Magic Templates: A Spellbinding Approach to Logic Programs *

Raghu Ramakrishnan

University of Wisconsin-Madison

ABSTRACT

We consider a bottom-up query-evaluation scheme in which facts of relations are allowed to have nonground terms. The Magic Sets query-rewriting technique is generalized to allow arguments of predicates to be treated as bound even though the rules do not provide ground bindings for those arguments. In particular, we regard as ''bound'' any argument containing a function symbol or a variable that appears more than once in the argument list. Generalized ''magic'' predicates are thus defined to compute the set of all goals reached in a top-down exploration of the rules, starting from a given query goal; these goals are not facts of constants as in previous versions of the Magic Sets algorithm. The magic predicates are then used to restrict a bottom-up evaluation of the rules so that there are no redundant actions; that is, every step of the bottom-up computation must be performed by any algorithm that uses the same sideways information passing strategy (*sips*). The price paid, compared to previous versions of Magic Sets, is that we must store relations with nonground facts; and we must perform unifications, rather than equijoins, when evaluating the rules bottom-up. The method is applicable to general Horn clause logic programs.

1. Introduction

Several strategies have been proposed for evaluating recursive queries expressed using sets of Horn clauses [BeR87, BMSU86, DW87, GMN84, HN84, KL86a, KL86b, KL87, Na87, Na88, RL85, SZ86a, SZ86b, Ul85, VG86, Vi86, Vi88, etc.]. (See [BaR86, BaR88] for a survey and performance comparison.) It is our thesis that each of these strategies has two distinct components - a *binding passing* or *sideways information passing* strategy (*sip*) for each rule and a control strategy. A sip represents a decision on how information gained about some literals in a rule is to be used in subsequently evaluating other literals 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 any sip collection by first rewriting the given set of Horn clauses and then evaluating the rewritten set.

Our result generalizes Generalized Magic Sets and other related strategies presented in [BeR87]. The importance of the generalization is that the bottom-up methods now apply to *any* Horn clause logic program. The results presented here clarify the relationship between top-down strategies, such as that used in Prolog, and bottom-up strategies by showing that the execution of a top-down strategy can be ''mimicked'' by rewriting the program and then evaluating the fixpoint bottom-up. The rewriting introduces auxiliary ''magic'' predicates, and the ''magic'' facts (computed during the bottom-up evaluation) correspond to the goals that would be generated by the top-down strategy. Every other fact that is generated by the bottom-up evaluation is also generated by the top-down strategy in solving these goals. (We do not consider programs with extra-logical predicates such as *assert* and *retract*, or the *cut*, or programs containing negation. The approach can be used to evaluate some *stratified programs* containing negation, and we refer the reader to

* This work was supported in part by an IBM Faculty Development Award and NSF grant IRI-8804319.

[BNRST87], [BPR87].)

We believe that both Prolog-style evaluation and bottom-up approaches have their own advantages and are likely to be preferable in certain important domains. In this paper, we show the power of the bottom-up approach, and, by a careful separation of binding propagation and the flow of control (following [BeR87]), we also clarify the relationship between top-down and bottom-up evaluation strategies.

We also consider how these techniques can be used to implement Constraint Logic Programs ([JL87], [JM87], [HJLMSYY86]). For example, consider the following rule:

p(X,Y) :- X > Y.

This says that for all values x and y in the domain of the program, p(x,y) is true if the constraint x > y holds. The difficulty here is that this information cannot be represented as a fact in the relation associated with the predicate *p*. Rather, each fact must now be viewed as conditional upon an associated set of constraints. It is assumed that a *constraint solver* is given, and the problem is therefore one of collecting appropriate sets of constraints as the computation progresses.

The paper is organized as follows. Notation and definitions are introduced in the rest of this section. We define sips in Section 2. In Section 3, we consider nonground facts, and present an overview of our approach. In Section 4 we describe how to obtain an *adorned* program, and in Section 5, we present the Magic Templates algorithm for further rewriting the adorned program. We discuss several aspects of this algorithm through examples. In Section 6, we characterize the relationship between rewritten programs and the sips used in rewriting them, and also illustrate some aspects of safety and termination.

1.1. Preliminary Definitions

We assume the usual definitions of Horn clause rules, terms and literals. A *program* is a finite set of rules, $P = \{ r_1, ..., r_n \}$. A database D is a finite set of finite *relations*. A *relation p* is a collection of (possibly nonground) facts $p(t)$. Hence $P \cup D$ is a logic program. Predicates that name database relations are called *base* predicates; all other predicates are called *derived*. Without loss of generality, we will assume that no base predicate appears in the head of a rule in *P*, that is, the set of base predicates is disjoint from the set of derived predicates. This separation of program from database allows us to consider equivalence of programs with respect to all possible databases. In the following, Greek letters such as θ and ϕ are used to denote a vector of arguments. We also use these letters to denote substitutions, and when we do so, we state this explicitly. The *projection* of a vector of arguments θ on some subset of the set of argument positions is defined to be the vector of arguments in those positions.

A *substitution*, $\theta = \{ V_1 \leftarrow t_1, V_2 \leftarrow t_2, ... , V_n \leftarrow t_n \}$, assigns terms to variables. It is applied to a syntactic object o (term, rule, etc.) by simultaneously replacing all occurrences of each V_i in o by t_i, and the result $o\,\theta$ is called an *instance* of o. The set of variables $\{V_1, ... , V_n\}$ is called the domain of θ, and the set of variables appearing in $\{t_1, ... , t_n\}$ is called the range. We shall only consider substitutions such that *domain* $\cap$ *range* is empty, and this is understood unless otherwise stated in the rest of this paper. Such substitutions are idempotent [LMM88]. An element $V_i \leftarrow t_i$ of a substitution is called a *binding*. We use the notation *vars*(o) to denote the set of all variables in the object o. (We follow [LMM88] in this presentation.)

A *unifier* for two terms t_1 and t_2 is a substitution σ such that $t_1\,\sigma = t_2\,\sigma$. We call a unifier σ a *most general unifier* (mgu) of t_1 and t_2 if it has the following property: Let θ be any unifier for t_1

-- --

and t_2. Then, there is some substitution α such that $t\ \theta = t\ \sigma\ \alpha$, for all terms t.

Given a program-query pair $(P,\ Q)$ and a database D, the result of applying $(P,\ Q)$ to D, which we also refer to as the *set of answers* to the query on D, is the set of all facts q which are instances of the query Q, and are logical consequences of $P \cup D$. (See [VK76, LM83] for fixpoint characterizations of the set of answers.) We say that two programs with queries $(P,\ Q)$ and $(P',\ Q')$ are *equivalent* if, for every database D, $P \cup D$ and $P' \cup D$ produce the same answers for their respective queries.

2. Sideways Information Passing

The notion of sideways information passing was studied in [BeR87], and we refer the reader to it for a more detailed discussion. The discussion in this section follows the presentation there, but with some important differences. The only bindings considered there were of the form X = c, where X is some variable and c is some ground term. As we show later, this involves no loss of generality given the requirement that every variable in the head of a rule should also appear in the body. Without this requirement, which we relax in this paper, this definition of bindings precludes some information passing strategies. We therefore consider arbitrary bindings of the form X = t, where t is any term.

A *sideways information passing strategy*, henceforth referred to as a *sip*, is an inherent component of any query evaluation strategy. Intuitively, a sip describes how we evaluate a rule when a given set of head arguments are bound (through unification with a goal, to possibly nonground terms or variables). Our use of the term ‘‘bound arguments’’ is perhaps more accurately described by the term ‘‘potentially restricted’’. In particular, some confusion may arise since ‘‘bound’’ is used in the logic programming and deductive database literatures in a somewhat different sense.

Let r be a rule, with head predicate $p(\theta)$, and let θ^b denote the projection of θ on the arguments that are bound when the rule is invoked. The arguments that are not bound are called the *free* arguments. Thus, if the head literal is $p(X, [5 | Y], Z, W)$, and we assume that the first two arguments are bound when the rule is invoked, θ is $(X, [5 | Y], Z)$, and θ^b is $(X, [5 | Y])$. Similarly, we define θ^f to be the projection of θ on its free arguments - θ^f is (Z, W) in the above example - and follow these conventions in the rest of the paper. It is possible that a rule is invoked with more than one pattern of bound arguments. We analyze each rule for each pattern of bound arguments with which it is invoked, and the details should become clear as we proceed. Let p_h be a special literal, denoting the head literal restricted to its bound arguments. Thus, the arguments of p_h are θ^b. Let $P(r)$ denote the set of literals in the body. A sip for r, for a given set of bound head arguments, is a labeled graph that satisfies the following conditions:

1. Each node is a subset of $P(r) \cup \{p_h\}$.
2. Each arc is of the form $N \rightarrow \{q\}$, with label S, where N is a subset of $P(r) \cup \{p_h\}$, q is a member of $P(r)$, and S is a set of variables, such that each variable of S appears in q.
3. There exists a partial ordering of the literals of $P(r) \cup \{p_h\}$ such that for each arc, all literals at its tail precede the literal at its head, and such that the literals that do not appear in the sip follow all those that appear in the sip.

