ARGUMENT REDUCTION BY FACTORING

by

Jeffrey F. Naughton
Raghu Ramakrishnan
Y. Sagiv
J.D. Ullman

Computer Sciences Technical Report #1062

December 1991

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.

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.

---

*Some of the results in this paper were presented in a preliminary form in Proc. VLDB 89.

†Department of Computer Sciences, University of Wisconsin, Madison. Work supported by DARPA and ONR contracts N00014-85-C-0456 and N00014-85-K-0465, and by NSF grant IRI-8909795.

‡Computer Sciences Department, University of Wisconsin, Madison, Wisconsin. Work supported in part by an IBM Faculty Development award, a Presidential Young Investigator Award, a Packard Foundation Fellowship, 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.

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.

The separable recursions [Nau88a] and the left- and right-linear recursions [NRSU89] have also 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 [Nau88a] 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.

We conclude this introduction with two examples of the power of the factoring approach.

Example 1.1 Consider a definition of transitive closure including all three forms of the recursive rule.

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

We obtain the following program by first applying the Magic Sets transformation and then factoring:

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

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

The following example demonstrates that factoring is useful for programs with function symbols (not just for Datalog).

Example 1.2 Suppose we wish to compute the set of all members of a given list that satisfy some predicate p. We can do this by augmenting the standard Prolog member procedure

$$\begin{aligned} pmem(X,[X|T]) &\text{ :- } p(X). \\ pmem(X,[H|T]) &\text{ :- } pmem(X,T). \end{aligned}$$

and the following query

$$q(X) \text{ :- } pmem(X,[x_1,x_2,\ldots,x_n]).$$

where $[x_1,x_2,\ldots,x_n]$ is the given list. On this program and query, if all members of the given list satisfy the predicate p, Prolog will compute the $O(n^2)$ facts $pmem(x_i,[x_j,\ldots,x_n])$ for $1 \leq j \leq n$ and $j \leq i \leq n$.

By factoring, we get

$$\begin{aligned} m\_pmem([x_1,x_2,\ldots,x_n]). & \\ m\_pmem(T) &\text{ :- } m\_pmem([H|T]). \\ fpmem(X) &\text{ :- } m\_pmem([X|T]),\ p(X). \\ query(X) &\text{ :- } fpmem(X). \end{aligned}$$

Assuming a structure-sharing implementation of lists, each inference can be made in constant time (i.e. independently of the list size), and the factored program computes the answer in linear time. This example is worked out in detail in Example 4.6 □

2 Definitions

We consider Horn clause logic programs. We will assume the usual definitions of term, literal and rule [Llo87]. In the deductive database literature, a distinction is often drawn between a set of facts, called the *extensional database* or EDB, and the rest of the program, called the *intensional database* or IDB. [1] The motivation is that optimization strategies focus on the IDB, since the EDB can contain a large number of facts. We follow this convention, and by "program" we denote the IDB, unless otherwise noted.

The set of answers to a query, which is a partially instantiated literal, is the set of facts that unify with it in the least Herbrand model of IDB ∪ EDB. Let P be a program. Each IDB rule can be viewed as an operator that enables us to derive new (head) facts from known (body) facts. Indeed, the collection of program (IDB) rules can be viewed, by extension, as such an

[1] Although our results are applicable to programs containing function symbols, we work with programs in a restricted standard form, in which no function symbols or constants appear in program rules. This standard form is described further in Section 4.

operator, say T_P. It is well known that the least Herbrand model of a Horn clause logic program P is equal to the least fixpoint of the T_P operator that contains the EDB facts [vEK76]. We consider how the IDB can be transformed, say to IDB', such that the set of facts that unify with the query in the least model of IDB' $\cup$ EDB is identical to the set of answers, for all EDBs. (Of course, we expect that the use of IDB' will also lead to a more efficient computation of the set of answers.)

In this paper we use the notion of a derivation tree in several proofs.

Definition 2.1 *Consider a program P and an EDB D. Derivation trees constructed using $P \cup D$ (trees in $P \cup D$, for short) are defined recursively:*

1. *For each fact in D, there is a derivation tree consisting of a single node labeled with that fact.*
2. *Let there be an instance of a rule r of P such that p is the fact corresponding to the head and q_i, $1 \leq i \leq n$, are the facts corresponding to the (n) body literals. If there are derivation trees D_i, $1 \leq i \leq n$, such that the root of each tree is labeled with the corresponding q_i fact, then there is a derivation tree with D_i, $1 \leq i \leq n$, as subtrees of the root and with root label p. Each arc from the root to a child is labeled r.*
3. *Only trees defined by (1) and (2) are derivation trees.*

It is easily verified that there is a derivation tree with root label p (we also say "there is a derivation tree for p") if and only if the fact p is in the least fixpoint of $P \cup D$ (and thus in the least Herbrand model for P).

