Argument Reduction by Factoring

J. F. Naughton* R. Ramakrishnan† Y. Sagiv‡ J. D. Ullman§

Abstract

We identify a useful property of a program with respect to a predicate, called *factoring*. While we prove that detecting factorability is undecidable in general, we show that for a large class of programs, the program obtained by applying the Magic Sets transformation is factorable with respect to the recursive predicate. When the factoring property holds, a simple optimization of the program generated by the Magic Sets transformation results in a new program that is never less efficient than the Magic Sets program and is often dramatically more efficient, due to the reduction of the arity of the recursive predicate. We show that the concept of factoring generalizes some previously identified special cases of recursions, including separable recursions and right- and left-linear recursions, and that the specialized evaluation algorithms and rewriting strategies developed for those classes can be derived automatically by applying the Magic Sets transformation and then factoring the result.

*Department of Computer Science, Princeton University. Work supported by DARPA and ONR contracts NOOO14-85-C-0456 and N00014-85-K-0465, and by NSF Cooperative Agreement DCR-8420948

†Computer Sciences Department, University of Wisconsin, Madison, Wisconsin. Work supported in part by an IBM Faculty Development award and NSF grant IRI-8804319. Part of this work was done while visiting IBM Almaden Research Center.

‡Dept. of Computer Science, Hebrew University, Jerusalem, Israel. Work supported in part by grant 2545-2-87 from the Israeli National Council for Research and Development (ILNCRD).

§Stanford University. Work supported by NSF grant IRI-87-22886 and Air Force grant AFOSR-88-0266 and a grant of the IBM Corp.

Proceedings of the Fifteenth International Conference on Very Large Data Bases

Amsterdam, 1989

1 Introduction

The Magic Sets transformation [BMSU86, BR87] is a rule rewriting technique that, given a query and a recursive program, produces a new program such that the semi-naive bottom-up evaluation of the new program constructs the answer to the query more efficiently than the original recursion. Magic Sets achieves its power by restricting the search of the underlying database to the portion of the database that is relevant to the query.

The Magic Sets transformation is conceptually simple and the potential savings gained by ignoring the irrelevant tuples in the database is large. However, for some important recursions much better algorithms are known. Intuitively, this is because Magic Sets does not reduce the arity (number of columns) of the recursive predicate. Since the size of the relation computed is bounded by n^k, where n is the number of distinct constants in the database and k is the arity of the recursive predicate, reducing the arity (k) can result in an order of magnitude increase in the efficiency of the algorithm.

In this paper we identify a useful property of a program with respect to a predicate, called *factoring*. If a program can be factored nontrivially with respect to a query, then the program can be rewritten to reduce the arity of the recursive predicate. Few programs and queries have the factoring property as written; however, in many important cases the Magic Sets transformation produces programs that do have the factoring property. While we prove that in general detecting factorable recursions is undecidable, we describe classes of recursions for which the Magic Sets transformation always produces a factorable recursion.

Recently the separable recursions [Nau88] and the left- and right-linear recursions [NRSU89] have been identified as significant classes of recursion for which there are arity reducing evaluation algorithms. In this work we show that these classes of recursions are proper subsets of the class of recursions for which Magic Sets produces a factorable recursion. Furthermore, the special purpose evaluation algorithms of [Nau88] and the special purpose rewriting techniques of [NRSU89] can be derived automatically by simple optimizations applied to the factored Magic program.

We introduce the notion of factoring in Section 2, and show that in general it is undecidable. We describe classes of programs for which the corresponding "Magic" programs are factorable in Section 3. In Section 4, we summarize some simple optimizations that can be used in conjunction with factoring to refine a program. We discuss the connections between our approach, that is, Magic Sets followed by factoring, and the Counting transformation and the Separable, One-sided, and Right- and Left-linear classes of programs in Section 5. In Section 6, we present conclusions and directions for future work.

Example 1.1 As an example of the power of our approach, consider a definition of transitive closure including all three forms of the recursive rule, shown in Figure 1. We obtain the following program by first applying the Magic Sets transformation and then factoring:

$$\begin{aligned} m\_t^{bf}(W) &:- ft(W). \\ m\_t^{bf}(5). & \\ ft(Y) &:- m\_t^{bf}(X),\ e(X,Y). \\ query(Y) &:- ft(Y). \end{aligned}$$

(This example is presented in detail in Section 3.) □

$$\begin{aligned} t(X,Y) &:- t(X,W), t(W,Y). \\ t(X,Y) &:- e(X,W), t(W,Y). \\ t(X,Y) &:- t(X,W), e(W,Y). \\ t(X,Y) &:- e(X,Y). \\ query(Y) &:- t(5,Y). \end{aligned}$$

Figure 1: The three rule transitive closure.