Since the head of a sip arc is always a singleton set, we omit the curly braces. We explain the meaning of such a graph by explaining how the computation of a rule uses an arc $N \rightarrow_S q$, † and

† For simplicity, we assume that there is at most one arc entering a given node q. Multiple arcs can be dealt with as in [BeR87].

then explaining the complete computation of a rule. This arc means that the variables in S are bound to terms by solving the literals in N (a ''join'' in database terminology). Thus, if we consider an arc $p1(W,X,Y),\ p2(Y,Z) \rightarrow_{X,Z} p(X,Z,[Z\,|\,U],V)$, and facts $p1(1,[2|3],3)$ and $p2(3,4)$, we have the binding X = [2|3], Z = 4. A binding for each variable in S partially instantiates the literal q. The sip is to be interpreted as requiring that if a q-fact is a substitution instance of this partially instantiated literal, and further, is a logical consequence of the program and database, then it ought to be inferred. Thus, in our example, we have $p([2|3],3,[3\,|\,U],V)$, and every p-fact that is a logical consequence of the program and is a substitution instance of this p-fact ought to be inferred. We observe that each partially instantiated q-fact corresponds to a goal in a top-down Prolog-style execution, and the definition of a sip requires that each such goal should be identified and all answers to it should be inferred. The sip provides a potential restriction of the set of facts to be computed, since we are not required to compute any other q-facts in solving this literal. When all the literals in the body of a rule have been solved, we obtain an instance of the rule, and the instantiated head literal is a logical consequence of the program and database. This is how new facts are inferred.

Suppose that we want to use the rule r, with some arguments of the head predicate bound. If an argument is designated as bound, it means that we are only interested in evaluating the rule when this argument is bound to one of a set of terms. The special node p_h may be thought of as a relation with fields corresponding to the bound arguments of the head predicate. Each tuple in it corresponds to a vector of bindings that is given for these arguments. Intuitively, each tuple contains the vector of bindings for the bound arguments in some call of this adorned predicate in a Prolog-style execution.

We emphasize that the above discussion of the interpretation of a sip is to be understood as an abstraction that conveys *what* is done rather than *how* it is done. For example, Prolog does not create special predicates p_h to store bindings for head arguments, nor does it explicitly evaluate the joins we mentioned. These operations are implicit in the way Prolog maintains variable bindings through unification and backtracking. The above interpretation of a sip forms the basis for the definition of a *sip−strategy* in Section 6.

Example 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).
Query: sg(john,X)?

This is a non-linear version of the same-generation program [BeR87, BMSU86]. 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:

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

We focus on the evaluation of derived predicates and we therefore ignore bindings that are passed to base predicates (and which can be used to retrieve a subset of the corresponding relations) for simplicity of exposition. []

The previous example introduced the notion of sips, and the sips in it satisfy the earlier definition of a sip in [BeR87]. The next example illustrates some differences.

Example 2:

p(X,Y) :- same(X,Y), q(X,Y).
same(X,X) :-
q(5,X) :-

This program would be disallowed in earlier approaches since the last two rules are not range-restricted. (A rule is *range-restricted* if every variable in the head appears in the body.) However, consider a query p(X,Z)?. This could be solved by solving the goal same(X,Y) to bind Y (to X) and then solving the goal q(X,Y) to bind X (and therefore Y) to 5. This is in fact what Prolog does, and the following sip describes this:

$\{same\} \rightarrow_{X,Y} q$

While this is a valid sip according to the definition presented in this paper, it is not a valid sip according to the definition in [BeR87] since X and Y are not bound to ground terms in the arc entering *q*. []

In the second condition defining a sip, if $N \rightarrow p$ is a sip arc with label *S*, we require that the variables in *S* should appear in arguments of *p*. This is simply a consequence of the fact that bindings for other variables are of no interest when evaluating that predicate occurrence. However, we depart from the definition of a sip in [BeR87] in an important way - we no longer require that these variables also appear in arguments of predicates in *N*. Variables that appear in *S* but not in *N* are essentially free variables, but as we see in the following example, they could still restrict the set of *p*-facts that we must compute.

Example 3:

p(X,Y,Z) :- b(X,U,V), b(W,Y,Z).

Let *b* be a base relation. The first two fields of *p* contain the crossproduct of the first and second fields of *b*. Consider the following rule:

q(X) :- p(X,X,Y).

Let the query be q(X)?, and let these be the only rules in the program. We only need to compute *p*-facts that have the same value in the first two arguments. This can be expressed using the following sip for the second rule:

$\{\ \} \rightarrow_{X} p$

If the relation *b* is large, the reduction in the number of inferences is significant. In later sections, we see how this sip can be implemented using a bottom-up fixpoint computation. However, this sip is clearly invalid according to the definition in [BeR87], and further, it cannot be implemented using the algorithms presented there. (As we will show later, a bottom-up implementation of this sip requires us to introduce rules that are not range-restricted.) This example illustrates the more general notion of ‘‘binding’’ used in this paper, in contrast to the earlier work (where ‘‘bound’’ meant ‘‘bound to a ground term’’). []

An important difference between our definition of a sip and the earlier definition has to do with *partially bound* arguments (i.e. arguments in which some variables are bound while others are free). The following example illustrates this.

Example 4:

```
p(X) :- q1(X,Y), q2([Y|Z]).
q1(2,0) :- .
q2([Y|U]) :- q1(Y,U).
```

This program illustrates a sideways information passing strategy that is disallowed in [BeR87] because it cannot be efficiently implemented using the rewriting methods presented there. (This reflects a decision that is followed in most of the deductive database literature, and was first discussed in [Ul85].)

Consider a query p(X)? The following is a possible sip for the first rule:

$\{q1\} \rightarrow_Y q2$

Thus, we compute the entire relation $q1$, but only a subset of $q2$. This is not a valid sip according to the definition in [BeR87] since Z is not bound in the (only) argument of $q2$. []

2.1. Full Sips

Consider a rule r with a given set of arguments bound in the head literal p. There are, in general, many sips that we could choose, and we now identify an important class of sips, called *full sips*. A full sip is a sip in which: (i) there is a unique total ordering induced by the sip over all literals in $P(r) \cup \{p_h\}$, and (ii) there is exactly one arc $N_i \rightarrow_S \{q_i\}$ entering the singleton node associated with each body literal q_i; N_i contains all predecessors of q_i, and S contains all variables in q_i.

A full sip indicates that (the body literals are solved in the sip-induced order, and) in solving a literal, all bindings made in solving preceding literals are used to restrict the goal. For each body literal, every argument is considered bound. Thus, there is a unique full sip associated with each distinct total ordering of the literals in $P(r)$, For example, Prolog uses a full sip with a left-to-right ordering of literals. The results in this paper hold for arbitrary choices of sips, but the case of full sips for every rule simplifies some details and shows the intuition more clearly. We return to this point at the end of sections 4 and 5.

3. An Overview of the Proposed Evaluation Strategy

This section has two main objectives: (i) to extend definitions of ‘‘rule application’’, ‘‘duplicate elimination,’’ and other deductive database concepts to the case of nonground terms, and (ii) to present an overview of our approach to evaluating logic programs.

We remarked earlier that the methods presented in this paper generalize previous work. Consider a program, possibly obtained from a given program and query after applying some program transformations, that is to be evaluated. The important restriction that is now relaxed is the requirement that (in the final program) every variable in the head of a rule also appear in the body. A central feature of this generalization is our (revised) notion of a *tuple*. A tuple is usually defined to be a *ground* fact in a (derived or) base relation. We allow tuples to be possibly nonground facts; so that tuple and fact are the same, for the purposes of this paper. It is easy to see that requiring rules to be range-restricted ensures that no relation contains a nonground fact.

Lemma 3.1: If every rule in a program is range-restricted, then every fact $p(t)$, where p is a predicate appearing in the program, is ground.

Proof: By induction. As the basis, every fact in a database predicate must be ground, since they are range-restricted rules with empty bodies. If all facts derived in n or less steps are ground, consider the derivation of a fact in n+1 steps. Consider the last step (i.e. rule application). Every fact in the rule body must be ground, by the hypothesis. It follows that the head fact must be ground since every head variable appears in the body, and is thus bound to a ground term. []

The notion of nonground facts is standard in the logic programming literature. To our knowledge, however, the first use of such facts in the deductive database literature is in [KL87]. In that paper, Kifer and Lozinskii present an evaluation method called Sygraf, and allow rules that are not range-restricted, with the objective of dealing with general logic programs. The method does not always fully restrict the search, but it should be possible to combine the ideas with more a sophisticated version of the method, called Dynamic Filtering, defined earlier for range-restricted rules [KL86b].