2.1 The Magic Sets Rewriting Algorithm

This is a program transformation that takes a program, say P and a query, and produces a new program, say P^{mg}. For all EDBs, the two programs compute the same answers. That is, if we consider the least fixpoint of $P \cup D$ and $P^{mg} \cup D$, the set of facts that unify with the query literal is identical.

The idea is to compute a set of auxiliary predicates that contain the goals. The rules in the program are then modified by attaching additional literals that act as filters and prevent the rule from generating irrelevant tuples. We assume familiarity with the Magic Sets algorithm, which we illustrate in Example 4.2. The reader is referred to [BMSU86, BR87, Ram88] for details.

3 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:

$$p_1(X_{i_1}, \ldots, X_{i_k}) \ :\!- \ p(X_1, \ldots, X_n).$$

$$p_2(X_{j_1},\ldots,X_{j_l}) \text{ :- } p(X_1,\ldots,X_n).$$
$$p(X_1,\ldots,X_n) \text{ :- } p_1(X_{i_1},\ldots,X_{i_k}), p_2(X_{j_1},\ldots,X_{j_l}).$$

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

Proof Consider the program

$$t(X,Y,Z) \text{ :- } a_1(X), q_1(Y,Z).$$
$$t(X,Y,Z) \text{ :- } a_2(X), q_2(Y,Z).$$

with the query $t(X,Y,Z)$?. Furthermore, let a_1 and a_2 be *EDB* relations, while q_1 and q_2 are *IDB* relations. There are two ways to factor t nontrivially: into $t_1(X)$ and $t_2(Y,Z)$, and into $t'_1(X,Y)$, $t'_2(Z)$. We consider the second alternative first.

By definition of factoring, t can be factored into t'_1 and t'_2 if and only if adding the rules

$$t'_1(X,Y) \text{ :- } t(X,Y,Z).$$
$$t'_2(Z) \text{ :- } t(X,Y,Z).$$
$$t(X,Y,Z) \text{ :- } t'_1(X,Y), t'_2(Z).$$

to the definition of t computes the same relation for t as the original definition (without the new rules) for all possible *EDB*s.

Now consider an *EDB* such that a_2 is empty, a_1 contains the fact $a_1(1)$, q_2 is empty, and q_1 contains the facts $q_1(2,3)$, and $q_1(4,5)$. The original program computes only $t(1,2,3)$ and $t(1,4,5)$, while the rewritten program also computes $t(1,2,5)$ and $t(1,4,3)$, so t cannot be factored into t'_1 and t'_2.

Next, consider factoring $t(X,Y,Z)$ into $t_1(X)$ and $t_2(Y,Z)$. Again by the definition of factoring, this is possible if and only if adding the rules

$$t_1(X) \text{ :- } t(X,Y,Z).$$
$$t_2(Y,Z) \text{ :- } t(X,Y,Z).$$
$$t(X,Y,Z) \text{ :- } t_1(X), t_2(Y,Z).$$

to the definition of t computes the same relation for t as the original definition (without the new rules) for all possible *EDB*s.

It is clear that the program with the new rules will compute the same relation for t for all *EDB*s in which a_1 and a_2 are identical. So consider an *EDB* in which a_1 and a_2 differ. In this case, the new program will compute the same relation for $t(X, Y, Z)$ as the original program if and only if q_1 and q_2 compute the same relation. Since q_1 and q_2 can be arbitrary Datalog queries, and containment for binary Datalog queries is undecidable [Shm87], detecting factorable programs is also undecidable. □

The proof of Theorem 3.1 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 3.1 *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* X*s 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.

4 Classes of Factorable Programs

The Magic Sets transformation [BMSU86, BR87, Ram88] 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.

4.1 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 emphasize that this translation is syntactic, and is done only during compile time to test for factorability; the actual program that is evaluated need not be in standard form.

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]. In the following, by *disjoint conjunctive queries* we mean conjunctive queries that do not share any variables.