2 The Factoring Property

Consider a program P, a query Q, and a predicate p that appears in P. Let P' be the program obtained by adding the following rules to P:

$$\begin{aligned} p_1(X_{i_1}, \ldots, X_{i_k}) &:- p(X_1, \ldots, X_n). \\ p_2(X_{j_1}, \ldots, X_{j_l}) &:- p(X_1, \ldots, X_n). \\ p(X_1, \ldots, X_n) &:- p_1(X_{i_1}, \ldots, X_{i_k}), \\ & \quad p_2(X_{j_1}, \ldots, X_{j_l}). \end{aligned}$$

where the X_i's are distinct variables. Here, $X_{i_1}, \ldots, X_{i_k}$ and $X_{j_1}, \ldots, X_{j_l}$ represent subsets of X_1 through X_n. We say that *(P, Q, p)* has the *factoring property* if P and P' compute the same answers to Q for all *EDB*s. More precisely, we say that $p(X_1, \ldots, X_n)$ can be factored into $p_1(X_i, \ldots, X_j)$ and $p_2(X_k, \ldots, X_l)$ in P with respect to Q. This holds trivially if either p_1 or p_2 contains all arguments of p. We say that p can be *non-trivially factored* if neither p_1 nor p_2 contains all arguments of p, and henceforth, we shall consider only non-trivial factoring of programs.

Note that factoring is defined for general logic programs, not just Datalog. The following theorem shows that factorability is undecidable even for Datalog programs.

Theorem 2.1 *It is undecidable whether a predicate in a given program is non-trivially factorable with respect to a given query.*

The proof of Theorem 2.1 is by reduction from the containment problem for Datalog queries, and assumes multiple IDB predicates. To our knowledge, the decidability of factoring for single IDB predicate programs is open.

We have the following simple observation, which suggests an equivalent definition of factoring.

Proposition 2.1 *Let P' be obtained from a given program P by the following transformation with respect to predicate p:*

- *Every body literal $p(t_1, \ldots, t_n)$ is replaced by the literals $p_1(t_{i_1}, \ldots, t_{i_k})$ and $p_2(t_{j_1}, \ldots, t_{j_l})$.*
- *Every rule with head $p(t_1, \ldots, t_n)$ is replaced by two rules with the same body, and with heads $p_1(t_{i_1}, \ldots, t_{i_k})$ and $p_2(t_{j_1}, \ldots, t_{j_l})$.*

P and P' compute the same answers to Q for all EDBs if and only if $p(X_1, \ldots, X_n)$, where the Xs are distinct variables, can be factored into $p_1(X_{i_1}, \ldots, X_{i_k})$ and $p_2(X_{j_1}, \ldots, X_{j_l})$ in P with respect to a query Q.

We refer to the transformation described in the above proposition as the *factoring transformation.* Note that applying this transformation results in a program that does not contain p, which is replaced by two predicates, p_1 and p_2, of strictly lower arity.

3 Classes of Efficiently Evaluable Programs

The Magic Sets transformation [BMSU86, BR87] rewrites a program with the objective of restricting the computation by propagating bindings in the query. We identify classes of programs for which the program produced by applying the Magic Sets transformation can be factored with respect to the recursive predicate.

3.1 Preliminary Definitions

We begin by introducing some terminology and conventions. We only consider programs in which there is a single (recursive) IDB predicate, say p, and there is a single reachable adornment, say p^α. We refer to such programs as *unit* programs.

A rule is said to be in *standard form* if every argument of p^α, in the head or the body, is a variable, and no variable appears in two arguments of the same p^α-literal. We require all rules to be in standard form, and we allow the use of a special predicates to ensure that this requirement does not entail a loss of generality. Thus, a literal $p^\alpha(X, X, 5, Y)$ could be replaced by $p^\alpha(X, U, V, Y), equal(V, 5), equal(X, U)$, while a literal $p^\alpha(X.Y, Z)$ must be replaced by the conjunct $p^\alpha(U, Z)$, $list(X, Y, U)$. Conceptually, *list* and *equal* are infinite EDB relations. Once this translation to standard form is done, the results in this paper can be used to test for factorability

We use $p^\alpha(\overline{X}, \overline{Y})$ to denote a p^α-literal, where $\overline{X}$ is the vector of variables in the bound argument positions of a p^α-literal, and $\overline{Y}$ is the vector of variables in the free argument positions.

Consider a rule in the adorned program with head literal $p^\alpha(\overline{X}, \overline{Y})$. A *left-linear* occurrence of p^α is a body literal $p^\alpha(\overline{X}, \overline{U})$, and a *right-linear* occurrence of p^α is a body literal $p^\alpha(\overline{V}, \overline{Y})$.

The following definitions generalize those in [NRSU89].

Definition 3.1 A rule is *left-linear* if it is of the form

$$p^\alpha(\overline{X}, \overline{Y}) \text{ :- } left(\overline{X}), p_1^\alpha(\overline{X}, \overline{U}_1), p_2^\alpha(\overline{X}, \overline{U}_2), \ldots, p_n^\alpha(\overline{X}, \overline{U}_n), last(\overline{U}_1, \overline{U}_2, \ldots, \overline{U}_n, \overline{Y}).$$

where

- The rule is in standard form.
- $left(\overline{X})$ and $last(\overline{U}_1, \ldots, \overline{U}_n, \overline{Y})$ are disjoint conjunctions of EDB predicates.

Definition 3.2 A rule is *right-linear* if it is of the form

$$p^\alpha(\overline{X}, \overline{Y}) \text{ :- } first(\overline{X}, \overline{V}), p^\alpha(\overline{V}, \overline{Y}), right(\overline{Y}).$$

where

- The rule is in standard form.
- $first(\overline{X}, \overline{V})$ and $right(\overline{Y})$ are disjoint conjunctions of EDB predicates.