A *rule application* is defined as follows. Consider a rule

$r:\ h \text{ :- } b_1, b_2, \ldots, b_k$

Let θ be a substitution such that for each body literal b_i, there is some fact f_i and substitution σ_i such $b_i\,\theta = f_i\,\sigma_i$. Further, if i ≠ j, $vars(f_i) \neq vars(f_j)$. Then, rule r can be applied on the set of facts $\{f_1, \ldots, f_k\}$ to generate the fact $h\,\theta$.

Consider a logic program $P \cup D$. We compute the least fixpoint of this program as follows. The set of *known facts* is initially the set of facts in D. We repeatedly apply the rules in P to the set of known facts and add the generated facts to the set of known facts, until no fact can be generated which is not an instance of some known fact.

In practice, we can perform a generalized *join* operation on the relations (containing the known facts) corresponding to the body literals, to generate a set of facts for the head. Thus, choosing the body literals in some order, we unify each literal with a known fact, and apply the resulting substitution (the *most general unifier*, or mgu) to the rule. The composition of the mgus for all the body literals, in order, corresponds to the substitution θ in our description of rule application.

A fact containing variables is used to denote a (potentially infinite) set of facts. The set consists of those facts that can be generated by a substitution that binds each variable X_i in the given fact to some term t_i. The set is infinite if terms can be constructed using function symbols or if the domain contains an infinite set of constants or variables. When a rule is used to generate a new fact, the scope of a variable in it is precisely this fact. That is, facts are equivalent upto variable renamings. Note that we do not maintain an environment of variable bindings, as for example, is done in Prolog implementations, where a given variable may be ‘‘bound’’, or instantiated, to different terms in the course of program execution.

A fact t_1 is *more general* than a fact t_2 if there is some substitution of terms for the variables in t_1 that makes it identical to t_2. Thus, p(X,[Y|Z]) is more general than p(5,[6|U]), but is not more general than p(5,U). p(U,V) is more general than p(X,[Y|Z]), but p(5,V) is not. Also, p(X,Y) is more general than p(U,V), but p(X,X) is not. (It may appear a little strange that p(X,Y) is more general than p(U,V), but this is simply a consequence of the way we use this property. If p(U,V) is a newly generated fact, and p(X,Y) is an existing fact, we only need to recognize that all the facts denoted by p(U,V) are also denoted by p(X,Y). We don’t need to know that they denote the same set of facts.)

When a fact is generated by applying a rule, we check whether the set of facts denoted by it is a subset of the set of facts denoted by a previously generated fact. If not, we add the new fact to the set of known facts. We observe that this operation is a generalization of *duplicate elimination* in database operations. We also note that it is introduced only to allow us to detect termination. It does not affect the completeness of bottom-up evaluation. It is possible to devise other termination conditions as well. For example, we could test whether some fact in *p* is identical to (rather than more general than) the newly generated fact, upto a renaming of variables. That is, we check if the two facts denote the same set of facts. While this can be checked more efficiently, it fails to detect termination in some cases. We discuss the termination issue in more detail in subsequent sections.

We now outline our approach to evaluating logic programs. The basic idea is to rewrite the given program and then evaluate the fixpoint of the rewritten program by repeatedly evaluating each rule (bottom-up) until no new facts are produced.† The objective of the rewriting is to produce a program whose bottom-up evaluation reflects the sips chosen for the original program (that is, only facts that agree with the values passed through sip arcs should be generated). The rewriting algorithms we consider proceed in two phases. Given a logic program and a query, we first produce an *adorned* program. The generation of the adorned program is done in conjunction with the choice of sips. Next, the final program is produced by a rewriting algorithm from the adorned program and sips. We present a generalization of the Generalized Magic Sets rewriting algorithm ([BMSU86, BeR87]), called the *Magic Templates* algorithm. (The generalization is actually at the level of sips and adornments. The rewriting algorithm is not affected, but we give it a different name to indicate the changed nature of the sips and adorned programs that it accepts and the final programs that it produces.)

4. 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*. Consider a literal *p* in the body of a rule, where *S* is the set of variables in the labels of sip arcs entering this literal. By solving the literals in the tails of these arcs, bindings are generated for the variables in *S*, and we wish to compute only those facts for *p* that match these bindings. Thus, if a variable X in *S* appears in some argument of *p*, we are only interested in facts in which this argument is bound to some term such that X is bound to one of the bindings passed through the sip. The set of interesting bindings for this argument is therefore potentially restricted, and we designate such an argument as a *bound argument*. (It is important to note that the solution of the predicates in the tail of a sip arc may leave a variable in the label free - in particular, the tail may be empty - and such a variable is still designated as bound.)

Let a program *P* and a query $q(\theta)$? be given. We construct a new, adorned version of the program, denoted by P^{ad}. 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 details are similar to the corresponding algorithm in [BeR87], but with some important differences that we discuss after presenting the algorithm.

The process starts from the given query. The query determines bindings for *q*, and we replace *q* by an adorned version, in which precisely the positions bound in the query are designated as bound, say q^e. In particular, we may treat as bound any argument position that contains a function symbol, constant or a repeated variable, since each of these could potentially restrict the computation of the answers. As we proceed, we have a collection of adorned predicates, and as

† *Seminaive* fixpoint computation is a refinement that avoids repeating inferences in different iterations. (See, e.g., [Ba85]. Note that non-linear rules are also treated.)

each one is processed, we will mark it, so that it will not be processed again. If p^a is an unmarked 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}; then p is marked as processed. The adorned version of a rule contains additional adorned predicates, and these are added to the collection, unless they already appear there. 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 finite.

Let r be a rule in P with head p. We generate an adorned version, corresponding to an (unmarked) 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 adornment a; that is, every argument in p_h for the sip is bound according to a. Thus, 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 a derived predicate in the rule, let S_i be the union of the labels of all arcs coming into p_i. (If there is no arc coming into p_i, let S_i denote the empty label.) We replace p_i by $p_i^{a_i}$, where an argument of p_i is bound in a_i only if some variable appearing in it is in S_i or it contains a constant.

Example 5: (From [BeR87]) The following is the adorned rule set corresponding to the nonlinear same generation example, for the sip of Example 1.

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}(Z,Z4), down(Z4,Y).

Query: sg^{bf}(john,Y)?

We will use these adorned rules to illustrate the rule rewriting algorithms presented later. []

Example 6:

The adorned rule set corresponding to the program and sip in Example 2 is the following:

p^{ff}(X,Y) :- $same^{ff}$(X,Y), q^{bb}(X,Y).
$same^{ff}$(X,X) :- .
q^{bb}(5,X) :- .

The adorned program contains a rule that is not range-restricted (rule 2). This program is disallowed in [BeR87]. []

Example 7:

The adorned rule set corresponding to the program and sip in Example 3 is the following:

q^f(X) :- p^{bbf}(X,X,Y).
p^{bbf}(X,Y,Z) :- b(X,U,V), b(W,Y,Z).

This adorned program would not be produced at all by the algorithm in [BeR87]. Only the adorned version p^{fff} would be considered for the occurrence of p in the first rule, since X is not bound to the left of p in the rule. (As we will see later, this leads to a rule that is not range-restricted in the rewritten ‘‘magic’’ program.) []

Example 8:

The adorned rule set corresponding to the program and sip in Example 4 is the following:

p^f(X) :- q1(X,Y), $q2^b$([Y|Z]).
q1(2,0) :- .

$q2^b$([Y|U]) :- q1(Y,U).

The following variant of the program in Example 4 illustrates a subtle point:

p(X) :- q1(X,Y), q2([Y|Z]).
q1(2,0) :- .
q2([Y|U]) :- q1(Y,Z), q2(U).
q2([]) :- .

Let the same sip be chosen for the first rule. Thus, the adornment $q2^b$ is reachable, that is, it appears in P^{ad}. Consider the recursive rule defining $q2$. With $q2^b$ as the head, we can choose the following sip for this rule: $q2_h \rightarrow_U q2$, and thereby bind the argument of $q2$ in the body of this rule. This gives us the following adorned program:

p^f(X) :- q1(X,Y), $q2^b$([Y|Z]).
q1(2,0) :- .
$q2^b$([Y|U]) :- q1(Y,Z), $q2^b$(U).
$q2^b$([]) :- .

The adornment that we use for the occurrence of $q2$ in the body of the second rule is optimistic. The adornment $q2^b$ was chosen in the first rule since Y was bound in [Y|Z]. By using this adornment for the head of the second rule, we (optimistically, and alas, mistakenly) assume that U is bound. (Technically, it is bound to a free variable. Recall that ‘‘bound’’ is to be understood as ‘‘potentially restricted’’. As we will see, the set of answers is still computed correctly.) []

In general, it is important to remember the sips that were used to generate the adorned program, since they are used in the subsequent rewriting. Note that a single adorned version of a rule is chosen for each adorned version of the head predicate. Thus, all goals whose binding pattern matches the adornment in an adorned head predicate are solved using the same adorned version of the rule, chosen at compile time.

