Extending Magic Sets Techniques to Deductive Databases with Uncertainty

Huang Qiong

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montréal, Québec, Canada

September 2008

Your file Votre référence
ISBN: 978-0-494-45486-2
Our file Notre référence
ISBN: 978-0-494-45486-2

ABSTRACT

Extending Magic Sets Techniques to Deductive Databases with Uncertainty

Huang Qiong

With the magic sets techniques having been proposed to improve the efficiency of bottom-up evaluations of Datalog programs by taking advantage of goal structure, we extend these techniques to deductive databases with uncertainty in the context of the parametric framework (PF). In our endeavor, we develop the generalized magic sets and generalized supplementary magic sets techniques, and establish their correctness. We have implemented the proposed techniques and have conducted numerous experiments for the assessment of the evaluation performance. Our experiment results reveal that different programs enjoy different efficiency gain, depending on the potential facts ratio, which measures the capacity to improve efficiency. When this ratio ranges from 1% to 20%, the efficiency observed was approximately 1 to 700 times faster. Our results also indicate that an integration of magic sets techniques and semi-naive evaluation with predicate partitioning yield the best performance.

ACKNOWLEDGEMENTS

My first thanks must go to Professor Nematollaah Shiri, my thesis supervisor, for his extraordinary and continuous guidance, support, encouragement throughout the development of this thesis. He introduced me to the world of knowledge base systems, logic programming and deductive databases. It is hard to count how much time and energy he spent in helping me amidst his tight schedule. His advise and patience made the development of this thesis possible.

I am grateful to my lab colleagues Jinzan Lai, Ali Kiani, Xi Deng, Hsueh-Ieng Pai, and Nima Mohajerin. Without their help and encouragement, the development of thesis would last longer.

I would also like to thank my friends Qiang Wu, Zhaoxing Huang, and Tahira Hasan for their help in the development of the first prototype of this work in the context of the knowledge base course COMP6591.

My deep love goes to my wife Demin Qu and my daughter Rachel. Without their love and support, I would not be able to finish this wonderful stage of my life, and my best wishes to my expected new baby.

Finally, I would like to thank my parents Beining Huang and Manjun Li, my sisters Qing, Lin and Chui, and my best friend Xiaochun Yang. They have never lost faith on my capabilities.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

Uncertainty is considered as a term describing unsettled information and human doubt, a measurement for the prediction of future events and the outcome of chance, and also as a potential deficiency due to the lack of knowledge and certainty. Uncertain data exist in a number of fields, including meteorology, astronomy, geology, medical science, economics, psychology, engineering, and science etc. Travelers are concerned with weather forecast. Geologists wish to have an accurate earthquake prediction. Bankers measure the risk of loans. Chaining uncertain data becomes an interesting topic that challenges researchers. Many systems have been developed and extended to manage uncertain information. For example, NASA satellites help observing cloud of uncertainty on climate change; Microsoft researchers use machine learning techniques to model uncertain information for the HIV vaccine; IBM's Avatar project tends to extend probabilistic databases for data analysis in its IES machine.

"Reasoning about uncertain data" has been identified as one of the thirteen topics of next-generation databases infrastructure in Lowell Database Research Meeting in 2003 [ea05]. Since databases are designed to manage large quantities of data, reasoning over databases have been important. A deductive database (DDB) allows reasoning based on given rules and facts stored in the database. Many systems have been proposed in the last two decades, such as Aditi, COL, Concept-Base, CORAL, LDL(++), LOGRES, LOLA, Glue-Nail, Starburst, and XSB [VRK+94, AG91, JS94, RSS92, CGK+90, CCCR+90, FSS92, DMP93, SSW94]. For basics of deductive databases (DDBs), interested readers are referred to [CGT89]. Uncertainty has been addressed in some systems, such as Coral, and XSB. Also numerous frameworks for DDBs with uncertainty have been proposed [DLP91, Fit91, KL88, KS92, LS94, NS92, NS93, Sub94, vE86]. On the basis of the way in which uncertainty is associated with facts and rules of a logic program, these frameworks are classified into annotation based (AB) [Sub94, KL88, NS92, NS93, KS92, Sub94], and implication based (IB) [DLP91, Fit91, LS94, vE86]. A generalized and unified IB framework, called the parametric framework (PF) was introduced in [LS96].

A major concern is the efficiency of query processing for DDBs with uncertainty. Programs in DDBs can express recursive queries that conventional databases cannot handle in general. This makes optimization of logic programs in DDBs harder than conventional relational databases. "Magic sets rewriting" is an optimization technique for DDBs, introduced by Banihon, Maier, Sagiv and Ullman [BMSU86]. Supplementary magic sets were introduced by Sacca and Zaniolo [SZ86]. Beeri and Ramakrishnan developed generalized (supplementary) magic sets [BR87]. More studies in this direction may be found in [MFPR96, Ros94, RRSS94, SSW94, MP94, Bra96, FGL07]. In this work, we investigate ways to extend magic sets technique for DDBs with uncertainty.

1.1 Motivation

Computing the least fixpoint (lfp) is important in DDBs because lfp contains the meaning of a logic program. There are two approaches to evaluate logic programs in DDBs with uncertainty and to compute the answer set: top-down and bottom-up. No matter which approach we choose, efficiency is a crucial issue. The two sources of major problems affecting the evaluation speed are (1) repeated evaluations of rules which do not contribute to deriving of new atoms, and (2) evaluation process generates facts unrelated to a given query.

In the context of bottom-up fixpoint evaluation with uncertainty, two algorithms have been proposed to solve efficiency problem of type (1), i.e., semi-naive evaluation (SN) [LS96] and semi-naive evaluation with predicate partitioning (SNP) [SZ04]. For each rule at every iteration in the SN, the evaluation may derive atoms improved at the previous iteration, but unrelated to the goal query. SNP is a refinement and extension of SN in which at each iteration, the tuples found for each relation are divided into two partitions, new tuples and old tuples. At the next iteration, a rule chosen for evaluation should have at least one tuple in one of the new partitions.

Given a logic program P and a query Q, a top-down evaluation starts from the goal, unifies the head of each possible rule, and propagates bindings of the variables in the rule body. Atoms unrelated to answering Q can will not be reached. The top-down approach seems to solve the type (2) inefficiency problem during the evaluation process since it ignores all non-related facts for answering Q in the database. In a recent work, [Str05] studies the building of a top-down evaluation system for logic programs with uncertainty, which use fuzzy sets as the foundation of uncertainty. In that proposal, the evaluation engine searches all ground atoms w.r.t. the query as the first step, disregarding certainty values. After this lookup process stops, all related derivations from rules are recorded, and a set of equations are generated based on the ground atoms found and the combination functions used. However, the number of the equations to be solved in this case is much larger than the number of rules in the program, which can be considered as the number of equations dealt with in the fixpoint evaluation.

When the input collection of facts is large, the bottom-up approach is desirable. First of all, termination is a problem in the top-down evaluation. Even in the standard case without uncertainty, evaluation of a program may be trapped into an infinite loop without returning answers. When some goals are recursively found, the evaluation could be lead to a "dead end." We may use a book-keeping strategy to keep track of atoms already computed, but detection algorithms are usually expensive to implement. Secondly, we need to generate a rule/goal tree that has a large number of nodes. Whenever there is a certainty change in a leaf (atom), it is necessary to update all duplicated leaves. Considering the case in [Str05] which ignores all certainties first, we have to deal with a system with many equations. Thirdly, bottom-up approach guarantees termination in finite steps in Datalog because the herbrand base including all atoms is finite under the closed world assumption (CWA) [Lif85]. Finally, top-down requires unification, while bottom-up algorithms uses term-matching for joins which are one-way unifications and hence easier. Existing optimization techniques such as indexing may be applied for joins of massive relations with ease.

A "combination" solution of top-down and bottom-up, called "magic sets technique", was introduced in the standard case [BMSU86]. It is an attractive technique implementing the goal-oriented feature of top-down in a bottom-up evaluation approach. While magic sets technique may generate more rules, it is the principal technique in standard deductive databases to avoid computation of irrelevant facts for recursive rules [BMSU86, Ull89, MFPR96, BR87, Sun92]. The basic idea in magic sets is that the bottom-up evaluation of a logic program should be restricted to those facts that are "potentially relevant" with respect to the known query. The magic sets rewriting starts with a logic program P and a query Q with some bound arguments. The rewriting procedure chooses an order to pass the query bindings from the rule head to the rule body, called "sideways information passing" (SIP) [BR87]. Within a rule, SIP will occur for a fixed ordering of subgoals. A "Magic predicate" is defined for each bound version of the head and added to the rule, in order to restrict the joins of subgoals that are indeed restricted by the bound arguments. A rule will not be fired at each iteration unless the magic sets predicates hold necessary tuples.

The magic sets technique has also been extended to programs with uncertainty [JZ02], which uses sets as the structure of semantics, and uses fuzzy sets as the uncertainty foundation. In this work, we extend magic sets technique to IB Frameworks in DDBs with uncertainty which is multi-set based and uses fuzzy sets as well as other mathematical foundation of uncertainty.

1.2 Contributions of the Thesis

The major contributions of this thesis are:

- We extend generalized magic sets rewriting (GMS) and generalized supplementary magic sets rewriting (GSMS) techniques (Chapter 3) to deductive databases (DDBs) with uncertainty, which is multi-set based and is capable of using different mathematical foundations of uncertainty, such as probability. We established the correctness of the proposed techniques (Section 3.4 and 3.5).

- We modify the existing evaluation algorithms to compute the least fixpoint (Section

3.5) of a magic rewritten program in PF.

- We develop a prototype system (Chapter 4), called UNLOG, to measure the proposed techniques GMS/GSMS. For this, we created a number of test cases, performed experiments, and report the experimental results (Chapter 5). We compare the difference of efficiency gain for different run-time optimization techniques.

1.3 Thesis Outline

The rest of this thesis is organized as follows.

In Chapter 2, we briefly review basic concepts from the parametric framework (PF), and study, in particular, the fixpoint theory and relevant existing algorithms in Section 2.2. Then, we recall magic sets techniques in Datalog in Section 2.3.

In Chapter 3, we focus on technical details of our work on extending magic sets techniques to PF. We begin by presenting *straightforward* GMS and *straightforward* GSMS in Section 3.1 and 3.2. In Section 3.3, we identify problems in *straightforward* methods and discuss the challenges to extend magic sets techniques. We discuss our solutions and introduce the required modifications to the *straightforward* GMS and GSMS in Section 3.4 and 3.5.

In Chapter 4, we present our prototype system "UNLOG." We present the system architecture and its components (Section 4.1), the data structures (Section 4.2), and the optimization techniques implemented (Section 4.3).

In Chapter 5, we report the results of our experiments for evaluating performance of the proposed techniques. We describe the test programs and test data generation in Section 5.2 and 5.3. Evaluation criteria are introduced in Section 5.4. Finally, we report the test results and our analysis in Section 5.5.

Concluding our remarks and future research directions are introduced in last chapter.

Chapter 2

Background and Related Work

This section provides a background, and reviews related work. For the background, we review basic concepts and ideas for the parametric framework and magic sets technique introduced in [LS96, SZ08, BR87, Ull89].

Numerous frameworks have been proposed to represent and manage uncertain information in deductive databases. These frameworks may use, as a mathematical foundation of uncertainty, fuzzy logic, probability theory, multi-valued logic, possibility theory, and hybrid of numeric and symbolic values. They may also differ from (i) uncertainty manipulation, and (ii) the way in which uncertainty is associated with facts and rules in a program. On the basis of (ii), these frameworks are classified into annotated-based (AB) and implication-based (IB).

A rule in AB framework is an expression of the form:

$$A : f(\beta_1, \cdots, \beta_n) \leftarrow B_1 : \beta_1, \cdots, B_n : \beta_n$$

which asserts that "if the certainty of B_i is at least β_i, then the certainty of A is at least $f(\beta_1, ..., \beta_n)$, $1 \leq i \leq n$." Here f is a n-ary computable function and β_i is a certainty variable or value over an appropriate certainty domain.

A rule in an IB framework is of the form:

$$A \overset{\alpha}{\leftarrow} B_1, \cdots, B_n$$

which asserts "the certainty that conjunction $B_1 \wedge \cdots \wedge B_n$ implies A is α". Given a certainty assignment ν to the B_i's, the certainty associated with the result of this implication is derived as $\nu(A) = f_p(\alpha, f_c(\nu(B_1), \cdots, \nu(B_n)))$, where f_c and f_p are pre-defined propagation and conjunction functions.

Evidently the AB frameworks subsumes the IB frameworks, we do not argue about which of the two approaches, AB or IB, is better because it has been shown in [Shi05] that IB frameworks extended with certainty constraints are equally expressive as AB frameworks. This is done by showing that any logic program in AB framework can be translated into a p-program in the parametric framework, and vice versa.

2.1 The Parametric Framework: A Review

The parametric framework (PF) is a generic IB framework. Every logic program in an IB framework can be expressed as a p-program with appropriate parameters. In this section, we review the basic syntax and semantics of PF. For more detail, please refer to [LS96].

2.1.1 Syntax and Notations

Definition 2.1.1. [Complete Lattice][ea97]: A lattice $< T, \preceq >$ is a set partially ordered $\preceq$, with a meet $\otimes$ and a join operator $\oplus$. $< T, \preceq >$ is said to be a complete lattice iff for every subset T' of T, T' has a unique least upper bound and a unique greatest lower bound.

Definition 2.1.2. [P-program]: A parametric program P is a 5-tuple $\langle T, R, D, P, C \rangle$, whose components are defined as follows.

- T is certainty domain assumed to be a complete lattice with meet $\otimes$ and join operator $\oplus$. We use $\perp$ to denote the least element in the lattice, and $\top$ to denote its greatest element.

- R is a finite set of rules of the form

$$A \overset{\alpha}{\leftarrow} B_1, \cdots, B_n; \langle f_d, f_p, f_c \rangle$$

  where A, $B_1, \cdots, B_n$ are atoms, and $\alpha \in T - \{\perp\}$.

- D is a mapping that associates a rule with a disjunction function $f_d \in F_d$, where F_d is the set of disjunction functions.

- P is a mapping that associates a rule with a propagation function $f_p \in F_p$, where F_p is the set of propagation functions.

- C is a mapping that associates a rule with a conjunction function $f_c \in F_c$, where F_c is the set of conjunction functions.

The PF uses multi-sets, which is also called bags, as the underlining structure of the semantics. A multiset is a collection of unordered elements, any of which may occur more than once. Let B be any set, called base set of a multiset, then a multiset X over B is a mapping from B to $N = \{0, 1, \cdots\}$. In our context, B is the Herbrand base that is the set of ground atoms built from the predicates and constants in a logic program. We use $\{|\cdots|\}$ to denote multisets. For instance, $\{|(A : \alpha) : 3|\}$ is a multiset that indicates 3 copies of an element $x = (A : \alpha)$, where A is the base part of x, and α is the certainty associated with A. We use $\dot{\emptyset}$ to denote the empty multiset. If a multiset X is contained in another multiset Y, denoted $X \dot{\subseteq} Y$, then $\forall\ x = (A : \alpha) : m \in X$, we will have that $\exists\ y = (A : \alpha) : n \in Y$, where $m \leq n$. In this thesis, the following conventions are adopted. Lower case letters represent predicate symbols and constants; upper case letters with alphabetic order from $A, B, \cdots$ represent ground atoms; $X, Y, \cdots$ represents multisets as well as arguments. The lower case Greek letters $\alpha, \beta, \cdots$ are used to represent the certainty values in T. Finally, $\pi(A)$ is used to denote the predicate symbol of A, and $\nu(A)$ for denoting the certainty value assigned to A by ν.

2.1.2 Combination Functions

In PF, three types of combination functions are defined. They are disjunction functions F_d, conjunction functions F_c and propagation functions F_p. Each type satisfies a subset of the following properties.

1. Monotonicity: $f(\alpha_1, \alpha_2) \preceq f(\beta_1, \beta_2)$ if $\alpha_i \preceq \beta_i$ for $i = 1, 2$ and $\alpha_i, \beta_i \in T$.

2. Continuity: f is continuous.

3. Bounded-Above: $f(\alpha_1, \alpha_2) \preceq \otimes(\alpha_1, \alpha_2)$.

4. Bounded-Below: $f(\alpha_1, \alpha_2) \succeq \oplus(\alpha_1, \alpha_2)$.

5. Commutativity: $f(\alpha, \beta) = f(\beta, \alpha)$ for $\forall \alpha, \beta \in T$.

6. Associativity: $f(\alpha, f(\beta, \gamma)) = f(f(\alpha, \beta), \gamma)$ for $\forall \alpha, \beta, \gamma \in T$.

7. $f(\{|\alpha|\}) = \alpha$, $\forall \alpha \in T$.

8. $f(\dot{\emptyset}) = \bot$.

9. $f(\dot{\emptyset}) = \top$.

Postulate 2.1.1. *Certain type of the combination functions in the parametric framework should satisfy certain properties, postulated as follows.*

- Every conjunction function $f_c \in F_c$ should satisfy properties 1,2,3,5,6,7, and 9.
- Every disjunction function $f_d \in F_d$ should satisfy properties 1,2,4,5,6,7, and 8.
- Every propagation function $f_p \in F_p$ should satisfy properties 1,2, and 3.

Disjunction functions F_d is classified into three types as follows.

Definition 2.1.3. Let f_d be a disjunction function in the parametric framework, then f_d is of one of the following types:

1. Type 1: $f_d = \oplus$, i.e., f_d coincides with the lattice join.
2. Type 2: $\oplus(\alpha, \beta) \prec f_d(\alpha, \beta) \prec \top, \forall \alpha, \beta \in T - \{\bot, \top\}$.
3. Type 3: $\oplus(\alpha, \beta) \prec f_d(\alpha, \beta) \preceq \top, \forall \alpha, \beta \in T - \{\bot, \top\}$, i.e., there are the certainty values $\alpha', \beta' \in T - \{\bot, \top\}$, such that $f_d(\alpha', \beta') = \top$.

A type 1 disjunction function coincides with the join operator $\oplus$ in the underlying certainty lattice T, such as $f_d = max$. A type 2 disjunction function always returns a "larger" value, while a type 3 disjunction function may return $\top$ when its arguments are different from $\top$. For instance, $f_d = ind$, where $ind(\alpha, \beta) = \alpha + \beta - \alpha * \beta$, is a type 2 disjunction function, whereas the negative correlated function $nc = min(1, \alpha + \beta)$ is a type 3 disjunction function.

2.1.3 Fixpoint Theory

The fixpoint theory in standard deductive database is concerned with computing the least model of the program in a bottom-up fashion, starting with the facts and applying the rules repeatedly until no new fact is derived. This has been extended in [LS96] to compute the fixpoint semantics of programs in parametric framework.

Definition 2.1.4. [LS96] Let P be any p-program, and P^* be the Herbrand instantiation of P. Also let Υ_P be the set of all valuations of P. The immediate consequence operator T_p is a mapping from Υ_P to Υ_P, such that for every valuation $\nu \in \Upsilon_P$ and every ground atom $A \in B_p$, $T_p(\nu)(A) = f_d(X)$, where B_p is the Herbrand base of P, f_d is the disjunction function associated with $\pi(A)$, the predicate symbol of A, and

$$X = \{|f_p(\alpha_r, f_c(\{|\nu(B_1), \cdots, \nu(B_n)|\}))|(A \overset{\alpha_r}{\leftarrow} B_1, \cdots, B_n; \langle f_d, f_p, f_c\rangle) \in P^*|\}$$

The bottom-up fixpoint evaluation of P is then defined as follows:

$$T_p{}^k = \begin{cases} \nu_\perp & \text{if } k = 0 \\ T_p(T_p{}^{k-1}) & \text{if } k \text{ is a successor ordinal} \\ \oplus\{T_p{}^l | l < k\} & \text{if } k \text{ is a limit ordinal} \end{cases}$$

Note that $\nu_\perp(A) = \perp$, $\forall A \in B_p$. It has been shown that T_p is monotone and continuous. For $k = 0$, T_p evaluates the facts with $f_p(\alpha_r, f_c(\dot{\emptyset})) = \alpha_r$. For any $A \in B_p$, if A does not unify with the head of any rule in P, then $T_p(\nu)(A) = \perp$.

2.2 Bottom-up Fixpoint Evaluation Algorithms: A Review

The basic bottom-up least fixpoint evaluation in the PF is the naive evaluation (N) extended from the standard naive method by taking into account the presence of certainties.

Given a p-program P, every atom is initially assigned the least certainty value $\perp$ in the naive evaluation. At each iteration i, we apply every possible rule. For every ground atom A derived, the disjunction function f_d associated with the predicate of A is used to combine the multiset of certainties associated with derivations of A into one. The evaluation terminates at the iteration in which the certainty of no atom is improved. Figure 2.1 shows the multiset

```
procedure Naive(P, D, $lfp(T_{P\bigcup D})$)
input: P : a set of parametric rules(IDB);
       D : a set of atom-certainty pairs(EDB).
output:$\nu$ : the least fixpoint valuation of $P \bigcup D$.
begin
  forall $A \in B_p$
    $\nu_0(A) := \bot$;
    $M_1(A) := \{|\alpha|(A : \alpha) \in D|\}$;
    $\nu_1(A) := f_d(M_1(A))$;
  end forall
  $new_1 := \{A|(A : \alpha) \in D\}; i := 1$;
  while($new_i \neq \varnothing$)
    i:=i+1
    forall $\forall(A \xleftarrow{\alpha_r} B_1, \ldots, B_n; \langle f_d, f_p, f_c \rangle) \in P^*$;
      $M_i(A) := \{|f_p(\alpha_r, f_c(\{|\nu_{i-1}(B_1), \ldots, \nu_{i-1}(B_n)|\}))|\}$;
    end forall;
    $\nu_i(A) := f_d(M_i(A))$;
    $new_i := \{A|A \in B_p, \nu_i(A) \succ \nu_{i-1}(A)\}$;
  end while
  $lfp(T_{P\cup D}) := \nu_i$
end procedure
```

Figure 2.1: A multiset based naive algorithm for p-programs [LS96]

based naive algorithm for p-programs.