Definition 3.3 A rule is a *combined rule* if it is of the form

$$p^\alpha(\overline{X}, \overline{Y}) \text{ :- } left(\overline{X}), p_1^\alpha(\overline{X}, \overline{U}_1), p_2^\alpha(\overline{X}, \overline{U}_2), \ldots, p_n^\alpha(\overline{X}, \overline{U}_n), center(\overline{U}, \overline{V}), p^\alpha(\overline{V}, \overline{Y}), right(\overline{Y}).$$

where

- The rule is in standard form.
- $left(\overline{X})$, $center(\overline{U}, \overline{V})$, and $right(\overline{Y})$ are disjoint conjunctions of EDB predicates.

We remark that some of the conjunctions of EDB predicates referred to in the above definitions may contain occurrences of the special EDB predicate *equal*. As a special case, a conjunction may contain only such occurrences.

3.2 Factorable Programs

We present theorems that identify classes of programs for which the corresponding Magic programs are factorable. The importance of these theorems lies in the technique that they exemplify: a two-step approach to optimizing programs in which the programs are rewritten using the Magic Sets transformation and subsequently factored if possible.

Let P be a program, Q a query, and P^{ad} the adorned program corresponding to a left-to-right evaluation of the rules of P. P^{mg} represents the program obtained by applying the Magic Sets transformation to P and Q.

Example 3.1 The rewriting algorithms presented in [NRSU89] were the first to derive automatically unary programs for single-selection queries for all three forms (left-linear, right-linear, non-linear) of the transitive closure. We achieve the same result here by first applying the Magic Sets transformation and

$$
\begin{aligned}
m\_t^{bf}(W) &:- m\_t^{bf}(X), t^{bf}(X,W).\\
m\_t^{bf}(W) &:- m\_t^{bf}(X), e(X,W).\\
m\_t^{bf}(5).&\\
t^{bf}(X,Y) &:- m\_t^{bf}(X), t^{bf}(X,W), t^{bf}(W,Y).\\
t^{bf}(X,Y) &:- m\_t^{bf}(X), e(X,W), t^{bf}(W,Y).\\
t^{bf}(X,Y) &:- m\_t^{bf}(X), t^{bf}(X,W), e(W,Y).\\
t^{bf}(X,Y) &:- m\_t^{bf}(X), e(X,Y).\\
query(Y) &:- t^{bf}(5,Y).
\end{aligned}
$$

Figure 2: P^{mg} for the three-rule transitive closure.

then factoring the rewritten program. To illustrate the technique, we again consider the single program that includes all three forms of the recursive rule presented in Figure 1. The Magic Sets algorithm rewrites this program to produce the program in Figure 2.

If we identify m_t^{bf} tuples with goals in a top-down evaluation, we see that only the last occurrence of t^{bf} in a rule body generates new goals, and further, the answer to a new goal is also an answer to the goal that invoked the rule. In fact, every answer to a subgoal is also an answer to the query goal m_t^{bf}. Also, if c is generated as an answer to a subgoal, then a new subgoal $m\_t^{bf}(c)$ is also generated. These observations imply that it does not matter which subgoal an answer corresponds to; its role in the computation is the same in any case. That is, $t^{bf}(X,Y)$ can be factored into $bt(X)$ and $ft(Y)$ in the Magic program. This yields the program shown in Figure 3.

Applying further optimizations, discussed in Section 4, we finally obtain the following unary program:

$$
\begin{aligned}
m\_t^{bf}(W) &:- ft(W).\\
m\_t^{bf}(5).&\\
ft(Y) &:- m\_t^{bf}(X), e(X,Y).\\
query(Y) &:- ft(Y).
\end{aligned}
$$

□

Definition 3.4 Let p be the only IDB predicate in a program P, and Q be a query on p. Then the combination of P and Q is an *RLC-stable program* if P consists only of right-linear, left-linear, and combined-linear rules plus one exit rule, and p^{α} is the only adorned version of p in P^{ad}.

We now define some auxiliary conjunctive queries that appear often later in this section.

$$
\begin{aligned}
m\_t^{bf}(W) &:- m\_t^{bf}(X), bt(X), ft(W).\\
m\_t^{bf}(W) &:- m\_t^{bf}(X), e(X,W).\\
m\_t^{bf}(5).&\\
bt(X) &:- m\_t^{bf}(X), bt(X), ft(W),\\
&\quad bt(W), ft(Y).\\
bt(X) &:- m\_t^{bf}(X), e(X,W),\\
&\quad bt(W), ft(Y).\\
bt(X) &:- m\_t^{bf}(X), bt(X),\\
&\quad ft(W), e(W,Y).\\
bt(X) &:- m\_t^{bf}(X), e(X,Y).\\
ft(Y) &:- m\_t^{bf}(X), bt(X), ft(W),\\
&\quad bt(W), ft(Y).\\
ft(Y) &:- m\_t^{bf}(X), e(X,W),\\
&\quad bt(W), ft(Y).\\
ft(Y) &:- m\_t^{bf}(X), bt(X),\\
&\quad ft(W), e(W,Y).\\
ft(Y) &:- m\_t^{bf}(X), e(X,Y).\\
query(Y) &:- bt(5), ft(Y).
\end{aligned}
$$

Figure 3: The factored version of P^{mg}.