4.1. The Case of Full Sips

The number of adorned versions of a n-ary predicate p is 2^n. However, the number of adorned versions that are reachable from the query, i.e., that appear in P^{ad}, is typically much smaller. In particular, if a full sip is chosen for each rule in the program P, all adornments are strings of *b*s. Thus, P and P^{ad} are identical, with the understanding that in P^{ad}, each n-ary predicate is adorned with a string of n *b*s.

5. Magic Templates

Henceforth, we only consider the adorned set of rules, P^{ad}. The next stage in the proposed transformation is to define additional predicates that 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 applies only when values for these additional predicates are available. These auxiliary predicates are called *magic predicates* and the sets of facts that they compute are called *magic templates*. † The intention is that the bottom-up evaluation of the modified set of rules simulate the sip that we have chosen for each adorned rule, thus restricting the search space.

† Hence the name for the strategy. The earliest version was called Magic Sets [BMSU86], and a generalization to arbitrary range-restricted rules is called Generalized Magic Sets [BeR87].

The rewriting algorithm to be described in this section is from [BeR87]. However, the evaluation of the rewritten programs, the notion of program equivalence, and the proofs of the theorems and lemmas characterizing the rewritten programs, all differ due to the different interpretations of sips and adornments presented in this paper, and must be reconsidered. In order to keep this paper self-contained, we present below the description of the rewriting algorithm:

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).

3. Each rule (with head, say, $p^a(\theta)$) in P^{ad} is *modified* by the addition of the literal $magic\_p^a(\theta^b)$ to the body.

4. We create a *seed* fact $magic\_q^e(\phi^b)$ from the query $q^e(\phi)$?.

We now explain the second step in more detail. Consider the adorned rule:

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

Let s_r be the sip associated with this rule. Assume that the predicates in the body are ordered according to the sip. (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 the special predicate denoting the bound arguments of the head is in *N*, we add $magic\_p^a(\phi^b)$ to the body of the magic rule. † (Note that every argument of the magic predicate corresponds to a *b* in adornment a_i, and that the *f*s, which correspond to free arguments in $q_i^{a_i}$, do not correspond to any arguments of the magic predicate. Thus, $magic\_q_i^{a_i}$ is to be thought of as ‘‘the magic predicate of $q_i^{a_i}$’’, rather than as a predicate $magic\_q_i$ with adornment a_i.)

The restriction on programs in [BeR87] can be stated in terms of the rewritten programs produced by the rewriting algorithms: ‘‘Every rule in the rewritten program should be range-restricted.’’ The restrictions on sips (in particular, the restriction that in a sip arc $N \rightarrow_S p$, every variable in *S* should appear in *N*) and adornments (the restriction that partially bound arguments should be considered free arguments) in [BeR87] are necessary to ensure this. We now illustrate the rewriting algorithm through several examples.

Example 9: Using the sips presented in Example 1, the Generalized Magic Sets strategy rewrites the adorned rule set corresponding to the non-linear same generation example into the following set of rules. (The rule numbers refer to the adorned rule set.)

$magic\_sg^{bf}$(john) [Seed; 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]

† We do not consider the case of multiple arcs entering a predicate; see [BeR87].

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] []

In the above example, we observe that some joins are repeated in the bodies of the rules defining $magic\_sg^{bf}$ (that are generated from rule 2) and the body of rule 2. There is a variant of Magic Sets, called Supplementary Magic Sets, presented in [BeR87], that avoids this duplication of effort by storing the results of these joins, after projecting out unnecessary arguments. We remark that the Supplementary algorithm is easily generalized along the lines indicated in this paper. Although we focus on the Magic Templates algorithm in this paper since it is easier to see the intuition behind it, the Supplementary variant may well be the rewriting algorithm of choice.

Example 10:

The rewritten program corresponding to the adorned program in Example 7 is the following:

p^{ff}(X,Y) :- $same^{ff}$(X,Y), q^{bb}(X,Y).
$same^{ff}$(X,X) :- .
q^{bb}(5,X) :- $magic\_q^{bb}$(5,X).
$magic\_q^{bb}$(X,Y) :- $same^{ff}$(X,Y).

The rewritten program contains a rule that is not range-restricted (rule 2). This program is disallowed by the restrictions in [BeR87]. The execution of this program proceeds as follows. The last rule can be applied to generate $magic\_q^{bb}$(U,U). The third rule can then be applied to generate q^{bb}(5,5), and the first rule can then be applied to generate p^{ff}(5,5). We observe that no other facts are generated, and the program then halts. []

Example 11:

The rewritten program corresponding to the adorned program in Example 8 is the following:

q^{f}(X) :- p^{bbf}(X,X,Y).
p^{bbf}(X,Y,Z) :- $magic\_p^{bbf}$(X,Y), b(X,U,V), b(W,Y,Z).
$magic\_p^{bbf}$(X,X) :- .

As we mentioned earlier, this program has a rule that is not range-restricted (the third rule). This is because the sip contained an arc ({ } $\rightarrow_X$ p) in which a variable in the label (X) did not appear in the tail of the arc. Since such arcs always lead to rules that are not range-restricted in the rewritten program, such arcs were disallowed by the definition of sips in [BeR87]. []

Example 12:

The rewritten program corresponding to the first adorned program in Example 9 is the following:

p^{f}(X) :- q1(X,Y), $q2^{b}$([Y|Z]).
q1(2,0) :- .
$q2^{b}$([Y|U]) :- $magic\_q2^{b}$([Y|U]), q1(Y,U).
$magic\_q2^{b}$([Y|Z]) :- q1(X,Y).

The execution of this program would proceed as follows. The last rule is applied to generate (only) the fact $magic\_q2^{b}$([0|Z]). The third rule is now prevented from producing any facts. Therefore, the first rule cannot produce any facts either, and so the computation halts.

The rewritten program corresponding to the second adorned program in Example 9 is the following:

p^{f}(X) :- q1(X,Y), $q2^{b}$([Y|Z]).

q1(2,0) :- .
$q2^b$([U|V]) :- $magic\_q2^b$([U|V]), q1(U,W), $q2^b$(V).
$q2^b$([]) :- $magic\_q2^b$([]).
$magic\_q2^b$([Y|Z]) :- q1(X,Y).
$magic\_q2^b$(V) :- $magic\_q2^b$([U|V]).

The second rule defining $magic\_q2^b$ is produced because of the occurrence of $q2^b$ in the body of the rule defining $q2^b$. We observe that this rule produces the fact $magic\_q2^b$(V). Thus, every fact is in this relation, and $magic\_q2^b$ becomes a trivial filter that does not restrict the computation. This is exactly what should happen since the argument of $q2^b$ in the body of the rule defining $q2^b$ was really a free variable, as discussed in Example 9. []

The previous examples illustrated the nature of the extension from Magic Sets to Magic Templates. The ability to deal with rules that are not range-restricted in the rewritten program provides an elegant solution to the problem of utilizing partially bound arguments (examples 4, 8 and 12). Example 11 illustrated another situation - repeated free variables - in which the ability to deal with such rules allowed us to extend the range of sips that can be implemented by bottom-up strategies. This ability is also required to utilize some standard logic programming techniques, such as *difference lists*.

Example 13:

The following is a program that appends two lists in constant time. However, the input lists must be represented as difference lists. A *difference list* is a term representing a list as the ‘‘difference’’ of two lists. For example, consider a list L composed of the elements 1, 2 and 3 in that order. We can think of it as the difference of the list composed of 1, 2, 3 and X, where X is some list, and the list X. We use the syntax dlist([1, 2, 3|X], X) to denote this term.

append(dlist(X,Y), dlist(Y,V), dlist(X,V)) :- .

We expect this rule to be used with the first two arguments bound to (difference) lists. The third argument in the (only) resulting fact is the list obtained by appending the second list to the first. Thus, the adorned version of this rule is:

$append^{bbf}$(dlist(X,Y), dlist(Y,V), dlist(X,V)) :- .

The call *append*(dlist([1,2|U],U), dlist([4,5|V],V),Z)? succeeds by generating the fact $append^{bbf}$(dlist([1,2|U],U), dlist([4,5|V],V), dlist([1,2,4,5|V],V)). The computation makes the binding U = [4,5|V], through unification. This rule is range-restricted. However, in the input arguments, the variables Y and V must be (bound to other) free variables. The rule that generates the fact(s) containing the input argument lists must therefore generate facts with free variables in them. This implies that the program must contain at least one rule that is not range-restricted.

Let us consider another program that uses this technique. The following program, from [MW88], breaks a list into two parts. The problem is that we may not know where to break the list until we process the first part. To do this efficiently without stepping through the list in the first part, we use difference lists. (For a better appreciation of the power of difference lists, the reader is referred to [MW88] for an alternative program that does not use difference lists.)

all(dlist(W,R)) :- firstpart(dlist(W,M)), secondpart(dlist(M,R)).
firstpart(dlist([1, 2|X], X)) :- .

When executed as a Prolog program, the only argument of *all* is bound to a difference list. Consider the call *all*(dlist([1,2,3,4,5], []))? This generates the goal *firstpart*(dlist(W,M))?, which is solved with *firstpart*(dlist([1,2,3,4,5], [3,4,5])). Thus, the input list has been (processed and) broken. We then solve the goal *secondpart*(dlist([3,4,5], []))? (using rules defining *secondpart*, that are not included here).