The naive evaluation in PF suffers from the same two problems as the naive method in Datalog: (1) a tuple $(A : \alpha)$ derived at iteration i continues to be derived at every next iteration, and (2) a goal structure is looked up only when the fixpoint is reached. A part of the redundant computation in the naive evaluation is avoided by using semi-naive (SN) algorithm [LS96] as follows. A pair $\langle M_i, \sigma_i \rangle$ is associated with every ground atom A at iteration i, where M_i is a multiset of all certainties of A derived from the program, and $\sigma_i = f_d(M_i)$ is the combined certainty of A. Every element (r, α) in M_i indicates a derivation of A with certainty α obtained by a ground instance of rule r. Only those rules with "new" subgoals in the body are re-evaluated at iteration $i+1$, and the certainty of the corresponding derivation of A is replaced. Otherwise, we use the certainty of A at iteration i as its certainty at iteration $i+1$. The SN algorithm yields the same result as the naive

```
1: procedure Semi_Naive(P, D, $lfp(T_{P\bigcup D})$)
2:   forall $A \in B_p$
3:     $\nu_0(A) := \perp$
4:     $M_1(A) := \{|\alpha|(A:\alpha) \in D|\}$;
5:     $\nu_1(A) := f_d(M_1(A))$;
6:   end forall
7:   $new_1 := \{A|(A:\alpha) \in D\}$; $i := 1$;
8:   while($new_i \neq \emptyset$)
9:     i:=i+1;
10:    forall $A \in new_i$
11:      if $\exists(A \xleftarrow{\alpha_r} B_1,\ldots,B_n; \langle f_d, f_p, f_c\rangle) \in P^*$,
           such that $\exists B_j \in new_i$, for $j \in \{1,\ldots,n\}$
12:      then begin
13:          $M_i(A) := M_{i-1}(A)$;
14:          forall $(r : A \xleftarrow{\alpha_r} B_1,\ldots,B_n; \langle f_d, f_p, f_c\rangle) \in P^*$
             such that $\exists B_j \in New_i$,for some $j \in \{1,\ldots,n\}$;
15:            $M_i(A) := M_i(A) - \{|\sigma^r_{i-1}(A)|\} \bigcup \{|\sigma^r_i(A)|\}$, where
16:            $\sigma^r_i(A) := f_p(\alpha_r, f_c(\{|\nu_{i-1}(B_1),\ldots,\nu_{i-1}(B_n)|\}))$;
17:          end forall;
18:          $\nu_i := f_d(M_i(A))$, where $f_d := Disj(\pi(A))$;
19:      end else $\nu_i(A) := \nu_{i-1}(A)$;
20:    end forall;
21:    $new_i := \{A|A \in B_p, \nu_i(A) \succ \nu_{i-1}(A)\}$;
22:  end while
23:  $lfp(T_{P\bigcup D}) := \nu_i$;
24:end procedure
```

Figure 2.2: A multiset based SN algorithm for p-programs [LS96]

method at every iteration and hence they are equivalent. Figure 2.2 shows this multiset based SN algorithm for p-programs.

A rule r in SN may be applied multiple times at iteration i, but not all of them may yield improved certainties. If a derivation of r yields an improved certainty, the SN method above replaces all previous derivations of r by the new derivations, even though most of them may not yield improved certainties. The efficiency of SN is further improved by the semi-naive with predicate partitioning (SNP) [SZ04], which distinguishes between improved certainties from different derivations at iteration i and unimproved certainties from others, and correspondingly partitions each IDB relation into the new and the old: improved and

```
1: procedure $Semi\_Naive\_Partion(P, D, lfp(T_{P\bigcup D}))$
2:     forall $A \in B_p$;
3:         $C_1(A) := \{|(\alpha : \emptyset)|(A : \alpha) \in D|\}$
4:         $\nu_1(A) := f_d(C_1(A)(\alpha))$;
5:     end forall
6:     $new_1 := \{A|(A : \alpha) \in D\}; i := 1$;
7:     while $(new_i \neq \varnothing)$
8:         ı:=i+1;
9:         forall $A \in B_p$ :
10:            $C_i(A) := C_{i-1}(A)$;
11:            forall $B \in new_{i-1} \bigwedge (\alpha, S_B) \in C_{i-1}(A) \bigwedge B \in S_B$
12:                $C_i(A) := C_i(A) - \{|(\alpha, S_B)|\}$
13:            end forall
14:            forall $(A \xleftarrow{\alpha_r} B_1, \ldots, B_n; \langle f_d, f_p, f_c \rangle) \in P^* \bigwedge \exists B_j \in new_i$,
               where $j \in \{1, \ldots, n\}$
15:                $C_i(A) := C_i(A) \dot{\bigcup} \{|(\alpha_i^r(A), S_B)|\}$, where
                   $\alpha_i^r(A) := f_p(\alpha_r, f_c(\{|\nu_{i-1}(B_1), \ldots, \nu_{i-1}(B_n)|\}))$, and
                   $S_B := \{B_j | j \in \{1, \ldots, n\}\}$
16:            end forall;
17:            $\nu_i := f_d(C_i(A)(\alpha))$;
18:        end forall;
19:        $new_i := \{A | A \in B_p, \nu_i(A) \succ \nu_{i-1}(A)\}$;
20:    end while
21:    $lfp(T_{P\bigcup D}) := \nu_i$;
22:end procedure
```

Figure 2.3: A multiset based SNP algorithm for p-programs [SZ04]

non-improved atoms. At iteration $i + 1$, if the certainty of a tuple t is not improved, compared to t's certainty at iteration i, then t will stay in or be moved to the old partition. There must be one tuple selected in a new partition to contribute to joins and hence, the evaluation process will continue.

In more details, the SNP algorithm replaces M_i with C_i that is a multiset with all certainties derived from different paths, and $\sigma_i = f_d(C_i)$ is the certainty of a ground atom A. A pair $\langle C_i, \sigma_i \rangle$ is associated with A at iteration i. Every element in C_i is of form (α, S_i), where α is A's certainty and S_i contains all IDB subgoals from the previous iterations. At iteration $i + 1$, derivations with at least one improved tuple in S_i are re-evaluated. Figure 2.3 shows the SNP algorithm for p-programs.

2.3 Magic Sets Techniques in Datalog

The idea of magic sets rewriting comes from sideways information passing (SIP) strategy [BR87]. Intuitively, SIP induces an order among the rules and subgoals of a rule, when evaluating a logic program. Magic sets act like filters that hold values for bound variables. A rule is applied if the instances of the variables in its IDB subgoals unify the instances of the variables in the corresponding magic predicates. A *magic predicate m_p* is created for an IDB predicate p with unique variables-binding pattern. A *supplementary magic predicate sup_p* is created for a subgoal p with unique variables-binding pattern. Magic predicates hold "bound" arguments, but supplementary magic predicates hold both "bound" and "free" arguments that appear in the path in SIP. Depending on the types of magic predicates used in a rewriting, magic set rewriting approaches are classified into magic sets rewriting (MS) and supplementary magic sets rewriting (SMS) [BR87]. Given a program P, EDB D and a query Q, the magic sets rewriting technique is to generate a magic rewritten program P^m with $EDB = D \cup M$, where M is the initial tuples for magic predicates such that P^m and P produce the same set of answers w.r.t. Q.

Definition 2.3.1. Subgoal-rectified: A rule is said to be subgoal-rectified if it includes no repeated argument in the body. For example, a rule

$$p(X,Y)\text{:- } e(X,Y),\, p(Y,Y).$$

is not subgoal-rectified because Y appears more than once in the body.

"Generalized magic sets rewriting" (GMS) and "generalized supplementary magic sets rewriting" (GSMS) were introduced in [BR87, Ull89]. For a non-subgoal-rectified rule, MS binds the same arguments in different positions of subgoals in multiple steps, but GMS will consider them in one step. Therefore, the MS and GMS techniques produce the same rewritten program in this case. Otherwise, MS will generate more magic rule than GMS in the rewritten program because a predicate might have more forms if its arguments are bounded in more steps. In this case, the rewritten program generated from MS cannot achieve the same performance as GMS does. In this thesis, we consider more general cases GMS and GSMS and extend them to the PF.

Example 2.3.1 shows the well-known same-generation cousin (SGC) program in Datalog and its GMS program. After the GMS procedure, one rule and one fact that holds the bound argument in the query has been added, and two rules have been modified. Let's consider rule $r1'$, a new predicate has been added to its body: $magic\_sgc(X)$. The presence of this new predicate restricts $r1'$ to be fired only if the person X is related to answer the query. Rule $r2'$ says that if X_1 is the parent of X, X_1 should contribute to answer the query and is stored into *magic_sgc*. In rule $r3'$, the added magic predicate $magic\_sgc(X)$ forces the argument X of the EDB relation *par* to assume some specific values which belong to *magic_sgc* and hence, X_1 in *par* is restricted to expect specific values in $sgc(X_1, Y_1)$. Rule $r4'$ simply stores the initial constant *anna* to start the evaluation.

Example 2.3.1. An Example: GMS transformation of a SGC program

Original program P:
$r1 : sgc(X, X) \leftarrow person(X).$
$r2 : sgc(X, Y) \leftarrow par(X, X_1), sgc(X_1, Y_1), par(Y, Y_1).$
$? - sgc(Anna, Y).$
The GMS rewritten program P^m:
$r1' : sgc(X, X) \leftarrow magic\_sgc(X), person(X).$
$r2' : magic\_sgc(X_1) \leftarrow magic\_sgc(X), par(X, X_1).$
$r3' : sgc(X, Y) \leftarrow magic\_sgc(X), par(X, X_1), sgc(X_1, Y_1), par(Y, Y_1).$
$r4' : magic\_sgc(anna).$

Let us consider the evaluation of the program shown in Example 2.3.1 on the following sample data set D.

$D = \{person(Anna), person(Tom), person(Jack), person(George), person(Sam),$
$person(Mike), par(Anna, Jack), par(Tom, Jack), par(Mike, sam),$
$par(Geoge, Sam)\}.$

The data set D represents relationships of two small families, shown as Figure 2.4.

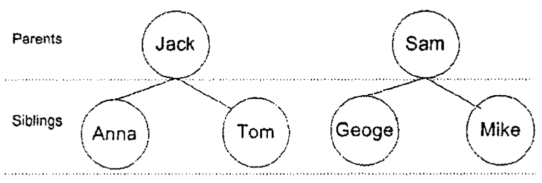

Figure 2.4: A family graph

On one hand, the program P on the data set D, at iteration 1 rule $r1$ is fired and yields $\{sgc(Anna, Anna), sgc(Tom, Tom), sgc(Jack, Jack), sgc(George, George), sgc(Sam, Sam), sgc(Mike, Mike)\}$. At iteration 2, rule $r1$ does not yield new tuples. Rule $r2$ yields $\{sgc(Anna, Tom)$, $sgc(Tom, Anna)$, $sgc(Mike, George)$, $sgc(George, Mike)\}$. The evaluation process terminates in iteration 3 since there is no new tuple discovered. The answer set for the query is $sgc(Anna, Y)$ is $\{sgc(Anna, Anna)$, $sgc(Anna, Tom)\}$.

On the other hand, considering the program P^m on the data set D, at iteration 1 rule $r1'$ is fired, but only $sgc(Anna, Anna)$ is yielded since $person(X)$ is restricted by $magic\_sgc(X)$ which contains only $Anna$ at that moment. At the same time, $Anna$'s parent, $Jack$, is discovered by $r2'$ and stored into the relation $magic\_sgc$. At iteration 2, $sgc(Jack, Jack)$ is discovered by rule $r1'$, but rule $r2'$ does not yield new tuples. At iteration 3, only rule $r3'$ yields a new tuple $sgc(Anna, Tom)$ since the variable X in $par(X, X1)$ is bound by $magic\_sgc(X)$, so X_1 is restricted to expect $sgc(Jack, Y_1)$ in $r3'$. Finally, the evaluation process stops. Some *sgc* tuples are not even derived, such as $sgc(Tom, Anna)$, and the facts generated are related to the answer set because joins for each rewritten rule are restricted by the magic tuples. Hence the rewritten program improves the evaluation speed by conduct less joins to achieve the answer set.

Example 2.3.1 also illustrates the idea of SIP. Given a rule r and a subgoal p in the body with some bound arguments, the binding information is used to obtain the bindings for non-instantiated variables in other argument positions. This process of passing information is repeatedly applied for each subgoal, and recursively on other relevant rules. Thus, known information is passed sideways within the whole program until the evaluation terminates.

Chapter 3

Extending Magic Sets Techniques to Parametric Framework

The objective to apply the magic sets techniques is to speed up the evaluation of goal-oriented programs with uncertainty. We are likely to encounter the following questions: How to manipulate the combination functions when rewriting a program? Can existing evaluation algorithms be applied for the rewritten programs? Does the rewritten program produce desired results? We answer these questions in this chapter by extending both generalized magic sets rewriting (GMS) and generalized supplementary magic sets rewriting (GSMS) techniques to parametric framework (PF).

The extended GMS includes two stages: *straightforward* GMS and magic tuples generation. We use the word *straightforward* to show that the proposed procedure inherits major steps of GMS in Datalog, while we extend the capability of each step to deal with combination functions. The challenge for the *straightforward* GMS is to establish its correctness when the type 2 disjunction function is applied. We claim that the second stage of the rewriting that pre-computes all magic tuples is necessary to adapt the proposed GMS technique in a practical application.

The extended GSMS also includes two rewriting stages: *straightforward* GMS and magic tuples generation. In the first stage, we carefully manipulate combination functions for the written rules so as to retain the original meaning or a program. Unlike GMS, we need to modify the existing evaluation algorithms to evaluate the GSMS rewritten program.

The rest sections are organized as follows. In Section 3.1, we introduce the *straightforward* GMS and establish its correctness. In Section 3.2 we introduce the *straightforward* GSMS and its correctness discussion. An example is introduced in Section 3.3 to identify the challenge in extending GMS to PF. We present the relevant modifications for GMS and GSMS in Section 3.4 and 3.5. Finally, the efficiency discussion of the rewriting algorithms is presented.

3.1 *Staightforward* GMS

In this section a *straightforward* GMS procedure is introduced to solve the second source of inefficiency, i.e., computing facts which are unrelated to the query. As in standard Datalog, GMS in PF follows sideways information passing strategy (SIPs) [BR87]. The difference is that here we take into account the presence of uncertainty and combination functions. The major steps to accomplish GMS transformation in PF are as follows:

- Adorned rules generation
- Magic rules generation
- Rewriting the adorned rules

Given a p-program P and a query Q, the rewriting process starts from Q, and generates a set of adorned rules (Section 3.1.1) which have the same IDB predicate as Q. More bound

IDB predicates are then discovered, so we use one of these bound predicates to generate more adorned rules until all discovered bound IDB predicates are considered. For each adorned rule, we generate a set of magic rules for it, and rewrite it by adding the proper magic predicates. Finally, we create a tuple for the magic predicate related to Q, called seed. As the result of the *straightforward* GMS, we get a version of the rewritten program of P, denoted as P^m which includes all magic rules, the rewritten adorned rules and the seed. The following sections describe the details of the major steps of the *straightforward* GMS through the rewriting process.

3.1.1 Adorned Rules Generation for GMS

A binding pattern for a predicate is a set of bound arguments. An adornment for an n-ary predicate p is a string of length n on the alphabet $\{b, f\}$, where b stands for bound argument and f for free. A predicate p adorned with a binding pattern a, denoted as p^a, indicates the bound and free arguments of p. For example, p^{bf} indicates that predicate p is a binary predicate, the first argument of which is bound and the second is free.

Let P be a program and Q be a query. The process of generating adorned rules starts by considering the binding patterns of Q. We create a collection C for all adorned predicates. Initially, the binding pattern of Q is in C. The adorned predicates in C are processed one at a time and are then marked, so that they are not processed again. Let p^a be an unmarked adorned predicate in C, for each rule r that has p in the head, we generate an adorned version of r, called adorned rule and denoted as r^{ad}. If r^{ad} includes more adorned predicates, they are added to the collection C unless they are dealt with before. The process terminates when there is no unmarked adorned predicate left in C. Termination of this process is guaranteed since the number of adorned predicates for any specific program is

```
procedure $BPG(r, h^a)$
input: $r : h(\bar{X}) \overset{\alpha}{\leftarrow} p_1(\bar{X_1}), ..., p_n(\bar{X_n}); \langle f_d, f_p, f_c \rangle$ and an adorned predicate $h^a$.
output:An adorned version of $r^{ad}$, where the subgoals are re-ordered
        depending on the order of bound variables.
begin
    Initialize a stack $S$;//used to save bound    variables
    forall variable $X_i$ of $h(\bar{X})$ and the $i_{th}$ character $c$ of $a$
        if $(c =' b')$ and $X_i$ is marked 'b';
        else $S.push(X_i)$;
    end forall
    while $(Y = Q.pop())$
        forall subgoal $q = p_i(\bar{X_i})$ of r
            forall variable $Y_j$ of $p_i(\bar{X_i})$
                if $(Y = Y_j$ and $Y$ is not from $q)$ {
                    $Y_j$ is marked 'b';
                    $q$ is marked 'Visited'; q.order++;
                    A copy of $S$ is attached to $q$;}
                else {
                    $Y_j$ is marked 'f';
                    $S.push(Y_j)$; }
            end forall
        end forall
    end while
    Sort the body or $r$ based on the order of attached variables;
end
```

Figure 3.1: Adorned rules generation algorithm for GMS

finite.

Generating an adorned rule follows sideways information passing strategy. For an (unmarked) adorned predicate h^a and a rule r with head h, we create the binding pattern a. The bound variables are added to a new collection S. Next, we replace each subgoal of r by an adorned version. If this version is new, we add it to C. To obtain the adorned form of a subgoal in the body, we select a variable Y from S. For every argument of each subgoal p, if p has Y as a argument, we mark it as 'b' and the rest of the variables are collected into S if they do not appear there. We then consider the next variable in S until all variables in S are considered. The number of predicates in an adorned rule is the same as its original rule, but the arguments of each predicate are adorned.

Figure 3.1 shows the process in which a stack is used to save the bound variables. In general, the order in which we bound the variables is not unique.

3.1.2 Generalized Magic Rules Generation for GMS

Once the adorned version r^{ad} of a rule r is generalized, we may generate generalized magic rules. Figure 3.2 illustrates the steps to generate generalized magic rules for an adorned rule. A generalized magic rule is a rule with magic predicate as the head and its body also include at least one magic predicate. A method called *getMagicPredicate*(p^{ad}) is defined to return the corresponding magic predicate for an adorned predicate p^{ad}. If p^{ad} is an IDB predicate, the method returns a generalized magic predicate m_p^{ad}, based on the binding pattern *ad*; otherwise, it returns p^{ad}. The arguments of m_p^{ad} are the bound arguments in p^{ad}. For each IDB subgoal ${p_i}^{ad_i}$ in an adorned rule r^{ad}, where i indicates the position of

```
1: procedure $GenerateGMSR(r^{ad})$
2: input: An adorned rule $r^{ad} : h^{ad_0}(\bar{X}) \overset{\alpha}{\leftarrow} {p_1}^{ad_1}(\bar{X}_1), ..., {p_n}^{ad_n}(\bar{X}_n); \langle f_d, f_p, f_c \rangle$
3: output: A collection of generalized magic rules for $r^{ad}$
4: begin
5:     Initialize a collection $C$;//used to save magic rules
6:     forall subgoal $q = {p_i}^{ad_i}(\bar{X}_i)$ 7: in $r^{ad}$
8:        if ($\pi(q)$ is a IDB predicate name) {
9:           Create a magic rule ${r_m}^{ad}$;
10:          ${r_m}^{ad}.f_d = max$; ${r_m}^{ad}.f_p = max$; ${r_m}^{ad}.f_c = max$; ${r_m}^{ad}.\alpha = \top$;
11:          ${r_m}^{ad}.head = getMagicPredicate(q)$;
12:          Add $m\_h = getMagicPredicate(r^{ad}.head)$ to the body of $r^m$;
13:          for $j = 1$ to $i$ do
14:             $q_j = {p_j}^{ad_j}(\bar{X}_j)$;// $j$ is the position in the body of $r$
15:             Add $m\_q_j = getMagicPredicate(q_j)$ to the body of $r^m$;
16:          end for
17:          Add ${r_m}^{ad}$ to $C$; }
18:    end forall
19:    return $C$;
20:end
```

Figure 3.2: Generalized magic rules generation algorithm for GMS

${p_i}^{ad_i}$ in the body, we generate a magic rule ${r_m}^{ad}$ with certainty $\top$, and defining predicate

$m\_p_i{}^{ad_i}$ as the magic predicate of p_i. As the combination functions associated with r^{ad}, we use $\langle max, max, max \rangle$. This associates $\top$ with all ground tuples of the magic predicates in the rewritten program. The rules extended with magic predicates are not affected when they are evaluated because $f(\alpha, \top) = \alpha$, for all certainty $\alpha \in T$. The head of $r_m{}^{ad}$ for the IDB subgoal $p_i{}^{ad_i}$ is the corresponding magic predicate of the head of r^{ad}. We then add *getMagicPredicate*($p_i{}^{ad_i}$) to the body of $r_m{}^{ad}$. For each subgoal at the position $j \in [1, i]$ in the body of r^{ad}, we add $p_j{}^{ad_j}$ to the body of $r_m{}^{ad}$. Note that an adorned predicate may have several occurrences in the same adorned rule, so several rules that define $m\_p_i{}^{ad_i}$ might be generated from a single adorned rule.

3.1.3 Rewriting the Adorned Rules for GMS

The remaining operations to complete a *straightforward* GMS include rewriting the adorned rules and seeding the facts for magic predicates. We add the magic predicate of h^a to the body of each adorned rule with h as the head and then get a collection of rewritten rules for the original rules, referred to as R^m. Then, we create a seed that contains the bound arguments in Q, and is with predicate name $\pi(Q)$ and certainty $\top$. This completes the rewriting process and the rewritten program includes the magic rules, the rules collection R^m, the tuples from original program, and the seed. Example 3.1.1 shows the adorned rules and the transformed program using the proposed GMS rewriting. Note that $*$ is the regular multiplication operation.

While users do not consider "max" as a propagation function because it does not satisfy property 3 mentioned in Section 2.1.2, we allow "max" in magic rules in order to preserve the original meaning of the program, viewing this use as "internal". The following definition define this "internal" property for the magic predicates.

Definition 3.1.1. [Definitely Relevant]: If a variable X in a rule is "definitely relevant" in computing an answers, then the certainty of X belonging to the magic predicate m_p is $\top$, i.e, $m_p(X)$ is certainly true.