Definition 3.5 The conjunctive query *free_exit* is defined as follows:

$$free\_exit(\overline{Y}) \;:-\; exit(\overline{X},\overline{Y}).$$

where $exit(\overline{X},\overline{Y})$ is the body of the exit rule.

The conjunctive query *bound_first* is defined for a given right-linear rule:

$$bound\_first(\overline{X}) \;:-\; first(\overline{X},\overline{U}).$$

where $first(\overline{X},\overline{U})$ appears in the body of the rule.

The conjunctive query *bound* is defined for a given left-linear or combined rule:

$$bound(\overline{X}) \;:-\; left(\overline{X}).$$

where $left(\overline{X})$ appears in the body of the rule.

The conjunctive query *bound* is defined for a given right-linear or combined rule:

$$free(\overline{Y}) \;:-\; right(\overline{Y}).$$

where $right(\overline{Y})$ appears in the body of the rule.

The conjunctive query *middle* is defined for a given combined rule:

$$middle(\overline{U},\overline{V}) \;:-\; center(\overline{U},\overline{V}).$$

where $center(\overline{U},\overline{V})$. appears in the body of the rule.

Often by a slight abuse of notation we will refer to *left*, *right*, and *center* as conjunctive queries instead of using *bound*, *free* and *middle*.

Our first theorem essentially generalizes of the results in [NRSU89], although it must be used in conjunction with the additional optimizations described in Section 4 in order to do so. It uses the following definition.

Definition 3.6 Let P, Q be an RLC-stable program with IDB predicate p. Then P, Q is *selection-pushing* if the following conditions hold:

- For any combined or right-linear rule r in P, the conjunctive query "*free_exit*" must be contained in the conjunctive query "*free*" for r.
- For any pair of rules r_1 and r_2 in P, if both r_1 and r_2 contain a "*left*" conjunctive query, these must be equivalent. If one contains a "*left*" query, and the other a "*first*" query, the conjunctive query "*bound_first*" must be contained in the conjunctive query "*bound*".

Theorem 3.1 *Let P, Q be an RLC-stable program with IDB predicate p, and let $\overline{X}$ be the vector of variables appearing in bound arguments of p^α in the heads of the rules of P^{ad}, and let $\overline{Y}$ be the vector of variables appearing in free arguments of p^α in P^{ad}. If P, Q is selection-pushing then $p^\alpha(\overline{X}, \overline{Y})$ can be factored into $bp(\overline{X})$ and $fp(\overline{Y})$ in P^{mg} with respect to the query Q.*

Example 3.2 We illustrate the intuition behind selection-pushing and show that violating any of the associated conditions could destroy this property.

$$
\begin{aligned}
p(X,Y) &:- l1(X), p(X,U), c1(U,V), \\
&\quad\; p(V,Y), r1(Y). \\
p(X,Y) &:- l2(X), p(X,U), c2(U,V), \\
&\quad\; p(V,Y), r2(Y). \\
p(X,Y) &:- f(X,V), p(V,Y), r3(Y). \\
p(X,Y) &:- e(X,Y). \\
query(Y) &:- p(5,Y).
\end{aligned}
$$

The Magic Sets algorithm rewrites this to

$$
\begin{aligned}
m\_p^{bf}(V) &:- m\_p^{bf}(X), l1(X), \\
&\quad\; p^{bf}(X,U), c1(U,V). \\
m\_p^{bf}(V) &:- m\_p^{bf}(X), l2(X), \\
&\quad\; p^{bf}(X,U), c2(U,V). \\
m\_p^{bf}(V) &:- m\_p^{bf}(X), f(X,V). \\
m\_p^{bf}(5). & \\
p^{bf}(X,Y) &:- m\_p^{bf}(X), l1(X), p^{bf}(X,U), \\
&\quad\; c1(U,V), p^{bf}(V,Y), r1(Y). \\
p^{bf}(X,Y) &:- m\_p^{bf}(X), l2(X), p^{bf}(X,U), \\
&\quad\; c2(U,V), p^{bf}(V,Y), r2(Y). \\
p^{bf}(X,Y) &:- m\_p^{bf}(X), f(X,V), \\
&\quad\; p^{bf}(V,Y), r3(Y). \\
p^{bf}(X,Y) &:- m\_p^{bf}(X), e(X,Y). \\
query(Y) &:- p^{bf}(5,Y).
\end{aligned}
$$

Factoring this program and applying further transformations described in detail in Section 4 yields

$$
\begin{aligned}
m\_p^{bf}(V) &:- bp(X), l1(X), \\
&\quad\; fp(U), c1(U,V). \\
m\_p^{bf}(V) &:- bp(X), l2(X), \\
&\quad\; fp(U), c2(U,V). \\
m\_p^{bf}(V) &:- m\_p^{bf}(X), f(X,V). \\
m\_p^{bf}(5). & \\
bp(X) &:- m\_p^{bf}(X), f(X,V), \\
&\quad\; bp(V), fp(Y), r3(Y). \\
bp(X) &:- m\_p^{bf}(X), e(X,Y). \\
fp(Y) &:- m\_p^{bf}(X), e(X,Y). \\
query(Y) &:- fp(Y).
\end{aligned}
$$

The transformations that produce the above program from the factored version of the Magic program preserve equivalence. We have applied these transformations in order to delete some unnecessary literals and rules in the factored program, thus making it easier to understand the essential ideas.