We can realize this by rewriting the program:

all^b(dlist(W,R)) :- $magic\_all^b$(dlist(W,R)),
    $firstpart^b$(dlist(W,M)), $secondpart^b$(dlist(M,R)).
$firstpart^b$(dlist([1,2|X], X)) :- $magic\_firstpart^b$(dlist([1,2|X], X).
$magic\_firstpart^b$(dlist(W,M)) :- $magic\_all^b$(dlist(W,R)).
$magic\_secondpart^b$(dlist(M,R)) :- $magic\_all^b$(dlist(W,R)), $firstpart^b$(dlist(W,M)).
$magic\_all^b$(dlist([1,2,3,4,5], [])).

The last rule is the seed, and is obtained from the call. We have not presented the details of the sip chosen, or the intermediate adorned program, but these should be clear. The point to note is that the rule defining $magic\_firstpart^b$ is not range-restricted (M appears in the head but not in the body). The bottom-up execution of this program mimics the Prolog execution, as the reader can easily verify. (Notice that the generation of ''magic'' facts corresponds to the generation of goals in the Prolog execution.) []

We now consider the correctness of the transformation. Let P^{mg} denote a program obtained from P^{ad} by the ''magic'' transformation. We must first resolve a small detail. 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 to not be a part of P^{mg}.

With respect to the original program P, an adorned predicate p^a can be viewed as a *query form*. It represents queries of the form $p(\theta)$, in which all arguments corresponding to b's in adornment a are bound. We say that (P, 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}. We must show that (P, q^e) and (P^{mg}, q^e) are equivalent, where q^e is the given query form. This follows from the stronger result presented below. We remind the reader that we only consider pure Horn clause programs without negation.

Theorem 5.1: Let P^{ad}, P^{mg} be as above, and let p^a be a predicate that appears in P^{ad}. Then (P, p^a) is equivalent to (P^{mg}, p^a).

Proof: First, we note that for each ''modified'' rule (i.e., those produced in step (3) of the transformation) of P^{mg}, if the predicate adornments and the magic literal corresponding to the head are dropped, we obtain a rule of P. Further, only the modified rules contain predicates from P^{ad} in the head. It follows that if a modified rule of P^{mg} is applied to some facts to produce a new fact, then the corresponding rule in P can be applied to the unadorned versions of those facts to generate the corresponding new unadorned fact. By induction, if an adorned version of a fact is generated in a bottom-up computation of P^{mg}, then the unadorned version of the same fact is generated in a bottom-up computation of P. Thus, the answer set for (P^{mg}, p^a) is contained in the answer set for (P, p^a).

For the other direction, if $p(\overline{c})$ is generated by a computation of P, then there exists a *derivation tree* for it. The fact $p(\overline{c})$ 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 (in P) that generates this fact from the facts labeling its children. We prove, by induction on the height of derivation trees in P, that given a derivation tree for a fact $p(\overline{c})$ in P, if p^a is in P^{ad}, there is a derivation tree in $P^{mg} \cup$ {

$magic\_p^a(\bar{c}^b)$ } for $p^a(\bar{c})$. The basis of the induction is the set of derivation trees of height one. These are simply base facts, and they are also derivation trees for P^{mg}. Let the hypothesis, which we will refer to subsequently as hypothesis (1), hold for trees of height less than n. Consider now a derivation tree of height n, and assume that the rule used to derive its root is the following:

r: $p(\theta)$:- $q_1(\phi_1), q_2(\phi_2), \ldots, q_n(\phi_n)$

We use the notation $\bar{c}$ and $\bar{d_i}$ to denote vectors of arguments in the rule instance in order to distinguish them from the corresponding arguments in the text of the rule. Let the rule instance corresponding to the derivation of the root using rule r be:

$$p(\bar{c}) :- q_1(\overline{d_1}), q_2(\overline{d_2}), \ldots, q_n(\overline{d_n})$$

The modified rule for r in P^{mg} has the form:

r': $p^a(\theta)$:- $magic\_p^a(\theta^b), q_1^{a_1}(\phi_1), q_2^{a_2}(\phi_2), \ldots, q_n^{a_n}(\phi_n)$

The corresponding rule instance is:

$$p^a(\bar{c}) :- magic\_p^a(\bar{c}^b), q_1^{a_1}(\overline{d_1}), q_2^{a_2}(\overline{d_2}), \ldots, q_n^{a_n}(\overline{d_n})$$

Each fact $q_i(\overline{d_i})$, i = 1 to n, is the root of a derivation tree in P of height less than n. To show that there is a derivation tree in $P^{mg} \cup$ { $magic\_p^a(\bar{c}^b)$ } for $p^a(\bar{c})$, we have to show that there exist derivation trees for the facts $q_i^{a_i}(\overline{d_i})$, i = 1 to n. If q_i is a base predicate, there is a derivation tree of height 1. If q_i is a derived predicate, this follows from induction hypothesis (1) if we show that there is a derivation tree (in $P^{mg} \cup$ { $magic\_p^a(\bar{c}^b)$ }) for the facts $magic\_q_n^{a_n}(\overline{d_i}^b)$, i = 1 to n.

The proof that there is a derivation tree for these magic facts is by induction on the position of the predicate occurrence in the body of rule r. As the basis case, let $q_k(\phi_k)$ be the first derived predicate occurrence in the body. By construction, there is a rule, say r1, in P^{mg} with head $magic\_q_k^{a_k}(\phi_k^b)$, such that the body only contains base literals that occur to the left of $q_k(\phi_k)$ in r, and $magic\_p^a(\theta^b)$. † The corresponding facts in the body of the rule instance for r' can be used in the body of rule r1 to obtain an instance of rule r1 with head fact $magic\_q_k^{a_k}(\overline{d_i}^b)$. Thus, there is a derivation tree in $P^{mg} \cup$ { $magic\_p^a(\bar{c}^b)$ } for $magic\_q_k^{a_k}(\overline{d_i}^b)$. It follows from induction hypothesis (1) that there is also such a derivation tree for $q_k^{a_k}(\overline{d_i})$.

Let $q_m(\phi_m)$ be the jth derived predicate occurrence in the body of rule r, and let the hypothesis, which we will refer to as hypothesis (2), hold for derived predicate occurrences 1 to j-1. By construction, P^{mg} contains a rule, say r2, with head $magic\_q_m^{a_m}(\phi_m^b)$, such that the body only contains $magic\_p^a(\theta^b)$, base literals, derived predicate occurrences 1 through j-1, and the corresponding magic literals. Thus, the facts corresponding to these literals in the instance of rule r', which are generated according to induction hypothesis (2), can be used in rule r2 to construct a derivation tree for $magic\_q_m^{a_m}(\overline{d_m}^b)$ as well. It follows that there is a derivation tree for $q_m^{a_m}(\overline{d_m})$ also. This completes the proof of Theorem 5.1. []

† Without loss of generality, we assume that the body literals are ordered according to the partial order induced by the sips.

5.1. Magic Programs for Full Sips

There is some syntactic simplification possible in that we need not explicitly indicate the adornments of predicates, since every argument is considered bound. We note that certain arguments, although designated as bound, really provide no restriction. Consider the original program. If argument position i of a body literal, say $q(\theta)$, in rule r contains just a variable, and this variable appears nowhere else in this literal or in preceding literals of $P(r) \cup p_h$, then this argument provides no restriction. The magic rule generated from this literal occurrence has *magic_q*(θ) as the head literal, and the variable in the ith argument can be replaced by a ‘‘don't-care’’ variable. (In fact, we can reduce the arity of the magic predicate by considering such arguments to be free, but this would obviously mean sacrificing the simplicity of having just one reachable adornment - a string of *b*s - for each predicate.)

We also observe that an argument position is potentially restricted in all other cases - if the variable appears in a preceding literal, more than once in the given literal, or if the position contains a non-variable term.

6. Properties of Magic Programs

In the previous section, we showed that the Magic Templates algorithm transformed the given program into an equivalent program with respect to the query. The fixpoint of the transformed program is computed using a bottom-up iteration, possibly with some refinements as in *Seminaive* evaluation (e.g. [Ba85]). In this section, we consider some properties of the fixpoint evaluation of the transformed program.

6.1. Optimality of the Magic Implemention of Sips

Our main result concerns the optimality of the Magic Templates strategy, in the sense that it implements a given set of sips by computing the minimum set of facts. We first define the class of strategies for which this claim of optimality is made, following [BeR87]. Our definition generalizes that in [BeR87] by including strategies that generate nonground facts. Essentially, the definition seeks to capture the work that must be done to establish that every answer has been computed; and to preclude strategies that behave like ‘‘oracles’’, in that they work with knowledge other than the logical consequences of the rules and the facts in the database. It also limits consideration to strategies that follow the given set of rules, according to the given collection of sips.