Example 3.1.1. An Example: *straightforward* GMS of a p-program

Original Program:

$p(X,Y) \overset{0.5}{\leftarrow} a(X,Y); \langle ind, *, * \rangle.$
$p(X,Y) \overset{0.5}{\leftarrow} p(Y,Z), p(Y,X); \langle ind, *, * \rangle.$
$a(1,2) \overset{0.5}{\leftarrow}; \langle \_, \_, \_ \rangle.$
$a(2,1) \overset{0.5}{\leftarrow}; \langle \_, \_, \_ \rangle.$
$a(1,1) \overset{0.5}{\leftarrow}; \langle \_, \_, \_ \rangle.$
$?p(1,Y).$

Adorned Rules:

$p^{bf}(X,Y) \overset{0.5}{\leftarrow} a(X,Y); \langle ind, *, * \rangle.$
$p^{fb}(X,Y) \overset{0.5}{\leftarrow} a(X,Y); \langle ind, *, * \rangle.$
$p^{bf}(X,Y) \overset{0.5}{\leftarrow} p^{fb}(Y,X), p^{bf}(Y,Z); \langle ind, *, * \rangle.$
$p^{fb}(X,Y) \overset{0.5}{\leftarrow} p^{bf}(Y,Z), p^{bf}(Y,X); \langle ind, *, * \rangle.$

The *Staightforward* GMS Rewritten Program:

$p\_bf(X,Y) \overset{0.5}{\leftarrow} magic\_p\_bf(X), a(X,Y); \langle ind, *, * \rangle.$
$magic\_p\_fb(X) \overset{1}{\leftarrow} magic\_p\_bf(X); \langle max, max, max \rangle.$
$magic\_p\_bf(Y) \overset{1}{\leftarrow} magic\_p\_bf(X), p\_fb(Y,X); \langle max, max, max \rangle.$
$p\_bf(X,Y) \overset{0.5}{\leftarrow} magic\_p\_bf(X), p\_fb(Y,X), p\_bf(Y,Z); \langle ind, *, * \rangle.$
$p\_fb(X,Y) \overset{0.5}{\leftarrow} magic\_p\_fb(Y), a(X,Y); \langle ind, *, * \rangle.$
$magic\_p\_bf(Y) \overset{1}{\leftarrow} magic\_p\_fb(Y); \langle max, max, max \rangle.$
$magic\_p\_bf(Y) \overset{1}{\leftarrow} magic\_p\_fb(Y), p\_bf(Y,Z); \langle max, max, max \rangle.$
$p\_fb(X,Y) \overset{0.5}{\leftarrow} magic\_p\_fb(Y), p\_bf(Y,Z), p\_bf(Y,X); \langle ind, *, * \rangle.$
$a(1,2) \overset{0.5}{\leftarrow}; \langle \_, \_, \_ \rangle.$
$a(2,1) \overset{0.5}{\leftarrow}; \langle \_, \_, \_ \rangle.$
$a(1,1) \overset{0.5}{\leftarrow}; \langle \_, \_, \_ \rangle.$
$magic\_p\_bf(1) \overset{0.5}{\leftarrow}; \langle \_, \_, \_ \rangle.$

3.1.4 Correctness of *Staightforward* GMS

Let P be a p-program, Q be a query, and P^m be the GMS rewritten program. We will show that P^m produces the same results w.r.t. Q. We construct three sequences $\{x_n\}$,$\{y_n\}$, and $\{z_n\}$ for each derived atom $A \in B_p$ (without certainty values), where B_p is Herbrand base of P. The sequence $\{x_n\}$ is the sequence of certainty values at different iterations in a bottom-up fixpoint computation of P^m. $\{y_n\}$ is the sequence of certainty values at different iterations for P. $\{z_n\}$ is a sub-sequence of $\{x_n\}$, selected as described in section 3.1.4. In general, $\{x_n\}$, $\{y_n\}$, and $\{z_n\}$ are infinite sequences, but in case they are finite and have different number of elements, we simply repeat the last element to maintain the sequences which have the same size because the last element of each sequence represents the certainty value of A in the fixpoint. We prove that these three sequences converge to the same certainty value, indicating A's certainty in the limit.

Definition 3.1.2. [Iterative Aggregated View D_i]: Let D_i be the set of atom-certainty pairs with their associated certainties obtained at iteration i in a fixpoint evaluation of the program.

Definition 3.1.3. [Atom Collection]: $Pure(D_i) = \{A | (A : \alpha) \in D_i, \alpha \succ \perp\}$.

Definition 3.1.4. We use $New(D_i)$ to represent the set of all atom-certainty pairs found the first time at iteration i with a certainty greater than $\perp$. In symbol, $New(D_i) = \{(A : \alpha) | (A : \alpha) \in D_i, (A : \beta) \notin D_{i-1}$ and $\alpha, \beta \succ \perp\}$

The next result shows that without new atom being derived at iteration i, there will be if no new atom derived at iteration $i+1$. In this case, a fixpoint evaluation continues beyond iteration $i+1$ only because the certainty of some atom A in $Pure(D_i)$ is improved.

Lemma 3.1.1. *Considering two consecutive iterations, if $New(D_i) = \emptyset$, then $New(D_{i+1}) = \emptyset$.*

Proof. Suppose that $New(D_i) = \emptyset$, then $Pure(D_i) = Pure(D_{i-1})$, i.e., for any $A \in Pure(D_i)$, we have $A \in Pure(D_{i-1})$. In this case, the program evaluation would not yield any new ground atoms based on D_{i-1}, no matter which rule is fired. On the other hand, suppose $New(D_{i+1}) \neq \emptyset$, then there is at least one new ground atom $A \in Pure(D_{i+1})$, but $A \notin Pure(D_i)$. In this case, the evaluation yields some new ground atom based on $Pure(D_i)$, which is the same as $Pure(D_{i-1})$ – a contradiction. □

Theorem 3.1.2. ***[Convergence]:*** *Considering the sequences $\{x_n\}$, $\{y_n\}$, and $\{z_n\}$ such that $x_n \leq y_n \leq z_n$, for all $n > i$ for some $i \in N$, if $\{x_n\}$ and $\{z_n\}$ both converge to the same limit α, then $\{y_n\}$ also converges to α.*

Proof. The above result is well-known, e.g., see page 84 in [Zor04] for a proof. $\square$

Note that a Datalog program is just a p-program where certainty values are restricted to $\{1, 0\}$, with *max* as the disjunction and *min* as propagation and conjunction functions. In standard Datalog, [BR87] has established the correctness of GMS transformation. Given a p-program P with a query Q and its *straightforward* GMS rewritten program P^m, Lemma 3.1.1 implies that all ground atoms derivable from P are found in finite number of iterations, denoted as I. Precisely, the number of iterations is polynomial in the number of constants in the EDB and the program. Therefore, the major task to show the correctness of transformation is to demonstrate that the certainties associated with every ground atom in P w.r.t Q in the fixpoint is the same as the one in P^m w.r.t Q.

Theorem 3.1.3. ***[Correctness of*** **Straightforward** ***GMS]:*** *Considering a p-program P, a query Q, and a set of EDB facts D. Suppose P^m is the* straightforward *GMS rewritten program with facts $D^m = D \cup M$, where M is the initialized tuples/facts in the magic predicates. Then naive fixpoint evaluation of P and P^m produce the same results w.r.t Q.*

Proof. Given a p-program P with a set of rules R and a set of facts D, each rule r in P can be rewritten as follows:

$$\text{r: } h(\bar{X_0}) \overset{\alpha}{\leftarrow} q_1(\bar{Y_1}), ..., q_t(\bar{Y_t}), p_1(\bar{X_1}), ..., p_n(\bar{X_n}); \langle f_d, f_p, f_c \rangle,$$

where q_j is an EDB predicate, and p_i is an IDB predicate. Each rule r is transformed into a set of magic rules and a number of transformations of r with adorned predicates. The adorned rule and magic rules are of the form:

$$h^{ad}(\bar{X_0}) \overset{\alpha}{\leftarrow} q_1(\bar{Y_1}), ..., q_t(\bar{Y_t}), p_1^{ad1}(\bar{X_1}), ..., p_n^{adn}(\bar{X_n}); \langle f_d, f_p, f_c \rangle.$$

$$h^{ad}(\bar{X_0}) \overset{\alpha}{\leftarrow} magic_h{}^{ad}, q_1(\bar{Y_1}), ..., q_t(\bar{Y_t}), magic_{p_1}{}^{ad_1}, p_1^{ad1}(\bar{X_1}), ...,$$

$$magic_{p_n}{}^{adn}, p_n^{ad_n}(\bar{X_n}); \langle f_d, f_p, f_c \rangle$$

For every ground instance of r with head atom $h(\bar{A})$ in P and $h^{ad}(\bar{A})$ in P^m, we construct three certainty sequences $\{x_n\}, \{y_n\}$ and $\{z_n\}$, where $x_0 = y_0 = z_0 = \bot$, and

$x_i = \nu_i(h^{ad}(\bar{A}))$

$y_i = \nu_i(h(\bar{A}))$

$z_i = \nu_{I+(i-1)}(h^{ad}(\bar{A}))$

I indicates the iteration that $Pure(New(D_I^m)) = \emptyset$, but $Pure(New(D_{I-1}^m)) \neq \emptyset$.

Case $i = 1$:

If there is a derivation "$h^{ad}(\bar{A}) \overset{\alpha}{\leftarrow} magic_h{}^{ad}, q_1(\bar{E_1}), ..., q_t(\bar{E_t})$" for h^{ad} at iteration 1, then there is also a derivation "$h(\bar{A}) \overset{\alpha}{\leftarrow} q_1(\bar{E_1}), ..., q_t(\bar{E_t})$" in P at iteration 1 because some rules in P^m may not have being fired since more tuples are yet to be derived for the magic predicates. Also if there is a derivation "$h(\bar{A}) \overset{\alpha}{\leftarrow} q_1(\bar{E_1}), ..., q_t(\bar{E_t})$" for P at iteration 1, then there is also a derivation "$h^{ad}(\bar{A}) \overset{\alpha}{\leftarrow} magic_h{}^{ad}, q_1(\bar{E_1}), ..., q_t(\bar{E_t})$" for P^m at iteration I because $Pure(New(P_I^m)) = \emptyset$, but not vice versa.

Let $X_i^m = \{|f_p(\alpha, f_c(\{|\nu_0(magic_h{}^{ad}), \nu_0(q_1(\bar{E_1})), ..., \nu_0(q_t(\bar{E_t}))|\}))|\}$, then $|X_i^m| = s^m$ which indicates the number of derivations for P^m at iteration i.

Let $X_i = \{|f_p(\alpha, f_c(\{|\nu_0(q_1(\bar{E_1})), ..., \nu_0(q_t(\bar{E_t}))|\}))|\}$, then $|X_i| = s$ which indicates the number of derivations for P at iteration 1.

Let $X_i^I = \{|f_p(\alpha, f_c(\{|\nu_0(magic_h{}^{ad}), \nu_0(q_1(\bar{E_1})), ..., \nu_0(q_t(\bar{E_t}))|\}))|\}$, then $|X_i^I| = s^I$ which indicates the number of derivations for P^m at iteration $I + i - 1$.

We have $s^m < s < s^I$. Therefore,

$$x_1 = \nu_1(h^{ad}(\bar{A})) = T_p^m(\nu_0)(h^{ad}(\bar{A})) = f_d(\{|X_1^m|\})$$

$$\preceq f_d(\{|X_1|\})$$
$$= \nu_1(h(\bar{A})) = T_p(\nu_0)(h(\bar{A})) = y_1 = f_d(\{|X_1|\})$$
$$\preceq f_d(\{|X_1^I|\})$$
$$= \nu_I(h^{ad}(\bar{A})) = T_p^m(\nu_{I-1})(h^{ad}(\bar{A})) = z_1.$$

Let us examine the intermediate results for case $i = 1$. First, for any head atom $h(\bar{A})$ in P and $h^{ad}(\bar{A})$ in P^m, we have $\nu_1(h^{ad}(\bar{A})) \preceq \nu_1(h(\bar{A})) \preceq \nu_I(h^{ad}(\bar{A}))$. Secondly, if $h^{ad}(\bar{A}) \in Pure(D_1^m)$, then $h(\bar{A}) \in Pure(D_1)$; if $h(\bar{A}) \in Pure(D_1)$, then $h^{ad}(\bar{A}) \in Pure(D_I^m)$ because all ground atoms are determined at iteration I. Thirdly, x_1, y_1 and z_1 represent three certainty values for any specific ground atom. That means for every non-magic ground atom w.r.t Q, the inequality holds.

Case $i = k + 1$:

For any head atom $h(\bar{A}) \in P$ and $h^{ad}(\bar{A}) \in P^m$, if $x_k \preceq y_k \preceq z_k$, then:

$$x_{k+1} = \nu_{k+1}(h^{ad}(\bar{A})) = T_p^m(\nu_k)(h^{ad}(\bar{A})) = f_d(\{|X_{k+1}^m|\})$$
$$\preceq f_d(\{|X_{k+1}|\})$$
$$= \nu_{k+1}(h(\bar{A})) = T_p(\nu_k)(h(\bar{A})) = y_{k+1} = f_d(\{|X|\})$$
$$\preceq f_d(\{|X_{k+1}^I|\})$$
$$= \nu_{I+k}(h^{ad}(\bar{A})) = T_p^m(\nu_{I+k-1})(h^{ad}(\bar{A})) = z_{k+1}$$

Correspondingly, if there is a derivation through "$h^{ad}(\bar{A}) \overset{\alpha}{\leftarrow} magic_h{}^{ad}, q_1(\bar{E_1}), ..., q_t(\bar{E_t}), magic_{p_1}{}^{ad1}, p_1^{ad1}(\bar{A_1}), ..., magic_{p_n}{}^{adn}, p_n^{adn}(\bar{A_n})$" for P^m at iteration $k + 1$, then a derivation is obtained for P through "$h(\bar{A}) \overset{\alpha}{\leftarrow} q_1(\bar{E_1}), ..., q_t(\bar{E_t}), p_1(\bar{A_1}), ..., p_n(\bar{A_n})$" at iteration $k + 1$. We have $s > s^m$ because some rules might not be fired when magic sets are not fully prepared. If a derivation "$h(\bar{A}) \overset{\alpha}{\leftarrow} q_1(\bar{E_1}), ..., q_t(\bar{E_t}), p_1(\bar{A_1}), ..., p_n(\bar{A_n})$" is obtained for P at iteration $k + 1$, then a derivation "$h^{ad}(\bar{A}) \overset{\alpha}{\leftarrow} magic_h{}^{ad}, q_1(\bar{E_1}), ..., q_t(\bar{E_t}$

$), magic_{p_1}{}^{ad1}, p_1^{ad1}(\bar{A}_1), ..., magic_{p_n}{}^{adn}, p_n^{adn}(\bar{A}_n)$" is obtained for P^m at iteration $(I + k)$ because $Pure(New(D^m_{I+k-1})) = \emptyset$.

Based on the discussion above, we then conclude that for every atom A, which is an instance of head atom $h(\bar{X})$ in P and $h^{ad}(\bar{X})$ in P^m, we construct three sequences $\{x_n\}$, $\{y_n\}$, and $\{z_n\}$, where $x_i \leq y_i \leq z_i$. $\{x_n\}$ represents a sequence of certainty values of A through $h^{ad}(\bar{X})$. Then, $\{x_n\}$ converges since the bottom-up fixpoint evaluation of any p-program in PF terminates at some steps and no more than ω [LS96]. Let this fixpoint value be $\nu^m_\omega(A)$, then $\{z_n\}$ also converges to $\nu^m_\omega(A)$. Correspondingly, let the certainty associated with $h(\bar{X})$ in the limit be $\nu_\omega(A)$. Based on Theorem 3.1.2,

$$\lim_{i\to\infty} x_i = \lim_{i\to\infty} z_i = \nu^m_\omega(A) = \lim_{i\to\infty} y_i = \nu_\omega(A).$$

which was to be proved. □

3.2 *Staightforward* GSMS

In Datalog, GMS succeeds in restricting facts to be potentially related to the given program, but it suffers from the drawback that many facts are evaluated repeatedly, especially in the non-linear rules. For instance, in Example 3.2.1, the joins "$magic\_sgc\_bf(X), par(X, X_1)$" for the magic rule will be re-done in the rewritten adorned rule. In Datalog, generalized supplementary magic sets rewriting (GSMS) was introduced to solve the problem. In this section we extend the GSMS technique to PF, called straightforward GSMS, which also includes three steps:

- Adorned rules generation
- Supplementary magic rules generation
- Rewriting the adorned rules

The step of "generating adorned rules" for GSMS is exactly the same as the one of the proposed for GMS. We will represent the last two steps in the following sections.

Example 3.2.1. GMS of same generation cousin p-program

Original Program:

$sgc(X,X) \xleftarrow{1} person(X); \langle ind,*,* \rangle.$

$sgc(X,Y) \xleftarrow{0.7} par(X,X_1), sgc(X_1,Y_1), par(Y,Y_1); \langle ind,*,* \rangle.$

$?p(a,Y).$

The *Straightforward* GMS Rewritten Program:

$sgc\_bf(X,X) \xleftarrow{1} magic\_sgc\_bf(X), person(X); \langle ind,*,* \rangle.$

$magic\_sgc\_bf(X_1) \xleftarrow{1} magic\_sgc\_bf(X), par(X,X_1); \langle max, max, max \rangle.$

$sgc\_bf(X,Y) \xleftarrow{0.7} magic\_sgc\_bf(X), par(X,X_1), sgc(X_1,Y_1), par(Y,Y_1);$

$\langle ind,*,* \rangle.$

$magic\_sgc\_bf(a) \xleftarrow{1}; \langle \_,\_,\_ \rangle.$

3.2.1 Supplementary Magic Rules Generation for GSMS

Instead of generating magic rules in GMS, the GSMS needs to generate supplementary magic rules. During the adorned rules generation, each adorned IDB predicate is associated with a list of bound variables S. Then we can define supplementary magic sets predicate for an adorned predicate p^{ad} by a defined method called $getSupMagicPredicate(p^{ad}, p^{ad}.S)$. The method returns a generalized supplementary magic predicate sup_p^{ad}, where all variables existed in p^{ad} and S.

Figure 3.3 shows the steps to generate supplementary magic rules for an adorned version of a rule r, referred to as r^{ad}. For each IDB subgoal ${p_i}^{ad_i}$ in r^{ad}, we generate a generalized supplementary rule ${r_{sup}}^{ad}$ in which "$\langle max, min, f_d \rangle$" is used as the combination functions. The major difference between a supplementary magic rule and a magic rule is that for each

```
1: procedure $GenerateGSMSR(r^{ad})$
2: input:   An adorned rule $r^{ad} : h^{ad_0}(\bar{X}) \overset{\alpha}{\leftarrow} p_1{}^{ad_1}(\bar{X_1}), ..., p_n{}^{ad_n}(\bar{X_n}); \langle f_d, f_p, f_c \rangle$
3: output: A collection of GSMS rules for $r^{ad}$
4: begin
5:     Initialize a collection C;//used to save GSMS rules
6:     k=0;//a pointer indicating the position of the subgoal visited at last
7:     forall subgoal $q_i = p_i{}^{ad_i}(\bar{X_i})$ of $r^{ad}$
8:         if ($\pi(q_i)$ is an IDB predicate) {
9:             Create a supplementary rule $r_{sup}{}^{ad}$;
10:            $r_{sup}{}^{ad}.\langle f_d, f_p, f_c \rangle = \langle max, min, r^{ad}.f_d \rangle$; $r_{sup}{}^{ad}.\alpha = \top$;
11:            $r_{sup}{}^{ad}.head = getSupMagicPredicate(q_i, q_i.S)$;
12:            if (k=0)  $sup\_h = getMagicPredicate(r^{ad}.head)$; k++;
13:            else $sup\_h = getSupMagicPredicate(q_k, q_k.S)$;
14:            Add $sup\_h$ to the body of $r_{sup}{}^{ad}$;
15:            for $j = k$ to $i$ do
16:                $q_j = p_j{}^{ad_j}(\bar{X_j})$;// $j$ is the predicate position in the body
17:                Add $sup\_q_j = getSupMagicPredicate(q_j)$ to the body of $r_{sup}{}^{ad}$;
18:            end for
19:            Create a transformation rule $r_m{}^{ad}$;
20:            $r_m{}^{ad}.\langle f_d, f_p, f_c \rangle = \langle max, max, max \rangle$; $r_m{}^{ad}.\alpha = \top$;
21:            $r_m{}^{ad}.head = sup\_h$;
22:            Add $sup\_h$ to the body of $r_m{}^{ad}$, and $r_{sup}{}^{ad}$, $r_m{}^{ad}$ to C; }
23:        end if
24:    end forall
25:    return C;
26: end
```

Figure 3.3: Generalized supplementary magic rules generation algorithm

predicate subgoal $p_j{}^{ad_j}$ in r^{ad}, where $j \in [1, i]$, GMS adds $getMagicPredicate(p_j{}^{ad_j})$ to the body of the magic rule, but GSMS adds the supplementary magic predicate of $p_k{}^{ad_k}$ and the subgoals in the position $j \in [k, i]$, where k is the position of the previous IDB predicate considered. A magic rule in GMS may be divided into several supplementary rules in GSMS. This difference results in saving the intermediate joins into a relation and hence avoids unnecessary or repeated joins from the rewriting, especially for non-linear rules. Note that for GSMS, we need to generate a magic rule for each adorned IDB subgoal in r^{ad}. The head of the magic rule is the magic predicate related to the head of r^{ad}. The body of the magic rule has one subgoal which is the supplementary magic predicate generated for the considered subgoal in the body of the adorned rule. We used to call this magic rule a transformation magic rule.