Definition 4.1 A rule is *left-linear* if it is of the form

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

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

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

where

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

Definition 4.3 A rule is a *combined rule* if it is of the form

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

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.

Also, note that we have used the terms "right" and "left" and "combined" in order to clarify the exposition. Trivially, any transformation that simply permutes the order of arguments in predicates (the same permutation for all instances of a predicate) and reorders predicate instances in the body of rules computes the same relation as the original program (up to renaming of columns.) For that reason, any program that matches our right (left, combined) definitions after some permutation of variables in predicates (the same permutation for all instances of a predicate) and some reordering of predicates in the body of rules, will also be considered right (left, combined).

Example 4.1 Consider the rule

$$t^{bfb}(X, Y, Z) \text{ :- } e(Y, W),\ t^{bfb}(X, W, Z).$$

In this form, the rule does not fit the definitions of right, left, or combined linear rules. Rearranging and permuting, we get the new rule

$$t'^{bbf}(X, Z, Y) \text{ :- } t'^{bbf}(X, Z, W),\ e'(W, Y).$$

which is a left-linear rule. □

4.2 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 4.2 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 then factoring the rewritten program. To illustrate the technique, we again consider the single program that includes all three forms of the recursive rule for the

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

Figure 1: P^{mg} for the three-rule transitive closure.

transitive closure presented in Example 1.1. The Magic Sets algorithm rewrites this program to produce the program in Figure 1.

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 2.

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 4.4 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}.

It is convenient to be able to associate sets of tuples with conjunctions of body literals. We do this by introducing some new predicates and rules that define them. These predicates can be viewed conjunctive queries (i.e. each predicate is defined by a rule that contains only EDB predicates in the body). We will refer to them as "conjunctions" to emphasize the fact that

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

Figure 2: The factored version of P^{mg}.

they correspond to conjunctions of body literals. In the following definition, *first*, *left*, *right*, and *center* are used in the same way as in the definitions of left-, right-, and combined-linear rules.

Definition 4.5 The conjunctions *bound_exit* and *free_exit* are defined as follows:

$$
\begin{aligned}
bound\_exit(\overline{X}) &:- exit(\overline{X},\overline{Y}).\\
free\_exit(\overline{Y}) &:- exit(\overline{X},\overline{Y}).
\end{aligned}
$$

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

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

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

The conjunction *free_last* is defined for a given left-linear rule:

$$
free\_last(\overline{Y}) :- last(\overline{U}_1,\overline{U}_2,\ldots,\overline{U}_n,\overline{Y}).
$$

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

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

The conjunction $free$ is defined for a given right-linear or combined rule:

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

The conjunction *middle* is defined for a given combined rule:

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

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

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

Definition 4.6 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 conjunction "$free\_exit$" must be contained in the conjunction "$free$" for r.
- For any pair of rules r_1 and r_2 in P, if both r_1 and r_2 contain a "*left*" conjunction, these must be equivalent. If one contains a "*left*" query, and the other a "$first$" query, the conjunction "$bound\_first$" must be contained in the conjunction "*bound*".

We use the following notation: $p^{\alpha}(x, a)$ denotes a tuple in the (only) recursive predicate p, with x being the vector of values in the bound arguments and a being the vector of values in the free arguments. We sometimes refer to x as a goal and a as an answer to the goal x. The original query is denoted as the goal x_0.

We begin by proving a simple technical lemma.

Lemma 4.1 *Let P be a selection-pushing program, $m\_p^{\alpha}(x_0)$ be a seed, and D be an EDB. If there is a derivation tree in $P^{mg} \cup D$ or in $P^{fact} \cup D$ for $m\_p^{\alpha}(x_i)$, $x_0 \neq x_i$, then x_0 must be contained in the (set of answers to the) conjunctive query "bound", which is uniquely defined for selection-pushing programs.*

Proof All rules in P^{mg} or P^{fact} that define magic predicates are generated from right-linear or combined rules of P. Consider the first magic fact $m\_p^{\alpha}(x_i)$ that is generated using a magic rule (i.e. that is not the seed $m\_p^{\alpha}(x_0)$). If the magic rule is generated from a right-linear rule, the body of the magic rule contains a conjunction of the form "$m\_p^{\alpha}(\overline{X}), first(\overline{X}, \overline{V})$", and since $m\_p^{\alpha}(x_0)$ is the only magic fact that can be used to instantiate the m_p^{α} literal in the body, x_0 must be contained in the conjunctive query "$bound\_first$". ¿From the second condition

on selection-pushing programs, it follows that x_0 must be contained in the conjunctive query "*bound*". If the magic rule is generated from a combined rule, the body contains a conjunction of the form form "$m\_p^\alpha(\overline{X}), left(\overline{X})$", and it follows immediately that x_0 must be contained in "*bound*". □

The following theorem identifies a class of factorable programs.