Consider the following EDB instance: $f(5,1)$, $e(5,6)$, $e(1,7)$, $e(2,8)$, $l1(1)$, $c1(6,2)$, $r1(7)$, $r1(8)$. Because the condition that *bound_first* should be a subset of $l1$ is violated by this EDB, 8 is incorrectly derived as an answer. Indeed, $m\_p^{bf}(1)$ is generated using $m\_p^{bf}(5)$ and $f(5,1)$. This generates $bp(1)$ using $e(1,7)$. Also, the tuple $e(5,6)$ gives us $fp(6)$. The critical step follows: the fact $fp(6)$ is used in the first rule with $bp(1)$, $l1(1)$ and $c1(6,2)$ to generate the fact $m\_p^{bf}(2)$. That is, the fact $fp(6)$, which is an answer to the goal $m\_p^{bf}(5)$, is incorrectly used where an answer to the goal $m\_p^{bf}(1)$ is required, thereby generating a spurious subgoal. One can verify that 8 is a valid answer if $l1(5)$ is added to the EDB. A similar example can be constructed if $l1$ and $l2$ are not identical, since the answer generated in response to a subgoal that satisfies $l1$ but not $l2$ can be used in the second rule to generate spurious subgoals.

Now consider the EDB instance: $f(5,1)$, $e(5,6)$, $e(1,7)$, $l1(5)$, $c1(6,1)$. The fact $fp(7)$ is incorrectly generated. The first rule is used to generate $m\_p^{bf}(1)$ from the query goal and the fact $e(5,6)$. The fact

$fp(7)$ is generated in response to this subgoal, but it cannot be an answer to $m\_p^{bf}(5)$ unless $r1(7)$ is true. The EDB instance violates the condition that *free_exit* should be contained in $r1$. This made it possible to generate subgoals whose answers are not answers to the original goal. □

Intuitively, we are separating the bound arguments from the free arguments, and we must ensure that every answer to a subquery (keeping in mind a top-down evaluation of the program) is also an answer to the original query. (We refer to the vector of values in the free arguments as the answer, corresponding to a query that is the vector of values in the bound arguments of a p^α-fact.) For this, we require that the *right* conjunctive queries be satisfied by every potential answer tuple, that is, *free_exit* is contained in every *right* conjunctive query. (Some answer tuples may be generated from left-linear rules, but these need not satisfy the *right* queries since there is a derivation of these answers to the original query that does not propagate these answers through right-linear occurrences of p^α.)

In addition, we must ensure that no spurious answers are generated. The main idea is that for every derivation of a fact using P^{mg}, there is an equivalent derivation in which the bound arguments of every left-linear p^α fact is identical to the bound arguments in the query. That is, in every recursive rule that contains a left-linear occurrence of p^α, we can replace the variables $X_1, \ldots, X_m$ in the bound arguments by the constants provided in the original query. This is in fact the motivation for the term "selection-pushing."

When a right-linear rule is applied to generate new subqueries, the answers to these subqueries could be used in left-linear occurrences of p^α. To justify this, we must ensure that a subquery invoking the right-linear rule is reachable from a subquery that satisfies the conditions on the bound arguments of the left-linear occurrences of p^α. Since every subquery is reachable from the initial goal, this is guaranteed if the initial query satisfies the (unique, for the given program) *left* conjunctive query. If the initial goal does not satisfy the *left* conjunctive query, then we cannot apply the right-linear rule, and the condition that the *bound_first* conjunctive queries should be contained in the *left* conjunctive query ensures this.

We can identify further classes of programs that can be factored.

Definition 3.7 Let P, Q be an RLC-stable program containing only combined recursive rules. Then P, Q is *symmetric* if the following conditions hold:

- Each rule contains exactly two occurrences of p^α in the body, and the "*middle*" conjunctive queries are all equivalent.
- For any recursive rule r in P, the conjunctive query "*free_exit*" must be contained in the conjunctive query "*right*" for r.

Theorem 3.2 *Let P, Q be an RLC-stable program with IDB predicate p, and let $\overline{X}$ be the vector of variables appearing in bound arguments of p^α in the heads of the rules of P^{ad}, and let $\overline{Y}$ be the vector of variables appearing in free arguments of p^α in P^{ad}. If P, Q is symmetric, then $p^\alpha(\overline{X}, \overline{Y})$ can be factored $bp(\overline{X})$ and $fp(\overline{Y})$ in P^{mg} with respect to the query Q.*

The Magic program for the following query can thus be factored:

$$
\begin{aligned}
p(X,Y) &:- l1(X), p(X,U), c(U,V), \\
&\quad\ p(V,Y), r1(Y). \\
p(X,Y) &:- l2(X), p(X,U), c(U,V), \\
&\quad\ p(V,Y), r2(Y). \\
p(X,Y) &:- e(X,Y). \\
query(Y) &:- p(5,Y).
\end{aligned}
$$

In summary, the results in this section are illustrative of a general approach to optimizing programs, in which we first apply the Magic Sets transformation and then factor. When we factor a Magic program and separate the bound and free arguments, we must establish two things:

- Every answer to a subquery is also an answer to the original query.
- No spurious subqueries or answers are generated.