3.2.2 Rewriting the Adorned Rules for GSMS

Example 3.2.2. An Example: *Straightforward* GSMS rewritten program of P

The *Straightforward* GSMS Rewritten Program:

$r_1 : p\_bf(X,Y) \overset{0.5}{\longleftarrow} magic\_p\_bf(X), a(X,Y); \langle ind, pro, pro\rangle.$
$r_2 : sup2\_1\_0(X) \overset{1}{\longleftarrow} magic\_p\_bf(X); \langle max, min, pro\rangle.$
$r_3 : magic\_p\_fb(X) \overset{1}{\longleftarrow} sup2\_1\_0(X); \langle max, max, max\rangle.$
$r_4 : sup2\_1\_1(X,Y) \overset{1}{\longleftarrow} sup2\_1\_0(X), p\_fb(Y,X); \langle max, min, pro\rangle.$
$r_5 : magic\_p\_bf(Y) \overset{1}{\longleftarrow} sup2\_1\_1(X,Y); \langle max, max, max\rangle.$
$r_6 : p\_bf(X,Y) \overset{0.5}{\longleftarrow} sup2\_1\_1(X,Y), p\_bf(Y,Z); \langle ind, pro, pro\rangle.$
$r_7 : p\_fb(X,Y) \overset{0.5}{\longleftarrow} magic\_p\_fb(Y), a(X,Y); \langle ind, pro, pro\rangle.$
$r_8 : sup4\_1\_0(Y) \overset{1}{\longleftarrow} magic\_p\_fb(Y); \langle max, min, pro\rangle.$
$r_9 : magic\_p\_bf(Y) \overset{1}{\longleftarrow} sup4\_1\_0(Y); \langle max, max, max\rangle.$
$r_{10} : sup4\_1\_1(Y,Z) \overset{1}{\longleftarrow} sup4\_1\_0(Y), p\_bf(Y,Z); \langle max, min, pro\rangle.$
$r_{11} : magic\_p\_bf(Y) \overset{1}{\longleftarrow} sup4\_1\_1(Y); \langle max, max, max\rangle.$
$r_{12} : p\_fb(X,Y) \overset{0.5}{\longleftarrow} sup4\_1\_1(Y,Z), p\_bf(Y,X); \langle ind, pro, pro\rangle.$
$a(1,2) \overset{0.5}{\longleftarrow}; \langle \_, \_, \_\rangle.$
$a(2,1) \overset{0.5}{\longleftarrow}; \langle \_, \_, \_\rangle.$
$a(1,1) \overset{0.5}{\longleftarrow}; \langle \_, \_, \_\rangle.$
$magic\_p\_bf(1) \overset{1}{\longleftarrow}; \langle \_, \_, \_\rangle.$

The rest of operations to complete a *straightforward* GSMS process are rewriting the original rules, and seeding the magic facts. For each adorned version rule r^{ad}, assume k is the position of the last visited IDB subgoal in the body of r^{ad}. We add the supplementary predicate of ${q_k}^{ad_k}$ to the rule body. subgoals in positions 1 to $(k-1)$ are eliminated from r^{ad}. This yields a collection of rewritten rules for the original rules, referred to as R^m. Like GMS, we create a seed for GSMS. The rewritten program includes the supplementary magic rules, the transformation magic rules, the collection of rewritten adorned rules R^m, the facts from original program, and the seed. Example 3.2.2 shows the result of GSMS transformation of the p-program given in Example 3.1.1.

3.2.3 Correctness of *straightforward* GSMS

Given a p-program P with a set of rules R and a set of facts D, each rule r in R can be rewritten as follows:

$$\text{r: } h(\bar{X_0}) \xleftarrow{\alpha} q_1(\bar{Y_1}), \cdots, q_t(\bar{Y_t}), p_1(\bar{X_1}), \cdots, p_n(\bar{X_n}); \langle f_d, f_p, f_c \rangle$$

where $q_j' s$ are EDB predicates, and $p_i' s$ are IDB predicates. We may classify these rules into two groups:

- Group 1: rules with no IDB predicates ($n = 0$)
- Group 2: rules with at least one IDB predicate ($n \geq 1$)

The following theorem shows that the naive fixpoint evaluation of a p-program P and its *straightforward* GSMS rewritten program P^m produce the same result w.r.t a given Q.

Theorem 3.2.1. *(Correctness of* **Straightforward** *GSMS): Given a p-program P with a query Q and a set of facts D. The* Straightforward *GSMS rewritten program of P is P^m with facts collection $D^m = D \cup M$, where M is the initialized facts for the magic predicates. A Naive fixpoint computations of P and P^m produce the same result w.r.t. Q.*

Proof. Recall that after a GSMS transformation, an adorned rule is replaced by a set of supplementary magic rules and the rewritten adorned rules:

- $sup_1{}^{ad1} \xleftarrow{\top} magic_h{}^{ad}, q_1(\bar{Y_1}); \langle max, min, f_c \rangle.$
- $sup_2{}^{ad2} \xleftarrow{\top} sup_1{}^{ad1}, q_2(\bar{Y_2}), p_1{}^{ad1}(\bar{X_1}); \langle max, min, f_c \rangle.$
- ... ...

- $h^{ad}(\bar{X_0}) \xleftarrow{\alpha} sup_n{}^{adn}, q_{n+1}(\bar{Y_{n+1}}), ..., q_t(\bar{Y_t}); \langle f_d, f_p, f_c \rangle$, called base rule.

For every head atom $h(\bar{A})$ in P and $h^{ad}(\bar{A})$ in P^m, we construct three sequences $\{x_n\}$, $\{y_n\}$ and $\{z_n\}$, where $x_0 = y_0 = z_0 = \bot$, and

$$x_i = \nu_i(h^{ad}(\bar{A}))$$

$$y_i = \nu_i(h(\bar{A}))$$

$$z_i = \nu_{I+(i-1)}(h^{ad}(\bar{A}))$$

I indicates the iteration that $Pure(New(D_I^m)) = \emptyset$, but $Pure(New(D_{I-1}^m)) \neq \emptyset$.

We are to show $x_i \preceq y_i \preceq z_i$. Since only rules in group 1, showed as above, can be fired at iteration 1, the proof of the case 1 for GSMS is the same as the one for GMS. Here we just highlight the differences at case $k+1$

Case $i = k + 1$:

Suppose in case $i = k$, for any head atom $h(\bar{A})$ in P and $h^{ad}(\bar{A})$ in P^m, we have $x_k \preceq y_k \preceq z_k$ i.e. $\nu_k(h^{ad}(\bar{A})) \preceq \nu_k(h(\bar{A})) \preceq \nu_{I+k-1}(h^{ad}(\bar{A}))$. if $h^{ad}(\bar{A}) \in Pure(D_k^m)$, then $h(\bar{A}) \in Pure(D_k)$ but not vice versa; if $h(\bar{A}) \in Pure(D_k)$, then $h^{ad}(\bar{A}) \in Pure(D_{I+(k-1)}^m)$. Note $I > 2$, or there are no supplementary magic rules.

(I) consider the iteration $k+1$ for either P or P^m, suppose s rules are fired for P, and s^m rules are fired for P^m. If there is a derivation "$h^{ad}(\bar{A}) \xleftarrow{\alpha} sup_h{}^{adn}, q_{n+1}(\bar{E_{n+1}}), ..., q_t(\bar{E_t})$" for P^m at iteration $k+1$, then there is also a derivation "$h(\bar{A}) \xleftarrow{\alpha} q_1(\bar{E_1}), ..., q_t(\bar{E_t}), p_1(\bar{A_1}$ $), ..., p_n(\bar{A_n})$" for P at iteration $k+1$. We have $s \geq s^m$ because if $h^{ad}(\bar{A}) \in Pure(D_k^m)$, then $h(\bar{A}) \in Pure(D_k)$, and some rules are not fired when (supplementary) magic sets are not fully prepared. Note that

$$\nu_k(sup_h{}^{adj}) = T_p^m(\nu_{k-1})(sup_h{}^{adj})$$

$$= f_d(\{|f_p(\alpha, f_c(\{|\nu_{k-1}(sup_h{}^{adj}), \nu_{k-1}(p_j^{adj}(\bar{A_j}))|\}))|\})$$

$$\preceq f_d(\{|f_p(\alpha, f_c(\{|\nu_k(sup_h{}^{adj}), \nu_k(p_j^{adj}(\bar{A_j}))|\}))|\})$$

$$= T_p^m(\nu_k)(sup_h{}^{adj}) = \nu_{k+1}(sup_h{}^{adj})$$

Therefore,

$$x_{k+1} = \nu_{k+1}(h^{ad}(\bar{A})) = T_p^m(\nu_k)(h^{ad}(\bar{A}))$$

$$= f_d(\{|f_p(\alpha, f_c(\{|\nu_k(sup_h{}^{adn}), \nu_k(q_{n+1}(\bar{E}_{n+1})), ..., \nu_k(q_t(\bar{E}_t))|\})), s^m\ derivations|\})$$

$$\preceq f_d(\{|f_p(\alpha, f_c(\{|\nu_k(sup_h{}^{ad(n-1)}), \nu_k(q_n(\bar{E}_n)), ..., \nu_k(q_t(\bar{E}_t)), \nu_k(p_n^{adn}(\bar{A}_n))|\})),$$
$$s^m\ derivations|\})$$

... ...

$$\preceq f_d(\{|f_p(\alpha, f_c(\{|\nu_k(q_1(\bar{E}_1)), ..., \nu_k(q_t(\bar{E}_t)), \nu_k(p_1^{ad1}(\bar{A}_1)), ..., \nu_k(p_n^{adn}(\bar{A}_n))|\})),$$
$$s^m\ derivations|\})$$

$$\preceq f_d(\{|f_p(\alpha, f_c(\{|\nu_k(q_1(\bar{E}_1)), ..., \nu_k(q_t(\bar{E}_t)), \nu_k(p_1^{ad1}(\bar{A}_1)), ..., \nu_k(p_n^{adn}(\bar{A}_n))|\})),$$
$$s\ derivations|\})$$

$$= \nu_{k+1}(h(\bar{A})) = T_p(\nu_k)(h(\bar{A})) = y_{k+1}$$

(II) On the other hand, consider the iteration $k+1$ for P and the $(I+k)_{th}$ iteration for P^m.

$$y_{k+1} = \nu_{k+1}(h(\bar{A})) = T_p(\nu_k)(h(\bar{A}))$$

$$= f_d(\{|f_p(\alpha, f_c(\{|\nu_k(q_1(\bar{E}_1)), ..., \nu_k(q_t(\bar{E}_t)), \nu_k(p_1(\bar{A}_1)), ..., \nu_k(p_n(\bar{A}_n))|\})),$$
$$s\ derivations|\})$$

$$\preceq f_d(\{|f_p(\alpha, f_c(\{|\nu_{I+k-2}(q_1(\bar{E}_1)), ..., \nu_{I+k-2}(q_n(\bar{E}_n)), \nu_{I+k-1}(q_{n+1}(\bar{E}_{n+1})),$$
$$..., \nu_{I+k-1}(q_t(\bar{E}_t)), \nu_{I+k-2}(p_1(\bar{A}_1)), ..., \nu_{I+k-2}(p_n(\bar{A}_n))|\})),$$
$$s\ derivations|\})$$

$$\preceq f_d(\{|f_p(\alpha, f_c(\{|\nu_{I+k-1}(sup_h{}^{adn}), \nu_{I+k-1}(q_{n+1}(\bar{E}_{n+1})), ..., \nu_{I+k-1}(q_t(\bar{E}_t))|\})),$$
$$s\ derivations|\})$$

$$\preceq f_d(\{|f_p(\alpha, f_c(\{|\nu_{I+k-1}(sup_h{}^{adn}), \nu_{I+k-1}(q_{n+1}(\bar{E}_{n+1})), ..., \nu_{I+k-1}(q_t(\bar{E}_t))|\})),$$
$$s^I\ derivations|\})$$

$$= \nu_{I+1}(h^{ad}(\bar{A})) = T_p^m(\nu_{I+k})(h^{ad}(\bar{A})) = z_{k+1}.$$

□

3.3 Magic Sets Do Not Play Magic Role!

An important difference between evaluating programs in PF and in Datalog is that while the fixpoint evaluation of Datalog program terminates in polynomial time (in the number of database constants), an evaluation for a p-program may terminate only at ω. This may happen when a type 2 combination function (see Section 2.1.2) is associated with a recursive predicate. With the T_p operator that is continuous, we may allow a fixpoint evaluation proceed until certainties derived are "close enough" to the fixpoint and within some desired precision. On the other hand, a real computation is bound to certain precision decided by memory word size. The question is how to balance these two issues: infinite number of iterations and finite precision. Given a p-program P and a query Q, consider a precision parameter c, w.r.t fixpoint evaluation. If $|\nu_{i+1}(A) - \nu_i(A)| \leq c$, $\forall A \in D_i$, and $A \in D_{i+1}$, then the evaluation terminates. Theorem 3.1.3 and Theorem 3.2.1 have shown that a *straightforward* GMS/GSMS rewritten p-program may converge to its least fixpoint in the limit, but under "precision control", certainties obtained by evaluating the original program might be different in a finite computation. The following example illustrates the problem.

Example 3.3.1. Consider Example 3.1.1, $T = [0, 1]$, and precision in controlled less than 0.001. The following table shows the database for IDB predicates at every iteration. We use **** to indicate when the aggregated terms is considered as new.

Iteration i	Original Program P	GMS Rewritten Program p^m
1	p(2,1):0.25 **** p(1,2):0.25 **** p(1,1):0.25 ****	p_bf(1,1):0.25 **** magic_p_fb(1):1.0 **** p_bf(1,2):0.25 ****
2	p(2,1):0.29614258 **** p(1,2):0.2734375 **** p(1,1):0.29614258 ****	p_bf(1,1):0.25 p_fb(2,1):0.29614258 **** magic_p_fb(1):1.0
3	p(2,1):0.307269 ****	p_bf(1,1):0.304499 ****

	p(1,2):0.28288764 ****	p_fb(2,1):0.29614258
	p(1,1):0.31192228 ****	magic_p_fb(1):1.0
		p_fb(1,1):0.29614258 ****
		magic_p_bf(2):1.0 ****
		magic_p_bf(1):1.0 ****
		p_bf(1,2):0.25 ****
4	p(2,1):0.30882657 ****	p_fb(1,1):0.31199324 ****
	p(1,2):0.28540534 ****	magic_p_bf(2):1.0
	p(1,1):0.31691408 ****	magic_p_bf(1):1.0
		magic_p_fb(2):1.0 ****
		p_fb(2,1):0.30109218 ****
		magic_p_fb(1):1.0
		p_bf(1,2):0.25
		p_bf(1,1):0.31032723 ****
		p_bf(2,1):0.25 ****
5	p(2,1):0.30882657	p_fb(1,2):0.2734375 ****
	p(1,2):0.28540534	p_fb(1,1):0.31199324
	p(1,1):0.31691408	magic_p_bf(2):1.0
		magic_p_bf(1):1.0
		magic_p_fb(2):1.0
		p_fb(2,1):0.30109218
		magic_p_fb(1):1.0
		p_bf(1,2):0.27776337 ****
		p_bf(1,1):0.31348562 ****
		p_bf(2,1):0.25
6		p_fb(1,2):0.2734375
		p_fb(1,1):0.31501645 ****
		magic_p_bf(2):1.0
		magic_p_bf(1):1.0
		magic_p_fb(2):1.0
		p_fb(2,1):0.30942565 ****
		magic_p_fb(1):1.0
		p_bf(1,2):0.27776337 ****
		p_bf(1,1):0.31657955 ****
		p_bf(2,1):0.30851895 ****
7		p_fb(1,2):0.28569397 ****
		p_fb(1,1):0.31890637 ****
		magic_p_bf(2):1.0
		magic_p_bf(1):1.0
		magic_p_fb(2):1.0
		p_fb(2,1):0.31063545 ****
		magic_p_fb(1):1.0
		p_bf(1,2):0.28579888 ****
		p_bf(1,1):0.3179317 ****
		p_bf(2,1):0.30971062 ****
8		p_fb(1,2):0.28569397
		p_fb(1,1):0.31890637

		magic_p_bf(2):1.0 magic_p_bf(1):1.0 magic_p_fb(2):1.0 p_fb(2,1):0.31063545 magic_p_fb(1):1.0 p_bf(1,2):0.28579888 p_bf(1,1):0.3179317 p_bf(2,1):0.30971062

In the example 3.3.1, the answer set of P regarding the query p(1,Y) is {p(1,2):0.28540534, p(1,1):0.31691408}; whereas, the answer set of P^m w.r.t. Q is $\{p\_bf(1,2):0.28579888, p\_bf(1,1):0.3179317\}$.

Given a p-program P and its MS rewritten program P^m. The evaluation order of P^m is changed when the magic atoms are not fully prepared. Atoms evaluated at the same iteration in P might be separately evaluated at different iterations. For P^m, let $T_p(T_p(X) \cup Y)$ be the evaluation results for a specific atom A, where T_p is the immediate consequence operator, X related to derivations which magic atoms are prepared, and Y are those potential derivations which magic atoms are to be prepared. Then for P, $T_p(X \cup Y)$ represents all derivations may be fired at corresponding iteration. When type 2 f_d is applied, $T_p(T_p(X) \cup Y) \neq T_p(X \cup Y)$ which is the major barrier in extending GMS to PF. Although we may prove that the fixpoint values converge finally in an infinite procedure, we realize that bias might occur in a finite steps evaluation. Providing the error estimation analysis seems to be an interesting and complex topic, we do not discuss this topic in this thesis; instead, we add an additional stage "magic tuples generation" for the proposed techniques to avoid the bias in a finite numeric computation.

3.4 Magic Tuples Generation for GMS

In general, for any two adorned facts $p^{ad1}(A)$ and $p^{ad2}(A)$ of a predicate $p(X)$ in a magic sets rewritten program, semantically they should have the same certainty because they represent the same fact in the original program. Let us go back to Example 3.3.1, $p(1,1)$, $p(1,2)$, and $p(2,1)$ are computed at the first iteration in P, but only $p\_bf(1,1)$ and $p\_bf(1,2)$ are computed at the first iteration in P^m because not all necessary magic atoms are prepared at the first iteration, such as $magic\_p\_bf(2)$. For P^m, iterations 2,3,4 are used to prepare $magic\_p\_bf(2)$. However, $p\_bf(1,1)$, and $p\_bf(1,2)$ are improved during these iterations. This results in $p\_bf(1,1)$, and $p\_bf(1,2)$ having the different certainties from $p\_fb(1,1)$, and $p\_fb(1,2)$, and these differences would not converge in a finite steps evaluation. When the type 2 disjunction function (f_d) is adapted in a p-program, the evaluation order of the MS rewritten program will be changed when the magic atoms are not fully prepared at every iteration.

In this section, we introduce an stage after *straightforward* GMS such that the new rewritten program, called *alternative* GMS rewritten program, may generate the same results set w.r.t. the given query at each iteration. Based on the assumption that "the disjunction functions should combine the certainties of atoms derived at the same iteration, and should not combine newly derived certainties with prior certainties of the same atom from the same rule" [SZ04], we pre-compute all magic atoms before the evaluation.

Magic tuples generation is conducted as follows. Given a p-program P, first, we make a copy of its *straightforward* rewritten program p^m, and standardize it to be p^s ,i.e., for any rule r in p^s, we apply $\langle max, min, min\rangle$ as combination functions and $\top$ as the certainty. Second, we evaluate p^s, and found magic atoms are added to p^m. Thirdly, we construct a new program $(p^m)'$, *alternative* GMS rewritten program, in which all magic rules are eliminated. Example 3.4.1 shows the intermediate results of a magic tuples generation by considering the program in Example 3.1.1.

Example 3.4.1. Intermediate Results of magic tuples generation for GMS

Standardized GMS program p^s:

$p\_bf(X,Y) \xleftarrow{1} magic\_p\_bf(X), a(X,Y); \langle max, min, min \rangle.$
$magic\_p\_fb(X) \xleftarrow{1} magic\_p\_bf(X); \langle max, min, min \rangle.$
$magic\_p\_bf(Y) \xleftarrow{1} magic\_p\_bf(X), p\_fb(Y,X); \langle max, min, min \rangle.$
$p\_bf(X,Y) \xleftarrow{1} magic\_p\_bf(X), p\_fb(Y,X), p\_bf(Y,Z); \langle max, min, min \rangle.$
$p\_fb(X,Y) \xleftarrow{1} magic\_p\_fb(Y), a(X,Y); \langle max, min, min \rangle.$
$magic\_p\_bf(Y) \xleftarrow{1} magic\_p\_fb(Y); \langle max, min, min \rangle.$
$magic\_p\_bf(Y) \xleftarrow{1} magic\_p\_fb(Y), p\_bf(Y,Z); \langle max, min, min \rangle.$
$p\_fb(X,Y) \xleftarrow{1} magic\_p\_fb(Y), p\_bf(Y,Z), p\_bf(Y,X); \langle max, min, min \rangle.$
$a(1,2) \xleftarrow{1}; \langle \_, \_, \_ \rangle.$
$a(2,1) \xleftarrow{1}; \langle \_, \_, \_ \rangle.$
$a(1,1) \xleftarrow{1}; \langle \_, \_, \_ \rangle.$
$magic\_p\_bf(1) \xleftarrow{1}; \langle \_, \_, \_ \rangle.$

Evaluation Results:

$p(1,2) : 1.$	$p(1,1) : 1.$	$p(2,1) : 1.$
$a(1,2) : 1.$	$a(1,1) : 1.$	$a(2,1) : 1.$
$p\_bf(1,2) : 1.$	$p\_bf(1,1) : 1.$	$p\_bf(2,1) : 1.$
$p\_fb(1,2) : 1.$	$p\_fb(1,1) : 1.$	$p\_fb(2,1) : 1.$
$magic\_p\_bf(1) : 1.$	$magic\_p\_bf(2) : 1.$	
$magic\_p\_fb(1) : 1.$	$magic\_p\_fb(2) : 1.$	

Alternative GMS Program $(p^m)'$:

$p\_bf(X,Y) \xleftarrow{0.5} magic\_p\_bf(X), a(X,Y); \langle ind, pro, pro \rangle.$
$p\_bf(X,Y) \xleftarrow{0.5} magic\_p\_bf(X), p\_fb(Y,X), p\_bf(Y,Z); \langle ind, pro, pro \rangle.$
$p\_fb(X,Y) \xleftarrow{0.5} magic\_p\_fb(Y), a(X,Y); \langle ind, pro, pro \rangle.$
$p\_fb(X,Y) \xleftarrow{0.5} magic\_p\_fb(Y), p\_bf(Y,Z), p\_bf(Y,X); \langle ind, pro, pro \rangle.$
$a(1,2) \xleftarrow{0.5}; \langle \_, \_, \_ \rangle.$
$a(2,1) \xleftarrow{0.5}; \langle \_, \_, \_ \rangle.$
$a(1,1) \xleftarrow{0.5}; \langle \_, \_, \_ \rangle.$
$magic\_p\_bf(1) \xleftarrow{1}; \langle \_, \_, \_ \rangle.$
$magic\_p\_bf(2) \xleftarrow{1}; \langle \_, \_, \_ \rangle.$
$magic\_p\_fb(1) \xleftarrow{1}; \langle \_, \_, \_ \rangle.$
$magic\_p\_fb(2) \xleftarrow{1}; \langle \_, \_, \_ \rangle.$

We then introduce Theorem 3.4.1 to show that the *alternative* GMS rewritten program may generate the same intermediate results at every iteration, and thus the same fixpoint as the original program w.r.t. the query.

Theorem 3.4.1. Alternative *GMS Correctness:* *Given a query-related p-program P with a query Q and facts collection D, P's alternative GMS rewritten program is P^m with facts collection $D^m = D \cup M$, where M is the set of all pre-computed facts for the magic predicates. A naive fixpoint computation of P and P^m produces the same result regarding Q.*

Proof. **Basis:** On one hand, for any $h(\bar{A}) \in D_1$ in P, assume s rules are fired. They are of the form:

$$h(\bar{X_0}) \xleftarrow{\alpha} q_1(\bar{Y_1}), \cdots, q_t(\bar{Y_t}), p_1(\bar{X_1}), \cdots, p_n(\bar{X_n}); \langle f_d, f_p, f_c \rangle$$

Then

$\nu_1(h(\bar{A})) = T_p(\nu_0)(h(\bar{A})) = f_d(\{|f_p(\alpha, f_c(\{|\nu_0(q_1(\bar{E_1})), ..., \nu_0(q_t(\bar{E_t}))|\})), 1 \leq j \leq s|\}).$

Note that only those rules without IDB predicates in the body are fired in the first iteration because no IDB facts existed in the initialized database.