Accordingly, we define a *sip-strategy* for computing the answers to a query expressed using a set of Horn clause rules, and a set of sips, one for each adornment of a rule head, as follows. We assume that a strategy constructs queries, and for each query it constructs answers by using the rules in the program to compute new facts. The set of queries and the set of facts generated during a computation must satisfy certain conditions, which express the requirement that the strategy follow the sips in computing the answers.

A sip-strategy takes as input

i. A query, and

ii. A program with a collection of sips, where for each rule, there is exactly one sip per head adornment.

The computation must satisfy the following conditions:

1. If $p(\theta)$? is a query, and $p(\phi)$ is an answer, then $p(\phi)$ is computed.
2. If $p(\theta)$? is a query, then for every rule with head predicate p, a query is constructed for every predicate in the rule body according to the sip for the rule.

A sip-strategy is initially called with the given set of rules, the facts in the database, and the given query. The first condition requires that it compute all answers to each query that it generates. The second condition describes how answers are generated for a query. For every rule head matching the query, we invoke the rule, thus determining an adornment, and selecting a sip to follow. Next, the rule's body is evaluated. For every body literal, subqueries are generated according to the sip. Consider a body literal $p^a(\theta)$, and let $p^a(\phi)$? be a subquery generated from it. The vector of bound arguments ϕ^b in the subquery is obtained from θ^b by substituting, for the variables in it, terms that are passed through the sip arc entering the node corresponding to this literal. Each free argument in ϕ is a unique variable; the subquery does not restrict the terms that may appear in the free argument positions. (In defining adornments and passing bindings, an argument *must* be considered bound if any variable in it is bound.) For each subquery generated, there is a set of answers. These are used to pass bindings, as per the sip, to create additional subqueries. By combining the answers to all these subqueries, we generate answers for the original query involving the rule head.

A *sip-optimal* strategy is defined to be a sip-strategy that generates only a set of facts and queries as required by the above definition for the predicates in the program.

Example 14:

This example is designed to illustrate the above definition of sip-optimality. Consider the program:

p(X) :- q(X).
q(Y) :- b1(Y).
q(W) :- b2(W).
b1(5).
b2(V).
Query: p(U)?

The magic program is:

p^b(X) :- mp^b(X), q^b(X).
q^b(Y) :- mq^b(Y), b1(Y).
q^b(W) :- mq^b(W), b2(W).
mq^b(X) :- mp^b(X).
b1(5).
b2(V).
mp^b(U).

Denote the collections of queries and facts in conditions 1 and 2 in the definition of a method by Q and F respectively.

The following subqueries are generated: mp^b(U), mq^b(U). (We ignore subqueries on the base predicates $b1$ and $b2$ for simplicity.) The following facts must be generated: q^b(5), q^b(U). Now, it is possible to generate both p^b(5) or p^b(U). A strategy that does so would still be considered a sip-optimal by our definition. In other words, F includes both p^b(5) and p^b(U).

In practice, we could use subsumption checks to discard $q^b(5)$ before generating $p_b(5)$; this would result in our not generating $p^b(5)$ at all. This would also be a sip-optimal strategy. The reasons that we do not distinguish these two different executions in our definition of sip-optimality are the following: (1) First, even with subsumption checks, it is possible that we will generate $p^b(5)$ if the first rule is used with $q^b(5)$ *before* $q^b(U)$ is generated. Thus, these executions can be produced by different rule application orders of the same evaluation method, and we choose to consider the worst-case possibility uniformly. (2) The effect of rule ordering is, in our opinion, outside the scope of sips, which specify essentially a binding propagation order within rules. []

We have the following theorem.

Theorem 6.1: Consider a query over a set of rules P, where a sip is associated with each adornment of a rule's head. Let P^{mg} be the set of rewritten rules produced by the Magic Templates method. The bottom-up evaluation of P^{mg} is sip-optimal.

Proof: Denote the collections of queries and facts in conditions 1 and 2 in the definition of a method by Q and F respectively.

Let us consider a bottom-up computation of P^{mg}. First, we need to identify the facts generated in such a computation. The magic seed is a generated fact. Further, suppose that f_1, ... f_m are generated facts corresponding to derived predicates in the body of a rule, and g_1, ... , g_l are facts in base predicates in the body, such that the body is satisfied and generates the fact f for the head. Then f is also a generated fact.

It remains to show that every fact generated for a predicate in a bottom-up computation of P^{mg} is an answer to a query in Q, or denotes a query in Q. More precisely, we claim that for each generated fact, if it is a magic fact $magic\_p^a(\phi)$, then there is a query $p^a(\theta)$? in Q, such that $\theta^b = \phi$; and if the generated fact is a fact $p^a(\phi)$, then there is a query $p^a(\theta)$? in Q such that ϕ is an instance of θ.

The proof is by induction on the height of derivation trees. For the basis of the induction, we have the seed, say $magic\_q^e(\phi)$, which corresponds to the given query.

Suppose the claim holds for all facts generated using a derivation tree of height n or less. Consider a fact f that is generated using a rule r with derived facts f_1, ... , f_m in the body that all have derivation trees of height n or less. If the fact f is $p^a(\phi)$, where p^a is in P^{ad}, then one of the f_i, say f_1, must be $magic\_p^a(\phi^b)$, by construction of the ''modified'' rules in P^{mg}. By the induction hypothesis, $magic\_p^a(\phi^b)$ corresponds to a query $p^a(\theta)$ in Q, such that $\theta^b = \phi^b$. Since θ^f is a vector of distinct variables, ϕ must be a substitution instance of θ, and the claim holds for fact f as well.

If the fact f is a magic fact $magic\_p^a(\phi)$, generated using a rule r with head, say, $magic\_p^a(\theta)$, then consider the adorned rule in P^{ad}, say $r1$, and sip arc $N \rightarrow p$, that produced the magic rule r. By construction of magic rules in P^{mg}, if q is a predicate in N, and corresponds to the literal $q^{a1}(\eta)$ in $r1$, then $q^{a1}(\eta)$ and $magic\_q^{a1}(\eta^b)$ appear in the body of r.† Further, each of these facts corresponds to a query in Q or an answer to a query in Q, by the induction hypothesis. From the interpretation of a sip, the terms that are substituted for the variables in θ to obtain ϕ must be passed through the sip arc entering the node for p^a as bindings for those variables. By condition

† For simplicity of exposition, we assume here that base predicates are also adorned.

(2) in the definition of a method, the query $p^a(\phi)$? must be in Q. This completes the proof of Theorem 6.1. []

We consider the significance of this result. First, our definition tries to capture the intuitive idea of a strategy that evaluates a program using a given sip collection. A method that does not generate some of the queries or facts in Q and F cannot be considered as using the given collection of rules and sips. For if it does, then there must be a stage in the computation (corresponding to the missing queries or facts) where it is ‘‘guessing’’, or using an oracle. However, there are strategies which use auxiliary information to avoid generating all answers to some subqueries, and these are not covered by our definition of a ‘‘method’’. An example of such a method is QoSaQ [Vi88], when ‘‘global optimization’’ is used. (An appropriate comparison with such methods must weigh the advantage of inferring fewer facts and goals with the cost of maintaining and using the auxiliary information.) There are also strategies that do not proceed by generating subgoals on the given program, and these strategies are not covered either. Examples of such methods include Counting [BeR87, BMSU86, SZ86a, SZ86b], the methods proposed in [HN84, Na87, Na88]. Typically, such methods seek to exploit the structure of the rules in some way that goes beyond sideways information passing, and are less generally applicable.

However, our definition is sufficiently broad to include a large number of proposed strategies. These include Prolog, versions of top-down evaluation with memoing such as QSQ [Vi86] and Extension Tables [DW87], Static and Dynamic Filtering [KL86a, KL86b], and several parallel evaluation strategies proposed in the logic programming literature including those in [Ka87, VG86].

A strategy may generate additional queries, or facts, in addition to those that must be generated by conditions (1) and (2), and then we have good reason to consider it inferior to a sip-optimal strategy such as Magic Templates. Sip-optimality does not imply that facts and queries are not generated more than once. However, if the bottom-up computation is done using Seminaive evaluation (e.g., [Ba85]), no fact is inferred twice using the same derivation. (In general, it is not possible to avoid inferring the same fact by two distinct derivations without sacrificing completeness of the evaluation strategy.) Prolog does not have this property, and the importance of this distinction between Magic Templates and Prolog is emphasized by the study in [BaR86, BaR88]. (In the examples considered there, Magic Templates is identical to Generalized Magic Sets, which is the method used in the comparison. Also, there are some examples in which that study assigns a lower cost to Prolog, in terms of the number of facts inferred. We note that this does not contradict the results in this paper since the goals generated were counted in the cost for Magic Sets, since they are facts generated by rules in P^{mg}, but not in the cost for Prolog, since they are not facts generated by rules in P, which is the program used by Prolog.)