Because testing for these classes of recursions in general requires testing for containment of conjunctive queries, and testing for conjunctive query containment is NP-complete [CM77, ASU79], testing for membership in these classes is also NP-complete. It is important that the measure of size here is the size of the recursion and query, not the database. An algorithm that is exponential in the size of the recursion and query (small) may be worth running during query planning in order to save time proportional to the size of the database (large) during query evaluation. Furthermore, in many cases, the conjunctive queries will be empty, in which case polynomial time algorithms for testing if a recursion satisfies Theorems 3.1 and 3.2 recursions exist.

4 Some Additional Optimizations

We use the following definitions.

Definition 4.1 A bound argument position of p^α is a *static argument position* if for every p^α-literal in the body of a rule, the variable in this argument position also appears in the same argument position in the head of the rule. (Recall that the head must also be a p^α literal, since we only consider unit programs.)

Definition 4.2 Let (P, Q) be a unit program — query pair, and let the ith argument of p^α be a static argument. Without loss of generality, let the variable in the ith argument of P^α always be X, and let the constant in the ith argument of the query Q be c. The program P is *reduced with respect to argument position i* as follows:

- Every rule r is replaced by $\sigma(r)$, where σ is the substitution $X \leftarrow c$.
- Every p^α-literal — in the head or the body of a rule — is replaced by a s^θ-literal with the same vector of arguments except for the ith argument, which is deleted. s is a new predicate with one fewer argument position, and θ is identical to the adornment α, but with the b corresponding to the ith argument deleted.

We begin with a result that augments the theorems presented in the previous section. Some programs that do not satisfy the conditions of these theorems can be transformed into programs that do by applying the following lemma.

Lemma 4.1 *Let (P, Q) be a unit program — query pair, let the ith argument of P^α be a static argument, and let P' be the reduced program. Then P and P' are equivalent with respect to Q.*

In the rest of this section, we summarize a few simple optimizations that are often applicable to factored programs.

If p^α is factored into bp and fp in P^{mg}, then the relation bp is contained in $magic\_p^\alpha$, since every rule defining bp contains $magic\_p^\alpha$ (with identical arguments) in the body. Further, for every rule defining fp (resp. bp) there is a rule with an identical body describing bp (resp. fp). Therefore, the goal $bp(\_)$, where _ denotes an "anonymous" variable, succeeds if any fp goal succeeds, and vice-versa. These observations lead to the following propositions.

Proposition 4.1 *If a rule contains both bp and $magic\_p^\alpha$ in the body, with identical arguments, then we may delete the $magic\_p^\alpha$ literal.*

Proposition 4.2 *If a rule contains the literal $bp(\_)$ and also an fp literal, the literal $bp(\_)$ can be deleted.*

A symmetric proposition allows us to delete some $fp(\_)$ literals.

A similar observation is that if $m\_p^\alpha(\bar{c})$ is the original query goal, then $bp(\bar{c})$ is true if any fp goal succeeds. This is because every fp fact, in particular the successful fp goal, is an answer to the original query. However, note that in general, p^α may be factored but the original query may not be on predicate p^α.

Proposition 4.3 *Let the original query correspond to the fact $m\_p^\alpha(\bar{c})$. If a rule contains the literal $bp(\bar{c})$ and also an fp literal, then the literal $bp(\bar{c})$ can be deleted.*

Some additional simple observations that are useful are mentioned below.

Proposition 4.4 *We may delete a rule if the head literal also appears in the body, or if the head predicate is not reachable from the query predicate.*

This is a special case of deletion under uniform equivalence [Sag87].

Proposition 4.5 *We may introduce an "anonymous" variable in an argument position if the variable in it appears nowhere else in the rule.*

As shown in [RBK88], the preceding proposition can be strengthened to prove that an anonymous variable can be introduced in any existential argument position.

Example 4.1 Consider again the factored version of P^{mg} from the three-rule transitive closure (Figure 3.) We can delete the first and the third rules defining bt and the first two rules defining ft because the head literal also appears in the body. We can also delete the literal $m\_t^{bf}(X)$ from every rule that also contains $bt(X)$, and then replace all variables that only appear once in a rule by anonymous variables. This yields:

$$
\begin{aligned}
m\_t^{bf}(W) &:- bt(\_), ft(W).\\
m\_t^{bf}(W) &:- m\_t^{bf}(X), e(X, W).\\
m\_t^{bf}(5).&\\
bt(X) &:- m\_t^{bf}(X), e(X, W),\\
&\quad\ bt(W), ft(Y).\\
bt(X) &:- m\_t^{bf}(X), e(X, Y).\\
ft(Y) &:- bt(\_), ft(W), e(W, Y).\\
ft(Y) &:- m\_t^{bf}(X), e(X, Y).\\
query(Y) &:- bt(5), ft(Y).
\end{aligned}
$$