On the other hand, for any adorned version $h^{ad}(\bar{A})$ in P^m, the same number s of rules are fired since all the magic facts are pre-computed. They are of the form:

$$h^{ad}(\bar{X_0}) \xleftarrow{\alpha} magic_h{}^{ad}, q_1(\bar{Y_1}), \cdots, q_t(\bar{Y_t}), magic_{p_1}{}^{ad_1}, p_1^{adl}(\bar{X_1}), \cdots,$$
$$magic_{p_n}{}^{adn}, p_n^{ad_n}(\bar{X_n}); \langle f_d, f_p, f_c \rangle$$

Since $\nu_0(magic_h{}^{ad}) = \top$,

$\nu_1(h^{ad}(\bar{A})) = T_p(\nu_0)(h^{ad}(\bar{A}))$

$= f_d(\{|f_p(\alpha, f_c(\{|\nu_0(magic_h{}^{ad}), \nu_0(q_1(\bar{E_1})), ..., \nu_0(q_t(\bar{E_t}))|\})), 1 \leq j \leq s|\})$

$= f_d(\{|f_p(\alpha, f_c(\{|\nu_0(q_1(\bar{E_1})), ..., \nu_0(q_t(\bar{E_t}))|\})), 1 \leq j \leq s|\})$

$= T_p(\nu_0)(h(\bar{A})) = \nu_1(h(\bar{A}))$

Induction: Suppose for any $h(\bar{A}) \in D_k$ in P, and $h^{ad}(\bar{A}) \in D_k^m$ in P^m, if $\nu_k(h(\bar{A})) = T_p(\nu_{k-1})(h(\bar{A})) = T_p(\nu_{k-1})(h^{ad}(\bar{A})) = \nu_k(h^{ad}(\bar{A}))$, and $\nu_k({magic_h}^{ad}) = \top$, then

$$\begin{aligned}
\nu_{k+1}(h(\bar{A})) &= T_p(\nu_k)(h(\bar{A})) \\
&= f_d(\{|f_p(\alpha, f_c(\{|\nu_k(q_1(\bar{E_1})), ..., \nu_k(q_t(\bar{E_t})), \nu_k(p_1(\bar{A_1})), ..., \nu_k(p_n(\bar{A_n}))|\})), \\
&\qquad 1 \leq j \leq s|\}), \\
&= f_d(\{|f_p(\alpha, f_c(\{|\nu_k({magic_h}^{ad}), \nu_k(q_1(\bar{E_1})), ..., \nu_k(q_t(\bar{E_t})), \nu_k({magic_{p_1}}^{ad1}), \\
&\qquad \nu_k(p_1^{ad1}(\bar{A_1})), ..., \nu_k({magic_{p_1}}^{adn}), \nu_k(p_n^{adn}(\bar{A_n}))|\})), 1 \leq j \leq s|\}) \\
&= T_p(\nu_{k+1})(h^{ad}(\bar{A})) = \nu_{k+1}(h^{ad}(\bar{A}))
\end{aligned}$$

Based on the discussion above, we may conclude that for every atom A in P and its adorned form A^m in P^m, the certainty of A at every iteration is exactly same as A^m. Therefore, their certainties in the limit are also the same. □

3.5 Magic Tuples Generation for GSMS

In this section, we discuss how to evaluate GSMS rewritten program such that the rewritten program may generate the same set of results as the original program at each iteration of evaluation. The idea is that we pre-compute all magic facts and add them into the *straightforward* GSMS rewritten program.

Given a p-program P, all rules generated for the same adorned rule r^{ad} are classified into the same group. To do that, we can simply attach a symbol to each rule which indicates rules generated from the same r^{ad} have the same symbel. We make a copy of the rewritten program P^m, and standardize it to be P^s ,i.e., for any rule r in P^s, we apply $\langle max, min, min\rangle$ as combination functions and $\top$ as the certainty. Then we evaluate P^s to obtain all magic tuples and construct a new rewritten program $(p^m)'$, *alternative* GSMS rewritten program, in which all magic rules are eliminated, but all magic tuples pre-computed are added.

Example 3.5.1 shows the intermediate results of a magic tuples generation for GSMS by considering the program in Example 3.1.1.

Example 3.5.1. Intermediate Results of magic tuples generation for GSMS

Standardized GMS program p^s:

$g1 : p\_bf(X,Y) \xleftarrow{1} magic\_p\_bf(X), a(X,Y); \langle max, min, min\rangle.$
$g2 : sup2\_1\_0(X) \xleftarrow{1} magic\_p\_bf(X); \langle max, min, min\rangle.$
$g2 : magic\_p\_fb(X) \xleftarrow{1} sup2\_1\_0(X); \langle max, min, min\rangle.$
$g2 : sup2\_1\_1(X,Y) \xleftarrow{1} sup2\_1\_0(X), p\_fb(Y,X); \langle max, min, min\rangle.$
$g2 : magic\_p\_bf(Y) \xleftarrow{1} sup2\_1\_1(X,Y); \langle max, min, min\rangle.$
$g2 : p\_bf(X,Y) \xleftarrow{1} sup2\_1\_1(X,Y), p\_bf(Y,Z); \langle max, min, min\rangle.$
$g3 : p\_fb(X,Y) \xleftarrow{1} magic\_p\_fb(Y), a(X,Y); \langle max, min, min\rangle.$
$g4 : sup4\_1\_0(Y) \xleftarrow{1} magic\_p\_fb(Y); \langle max, min, min\rangle.$
$g4 : magic\_p\_bf(Y) \xleftarrow{1} sup4\_1\_0(Y); \langle max, min, min\rangle.$
$g4 : sup4\_1\_1(Y,Z) \xleftarrow{1} sup4\_1\_0(Y), p\_bf(Y,Z); \langle max, min, min\rangle.$
$g4 : magic\_p\_bf(Y) \xleftarrow{1} sup4\_1\_1(Y); \langle max, min, min\rangle.$
$g4 : p\_fb(X,Y) \xleftarrow{1} sup4\_1\_1(Y,Z), p\_bf(Y,X); \langle max, min, min\rangle.$
$a(1,2) \xleftarrow{1}; \langle \_, \_, \_\rangle.$
$a(2,1) \xleftarrow{1}; \langle \_, \_, \_\rangle.$
$a(1,1) \xleftarrow{1}; \langle \_, \_, \_\rangle.$
$magic\_p\_bf(1) \xleftarrow{1}; \langle \_, \_, \_\rangle.$

Evaluation Results:

$p(1,2) : 1.$	$p(1,1) : 1.$	$p(2,1) : 1.$
$a(1,2) : 1.$	$a(1,1) : 1.$	$a(2,1) : 1.$
$p\_bf(1,2) : 1.$	$p\_bf(1,1) : 1.$	$p\_bf(2,1) : 1.$
$p\_fb(1,2) : 1.$	$p\_fb(1,1) : 1.$	$p\_fb(2,1) : 1.$
$sup2\_1\_0(1) : 1.$	$sup2\_1\_0(2) : 1.$	
$sup4\_1\_0(1) : 1.$	$sup4\_1\_0(2) : 1.$	
$sup4\_1\_1(1,2) : 1.$	$sup4\_1\_1(1,1) : 1.$	$sup4\_1\_1(2,1) : 1.$
$sup2\_1\_1(1,2) : 1.$	$sup2\_1\_1(1,1) : 1.$	$sup2\_1\_1(2,1) : 1.$
$magic\_p\_bf(1) : 1.$	$magic\_p\_bf(2) : 1.$	
$magic\_p\_fb(1) : 1.$	$magic\_p\_fb(2) : 1.$	

Alternative GSMS Program $(p^{sup})'$:

$g1 : p\_bf(X,Y) \xleftarrow{0.5} magic\_p\_bf(X), a(X,Y); \langle ind, pro, pro\rangle.$
$g2 : sup2\_1\_0(X) \xleftarrow{1} magic\_p\_bf(X); \langle max, min, pro\rangle.$
$g2 : sup2\_1\_1(X,Y) \xleftarrow{1} sup2\_1\_0(X), p\_fb(Y,X); \langle max, min, pro\rangle.$
$g2 : p\_bf(X,Y) \xleftarrow{0.5} sup2\_1\_1(X,Y), p\_bf(Y,Z); \langle ind, pro, pro\rangle.$
$g3 : p\_fb(X,Y) \xleftarrow{0.5} magic\_p\_fb(Y), a(X,Y); \langle ind, pro, pro\rangle.$
$g4 : sup4\_1\_0(Y) \xleftarrow{1} magic\_p\_fb(Y); \langle max, min, pro\rangle.$
$g4 : sup4\_1\_1(Y,Z) \xleftarrow{1} sup4\_1\_0(Y), p\_bf(Y,Z); \langle max, min, pro\rangle.$
$g4 : p\_fb(X,Y) \xleftarrow{0.5} sup4\_1\_1(Y,Z), p\_bf(Y,X); \langle ind, pro, pro\rangle.$
$a(1,2) \xleftarrow{0.5}; \langle \_, \_, \_\rangle.$

```
$a(2,1) \overset{0.5}{\longleftarrow}; \langle \_, \_, \_ \rangle.$
$a(1,1) \overset{0.5}{\longleftarrow}; \langle \_, \_, \_ \rangle.$
$magic\_p\_bf(1) \overset{1}{\longleftarrow}; \langle \_, \_, \_ \rangle.$
$magic\_p\_bf(2) \overset{1}{\longleftarrow}; \langle \_, \_, \_ \rangle.$
$magic\_p\_fb(1) \overset{1}{\longleftarrow}; \langle \_, \_, \_ \rangle.$
$magic\_p\_fb(2) \overset{1}{\longleftarrow}; \langle \_, \_, \_ \rangle.$
```

Algorithm 3.5.1. A modified SNP for the GSMS rewritten p-programs

```
1: Procedure$Semi\_Naive\_Partion\_GSMS(P, D, lfp(T_{P \bigcup D}))$
2:   forall $A \in B_p$;
3:     $C_1(A) := \{|(\alpha : \emptyset)|(A : \alpha) \in D|\}$
4:     $\nu_1(A) := f_d(C_1(A)(\alpha)); where f_d := Disj(\pi(A))$;
5:   end forall
6:   $new_1 := \{A|(A : \alpha) \in D\}; i := 1$;
7:   while $(new_i \neq \varnothing)$
8:     i:=i+1;
9:     forall $A \in B_p$ :
10:        $C_i(A) := C_{i-1}(A)$;
11:        forall $B \in new_{i-1} \bigwedge (\alpha, S_B) \in C_{i-1}(A) \bigwedge B \in S_B$:
12:          $C_i(A) := C_i(A) - \{|\alpha, S_B)|\}$
13:        end forall
14:        forall $(r : A \overset{\alpha_r}{\longleftarrow} B_1, \dots, B_n; \langle f_d, f_p, f_c \rangle) \in P^* \bigwedge \exists B_j \in new_i$,
           where $j \in \{1, \dots, n\}$)
15:          $C_i(A) := C_i(A) \bigcup \{|(\alpha_i^r(A), S_B)|\}$, where $\alpha_i^r(A) :=$
               $f_p(\alpha_r, f_c(\{|\nu_{i-1}(B_1), \dots, \nu_{i-1}(B_n)|\}))$,and $S_B :=$
               $\{B_j | j \in \{1, \dots, n\}\}$
16:          $sb = r.gs//$gs is the group symbol for r
17:          forall$((r_g : A_g \overset{\alpha_r}{\longleftarrow} B_{1g}, \dots, B_{ng}; \langle f_d, f_p, f_c \rangle) \in P^* \bigwedge \exists B_{jg} \in new_i$
             $\bigwedge r_g.gs == sb$, where $j \in \{1, \dots, n\}$)
18:            $C_i(A_g) := C_i(A_g) \bigcup \{|(\alpha_i^r(A_g), S_B)|\}$, where $\alpha_i^r(A_g) :=$
                 $f_p(\alpha_r, f_c(\{|\nu_{i-1}(B_{1g}), \dots, \nu_{i-1}(B_{ng})|\}))$,and $S_B :=$
                 $\{B_{jg} | jg \in \{1, \dots, n\}\}$
19:          end forall;
20:        end forall;
21:        $\nu_i := f_d(C_i(A)(\alpha)), where f_d := Disj(\pi(A))$;
22:     end forall;
23:     $new_i := \{A | A \in B_p, \nu_i(A) \succ \nu_{i-1}(A)\}$;
24:   end while
25:   $lfp(T_{P \bigcup D}) := \nu_i$;
26: end procedure
```

Different from GMS we need modified evaluation algorithms to evaluate $(p^m)'$ because rules with the same group symbol conduct the same task for an adorned rule r^{ad}. If these rules are evaluated in several iterations, found atoms might do the self-improvement while supplementary magic facts are prepared. Updated evaluation algorithm (Figure 3.5.1) guarantee the rules with the same group symbol are fired at the same iteration.

Comparing "modified SNP" and "SNP" (Section 2.2), we will see lines from 16 to 19 are added in Algorithm 3.5.1. These lines are used to force rules generated from the same r^{ad}, i.e. with the same group symbol, to be fired at the same iteration. For N and SN, we can simply insert these lines to the proper places to obtain the modified algorithms, i.e., we can add these lines after line 18 for both algorithms shown in Figure 2.1 and 2.2.

Theorem 3.5.1 establishes the correctness of the *alternative* GSMS. Here, we just highlight the difference from Theorem 3.4.1 since the two proofs are quite similar.

Theorem 3.5.1. Alternative *GSMS Correctness:* *Given a p-program P with a query Q and facts collection D, P's alternative GSMS rewritten program is P^m with facts collection $D^m = D \cup M$, where M is the set of all pre-computed facts for the magic predicates. A Naive fixpoint computation of P and P^m produces the same result regarding Q.*

Proof. **Basis:** We ignore the proof here since it is the same as the basis part in the proof of theorem 3.4.1. In this case, only rules without IDB predicates are fired.

Induction: Suppose for any $h(\bar{A}) \in D_k$ in P, and $h^{ad}(\bar{A}) \in D_k^m$ in P^m, if $\nu_k(h(\bar{A})) = T_p(\nu_{k-1})(h(\bar{A})) = T_p(\nu_{k-1})(h^{ad}(\bar{A})) = \nu_k(h^{ad}(\bar{A}))$, then

$$\begin{aligned}
\nu_{k+1}(h(\bar{A})) &= T_p(\nu_k)(h(\bar{A})) \\
&= f_d(\{|f_p(\alpha, f_c(\{|\nu_k(q_1(\bar{E_1})), ..., \nu_k(q_t(\bar{E_t})), \nu_k(p_1(\bar{A_1})), ..., \nu_k(p_n(\bar{A_n}))|\})), \\
&\qquad 1 \le j \le s|\}), \\
&= f_d(\{|f_p(\alpha, f_c(\{|\nu_k(sup_h{}^1), \nu_k(q_1(\bar{E_1})), ..., \nu_k(q_t(\bar{E_t})), \nu_k(p_2^{ad2}(\bar{A_2})), ..., \\
&\qquad \nu_k(p_n^{adn}(\bar{A_n}))|\})), 1 \le j \le s|\}) \\
&= f_d(\{|f_p(\alpha, f_c(\{|\nu_k(sup_h{}^2), \nu_k(q_1(\bar{E_1})), ..., \nu_k(q_t(\bar{E_t})), \nu_k(p_3^{ad3}(\bar{A_3})), ...,
\end{aligned}$$

$$\nu_k(p_n^{adn}(\bar{A}_n))|\})), 1 \leq j \leq s|\})$$

$$= f_d(\{|f_p(\alpha, f_c(\{|\nu_k(sup_h{}^n), \nu_k(q_1(\bar{E}_1)), ..., \nu_k(q_t(\bar{E}_t)), 1 \leq j \leq s|\})$$

......

$$= T_p(\nu_{k+1})(h^{ad}(\bar{A})) = \nu_{k+1}(h^{ad}(\bar{A})). \qquad \square$$

3.6 Efficiency Discussion

The disadvantage of magic sets technique is that rewriting the program incurs an overhead. The more rules in a logic program, the more time is required to generate the rewriting program. In this section, we study the growth of the running time of the proposed rewriting algorithms introduced. For convenience, given a p-program P with a query Q, we use 'm' to denote the number of rules, 'n' to denote the number of constants existing in known tuples. We use 'k' to denote the largest arity of the predicates in P, and 's' to denote the largest number of subgoals of rules in P. Therefore, in Datalog, the compiling time for both GMS and GSMS can be represented as $O(2^k mks)$, and the evaluation time of the Datalog program is $O(n^k k^s m)$. However, the compiling time of the proposed techniques is $O(n^k k^s m)$ in the worst case.

First, for adorned rule generation, the running time of the algorithm in Figure 3.1 is $O(ks)$. Since there are $2^k m$ rules are considered for the rewriting, the total running time for the adorned rule generation is $O(2^k mks)$.

Secondly, for magic rules generation, the algorithms in Figure 3.2 and Figure 3.3 run in $O(2^k sm)$ to generate all (supplementary) magic rules since each adorned rule generation takes $O(s)$, and the rewritten program includes $2^k m$ adorned rules.

Thirdly, rewriting the adorned rules requires $O(2^k ms)$. In the worst case, for each rule r in P, there are 2^k adorned forms, then we will get $2^k m$ adorned rules. For each adorned rule, each subgoal is determined, and hence the running time for rewriting the adorned rules is $O(2^k ms)$.

Finally, generating magic tuples for the given query takes $O(k)$, but in the worst case when type 2 disjunction function is used, this time is the evaluation time of the "standardized" rewritten program, i.e., $O(n^k k^s m)$, which is the complexity of evaluating Datalog programs. Therefore, the total time complexity of magic sets rewriting is $O(n^k k^s m)$ when type 2 disjunction function is used; otherwise it is $O(2^k mks)$.

Chapter 4

UNLOG : A Prototype System

To study the performance of magic sets techniques, we have developed a prototype deductive database with uncertainty, which implements the proposed rewriting algorithms. This prototype, called UNLOG, was implemented in Java. It is platform-independent. The certainty domain is considered as [0..1]. In this chapter, we describe the details of the proposed system. We first show the architecture of the system and its major components. Then we describe the data structure. Optimization techniques incorporated are described at the end of the chapter.

4.1 Architecture and Components

UNLOG is a single user, memory-based system. Figure 4.1 shows the system architecture. When a p-program, saved in a file, is read into the system by the coordinator, the p-program parser parses the source code and stores the rules and tuples into the storage space. Necessary metadata is generated and collected at the same time, such as the type of the combination functions, the type of query and the size of the tables. Next, an optimizer is invoked to determine useful optimization techniques based on the metadata, such as magic sets rewriting and simplification. Then the evaluation process starts to compute the fix-point. An appropriate evaluation algorithm is selected based on the metadata and system configuration. Once the evaluation terminates, results are returned to the user through the system interface or saved into a target file if desired. The components descriptions are as

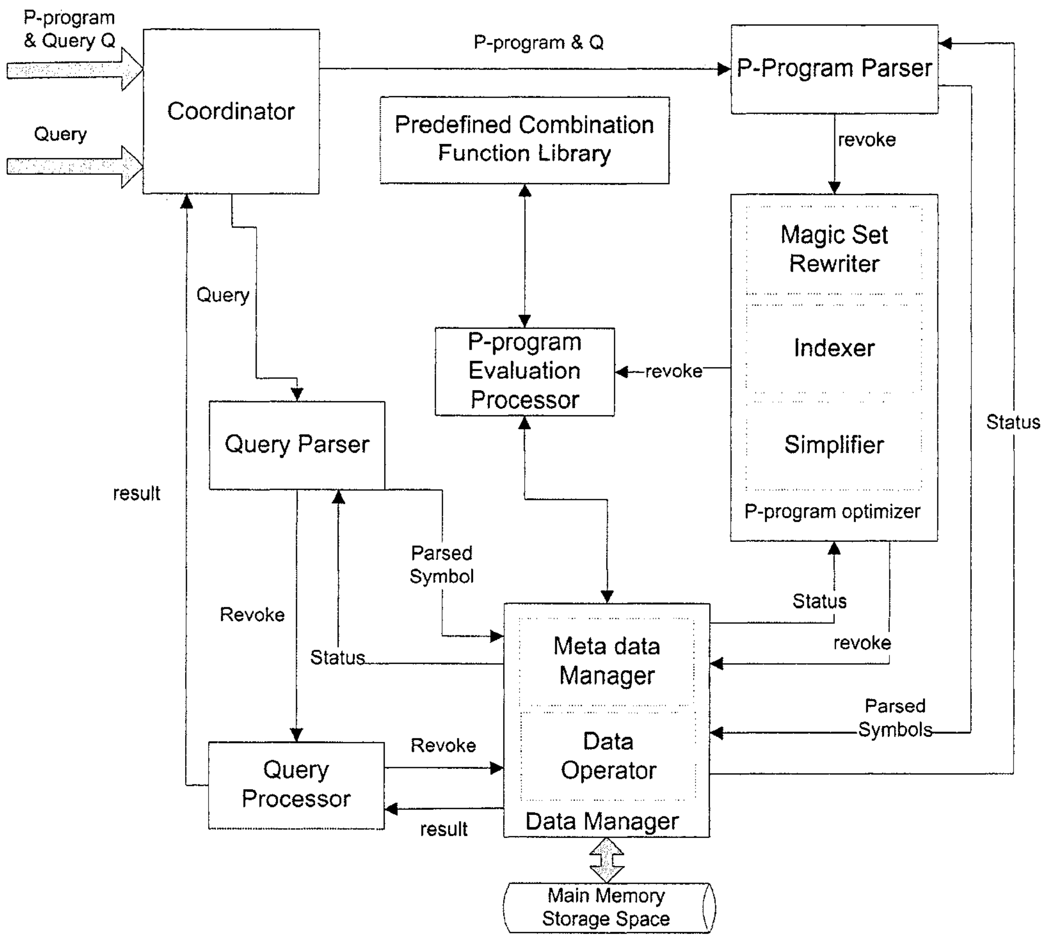

Figure 4.1: System Architecture and Components

the following.

- **Coordinator**

The coordinator is the central component of the prototype system. It interacts with other modules, controls the data flow, and determines the execution order between the modules. It provides interface for the user, accepts the request, invokes different components and outputs the final results.