Finally, sip-optimality does not necessarily imply that the computation is efficient in the resources it consumes. In particular, it does not reflect the overhead associated with a given method in inferring a new fact or goal from given facts and goals. For example, bottom-up evaluations offer the potential advantage of utilizing efficient set-at-a-time join algorithms, especially for relations containing only ground facts. On the other hand, each fact and query (i.e., magic fact) must be stored, and each new fact must be checked to see if it is previously generated. This is an overhead that is not incurred by Prolog. Of course, this has other consequences for Prolog, including repeated inferences and potential looping.

We observe that in this paper we have assumed that all answers to a query are required. If only one answer, or a subset of answers, is desired, the ability to explicitly direct the search as in Prolog may be useful. We have also restricted our discussion to pure Horn clause programs without negation.

6.2. Termination Issues

We now consider the question of whether the bottom-up evaluation of P^{mg} terminates after computing all answers to the query. We have the following corollary from Theorem 6.1.

Corollary 6.2: Consider a query over a set of rules *P*, where a sip is associated with each adornment of a rule's head. Let P^{mg} be the set of rewritten rules produced by the Magic Templates method. The bottom-up evaluation of P^{mg} terminates if any terminating sip-strategy exists for evaluating *P* according to the associated sips. []

Further, the bottom-up evaluation of P^{mg} terminates if the set of goals and facts to be generated is finite. Thus, it always terminates for programs that do not contain function symbols.

The following example illustrates some aspects of termination that arise due to our consideration of nonground terms. We compare Prolog and bottom-up evaluation. (The latter with and without rewriting).

Example 14:

p(X) :- p([X|X]).
p([[5]|[5]]) :-

We consider three queries: (1) p(X)? (2) p([5])? (3) p([6])?

Prolog does not terminate on any of these queries. If the program is evaluated bottom-up without any rewriting, it terminates after producing the single new fact p([5]), regardless of the query. (Not rewriting can be thought of as rewriting according to a sip that contains no arcs. That is, it simply reflects a certain choice of sips.)

Suppose we rewrite this program using the same sip as Prolog: $p_h \rightarrow_X p$. This gives us the following adorned program for query (1):

p^f(X) :- p^b([X|X]).
p^f([[5]|[5]]) :- .
p^b(Y) :- p^b([Y|Y]).
p^b([[5]|[5]]) :- .

The rewritten program is:

p^f(X) :- p^b([X|X]).
p^f([[5]|[5]]) :- .
p^b(Y) :- *magic_p*^{b}(Y), p^b([Y|Y]).
p^b([[5]|[5]]) :- *magic_p*^{b}([[5]|[5]]).
magic_p^b([X|X]) :- .
magic_p^b([Y|Y]) :- *magic_p*^{b}(Y).

The execution proceeds as follows. We can apply the last rule to produce *magic_p*^{b}([[U|U]|[U|U]]), but this can be generated from *magic_p*^{b}([X|X]) by assigning X = [U|U]. Thus, *magic_p*^{b}([X|X]) is more general and *magic_p*^{b}([[U|U]|[U|U]]) is discarded. We can apply the fourth rule to produce p^b([[5]|[5]]). We can then apply the first rule to produce p^f([5]). No further new facts can be produced, and the computation halts.

Consider the second query. The adorned program is:

p^b(Y) :- p^b([Y|Y]).

-- --

p^b([[5]|[5]]) :- .

The rewritten program is:

p^b(Y) :- *magic_p*^{b}(Y), p^b([Y|Y]).
p^b([[5]|[5]]) :- *magic_p*^{b}([[5]|[5]]).
magic_p^b([Y|Y]) :- *magic_p*^{b}(Y).

The seed is *magic_p*^{b}([5]). The last rule can be repeatedly applied to produce *magic_p*^{b}([[5]|[5]]), *magic_p*^{b}([[[5]|[5]]|[[5]|[5]]]), etc. However, once we apply the first rule to produce p^b([5]), it is easy to see that the computation can be stopped since this is the only answer fact.

Consider the third query. The adorned and rewritten programs are the same as for the second query, but the seed is now *magic_p*^{b}([6]). The last rule can be repeatedly applied to produce *magic_p*^{b}([[5]|[5]]), *magic_p*^{b}([[[5]|[5]]|[[5]|[5]]]), etc. In this case, the fact p^b([6]) will never be produced, but there is no way to stop the computation, which produces facts for *magic_p*^{b} forever. We note that this is a faithful implementation of the Prolog sip - in the first two queries, the bottom-up computation terminated due to its duplicate elimination or because the only answer had been generated. In this case, the computation mimics Prolog all too faithfully. []

We do not consider how to test whether bottom-up evaluation terminates on a given program. We refer the reader to [APPRSU86, KL87, KRS88] for some work on this problem.

7. Constraint Logic Programming

We now describe how Constraint Logic Programming (CLP) can be implemented using the bottom-up approach we described in earlier sections. The ability to deal with rules that are not range-restricted is crucial. Given this ability, the extension to deal with CLP programs is natural: essentially, with each fact we now associate a set of constraints. As we observed earlier, a fact with variables in it denotes a (possibly infinite) set of facts. Constraints limit this set of facts by excluding those which violate the set of constraints associated with the fact. Obviously, facts which do not contain variables have no associated constraints (and represent a single fact).

Consider the following rule:

p(X,Y) :- X < Y.

Given the query p(X,Y)?, a CLP system returns the constraint X < Y as the ‘‘answer’’. That is, the answer is a constrained fact, ‘‘(X,Y) where X < Y’’.

The notion of constraints is not really new - we have been using some form of constraints throughout the paper. For example, the fact p([X|Y]) can be thought of as a constrained fact ‘‘p(Z) where there exist some X,Y such that Z = [X|Y]’’. Indeed, the rule ‘‘p(X) :- b(X).’’ can be thought of as a constrained fact ‘‘p(X) where b(X) is true’’. However, it is useful to distinguish these from constraints such as ‘‘X < Y’’, which are resolved using a specialized *constraint solver*. Henceforth, we use ‘‘constraint’’ to refer only to these new forms of constraints; while all our examples are of arithmetic constraints, this is not necessary, so long as we have a sound and complete constraint solver. We use **where** as a keyword to associate constraints with facts; for example, p(X,Y) **where** X < Y.

A rule in a CLP program contains a set of constraints in addition to a conjunction of literals. As for logic programs, applying a rule bottom-up involves taking the join of the body literals to generate facts for the head predicate, but now generated facts which do not satisfy the constraints in the rule are discarded. This is a slight simplification since each fact that is unified with a body literal may have a set of constraints associated with it. These constraints, after applying the unifying substitution, must be added to the set of constraints associated with the rule in order to generate a new constrained fact corresponding to the head.

Example 15:

Consider the following program:

```
q1(X,Y) :- q3(Z,Y), q2(X,Z).
q4(X,Y) :- q3(X,Y), q3(U,V), X = U, Y = V.
q5(X,Y) :- q2(X,U), q2(V,Y), q3(U,V).
q3(X,Y) :- q6(X,Y).
q6(X,Y) :- X < Y.
q2(1,2) :- .
q2(1,5) :- .
q2(4,3) :- .
```

We assume that no rewriting is done, in order to focus on the basic properties of CLP programs and their (bottom-up vs. top-down) execution. Consider the query q1(X,Y)? By using the rules defining $q3$ and $q6$, we generate $q3$(U,V) **where** U < V. Now, using the facts for $q2$, we can generate the facts $q1$(1,Y) **where** Y > 2, $q1$(1,Y) **where** Y > 5, and $q1$(4,Y) **where** Y > 3. It is easy to see that the second fact is subsumed by the first. This is a natural extension of our definition of when a fact is more general than another, and (depending on how intelligent the constraint solver is) we can discard the second fact.

Consider the query q4(X,Y)? By substituting the (only) fact for $q3$ into the rule defining $q4$, we have $q3$(X,Y), X < Y, $q3$(V,U), V < U, X = U, Y = V. If the constraint solver can now detect that there is a conflict, no facts are produced for $q4$. This query also allows us to make an important comparison between top-down and bottom-up strategies. A top-down strategy would solve each $q3$ goal independently. Thus, the fact in $q3$ (and also $q6$) is derived twice. The bottom-up strategy only infers it once. With recursion, this effect could have a major impact on performance.

Consider the query q5(X,Y)? The answer is just the fact $q5$(1,3). We leave it to the reader to work out the details of the execution. []

Example 16:

We rewrite the following program to reflect a top-down execution, and compare the bottom-up and top-down approaches with respect to termination.

```
q1(X,Y) :- q2(X,Z), q1(Z,Y).
q1(10,10) :- .
q2(X,Y) :- X < Y.
Query: q1(5,Y)?
```

The rewritten program is:

$q1^{bf}$(X,Y) :- $magic\_q1^{bf}$(X), $q2^{bf}$(X,Z), $q1^{bf}$(Z,Y).
$q1^{bf}$(10,10) :- $magic\_q1^{bf}$(10).
$q2^{bf}$(X,Y) :- $magic\_q2^{bf}$(X), X < Y.
$magic\_q1^{bf}$(Z) :- $magic\_q1^{bf}$(X), $q2^{bf}$(X,Z).
$magic\_q2^{bf}$(X) :- $magic\_q1^{bf}$(X).
$magic\_q1^{bf}$(5).