Theorem 4.1 *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.*

Proof Let P^{fact} denote the factored program. We will prove the following.
For any database D

1. If there is a derivation tree for a fact $fp(a)$ in $P^{fact} \cup D$, then there is a derivation tree for the fact $p^\alpha(x_0, a)$ in $P^{mg} \cup D$, and
2. If there is a derivation tree for a fact $m\_p^\alpha(c)$ in $P^{fact} \cup D$, then there is a derivation tree for $m\_p^\alpha(c)$ in $P^{mg} \cup D$.

The converses of the above two statements are easy to show from the structure of P^{fact} and P^{mg}. From the first statement and its converse, it follows that relation fp contains exactly the set of answers to Q, thereby establishing the theorem.

The proof is by induction on the height of derivation trees. As the basis, the only tree of height 1 for an m_p^α or fp fact in $P^{fact} \cup D$ is $m\_p^\alpha(x_0)$, and this fact is also in $P^{mg} \cup D$.

For the induction, assume that the claim is true for trees of height less than N.

Case 1 (m_p^α facts):
Consider a derivation tree T of height N for $m\_p^\alpha(c)$ in $P^{fact} \cup D$. Since the rules for m_p^α obtained from left-linear rules are redundant, $m\_p^\alpha(c)$ must have been generated using a magic rule obtained from either a right-linear or combined rule, say r.

If r is a right-linear rule, the children of $m\_p^\alpha(c)$ in T are facts in D, except for one magic fact, say $m\_p^\alpha(c_1)$. By the induction hypothesis, $m\_p^\alpha(c_1)$ also has a derivation tree in $P^{mg} \cup D$. Thus, we have a derivation tree T' for $m\_p^\alpha(c)$ in $P^{mg} \cup D$. (See Figure 3. This figure, and other figures in the proofs of this section, use two conventions: (1) a triangle hanging off of a goal represents a subtree that must exist by induction, and (2) a goal with no subtrees is a basis fact.)

If r is a combined rule, the children of $m\_p^\alpha(c)$ in T include facts in D, plus one magic fact, say $m\_p^\alpha(c_1)$, and some fp facts. By the induction hypothesis, there is a derivation tree for $p^\alpha(x_0, a_i)$ in $P^{mg} \cup D$ for each child fact $fp(a_i)$. Further, $left(x_0)$ holds by Lemma 4.1, and we have $m\_p^\alpha(x_0)$. Using these facts in the body of the magic rule obtained from r in P^{mg}, we obtain an $m\_p^\alpha(c)$ in the head, which can be used in a derivation tree T' in $P^{mg} \cup D$ for $m\_p^\alpha(c)$. (See Figure 4.)

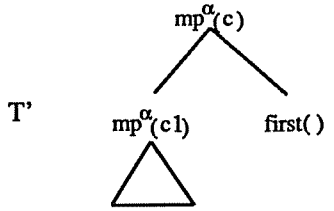

Figure 3: T' — a derivation tree for $m\_p^{\alpha}(c)$ in $P^{mg} \cup D$, right-linear rule case.