- **Parser**

The parser performs two tasks: program parsing and query parsing. Each of these reads the input and collects information to identify basic logical units. A logical unit is a meaningful element in the source language called token. The parser takes the collected tokens in stream and builds the syntax representation in the memory. Then, a top-down parsing technique is used to determine the grammar of the program based on the syntax representation. According to the p-program grammar, we create a well-defined data structure into which we store the rules and atom-certainty pairs (tuples).

- **Evaluation Processor**

This component computes the aggregation view for the input p-program. Three multiset based evaluation algorithms are supported in our system. They are naive (N), semi-naive (SN) and semi-naive with partitioning (SNP). The practical evaluation algorithms for evaluating the generalized magic sets rewritten programs are also implemented.

- **Data Manager**

The data manager is a set of libraries which communicates with the main storage manager and other components. The basic operations of the data manager include data storage, data deletion, data updating and data retrieval.

- **Query Processor**

The query processor in UNLOG is a component that allows a user to report structured data pulled from the storage. The major tasks of the query processor include (1)receiving queries from the coordinator, (2)generating optimized query plan, (3)informing data manager to manipulate the data based on the query plan (4) and encapsulating the results.

- **P-Program Optimizer**

The P-program optimizer is a key component that implements the optimization algorithms when computing the fixpoint of a given p-program. This component includes three modules. They are the magic sets rewriter, the indexer, and the simplifier modules.

Magic sets rewriter implements GMS and GSMS algorithms. The input to this module is a parsed p-program and a query. Its output is a new set of rules and the seeded magic atom-certainty pair referring to the bound arguments in the query.

The indexer is used to speed up locating tuples in the database. In our system, indexer generates indices for EDB atoms and IDB atoms derived during query processing. It is also responsible for creating indices for intermediate data structure and using them.

Simplification is an important technique to discard unnecessary rules in a logic program. Reducing the size of the program may result in increasing efficiency during the evaluation of programs. However, parametric framework is based on multisets and uses uncertainty and hence, a careful use of simplification is required (see Section 4.3.3). In our current engine, the simplifier conducts two major tasks: (1) eliminates rules that do not contribute to results of a given query when magic sets rewriting technique is not applicable; (2) reduces duplicated magic rules.

4.2 Data Structures

A suitable data structure is crucial to achieve better performance. Since Sun JDK is our implementation environment, we used existing classes, data structures and implemented algorithms in Sun JDK. For example, we use the quick sort algorithm is used to re-order subgoals, hash function for indexing, and existing data structures such as hash table, queue and stack are used to store tuples and intermediate results.

4.2.1 Representation of Indices

Indexing is a technique for fast locating elements in a structured data collection. Many types of indices exist such as B-tree, R-tree, Hash index and bitmap index etc. In UNLOG, we use indices to locate tuples, tables, and intermediate results. As most of the search operations are single search we mainly use hash indexing in our prototype. For any element to be indexed, we convert it into a string, which makes it unique in the context, described later. Then, we use the function called $HashCode(s : String)$ in Sun JDK to generate a hash key for the element. A pair combining the hash key and the reference of the element is pushed into the relevant hash table.

4.2.2 Tuples

A tuple is an atom-certainty pair of the form "$p(\bar{A}) : \alpha$", where $\bar{A}$ is a sequence of constants, and $\alpha \in [0, 1]$ is its associated certainty. Since tuples in the same table have the same predicate name, we do not save the predicate name; instead, we use hash keys to identify tuples. We first concatenate all terms with "_" to create a string. Then, we use the function $HashCode(s : String)$, which returns a hash key for the tuple. For example, for $p(1, 2) : 0.5$, we first create string "p_1_2", then convert it to a hash code using the function. The hash code is used as the ID of the tuple. The IDs may be used for further indexing. To save the memory space, we do not save every ground term for tuples. All ground terms are stored in a link list. References are assigned to the corresponding positions in tuples, as showed in Figure 4.2.

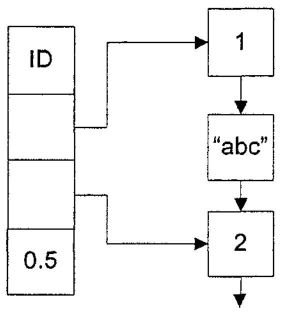

Figure 4.2: Internal representation of a tuple $p(1,2) : 0.5$ in UNLOG

4.2.3 Tables

A table in our implementation, similar to a table in relational databases, is a collection of tuples. Every table has a name, a list of tuples, and a hash table used to save the index. The ID of the table is generated by the function "HashCode(s:String)" using the table name as input. A hash table is used an index for the table, where each element in the hash table points to a tuple in the table. Figure 4.3 shows the structure of a table. While this index

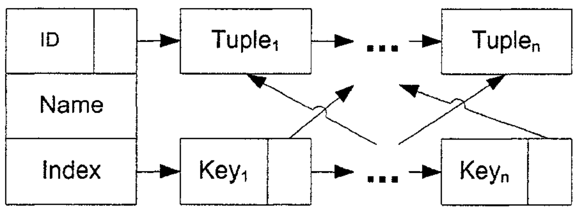

Figure 4.3: The structure of index for a table in UNLOG

is useful for N and SN evaluations, it is not beneficial much for SNP method since for a rule to be fired , it must have at least one new tuple for one of the subgoals found in the previous iteration. To avoid exhaustive search of the table, we partition the tuples in the table into two linked lists. One list is used to save new tuples obtained at last iteration. Another list is used to save the tuples in that table whose certainties were not changed at previous iteration. If the partition for "new" tuples is empty, we know that there would be no new tuples for the table. Figure 4.4 shows the structure of a partitioned table.

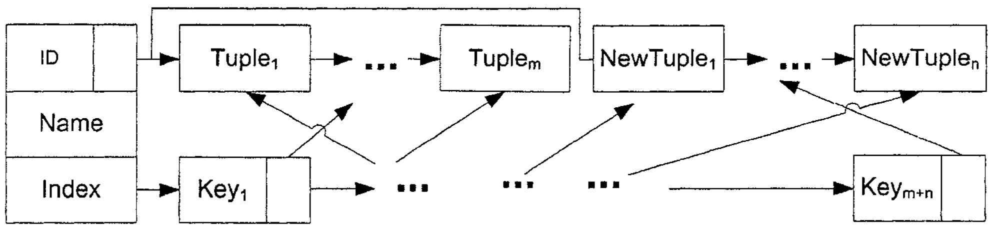

Figure 4.4: The structure of index for a partitioned table in UNLOG

4.2.4 Rules and Queries

To represent a rule r in our implementation, we create data structures for predicates, implicit constraints implied between subgoals and combination functions. A predicate $p(X_1, ..., X_n)$ in a rule represents a relation. We use a field "ID=HashCode(name)" to identify a predicate, and a link list to save its terms. The head of a rule and its subgoals in the body are all predicates. Being expressed within the same rule, implicit constraints are introduced through the typing mechanism of the terms. For example, a rule:

$$r : h(X,Y) \overset{\alpha}{\leftarrow} p(X,Y), q(Y,a); \langle f_d, f_p, f_c \rangle$$

we know that $p.Y = q.Y$ and the second argument of q is a constant 'a'. Depending on the number of parameters, we classify implicit constraints into two groups.

- **Unary constraint** is of the form X=a, where X is a reference pointing to an argument in a rule, and a is a constant.
- **Binary constraint** is of the form X=Y, where X and Y are the references pointing to two variables with the same name but different locations in subgoals.

Due to a well known heuristic "it is better to make selections before joins" in query optimizations, we classify the implicit constraints into two parts in the data structure. Those constraints verified by selections are grouped together. Thus, we create the data structure from rules shown in Figure 4.5. A query is a special case of a rule which does not have body. Hence a query includes head predicate, implicit constraints, and a unique ID.

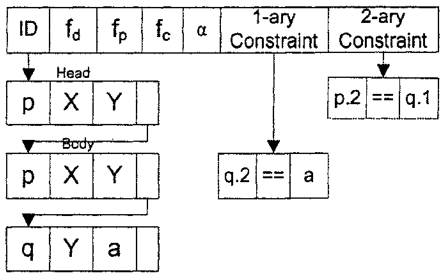

Figure 4.5: Rules in UNLOG

4.3 Optimization Techniques

In the last two decades, a number of query optimization techniques have been introduced for deductive databases, such as [HN84, BMSU86, Vie86, Sun92, Lus92, RRSS94, Hav97, JS94, SZ04]. While most of these techniques are not directly applicable to programs with uncertainty, we may consider the idea, and adapt and extend them to our context. For instance, in section 3.3, we discussed the problems of using magic sets in PF. In our system, we consider such techniques, which can be adapted and applied in our work. They are studied in the following sections.

4.3.1 Subgoals Reordering in Magic Sets Rewriting

We proposed extended magic sets techniques to PF in Chapter 3. In the algorithm in Figure 3.1.1, we proposed that re-ordering subgoals is based on the order of variables which are bounded. A variable called "*order*" is attached to every subgoal. We use a counter to record the number of subgoals visited. Each time a subgoal is visited, this counter is incremented by 1. Finally, we sort the subgoals based on "$order_s$". There are two justifications that are associated with counters for doing this reordering. On one hand, the re-ordered subgoals clearly represent which variables are bounded. There is no need to use extra data structure to represent sideways information passing graph which was used in the magic sets transformation. On the other hand, re-ordering plus materialization corresponds to "pushing selections into joins". For example, consider the following rule:

$$h(X, W) \xleftarrow{\top} r(W), p(X, Y), q(Y, Z); \langle max, min, min\rangle.$$

In general, the join plan $(p(X,Y) \bowtie q(Y,Z)) \bowtie r(W)$ performs faster than $(r(W) \bowtie p(X,Y)) \bowtie q(Y,Z)$ because the join $p(X,Y) \bowtie q(Y,Z)$ is restricted by implicit constraint "$p.Y = q.Y$". Certainly, when r(W) is empty or extremely small, the first execution plan is better. These, however, are exceptional cases, which we do not consider.

4.3.2 Predicate Partitioning

In the context of bottom-up evaluation, given two relations R and S in a p-program P, we use Δ to denote all improved atom-certainty pairs at iteration i and use Λ to denote the rest of atom-certainty pairs. An obvious way to compute $R \bowtie S$ is to reduce unnecessary joins at every iteration, that is,

$$\Delta(R \bowtie S) = \Delta((\Delta(R) \bowtie \Delta(S)) \dot{\cup} (\Delta(R) \bowtie \Lambda(S)) \dot{\cup} (\Lambda(R) \bowtie \Delta(S)))$$

The expression on the right hand side (RHS) excludes the uncertainty join $\Lambda(R) \bowtie \Lambda(S)$ because we may obtain the results for $\Lambda(R) \bowtie \Lambda(S)$ from the previous iteration by the bookkeeping technique. For example, if $|\Delta(R)| = 50$, $|\Lambda(R)| = 200$, $|\Delta(S)| = 50$, and $|\Lambda(S)| = 300$, then RHS does not perform $\Lambda(R) \bowtie \Lambda(S)$, which saves $200 \times 300 = 60000$ joins of tuples. Note that in PF atoms derived at some iteration will continue to be derived at all the subsequent iterations in the fixpoint bottom-up evaluation. Less and less derivations produce "better" certainties, so the potential of the idea in improving performance is huge. This idea is developed in [SZ04], called partitioning.

The idea of partitioning for programs with uncertainty was developed in [SZ04] and the algorithm proposed for partitioning was called semi-naive with partitioning, or SNP for short. In our system, we implement SNP. Every relation saved in the database is divided into two partitions: old and new. We implement joins in a rule at avoiding performing the joins which consists of only old tuples, denoted as partition.

4.3.3 Program Simplification

There is no doubt that evaluations of a logic program is faster with less number of rules. Simplification is a technique to remove unnecessary rules. In our implementation, we execute simplification procedure under two situations. One situation is when we wish to apply magic sets rewriting technique but there is no bound argument in the given query. For instance, consider the following example:

Example 4.3.1.

Original Program:

$r_1 : p(X,Y) \overset{0.5}{\leftarrow} a(X,Y); \langle ind, pro, pro \rangle.$

$r_2 : p(X,Y) \overset{0.5}{\leftarrow} p(Y,Z), p(Y,X); \langle ind, pro, pro \rangle.$

$r_3 : q(X,Y) \overset{0.7}{\leftarrow} p(X,Y); \langle ind, pro, pro \rangle.$

$a(1,2) \overset{0.5}{\leftarrow}; \langle \_, \_, \_ \rangle.$

$a(2,1) \overset{0.5}{\leftarrow}; \langle \_, \_, \_ \rangle.$

$a(1,1) \overset{0.5}{\leftarrow}; \langle \_, \_, \_ \rangle.$

$?p(X,Y).$

Rule "r_3" in this example does not contribute to answer query $Q = p(X,Y)$. Besides, the system will not generate the magic sets rewritten program because there is no bound argument in Q. To simplify the program, in this example, we start from the query with predicate name p. For every rule r with p in the head, we recursively check the IDB subgoals and collect their names a collection C. Such a rule "r" is then marked "*scanned*". Then, we get the next predicate name from C, and continue scanning until C is exhausted. This yields all useful rules, which are marked "*scanned*". Algorithm 4.3.1 shows the steps to eliminate unnecessary rules.

Another situation in which we may be able to simplify a program is when the magic sets rewritten program is generated, which may include duplicated magic rules. In general, we should not eliminate the rules in the input program because the parametric framework is based on multisets, and atoms derived from rules are combined by disjunction functions. If there are two rules that are exactly the same, we cannot eliminate any one of them because any such rule might result in a new derivation of instances of tuples, which are duplicately derived by another rule. However, magic rules are different, in that each magic fact derived by magic rules is always associated with certainty $\top$. The number of instances of such tuples does not change the certainties of the magic predicates. Therefore, we may remove the duplicate magic rules from the MS rewritten program.

Algorithm 4.3.1. Simplify Program

```
1: procedure $Simplify_P(Q, P)$
2: input: A parsed p-program P and a query Q
3: output: A collection of useful rules w.r.t Q
4: begin
5:     Initialize a queue C=$\emptyset$;
6:     Initialize a collection S=$\emptyset$;
7:     $C.enqueue(\pi(Q))$;
8:     while ($N = C.dequeue()$)
9:         forall $r \in P$, $\pi(r.head) = N$ and $r.scanned = false$
10:            forall subgoal $A$ in the body of r.
11:                if ($\pi(A) = p$ is an IDB)
12:                    $C.enqueue(p)$;
13:            end forall
14:            S.add(r);
15:            r.scanned=true;
16:        end forall
17:    end while
18:    return S;
19: end
```

4.3.4 Materialization and Pipelining

When evaluating a rule, the pipelining approach considers a sequence of joins of subgoals as a bunch of small pieces of jobs. A sets of pointers are used to locate the tuples concerned. The tuples currently pointed to are chosen to participate in a join, and results are materialized. Then, the pointers are moved (piped) to a new combination of tuples for the next join. This process is repeated until all possible combinations are discovered.

In materialization, we consider a sequence of joins of subgoals as sequential tasks. An intermediate relation is introduced to save the result of the first two subgoals after applying joins. Then, the intermediate relation and the third subgoal are considered as the input to the next join, resulting in a newer intermediate relation. This process is repeated until all subgoals are joined.

Both materialization and pipelining have advantages and disadvantages, but we prefer materialization for the following reasons. First, a major advantage of pipelining is that it requires less memory space. However, for both SN and SNP evaluation, we have to book-keep intermediate results, which include all derivations at previous iterations but not for the current iteration, the structures for book-keeping are M_i for SN and C_i for SNP (Section 2.2). If we use M_i or C_i to store the temporary results for joins, the major disadvantage of materialization is eliminated, compared to pipelining in our context. Secondly, a well-known heuristic to improve the joins operation performance in database is to "push selections before joins". If there are any implicit constraints of the form $X = a$, we may performs joins faster. One may argue that if the indices of tuples are attached with some metadata, which indicates the implicit constraints, they may achieve the same performance. For instance, [SZ04] uses index plans to implement the idea. However, when the size of the indices is large, the overhead of using indices may be costly. In our work, we do not have heavy index operations for SNP. Materialization benefits from disk I/O cost. Although our current system prototype is memory-based, materialization helps to deal with large data when they are stored on disk.

Chapter 5

Experiments and Results

In this chapter, we present our experiments on evaluating performance of the proposed techniques and other optimization techniques implemented. We report the test results and relevant analysis.

Different test cases might lead to different test results. Some optimization techniques benefit from some test cases and some might benefit from other techniques. Combining different optimization techniques may result in overhead. To assess the performance of the proposed techniques, we conduct a number of experiments under different test cases. For each test case, we consider different input parameters, such as combination functions, types of data sets, different certainty of rules and different combinations of optimization techniques.

To show the correctness of the implementation, we do the following comparisons. First, we compare the rewritten program of a program P with the one generated by CORAL for the standard case of P. Secondly, we compare our evaluation results with Zheng's implementation [SZ04]. Finally, we study the performance of our system with different optimization including magic sets.

5.1 Experiment Environment

To perform the experiments, we use Sun's Java running time environment (JRE) 1.5.1. The Java virtual machine is installed in a Dell desktop computer with Pentium 4 CPU of 2.4GHz, 2G RAM, 250GB hard disk, and runs under windows XP professional 2003.

5.2 Test Programs Generation

Recursion is an attractive feature of deductive databases. A recursive program is the one in which the depth graph has a cycle. The recursion could come from the same rule or several rules. A p-program P is linear if there is at most one occurrence of a head predicate in the body of any rule, defining in P; otherwise, P is non-linear. To study the performance of magic sets, we adopt the same-generation cousin (SGC) problem, which is widely used as the test program in standard deductive databases. A linear version as well as a non-linear version of SGC program are introduced as follows.

Example 5.2.1. P1: Linear version of the SGC program

$r_1 : sgc(X, X) \overset{\alpha}{\leftarrow} person(X); \langle f_d, f_p, f_c \rangle.$
$r_2 : sgc(X, Y) \overset{\alpha}{\leftarrow} par(X, Z), sgc(Z, W), par(Y, W); \langle f_d, f_p, f_c \rangle.$
$\cdots$ { Relevant facts } $\cdots$

Example 5.2.2. P2: Non-linear version of the SGC program

$r_1 : sgc(X, Y) \overset{\alpha}{\leftarrow} flat(X, Y); \langle f_d, f_p, f_c \rangle.$
$r_2 : sgc(X, Y) \overset{\alpha}{\leftarrow} up(X, Z1), sgc(Z1, Z2), flat(Z2, Z3), sgc(Z3, Z4), down(Z4, Y);$
$\langle f_d, f_p, f_c \rangle.$
$\cdots$ { Relevant facts } $\cdots$

Each of either program of the above has two rules. To experiment programs with more rules, we define a $N \times N$ structure which represents a SGC program holding $2 * N^2$ rules. A $N \times N$ SGC program has N^2 sets of rules. Each set R_{ij}, where $1 \leq i, j \leq N$, contains two rules. If $i = 1$, R_{1j} includes ${r_1}^{1j}$ and ${r_2}^{1j}$ which are the rules in either Example 5.2.1 or Example 5.2.2, and attached with subscript $1j$. If $1 < i \leq N$, R_{ij} also includes two rules: ${r_2}^{ij}$ and a new rule with attached subscript, denoted as ${r_3}^{ij}$:

$$ {r_3}^{ij} : sgc_{ij}(X,Y) \overset{\alpha}{\leftarrow} sgc_{(i-1)j}(X,Y); \langle max, min, min \rangle. $$

For example, a 2×2 p-program for P1 in Example 5.2.1 is shown in Figure 5.1, denoted as "$P1_2X2$". In the graph, the program includes 8 rules divided into four sets and each set contains 2 rules. In the sense of evaluation, atoms for sgc_{21} cannot be derived until sgc_{11} is prepared. We apply combination functions "$\langle max, min, min \rangle$" to r_3 in order to initialize the certainty of sgc_{21} at the start point is 0.5. In this case, the 2×2 P1 combines four copies of P1 and evaluate them at the same time.

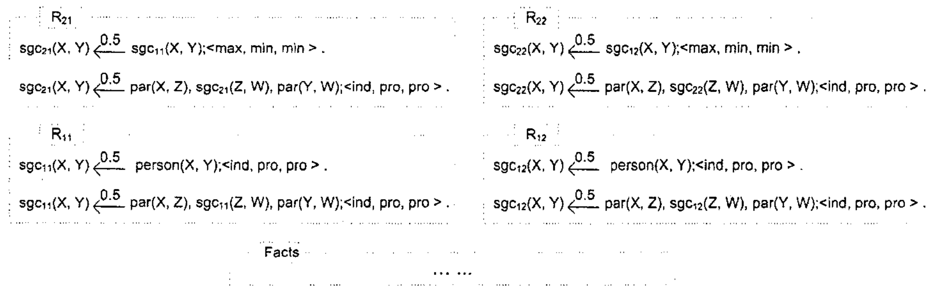
R21
sgc21(X, Y) ←0.5 sgc11(X, Y);<max, min, min >.
sgc21(X, Y) ←0.5 par(X, Z), sgc21(Z, W), par(Y, W);<ind, pro, pro >.
R22
sgc22(X, Y) ←0.5 sgc12(X, Y);<max, min, min >.
sgc22(X, Y) ←0.5 par(X, Z), sgc22(Z, W), par(Y, W);<ind, pro, pro >.
R11
sgc11(X, Y) ←0.5 person(X, Y);<ind, pro, pro >.
sgc11(X, Y) ←0.5 par(X, Z), sgc11(Z, W), par(Y, W);<ind, pro, pro >.
R12
sgc12(X, Y) ←0.5 person(X, Y);<ind, pro, pro >.
sgc12(X, Y) ←0.5 par(X, Z), sgc12(Z, W), par(Y, W);<ind, pro, pro >.
Facts
... ...

Figure 5.1: A 2×2 structured P1

Moreover, for magic sets rewriting technique, an important parameter affecting the efficiency is the input query Q. Different types of queries may lead to different potential set of relevant facts, which contribute to the answers of Q.

5.3 Test Data Sets Generation

In standard deductive databases, a number of data sets for SGC have been introduced and used to measure the efficiency of query processing and optimization techniques [RSS94, DMP93, BR86, KNSSS90]. Zheng [SZ08] adopted these data sets and made them suitable for p-programs by incorporating uncertainty rules and combination functions. In this thesis, we employed the relevant data sets from Zheng for P2 and developed our own data sets T_n for P1. To demonstrate the relations between tuples, we introduce the notation of data set graph (DSG), which is limited to generation of binary tuples and is suitable for P1 and P2. DSG is a directed graph in which vertices are constants. An edge (A, B) from A to B is labeled with a predicate name p. To construct a data set, we exhaustedly search the whole data set graph, and for each edge, we generate a fact whose predicate is the label of the edge and the arguments are the labels of the vertices connected. The certainty associated with the tuple is generated by the data set generation program developed. Figure 5.2 shows how to generate facts based on a data set graph.