We can delete both body occurrences of $bt(\_)$ since the rules in which they appear also contain ft literals in the body. Similarly, we can delete the literal $bt(5)$ from the rule defining the query. This makes bt unreachable from the query, and we can delete all remaining rules for bt. This gives us:

$$\begin{aligned} m\_t^{bf}(W) &:- ft(W). \\ m\_t^{bf}(W) &:- m\_t^{bf}(X), e(X,W). \\ m\_t^{bf}(5). & \\ ft(Y) &:- ft(W), e(W,Y). \\ ft(Y) &:- m\_t^{bf}(X), e(X,Y). \\ query(Y) &:- ft(Y). \end{aligned}$$

The second rule defining m_t^{bf} and the first rule defining ft can be deleted under uniform equivalence, and we finally obtain the following program:

$$\begin{aligned} m\_t^{bf}(W) &:- ft(W). \\ m\_t^{bf}(5). & \\ ft(Y) &:- m\_t^{bf}(X), e(X,Y). \\ query(Y) &:- ft(Y). \end{aligned}$$

□

5 A Unifying Overview

We consider how the refinements of the Magic Sets transformation presented in this paper are related to some previously defined optimizations.

5.1 One-Sided Recursions

One-sided recursions were identified in [Nau87] as a class of recursions for which there are efficient evaluation algorithms. Here we restate the characterization of one-sided recursions.

Theorem 5.1 *(Theorem 3.1 from [Nau87]) Let D be a recursive definition with a single, linear recursive rule r. Then D is one-sided if and only if the full A/V graph for r has only one connected component with a cycle of nonzero weight, and that component has a cycle of weight 1.*

An important subset of the one-sided recursions are those such that the full A/V graph has one connected component with a cycle of nonzero weight, and that component contains exactly one cycle of nonzero weight, and that cycle is of weight 1. We call such a one-sided recursion a *simple one-sided recursion.* Any simple one-sided recursion can be "expanded" (by substituting the rule into itself some number of times) to produce a rule of the form

$$p(\overline{A},\overline{B}) \;:-\; p(\overline{A},\overline{C}), c(\overline{C},\overline{D},\overline{B}). \qquad (1)$$

where $\overline{A}$, $\overline{B}$, $\overline{C}$, and $\overline{D}$ are vectors of disjoint variables, and c is a conjunction of EDB predicates.

The preceding recursion is written in a form isomorphic to what we have called a left-linear recursive rule. However, the definition of left-linear is in terms of both the recursion and the specific query in question. By contrast, the one-sided recursions are defined independently of queries. Notice, however, that coupled with the query $p(\overline{c},Y)$?, the preceding rule is left-linear; while coupled with the query $p(X,\overline{d})$? it is right-linear.

A selection that binds either every variable in $\overline{A}$ or $\overline{B}$ is a *full-selection.* With this definition, we can formalize the preceding discussion with the following theorem.

Theorem 5.2 *Let P be a simple one-sided recursion, expanded so that it is of the form of Equation 1. Let Q be a full-selection query on p, the recursive predicate of P. Also, let P^{mg} be the output of the Magic Sets algorithm on P and Q. Then P^{mg} and Q factor with respect to p.*

5.2 Separable Programs

Separable programs, defined in [Nau88], were defined to be class of recursions for which selection queries have efficient evaluation algorithms. Essentially, [Nau88] gave conditions that determine if a given recursion is separable and a schema for evaluating selection queries over separable recursions. Given a specific selection query on a recursion that is separable, the schema can automatically be instantiated to produce an evaluation algorithm for the query.

As was the case with one-sided recursions, the variables appearing in the heads of rules in separable recursions can be divided into equivalence classes (see [Nau88] for details.) A selection query that binds every variable of some equivalence class is a *full-selection query*, as before.

Theorem 5.3 *Let P be a separable recursion, let Q be a full-selection query on p (the recursive predicate of P), and let P^{mg} be the result of the Magic Sets transformation applied to P,Q. Then the pair P^{mg},Q is factorable.*

The proof proceeds by showing that the conditions for separability given in [Nau88] guarantee that the pair P,Q will satisfy the conditions of Theorem 3.1.

To see that the converse is not true, that is, that there are factorable programs that cannot be viewed as full selections on separable recursions, note that separable recursions are all linear, whereas factorable programs need not be linear.

There is also a close connection between the instantiated separable recursion evaluation algorithm and the program resulting from Magic Sets followed by the factoring rewrite. Essentially, for a full selection on a separable recursion, the instantiated separable recursion evaluation schema represents the same computation as the semi-naive bottom-up evaluation of the output of the factoring rewrite applied to the Magic program.

5.3 Left- and Right-Linear Programs

In [NRSU89], recursions containing right-linear, left-linear, mixed-linear, and combined-linear recursions were identified and special rewriting algorithms in the spirit of the Magic Sets transformation were given. A simple check shows that the classes of programs defined in [NRSU89] are a proper subset of the programs satisfying the conditions of Theorem 3.1, and that Theorem 3.2 handles some additional programs. In addition, for the programs considered in that paper, the Magic Sets plus factoring transformation produces the same final program as the rewriting algorithms from that paper.