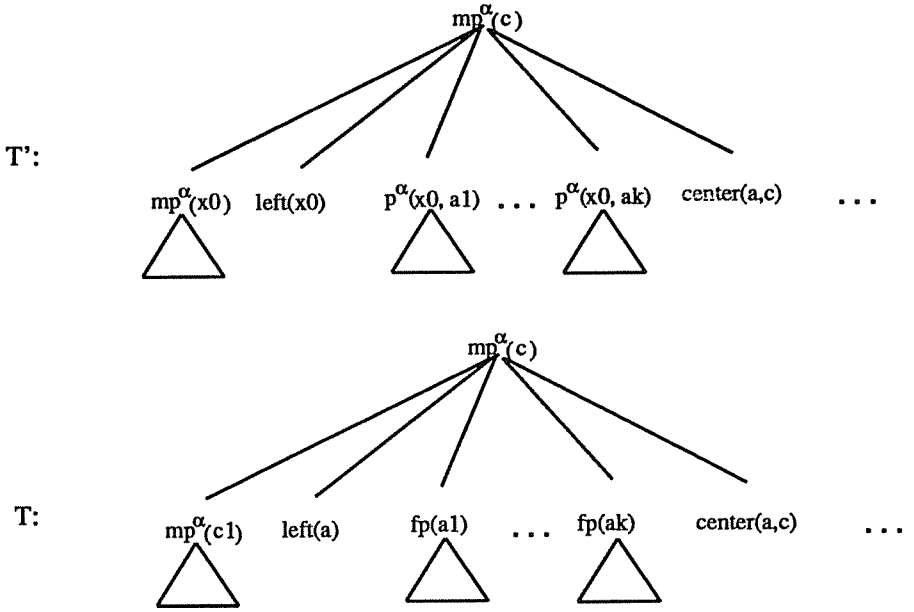

Figure 4: Corresponding derivation trees for $m\_p^{\alpha}(c)$ (with T' in $P^{mg} \cup D$ and T in $P^{fact} \cup D$), combined rule case.

Case 2 (fp facts):
Consider a derivation tree T of height N for $fp(a)$ in $P^{fact} \cup D$. Since the rules for fp obtained from right-linear and combined rules are redundant, $fp(a)$ must have been generated from a rule in P^{fact}, say f, obtained from either an exit or a left-linear rule, say r, in P. There are two subcases to consider.

Case 2(a) (r is an exit rule)
The children of $fp(a)$ in T are facts in D, except for one magic fact, say $m\_p^{\alpha}(c_1)$. We note that since $fp(a)$ was generated from an exit rule, a is contained in "$free\_exit$", and thus is in "$free$" for all right-linear and combined rules.

By the induction hypothesis, there is a derivation tree in $P^{mg} \cup D$ for $m\_p^{\alpha}(c_1)$. Let the magic rule from which $m\_p^{\alpha}(c_1)$ is generated be obtained from rule s in P. There are two subcases to consider.

Case 2(a)(i) (s is a combined rule) Clearly, there is a tree in $P^{mg} \cup D$, using the modified rule corresponding to r, for $p^{\alpha}(c_1, a)$. Consider the children of $m\_p^{\alpha}(c_1)$ in T. In addition to

facts in D, there is one magic fact, and some *fp* facts. By the induction hypothesis, there is a derivation tree for $p^\alpha(x_0, a_i)$ in $P^{mg} \cup D$ for each child fact $fp(a_i)$. Further, $left(x_0)$ holds by Lemma 4.1, and we have $right(a)$ (since a is contained in "$free$") and $m\_p^\alpha(x_0)$. Using these facts — plus the facts in D that are children of $m\_p^\alpha(c_1)$ in T and the fact $p^\alpha(c_1, a)$ — in the body of the rule corresponding to s in P^{mg}, we obtain an instance with $p^\alpha(x_0, a)$ in the head. This instance can be used in a derivation tree T' for $fp(a)$ in $P^{mg} \cup D$. (See Figure 5.)

