at a given node is solved as often as there are paths to the node. This is equivalent to QSQR operating on a similar data structure with E = F and D = 1. Thus, the cost is similar to that for QSQR, with F substituted for E:

$$(E)\mathrm{gsum}(E,h'-1) + \sum_{i=1}^{h'} (F^{i})\mathrm{gsum}(F,h'-i)$$