Applying the rule defining $magic\_q1^{bf}$, we can produce the facts:

$magic\_q1^{bf}$(U) **where** U > 5,
$magic\_q1^{bf}$(U) **where** U > V and V > 5,
etc.

The first fact is more general than the second. If the constraint solver is able to establish this, we can discard the second fact, and the computation of $magic\_q1^{bf}$ facts stops after producing the fact $magic\_q1^{bf}$(U) **where** U > 5. Using this fact in the first rule, we can produce the fact $q1^{bf}$(X,10) **where** X > 5, X < 10. (Using the fact $magic\_q1^{bf}$(5), we also generate the fact $q1^{bf}$(5,10), which is the only fact in the answer.) The computation then terminates. In contrast, the top-down strategy sets up goals forever (corresponding to the infinite set of ‘‘magic’’ facts listed above) without ever producing the answer fact. We note that even if the constraint solver is not smart enough to detect that the first fact in $magic\_q1^{bf}$ is more general than others (which would cause the bottom-up strategy to produce an infinite number of magic facts and therefore not terminate), the bottom-up strategy still produces the one answer fact. []

From a binding propagation standpoint, as we have shown, the distinction between top-down and bottom-up strategies is blurred since any top-down strategy can also be implemented bottom-up through rewriting. The appropriate control strategy depends upon the problem. While there are applications where a top-down strategy (say, Prolog) performs better, there are also cases where bottom-up strategies perform better, and, ideally, the choice must be made using careful cost estimates. However, bottom-up strategies have one important virtue: *completeness.* As we saw in the above example, all facts in the answer are eventually produced, unlike in the top-down strategy. Depending on the constraint solver, the bottom-up strategy often also terminates in cases when the top-down strategy does not.

8. Conclusions

We have presented a bottom-up strategy that is applicable to general Horn clause programs. The method is sound and complete, and also efficient in that it computes no facts or goals that are not also computed by a top-down strategy such as Prolog. Further, by virtue of the underlying Seminaive fixpoint evaluation, no fact is inferred twice using the same derivation, and in this respect it is superior to Prolog. The method often terminates when Prolog does not, and in particular, it always terminates if the program contains no function symbols. Bottom-up fixpoint evaluation also offers an opportunity for efficient set-oriented operations, such as joins. Thus, the method has several attractive properties. On the other hand, it requires that we store more facts, the relative real costs of inferences in the two approaches are hard to measure, and backtracking offers some advantages when we are only interested in one answer to a query.

Another contribution of this paper is that sideways information passing, or binding propagation, is distinguished from the control strategy used to implement it, thus providing insight into the relationship between top-down and bottom-up methods.

9. Acknowledgements

Catriel Beeri has had a great influence on the development of the ideas presented in this paper, and in particular, the idea of using sips to guide rewriting strategies arose in joint work with him, presented in [BeR87]. Discussions with Joxan Jaffar and Jean-Louis Lassez were helpful in in seeing the applicability of these results to Constraint Logic Programs. Jeff Ullman made several helpful comments on an earlier version, and revised the abstract. Michael Kifer commented on a draft, gave pointers to his work with Lozinskii, and clarified it in discussions. Saumya Debray provided extensive comments that greatly improved the paper. I thank them all for their help.

10. References

[APPRSU86] F. Afrati, C. Papadimitriou, G. Papageorgiou, A. Roussou, Y. Sagiv and J.D. Ullman, ‘‘Convergence of Sideways Query Evaluation,’’ *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1986.*

[Ba85] F. Bancilhon, ‘‘A Note on the Performance of Rule Based Systems,’’ *MCC Technical Report DB-022-85, 1985.*

[BaR86] F. Bancilhon and R. Ramakrishnan, ‘‘An Amateur’s Introduction to Recursive Query Processing Strategies,’’ *Proc. SIGMOD, 1986.*

[BaR88] F. Bancilhon and R. Ramakrishnan, ‘‘Performance Evaluation of Data Intensive Logic Programs,’’ *In Foundations of Deductive Databases and Logic Programming, Ed. J. Minker, Morgan Kaufman, 1988.*

[BeR87] C. Beeri and R. Ramakrishnan, ‘‘On the Power of Magic,’’ *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1987.*

[BMSU86] F. Bancilhon, D. Maier, Y. Sagiv and J. Ullman, ‘‘Magic Sets and Other Strange Ways to Implement Logic Programs,’’ *Proc. 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1986.*

[BNRST86] C. Beeri, S. Naqvi, R. Ramakrishnan, O. Shmueli and S. Tsur, ‘‘Sets and Negation in a Logic Database Language (LDL1),’’ *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1987.*

[BPR87] I. Balbin, G.S. Port, and K. Ramamohanarao, ‘‘Magic Set Computation of Stratified Databases,’’ *Technical Report 87/3, University of Melbourne, 1987.*

[DW87] S.W. Dietrich and D.S. Warren, ‘‘Extension Tables: Memo Relations in Logic Programming,’’ *Proc. Symposium on Logic Programming, 1987.*

[GMN84] H. Gallaire, J. Minker and J.-M. Nicolas, ‘‘Logic and Data Bases: A Deductive Approach,’’ *Computing Surveys, Vol. 16, No 2, June 1984.*

[HN84] L. Henschen and S. Naqvi, ‘‘On Compiling Queries in Recursive First-Order Data Bases,’’ *JACM, Vol 31, January 1984, pp 47-85.*

[JL87] J. Jaffar and J-L. Lassez, ‘‘Constraint Logic Programming,’’ *Proc. Conference on Principles of Programming Languages, 1987.*

[Ka87] L.V. Kale, ‘‘The Reduce-Or Process Model for Parallel Evaluation of Logic Programs,’’ *Proc. Intl. Conf. on Logic Programming, 1987.*

[KL86a] M. Kifer and E. Lozinskii, ‘‘Filtering Data Flow in Deductive Databases,’’ *Proc. Intl. Conf. on Database Theory, 1986.*

[KL86b] M. Kifer and E. Lozinskii, ‘‘ A Framework for an Efficient Implementation of Deductive Databases,’’ *Proc. Advanced Database Symposium, Tokyo, 1986.*

[KL87] M. Kifer and E. Lozinskii, ‘‘Implementing Logic Programs As a Database System,’’ *Proc. Intl. Conf. on Data Engineering, 1987.*

[KRS88] R. Krishnamurthy, R. Ramakrishnan and O. Shmueli, ‘‘A Framework for Testing Safety and Effective Computability of Extended Datalog,’’ *Proc. SIGMOD, 1988.*

[LM83] J-L. Lassez and M.J. Maher, ‘‘Closures and Fairness in the Semantics of Programming Logic,’’ *Theoretical Computer Science.*

[LMM88] J-L. Lassez, M.J. Maher, and K. Marriott, ‘‘Unification Revisited,*In Foundations of Deductive Databases and Logic Programming, Ed. J. Minker, Morgan Kaufman, 1988.*

[MW88] D. Maier and D.S. Warren, ‘‘Computing with Logic: Logic Programming with Prolog,’’ *The Benjamin/Cummings Publishing Company, 1988.*

[Na87] J.F. Naughton, ‘‘One-Sided Recursions,’’ *Proc. ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1987.*

[Na88] J.F. Naughton, ‘‘Compiling Separable Recursions,’’ *Proc. Sigmod, 1988.*

[RLK86] J. Rohmer, R. Lescoeur and J.M. Kerisit, ‘‘The Alexander Method: A Technique for the Processing of Recursive Axioms in Deductive Databases,’’ *New Generation Computing 4, 3, 1986.*

[SZ86a] D. Sacca and C. Zaniolo, ‘‘On the Implementation of a Simple Class of Logic Queries for Databases,’’ *Proc. ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1986.*

[SZ86b] D. Sacca and C. Zaniolo, ‘‘The Generalized Counting Method for Recursive Logic Queries,’’ *Proc. Intl. Conference on Database Theory, 1986.*

[Ul85] J.D. Ullman, ‘‘Implementation of Logical Query Languages for Databases,’’ *TODS, Vol. 10, No. 3, pp. 289-321, 1985.*

[VK76] M.H. Van Emden and R.A. Kowalski, ‘‘The Semantics of Predicate Logic as a Programming Language,’’ *JACM, 23, 4, Oct 1976.*

[VG86] A. Van Gelder, ‘‘A Message Passing Framework for Recursive Query Evaluation,’’ *Proc. SIGMOD, 1986.*

[Vi86] L. Vieille, ‘‘Recursive Axioms in Deductive Databases: The Query/Subquery Approach,’’ *Proc. Intl. Conference on Expert Database Systems, 1986.*

[Vi88] L. Vieille, ‘‘From QSQ Towards QoSaQ: Global Optimization of Recursive Queries,’’ *Proc. Intl. Conf. on Expert Database Systems, 1988.*