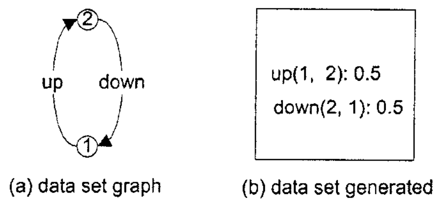

Figure 5.2: A data set graph and the tuples it denotes

Below, we present the DSGs for the data sets generated and used in our implementation.

- **Data set T_n**

The DSG of T_n (see Figure 5.3) looks like a "balanced" binary tree, in which all leaves are at the same depth. Each node has an outgoing edge to its parent, and each parent has two edges pointing to its children.

- **Data set A_n**

The DSG of A_n shown in Figure 5.4 looks like a pyramid. As the number of layers in this data set increases, the size of the bottom layer increases. There is at most one matched path from a node to its same generation node. It means that if there is an inference for an

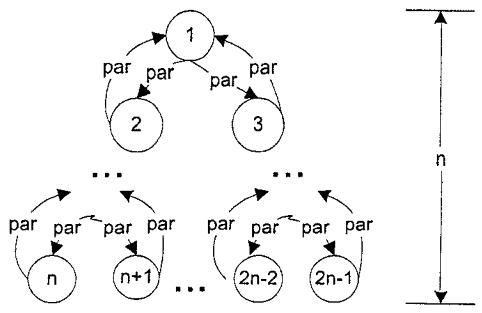

Figure 5.3: The DSG graph of data set T_n

answer, there is at most one match. To find a same generation node in level i, the derivation must go to the top-level layer and then down to the layer i. The more layers it has, the more computation time is required to find the match.

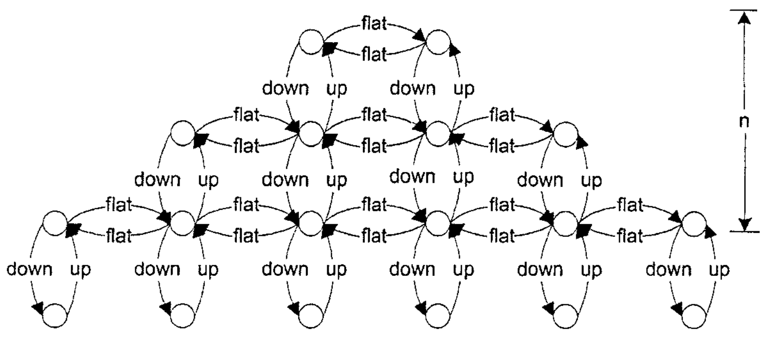

Figure 5.4: The DSG graph of data set A_n

• **Data set** B_n

The DSG of B_n shown in Figure 5.5 contains n layers of nodes. Each layer has 8 nodes. There are four arcs connecting a lower layer and its immediate higher layer, two of which are upward and the other two are downward.

• **Data set** C_n

The data set C_n is very similar to B_n. Each layer in C_n includes a single linked list of 8 nodes. Each node has an arc connected to the corresponding node at the higher layer. All arcs connecting a higher layer to its immediate lower layer are bi-directional.

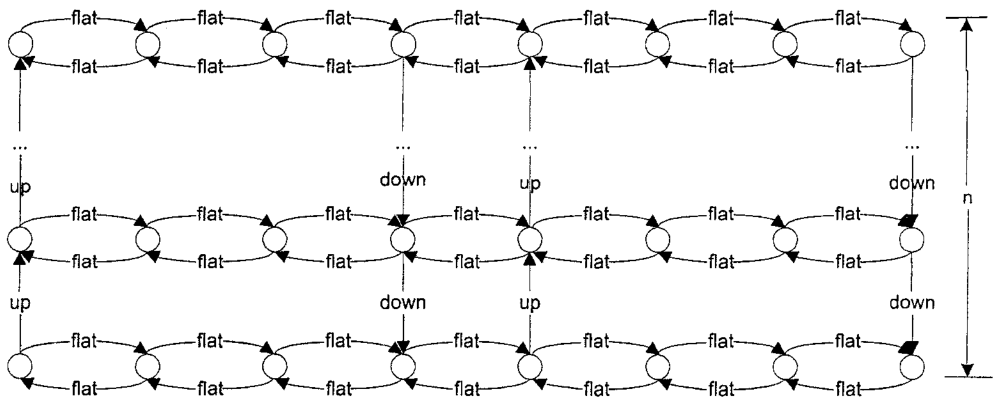

Figure 5.5: The DSG graph of data set B_n

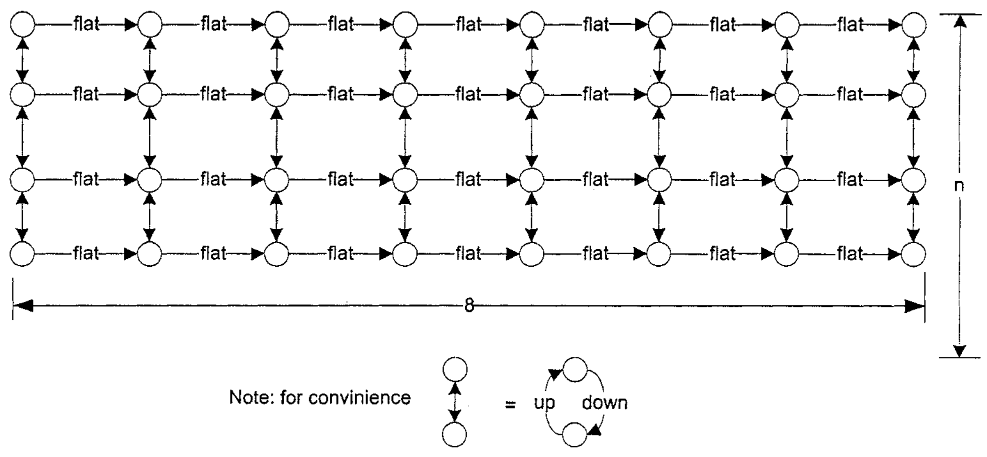

Figure 5.6: The DSG graph of data set C_n

• **Data set** F_n

F_n is a variant of C_n. Unlike C_n, the length of each layer and the number of layers in F_n are flexible. The number of layers is the same as the length of each layer. Each node in the lowest layer has an extra arc to the corresponding node of each higher layer.

• **Data set** S_n

S_n includes $2n$ linked lists, each of which has n nodes. The first n lists have upward links, while the second n lists have downward links. There is only one link between two nearest linked lists.

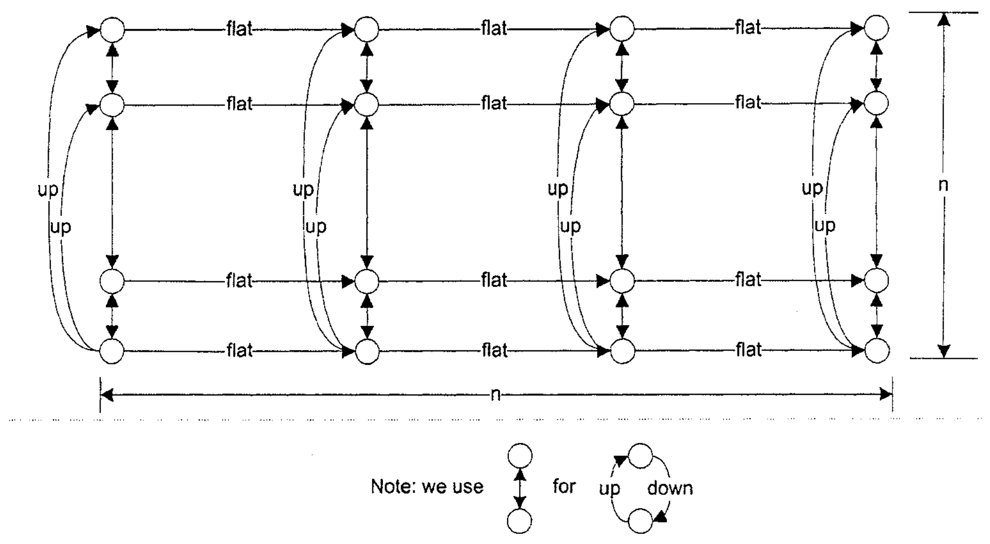

Figure 5.7: The DSG graph of data set F_n

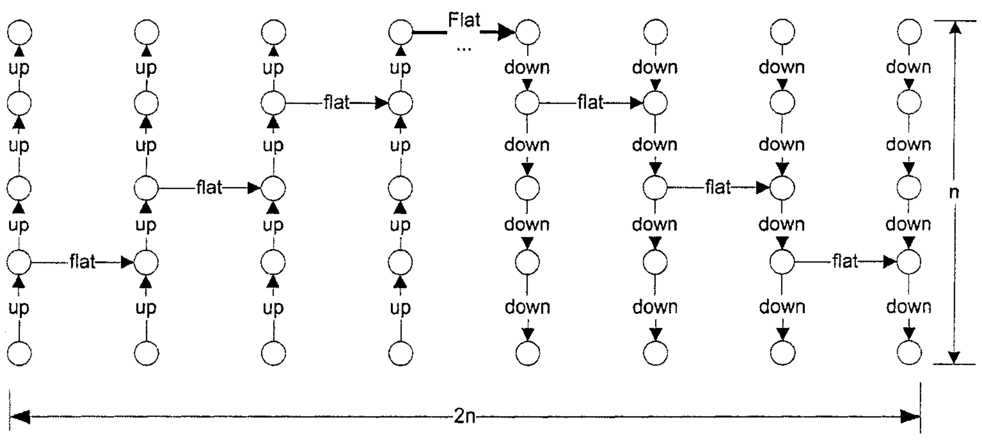

Figure 5.8: The DSG graph of data set S_n

- **Data set $T_{n,m}$**

This data set represents an m-ary tree, where n indicates the height of the tree and m is the number of children an internal node has. As an example, Figure 5.9 shows the DSG of $T_{2,3}$, where each internal node that has three children and provides many paths between any two same generation nodes. When the height n increases by 1, the total number of edges in the graph increases by $3m^{n-1}$.

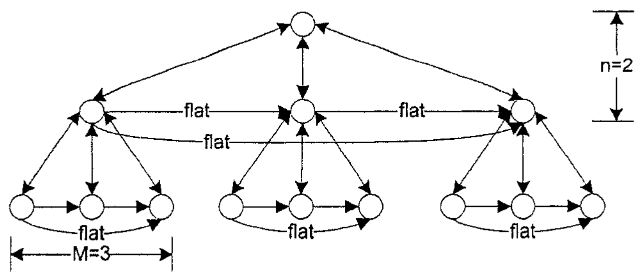

Figure 5.9: The DSG graph of data set $T_{2,3}$

• **Data set** $U_{n,m}$

$U_{n,m}$ is another variant of C_n. It makes the number of layers n and the length of each

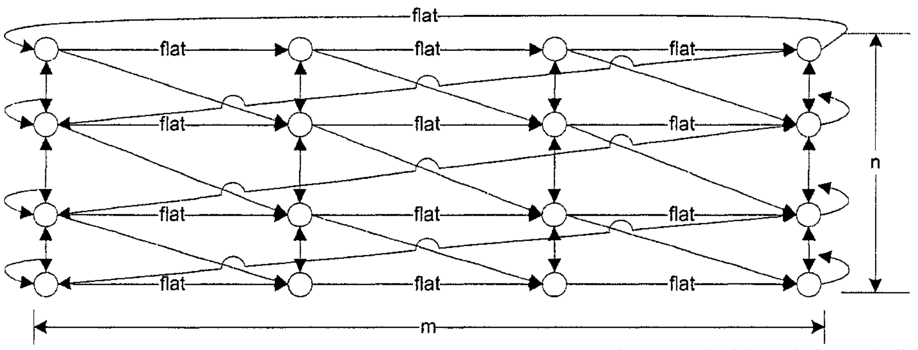

Figure 5.10: The DSG graph of data set U_{nm}

layer m flexible. Each layer is revised to be cycle-linked. This also causes the connections between two nearest layers to be more frequent.

5.4 Performance Evaluation Standards

We consider the following parameters to measure the performance.

• **The evaluation time:** the time to compute the least fixpoint. We use $\tau(D)$ to denote the computing time for data set D.

• **The rewriting time:** the time used to rewrite a p-program. We use $\ell(D)$ to denote this compile time for data set D.

• **Facts generated:** The number of IDB tuples generated, obtained by counting the number of tuples in the database in the fixpoint evaluation minus the number of tuples in the EDB. We use $\delta(D)$ to denote this number for data set D.

• **Potential Facts Ratio:** $\chi(D,Q) = |\{A|A \in \delta(D) \text{ and } Q \rhd A\}|/\delta(D)$, where A is a desired tuple subsumed by the query Q. This ratio indicates the portion of the derived facts that is matched or subsumed by the user query Q. A smaller value for $\chi(D,Q)$ is expected to bring advantage for using MS technique.

• **Speedup:** $\lambda(D) = \tau(D)/(\tau^m(D) + \ell^m(D))$.

5.5 Test Results and Analysis

Due to the limitation of the memory, the database size considered in our experiments was in the range from 5,000 to 100,000 tuples. We report our experimental results in the following sections.

5.5.1 N, SN and SNP

Without applying the proposed magic sets techniques, Table 5.3 shows the speedups of semi-naive (SN) over naive (N) and semi-naive with partition (SNP) over N by evaluating different types of test data. Row 1 records the speedup of SN for the programs with uncertainty, where $\alpha = 0.5$ and $\langle f_d, f_c, f_p\rangle = \langle ind, *, *\rangle$. Table 5.4 shows the running time for some test cases for P2 with "1 X 1" structure in our system and the running time for the same test cases in Zheng's implementation "Z3". Note that the numbers of "UNLOG" and "Z3" shown in Table 5.4 are not comparable because "UNLOG" was running under Java virtual machine, and Zheng's implementation was written in $C++$ and the target code was running above the operating system. However, we may compare the trend of performance gained from the different test cases. It seems that "UNLOG" achieves a better improvement

Table 5.3: Speedup λ of SN over N and SNP over N

	T_{16}	A_9	B_{64}	C_{64}	F_{32}	S_{64}	$T_{5,4}$	$U_{15,10}$
SN	2.87	1.05	1.04	1.03	1.13	1	1.03	1.07
SNP	7.17	2.56	1.40	1.18	2.31	0.75	1.05	1.03

Table 5.4: Running time of N, SN and SNP for some test cases with "1 × 1" program structure (time in millisecond)

	A_9		B_{64}		C_{64}		F_{32}	
	UNLOG	$Z3$	UNLOG	$Z3$	UNLOG	$Z3$	UNLOG	$Z3$
N	64,109	560,716	766	53,947	891	147,562	146,078	1417,768
SN	44,234	260,575	735	880	859	931	128,500	51,213
SNP	16,719	12,824	547	851	750	822	63,281	43,843

	S_{64}		$T_{5,4}$		$U_{15,10}$	
	UNLOG	$Z3$	UNLOG	$Z3$	UNLOG	$Z3$
N	47	37,263	89,219	94669,628	3,688	13,730
SN	47	711	86,625	113,644	3,438	6,199
SNP	62	60	85,016	82,008	3,578	6,159

of N and SN evaluation than "Z3" does. The reason is the difference of joins manipulation. We believe that materialization technique, compared to pipelining, results in "UNLOG" achieving better performance for N and SN. Relevant discussion is referred to in Section 4.3.4. Therefore, although we may see programs benefit from SN and SNP, the speedup reported in our implementation is much smaller than the speedup reported in [SZ04]. The speedup of Zheng's engine by comparing N over SNP ranges from 2 to 213 times for the same test data; whereas, we obtained speedup for SNP over N ranges from 0.75 to 7.17.

5.5.2 SN vs SN+GMS

As in the worst case, magic sets techniques may not result in an efficiency increase when χ is close to 100%. It happens when all the EDB facts are potentially relevant to the given query. We focus on finding the speedup when the potential facts ratio χ varies between 1% and 20%. Table 5.5 shows how SN is affected by GMS. Columns 2, 4, and 6 record the speedup ranges obtained for the standard case, where $\alpha = 1$, $\langle f_d, f_c, f_p \rangle = \langle max, min, min \rangle$.

Table 5.5: Speedup λ of GMS for SN

	$\chi = 1\%$		$\chi = 5\%$		$\chi = 10\%$	
	$\alpha = 1$	$\alpha = 0.5$	$\alpha = 1$	$\alpha = 0.5$	$\alpha = 1$	$\alpha = 0.5$
SN	$5 \sim 700$	**$2.9 \sim 550$**	$0.25 \sim 50$	$0.2 \sim 45.2$	$0.14 \sim 17.9$	$0.12 \sim 16$

Table 5.6: Impact of GMS for SN with different data sets, $\alpha = 0.5$

$\chi(A_n, Q)$	$\lambda(A_n)$	$\chi(B_n, Q)$	$\lambda(B_n)$	$\chi(C_n, Q()$	$\lambda(C_n)$	$\chi(F_n, Q)$	$\lambda(F_n)$
15.8%	12.0	18.4%	0.5	19.8%	0.35	19.1%	2.5
8.7%	18.1	10.1%	1.3	9.7%	1.2	10.5%	4.1
4.2%	51.2	5.3%	3.8	4.65%	4.4	4.5%	6.6
2.7%	95.3	3.0%	9.0	3.0%	8.6	3.8%	8.9
1.7%	169.0	1.77%	13.7	2.1%	13.6	2.5%	16.5

$\chi(T_n, Q)$	$\lambda(T_n)$	$\chi(T_{nm}, Q)$	$\lambda(T_{nm})$	$\chi(U_{nm}, Q)$	$\lambda(U_{nm})$	$\chi(S_n, Q)$	$\lambda(S_n)$
19.65%	4.03	2.62%	10.0	20.2%	3.0	13.6%	0.1
7.55%	14.12	**1%**	**26.3**	12.7%	13.4	9.0%	0.15
4.54%	22.6	0.61%	40.6	4.3%	49.0	3.97%	0.3
3.21%	31.1	0.43%	98.0	**1%**	**550.0**	2.4%	0.7
2.48%	41.49			0.83%	612.0	**1%**	**2.9**

For example, considering SN evaluation with certainty $\alpha = 0.5$ and $\chi = 1\%$, the speedup we obtained ranges from 2.9 to 550 times. Columns 3, 5, and 7 indicate ranges for programs with uncertainty, where $\alpha = 0.5$ and $\langle f_d, f_c, f_p \rangle = \langle ind, *, * \rangle$. From Table 5.5, the evaluation performance obtained for the standard p-programs is better than p-programs with uncertainty. The smaller the χ is, the bigger the difference between the two speedups is.

Table 5.6 shows more details for SN, where $\alpha = 0.5$, $\langle f_d, f_c, f_p \rangle = \langle ind, *, * \rangle$. There are eight types of test cases reported in this table. The first column for each case represents the χ and the second column represents the speedup. Figure 5.11 shows the speedup λ based on different *chi* values. The horizontal axis is *chi*, and the vertical axis is the speedup λ. The potential facts ratio χ is an influential parameter affecting the speed. The smaller χ is, the larger speedup we obtain. Whatever type of data sets used, the test program benefits from GMS when χ is small. The degree of impact of GMS is determined by the complexity of the data set and its structure.

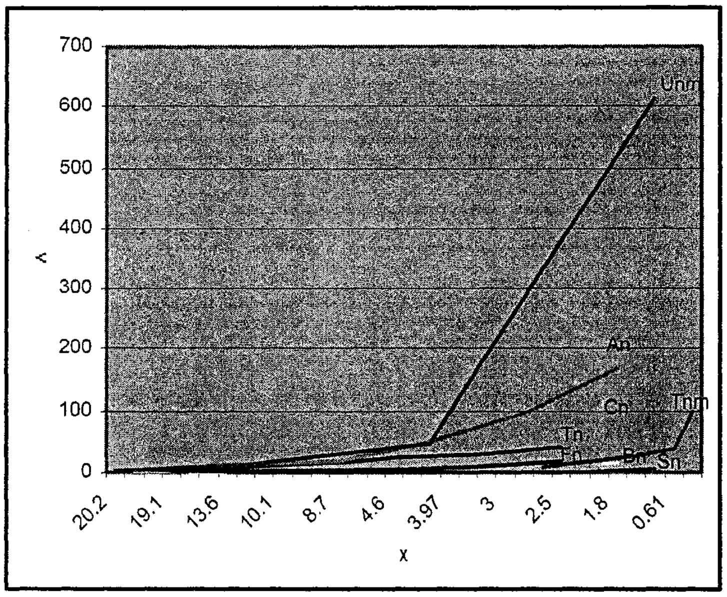

Figure 5.11: χ/λ graph of GMS for SN with different datasets

5.5.3 SNP vs SNP+GMS

In the previous section, we analyze how GMS affects SN evaluation. In this section, we are to report how GMS affects SNP evaluation. Table 5.7 shows how SNP is affected by GMS. Columns 2, 4, and 6 record the speedup ranges obtained for the standard p-programs, where $\alpha = 1$, $\langle f_d, f_c, f_p \rangle = \langle max, min, min \rangle$. Columns 2, 4, and 6 indicate the ranges for programs with uncertainty $\alpha = 0.5$, where $\langle f_d, f_c, f_p \rangle = \langle ind, *, * \rangle$. From Table 5.7, the smaller χ is, the larger the speedup benefiting from GMS is.