5.4 The Counting Transformation

The Counting transformation [BMSU86, BR87, SZ86] can be understood as a variant of the Magic Sets transformation. First, every derived predicate is augmented with some index fields, which, intuitively, encode the derivation of the fact. That is, the value of the index encodes the sequence of rule applications, and the literal that is expanded at each step, that was used to derive the fact. The program P^{mg} with these additional fields is then refined by deleting the fields corresponding to bound arguments in derived predicates.

When we describe Counting as reducing the arity of derived predicates, we ignore the new index fields that are introduced. The cost of computing the indices can be significant; in fact, this may make the Counting strategy more expensive than even Naive fixpoint evaluation, or cause non-termination.

There is an obvious parallel to factoring Magic programs, since the objective here is again to reduce the arity of derived predicates by separating the bound and free arguments. The connection is quite close — for the class of programs for which we have shown the Magic program to be factorable, the factored program (with some of the simple optimizations that we discussed in Section 4) is identical to the Counting program with all index fields deleted. In effect, this is a class of programs for which the benefits of the Counting strategy — reductions in predicate arity, and accompanying deletion of some literals and rules — can be obtained without the overhead of computing indices.

If a program contains left-linear or combined rules, the Counting program will not terminate since a rule is created that generates the same fact with an infinite number of values in the index fields. The following example is illustrative:

$$t^{bf}(X,Y) \;:-\; t^{bf}(X,Z), e(Z,Y).$$
$$t^{bf}(X,Y) \;:-\; e(Z,Y).$$

The first rule generates the Magic rule:

$$magic\_t^{bf}(X) \;:-\; magic\_t^{bf}(X).$$

With the indices added in the Counting transformation, this is:

$$cnt\_t^{bf}(X, I+1) \;:-\; cnt\_t^{bf}(X, I).$$

This is a rule whose fixpoint evaluation does not terminate, given an initial cnt_t^{bf} fact, which is obtained from the query.

Theorem 5.4 *If a program satisfies the conditions of the factoring theorems in Section 3, and no rule contains a left-linear literal, then the factored Magic program, after deleting trivially redundant rules, is identical to the Counting program with all index fields deleted.*

The factoring approach allows us to reduce arities of some programs with left-linear literals, whereas the Counting program would never terminate in such cases. On the other hand, the well-known same-generation program is the canonical example of a program that cannot be factored, and in which the index fields introduced in Counting are necessary.

6 Conclusion and Directions for Future Work

We have shown that the Magic Sets transformation followed by factoring produces programs on which bottom-up evaluation efficiently produces the answer to the query.

The results presented in this paper motivate several interesting problems.

- We have identified classes of programs for which the corresponding Magic program can be factored. However, there are other interesting programs that can also be factored. Identifying broader classes of factorable programs is an interesting research direction.
- We showed that for the classes of factorable Magic programs identified in this paper, the indices in Counting were unnecessary. Can we show that the Counting indices are unnecessary in factorable Magic programs, independently of the sufficient conditions that we use to ensure factorability?
- Not all one-sided recursions have arity-reducing evaluation algorithms, and not all one-sided recursions produce factorable Magic programs. Does Theorem 3.1 cover all one-sided recursions that have arity-reducing evaluation algorithms?
- Suppose the program for p^{α} is factorable, but this predicate is not the query predicate. How can we identify cases in which p^{α} can be factored even though it is not the top-level query?
- Consider the various techniques for deleting rules and literals in Section 4 (additional optimizations). Does the order in which these are applied to a program affect the final result? If so, can we identify classes of programs for which the final result is unique?

References

[ASU79] Alfred V. Aho, Yehoshua Sagiv, and Jeffrey D. Ullman. Equivalence of relational expressions. *SIAM Journal of Computing*, 8(2):218–246, 1979.

[BMSU86] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 1–15, Boston, Massachusetts, March 1986.

[BR87] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 269–283, San Diego, California, March 1987.

[CM77] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Conference Record of the Ninth Annual ACM Symposium on Theory of Computing*, pages 77–90, Boulder, Colorado, May 1977.

[Nau87] Jeffrey F. Naughton. One sided recursions. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 340–348, San Diego, California, March 1987.

[Nau88] Jeffrey F. Naughton. Compiling separable recursions. In *Proceedings of the SIGMOD International Symposium on Management of Data*, pages 312–319, Chicago, Illinois, May 1988.

[NRSU89] Jeffrey F. Naughton, Yehoshua Sagiv, Raghu Ramakrishnan, and Jeffrey D. Ullman. Efficient evaluation of right-, left-, and combined-linear rules. In *Proceedings of the SIGMOD International Symposium on Management of Data*, pages 235–242, Portland, Oregon, May 1989.

[Ram87] Raghu Ramakrishnan. *Magic Templates: A Spellbinding Approach to Logic Programs.* In *Proceedings of the International Conference on Logic Programming*, pages 140-159, Seattle, Washington, August 1988.

[RBK88] Raghu Ramakrishnan, Catriel Beeri, and Ravi Krishnamurthy. Optimizing existential datalog queries. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 89–102, Austin, Texas, March 1988.

[Sag87] Yehoshua Sagiv. Optimizing datalog programs. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 349–362, Austin, TX, March 1987.

[SZ86] Domenico Sacca and Carlo Zaniolo. The generalized counting methods for recursive logic queries. In *Proceedings of the First International Conference on Database Theory*, 1986.