Table 5.7: Speedup λ of GMS for SNP

	$\chi = 1\%$		$\chi = 5\%$		$\chi = 10\%$	
	$\alpha = 1$	$\alpha = 0.5$	$\alpha = 1$	$\alpha = 0.5$	$\alpha = 1$	$\alpha = 0.5$
SNP	$3.5 \sim 400$	$2 \sim 280$	$0.5 \sim 38$	$0.4 \sim 33.9$	$0.27 \sim 11.1$	$0.3 \sim 8$

Table 5.8 shows more detail for SNP, where $\alpha = 0.5$, $\langle f_d, f_c, f_p \rangle = \langle ind, *, * \rangle$. The first column

Table 5.8: Impact of GMS for SNP with different data sets, $\alpha = 0.5$

$\chi(A_n, Q)$	$\lambda(A_n)$	$\chi(B_n, Q)$	$\lambda(B_n)$	$\chi(C_n, Q()$	$\lambda(C_n)$	$\chi(F_n, Q)$	$\lambda(F_n)$
15.8%	5.84	18.4%	0.59	19.8%	0.44	19.1%	2.48
8.7%	7.98	10.1%	1.32	9.7%	1.42	10.5%	3.97
4.2%	21.54	5.3%	2.79	4.65%	3.85	4.5%	6.83
2.7%	34.5	3.0%	7.39	3.0%	6	3.8%	8.1
1.7%	50.4	1.77%	11.7	2.1%	8.86	2.5%	12.3

$\chi(T_n, Q)$	$\lambda(T_n)$	$\chi(T_{nm}, Q)$	$\lambda(T_{nm})$	$\chi(U_{nm}, Q)$	$\lambda(U_{nm})$	$\chi(S_n, Q)$	$\lambda(S_n)$
19.65%	4.88	2.62%	11.4	20.2%	2.5	13.6%	0.19
7.55%	9.84	**1%**	**19.3**	12.7%	10.3	9.0%	0.31
4.54%	16.29	0.61%	33.89	4.3%	39.36	3.97%	0.75
3.21%	26.04	0.43%	59.72	**1%**	**400**	2.4%	1.46
2.48%	31.52			0.83%	455.1	**1%**	**6.07**

in the table represents the χ ratio and the second column represents the speedup. Figure 5.12 shows the trend of how different types of datasets benefit from GMS.

As a result, SNP is also beneficial from GMS. Like SN, the degree of influence of GMS is

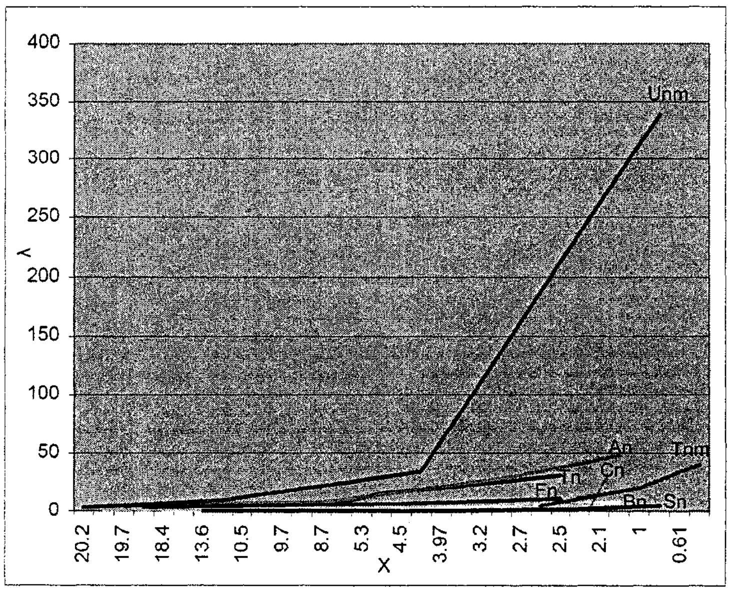

Figure 5.12: χ/λ graph of GMS for SNP with different datasets

determined by the potential facts ratio χ. The smaller χ is, the more benefit the programs gain from GMS technique.

5.5.4 SN+GMS vs SNP+GMS

By comparing the influence of GMS to the different algorithms, we examine the test results from two aspects. First, we examine the speedup gained for SN over SNP by comparing Table 5.6 with Table 5.8. For most test cases the evaluation of SN achieves better performance than that of SNP. For instance, "SN+magic" achieves a speedup of 12 times faster than SN, while the result of "SNP+magic" is only 5.84 times faster than SNP for A_n by considering $\chi = 15.8\%$. Figure 5.13 shows the significant difference. Secondly, we compare the actual running time between "SN+GMS" and "SNP+GMS". Table 5.9 shows the running time for some test cases that represent the general situation of different types of test data. The second row shows the SN running time of the GMS for P2, where $\alpha = 0.5$, $\langle f_d, f_c, f_p \rangle = \langle ind, *, * \rangle$. In most cases, "$SNP + GMS$" performs faster than "SN+GMS", except "T_n" which has the less complex data structure, i.e., less *cyclic* data than other data sets.

As a result, we conclude that different evaluation schemes yield different efficiency gains.

Table 5.9: The running time for SN+GMS and SNP+GMS for some test cases (time in millisecond) for P2 with "5 × 5" program structure, $\chi \approx 5\%$

	T_{16}	A_9	B_{64}	C_{64}	F_{16}	S_{64}	$T_{5,4}$	$U_{15,10}$
$SN + GMS$	250	688	1,766	5,250	7,234	2,735	144	564
$SNP + GMS$	94	766	1,656	4,672	6,250	1,641	297	547

Semi-naive evaluation benefits more from GMS rewriting than SNP does. However, "SNP + GMS" yields better performance than "SN + GMS" does, especially when the test data is in a more complex structure.

5.5.5 SN+GMS vs SN+GSMS

Table 5.10 shows how SN is affected by GSMS. Columns 2, 4, and 6 record the speedup ranges obtained for the standard p-programs. Columns 3, 5, and 7 indicate the ranges for

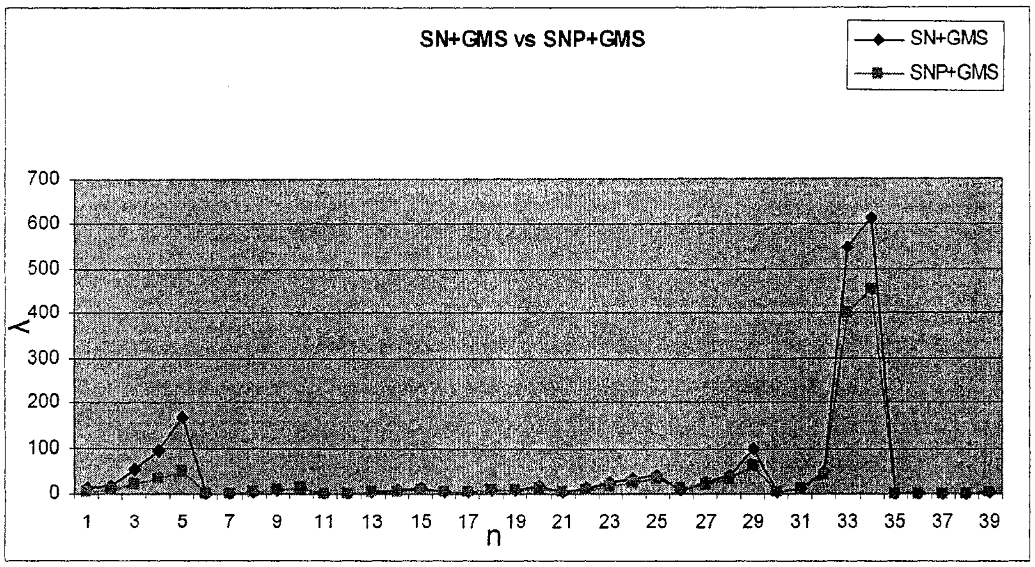

Figure 5.13: Speedup graph of SN+GMS vs SNP+GMS

programs with uncertainty, where the rule certainty set is $\alpha = 0.5$ and $\langle f_d, f_c, f_p\rangle = \langle ind, *, *\rangle$.

Comparing Table 5.10 with Table 5.5, "SN+GSMS" achieves better performance than

Table 5.10: Speedup λ of GSMS for SN and SNP

	$\chi = 1\%$		$\chi = 5\%$		$\chi = 10\%$	
	$\alpha = 1$	$\alpha = 0.5$	$\alpha = 1$	$\alpha = 0.5$	$\alpha = 1$	$\alpha = 0.5$
SN	$12 \sim 1700$	**$1.3 \sim 395$**	$0.76 \sim 64$	$0.1 \sim 33$	$0.2 \sim 19$	$(< 0.1) \sim 14.5$
SNP	$13 \sim 651$	**$1.03 \sim 311$**	$0.33 \sim 51$	$0.1 \sim 27$	$0.2 \sim 17$	$(< 0.1) \sim 13$

"SN+GMS" does for most standard p-programs, whose certainty are always 1. Like Datalog program, for standard p-programs, the GSMS rewritten program achieves better running time by reducing intermediate joins for magic predicates and supplementary magic predicates. However, to retain the dependency of the subgoals in a rule, we modified the evaluation algorithms for GSMS (Section 3.5) to avoid the evaluation bias of uncertainties between the original program and the GSMS rewritten program at every iteration. Computation for every supplementary magic predicate at every iteration has to be re-applied while most of these computation may be saved for the standard p-programs and hence, computing

Table 5.11: Impact of GSMS for SN with different data sets, $\alpha = 0.5$

$\chi(A_n, Q)$	$\lambda(A_n)$	$\chi(B_n, Q)$	$\lambda(B_n)$	$\chi(C_n, Q()$	$\lambda(C_n)$	$\chi(F_n, Q)$	$\lambda(F_n)$
15.8%	9.47	18.4%	0.37	19.8%	0.27	19.1%	2.6
8.7%	16.6	10.1%	0.9	9.7%	1.28	10.5%	4.4
4.2%	47.6	5.3%	4.1	4.65%	4.59	4.5%	6.6
2.7%	69	3.0%	9.9	3.0%	10.6	3.8%	8.7
1.7%	105	1.77%	19.4	2.1%	17.04	2.5%	11.7

$\chi(T_n, Q)$	$\lambda(T_n)$	$\chi(T_{nm}, Q)$	$\lambda(T_{nm})$	$\chi(U_{nm}, Q)$	$\lambda(U_{nm})$	$\chi(S_n, Q)$	$\lambda(S_n)$
19.65%	2.58	2.62%	< 0.1	20.2%	3.0	13.6%	< 0.1
7.55%	9.95	**1%**	**13**	12.7%	13.4	9.0%	0.1
4.54%	22.6	0.61%	54	4.3%	43.0	3.97%	0.22
3.21%	39.8	0.43%	92.0	**1%**	**395**	2.4%	0.56
2.48%	65.7			0.83%	453.0	**1%**	**1.3**

the results for supplementary magic predicates becomes overhead in GSMS rewritten programs with uncertainty. Therefore, "SN+GMS" performs better than "SN+GSMS" does for p-programs with uncertainty for most cases. Exceptionally, by examining Table 5.6, the GSMS performance for data sets "B_n" and "C_n" is better than GMS. The proposed GSMS rewriting includes two stages when the type 2 combination function is applied. In the first stage, we generate the *straightforward* GSMS rewritten program. The second stage is to obtain all magic facts by evaluating the conversion from from the *straightforward* GSMS transformation program to a standard p-program. Compared to GMS, the second stage may yield better performance in GSMS, like the Datalog program. This gain might result in GSMS winning the whole evaluation provided that the computation for the supplementary predicates are not very complex, and the potential facts rate is very small.

Chapter 6

Conclusion and Future Research

In this thesis, we studied techniques for efficient fixpoint evaluation of programs with uncertainty. Since magic sets rewriting techniques has been used in Datalog and standard logic programs, our goal was to incorporate the idea in our context, parametric framework (PF). Our studies cover four major aspects. They are (1) developing magic sets rewriting algorithms, (2) establishing the correctness of rewriting algorithms, (3) adapting the proposed techniques, and (4) reporting the evaluation performance.

Generalized magic sets rewriting technique (GMS) may result in better performance than magic sets rewriting technique when programs have subgoal-rectified rules, so we focus on GMS which is similar to GMS in Datalog, except rules in a p-program associates with combination function, and hence adopting proper combination functions in each rewriting step is the major difference to GMS in Datalog. Invisible problem arises when the type 2 disjunction function (f_d) is applied. The evaluation order of the magic sets rewritten program will be changed when the magic atoms are not fully prepared. Atoms evaluated at the same iteration in a p-program might be separately evaluated at different iterations and this is the major barrier extending GMS to PF. Our study has shown that no matter what type of disjunction functions is applied, the fixpoints of a p-program P and its magic sets rewritten program P^m meet in the limit.

To adapt GMS in a practical application with uncertainty under the precision control, we add an additional stage after the *straightforward* GMS such that the *alternative* rewritten program including all magic atoms prepared can be evaluated by the existing evaluation algorithms. This modification guarantees a p-program and its GMS rewritten program are evaluated at the same number of iterations, and certainties of "potentially relevant atoms" computed at each iteration are exactly the same. The tradeoff of the additional step is that the rewritten program has to be evaluated twice.

In Datalog, GSMS is a refinement of GMS. Many joins evaluated repeatedly in GMS can be eliminated, especially for non-linear rules. To extend GSMS for PF, carefully adopting appropriate combination functions is required to retain the original meaning of a p-program because the subgoals in a rule might be divided into several parts, and consequently the dependency of the predicates will be broken. To adapt GSMS in a practical application, we also need an additional stage to obtain all magic atoms ready before the rewritten program is evaluated. In addition, we revised the existing evaluation algorithms to keep the dependency of the subgoals from the original rules. However, this revision forces the evaluation of GSMS to conduct joins saved for GMS in Datalog.

A number of experiments were conducted to report the performance of the proposed techniques. We noted that different programs enjoy different efficiency gain, depending on the potential facts ratio, which measures the capacity for efficiency improvement. When this ratio ranges from 1% to 20%, the efficiency of semi-naive (SN) observed was about 1 to 700 times higher. Our results also indicate that while different evaluation schemes yield different efficiency gain, semi-naive (SN) benefits more from magic sets rewriting than semi-naive with predicate partitioning (SNP). However, SNP combined with GMS yield the best performance for p-programs with uncertainty. For the standard p-program, GSMS yields the best performance, the speedup observed for SN was about 1 to 1700 times and that for SNP was about 1 to 651 times when this ratio ranges from 1% to 20%.

In this thesis, the theoretical extension as well as practical extension of magic sets rewriting techniques to logic programs with uncertainty in this thesis suggest some avenues for future research.

So far, our studies focus on p-programs, which does not include functions, constraints and negation predicates, despite that these features enhance the expressive power of the PF. A "direct" adaption of magic sets techniques to p-programs with stronger expressive power may lead to new challenges. For example, evaluating a p-program associated with type 2 disjunction function is a infinite computation, while when combined with certainty constraints, it might be changed a finite computation. Therefore, the study of relevant problems is a topic for prospect.

When type 2 disjunction function is applied to a p-program, our solutions of magic sets were based on assumption that "the disjunction functions should combine the certainties with the atoms derived at the same iteration, and should not combine the newly derived certainties with prior certainties for the same atom from the same rule [SZ04].", and thus the magic sets rewritten programs are evaluated twice. Seeking better solutions seems to be interesting. Our test runs show that the evaluation results for programs without magic atoms fully prepared seem to be "close" enough to the fixpoint, so we conjecture the evaluation bias between the approximate certainties of the *straightforward* rewritten program and the fixpoint is less than the desired precision. "Analyzing the error range between computed certainties in a MS rewritten program and the fixpoint" seems to be an interesting direction for the future research.

Bibliography

[AG91] Serge Abiteboul and Stéphane Grumbach. A rule-based language with functions and sets. *ACM Trans. Database Syst.*, 16(1):1–30, 1991.

[BMSU86] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *PODS '86: Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 1–15, New York, NY, USA, 1986.

[BR86] François Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *SIGMOD Conference*, pages 16–52, 1986.

[BR87] C. Beeri and R. Ramakrishnan. On the power of magic. In *PODS '87: Proceedings of the sixth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 269–284, New York, NY, USA, 1987.

[Bra96] Stefan Brass. SLDMagic — an improved magic set technique. In *Proceedings of the Third International Workshop on Advances in Databases and Information Systems - ADBIS'96*, pages 75–83, Moscow, 1996.

[CCCR+90] Filippo Cacace, Stefano Ceri, Stefano Crespi-Reghizzi, Letizia Tanca, and Roberto Zicari. Integrating object-oriented data modeling with a rule-based programming paradigm. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 225–236, Atlantic City, NJ, USA, 1990.

[CGK+90] Danette Chimenti, Ruben Gamboa, Ravi Krishnamurthy, Shamim A. Naqvi, Shalom Tsur, and Carlo Zaniolo. The LDL system prototype. *IEEE Trans. Knowl. Data Eng.*, 2(1):76–90, 1990.

[CGT89] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

[DLP91] Didier Dubois, Jérôme Lang, and Henri Prade. Towards possibilistic logic programming. In *ICLP*, pages 581–595, 1991.

[DMP93] Marcia A. Derr, Shinichi Morishita, and Geoffrey Phipps. Design and implementation of the Glue-NAIL database system. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 147–156, Washington D.C., USA, 1993.

[ea97] Carlo Zaniolo et al. *Advanced database systems.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[ea05] Serge Abiteboul et al. The lowell database research self-assessment. *Commun. ACM*, 48(5):111–118, 2005.

[FGL07] Wolfgang Faber, Gianluigi Greco, and Nicola Leone. Magic sets and their application to data integration. *J. Comput. Syst. Sci.*, 73(4):584–609, 2007.

[Fit91] Melvin Fitting. Bilattices and the semantics of logic programming. *Journal of Logic Programming*, 11(1&2):91–116, 1991.

[FSS92] Burkhard Freitag, Heribert Schütz, and Günther Specht. Lola - a logic language for deductive databases and its implementation. In *Proceedings of the Second International Symposium on Database Systems for Advanced Applications*, pages 216–225. World Scientific Press, 1992.

[Hav97] Paul Havlak. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst.*, 19(4):557–567, 1997.

[HN84] Lawrence J. Henschen and Shamim A. Naqvi. On compiling queries in recursive first-order databases. *J. ACM*, 31(1):47–85, 1984.

[JS94] Manfred A. Jeusfeld and Martin Staudt. Query optimization in deductive object bases. In *Query Processing for Advanced Database Systems*, pages 145–176. 1994.

[JZ02] Karel Jezek and Martin Zíma. Magic sets method with fuzzy logic. In *ADVIS*, pages 83–92, 2002.

[KL88] M. Kifer and A. Li. On the semantics of rule-based expert systems with uncertainty. In *ICDT Conference*, pages 102–117, Bruges, Belgium, 1988.

[KNSSS90] Juhani Kuittinen, Otto Nurmi, Seppo Sippu, and Eljas Soisalon-Soininen. Efficient implementation of loops in bottom-up evaluation of logic queries. In

VLDB '90: Proceedings of the 16th International Conference on Very Large Data Bases, pages 372–379, 1990.

[KS92] Michael Kifer and V. S. Subrahmanian. Theory of generalized annotated logic programming and its applications. *J. Log. Program.*, 12(4):335–367, 1992.

[Lif85] Vladimir Lifschitz. Closed-world databases and circumscription. *Artif. Intell.*, 27(2):229–235, 1985.

[LS94] Laks V. S. Lakshmanan and Fereidoon Sadri. Probabilistic deductive databases. In *Symposium on Logic Programming*, pages 254–268, 1994.

[LS96] Laks V. S. Lakshmanan and Nematollaah Shiri. A parametric approach to deductive databases with uncertainty. In *Logic in Databases*, pages 61–81, 1996.

[Lus92] Ewing L. Lusk. Controlling redundancy in large search spaces: Argonne-style theorem proving through the years. In *LPAR*, pages 96–106, London, UK, 1992.

[MFPR96] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. Magic conditions. *ACM Trans. Database Syst.*, 21(1):107–155, 1996.

[MP94] Inderpal Singh Mumick and Hamid Pirahesh. Implementation of magic-sets in a relational database system. In *SIGMOD Conference*, pages 103–114, 1994.

[NS92] Raymond T. Ng and V. S. Subrahmanian. Probabilistic logic programming. *Information and Computation*, 101(2):150–201, 1992.

[NS93] Raymond T. Ng and V. S. Subrahmanian. A semantical framework for supporting subjective and conditional probabilities in deductive databases. *J. Autom. Reasoning*, 10(2):191–235, 1993.

[Ros94] Kenneth A. Ross. Modular stratification and magic sets for datalog programs with negation. *J. ACM*, 41(6):1216–1266, 1994.

[RRSS94] Raghu Ramakrishnan, Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. Efficient incremental evaluation of queries with aggregation. In *Symposium on Logic Programming*, pages 204–218, 1994.

[RSS92] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. CORAL - control, relations and logic. In *18th International Conference on Very Large Data Bases*, pages 238–250, Vancouver, Canada, 1992.

[RSS94] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Rule ordering in bottom-up fixpoint evaluation of logic programs. *IEEE Trans. on Knowl. and Data Eng.*, 6(4):501–517, 1994.

[Shi05] Nematollaah Shiri. Expressive power of logic frameworks with certainty constraints. In *18th Int'l FLAIRS Conference, Special Track on Uncertainty Reasoning*, pages 759–765, Florida, USA, 2005.

[SSW94] Konstantinos F. Sagonas, Terrance Swift, and David Scott Warren. XSB as an efficient deductive database engine. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 442–453, Minneapolis, Minnesota, USA, 1994. ACM Press.

[Str05] Umberto Straccia. Uncertainty management in logic programming: Simple and effective top-down query answering. In *KES (2)*, pages 753–760, 2005.

[Sub94] V. S. Subrahmanian. Amalgamating knowledge bases. *ACM Trans. Database Syst.*, 19(2):291–331, 1994.

[Sun92] Sundararajarao Sundarshan. *Optimizing bottom-up query evaluation for deductive databases.* PhD thesis, University of Wisconsin, Madison, WI, USA, 1992.

[SZ86] Domenico Saccà and Carlo Zaniolo. Differential fixpoint methods and stratification of logic programs. In *JCDKB*, pages 49–58, 1986.

[SZ04] Nematollaah Shiri and Zhi Hong Zheng. Challenges in fixpoint computation with multisets. In *In Proc. 3rd Int'l Symp. Foundations of Information and Knowledge Systems (FoIKS)*, pages 273–290, Vienna, Austria, 2004.

[SZ08] Nematollaah Shiri and Zhi Hong Zheng. Optimizing fixpoint evaluation of logic programs with uncertainty. In *Proc. 13 CSI Int'l Comp. Conf. (CSICC)*, Kish, Iran, March 9-11, 2008.

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

[vE86] Maarten H. van Emden. Quantitative deduction and its fixpoint theory. *Journal of Logic Programming*, 3(1):37–53, 1986.

[Vie86] Laurent Vieille. Recursive axioms in deductive databases: The query/subquery approach. In *Expert Database Conf.*, pages 253–267, 1986.

[VRK$^+$94] Jayen Vaghani, Kotagiri Ramamohanarao, David B. Kemp, Zoltan Somogyi, Peter J. Stuckey, Tim S. Leask, and James Harland. The aditi deductive database system. *VLDB Journal*, 3(2):245–288, 1994.

[Zor04] Vladimir A. Zorich. *Mathematical Analysis I*. Springer, 2004.