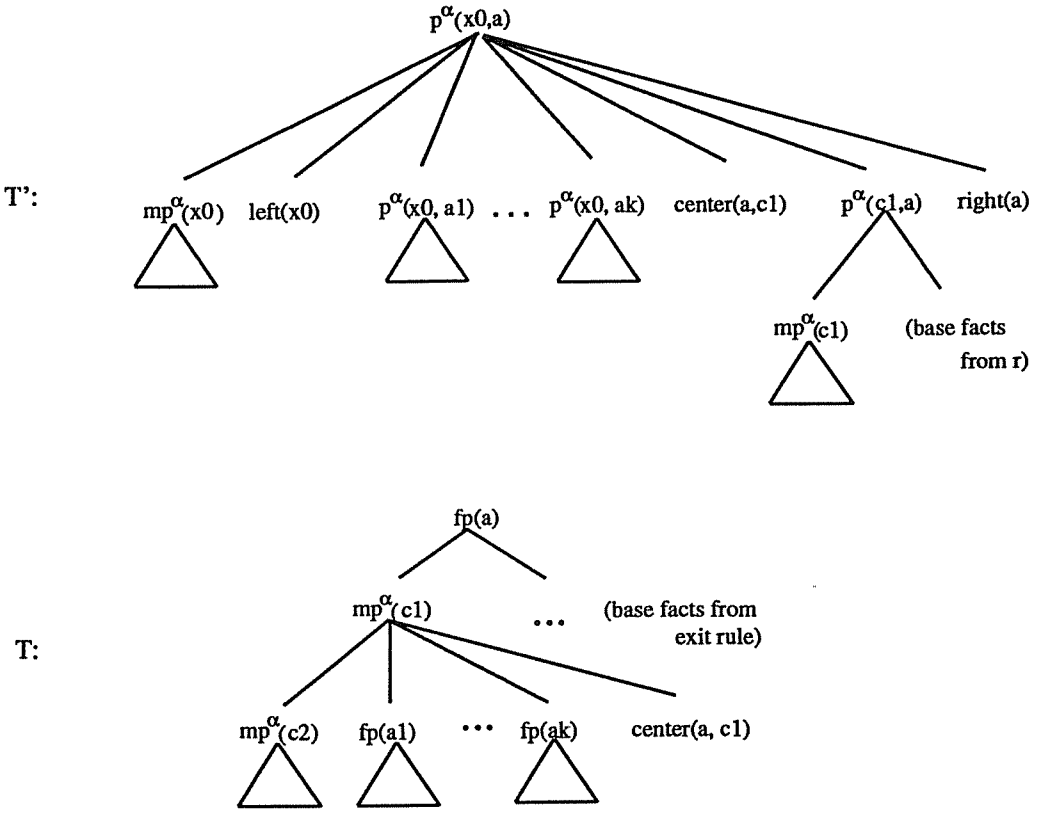

Figure 5: Corresponding derivation trees T' in $P^{mg} \cup D$ and T in $P^{fact} \cup D$, combined rule case.

Case 2(a)(ii) (s is a right-linear rule) Let $m\_p^\alpha(c_2)$ be a magic fact that is either the seed or is generated from a magic rule obtained from a combined rule, and let the magic fact $m\_p^\alpha(c_1)$ be obtained from $m\_p^\alpha(c_2)$ by one or more applications of a magic rule obtained from a right-linear rule. (Such a fact $m\_p^\alpha(c_2)$ must clearly exist.) We show that there is a derivation tree for $p^\alpha(x_0, a)$ in $P^{mg} \cup D$ in three steps. We show, in order, that there is a derivation tree in $P^{mg} \cup D$ for: (1) $p^\alpha(c_1, a)$, (2) $p^\alpha(c_2, a)$, and (3) $p^\alpha(x_0, a)$.

For part (1), by the induction hypothesis, there is a tree in $P^{mg} \cup D$ for $m\_p^\alpha(c_1)$. Thus, there is a tree in $P^{mg} \cup D$ for $p^\alpha(c_1, a)$ (using the rule corresponding to r).

For part (2), consider the derivation of $m\_p^\alpha(c_1)$ from $m\_p^\alpha(c_2)$. This derivation tree is a subtree of T, the derivation tree for $fp(a)$ in $P^{fact} \cup D$. By the induction hypothesis, there is a derivation tree in $P^{mg} \cup D$ for every magic fact that appears in T, and therefore in the subtree

for $m\_p^{\alpha}(c_1)$.

Let the sequence of magic rules used (proceeding from $m\_p^{\alpha}(c_2)$ to $m\_p^{\alpha}(c_1)$) be obtained from the sequence of right-linear rules $r_1', \ldots, r_n'$. Consider the reverse sequence $r_n', \ldots, r_1'$. Using the m_p^{α} and "*first*" facts that are the children of $m\_p^{\alpha}(c_1)$ in T, the fact $p^{\alpha}(c_1, a)$ and the fact $right(a)$, each of which has a derivation tree in $P^{mg} \cup D$, we obtain an instance of rule r_n'. (See Figure 6.) Proceeding similarly, and using the magic fact and the *first* fact used at the corresponding step in the derivation of $m\_p^{\alpha}(c_1)$ from $m\_p^{\alpha}(c_2)$, we can obtain a derivation tree T' with root $p^{\alpha}(c_2, a)$.

For part (3), if $m\_p^{\alpha}(c_2)$ is the seed, the claim holds trivially. If not, $m\_p^{\alpha}(c_2)$ is generated from a combined rule, say s'. The argument in this case is essentially that of Case 2(a)(i), using rule s' instead of s. Consider the children of $m\_p^{\alpha}(c_2)$ in T. In addition to facts in D, there is one magic fact, and some *fp* facts. By the induction hypothesis, there is a derivation tree for $p^{\alpha}(x_0, a_i)$ in $P^{mg} \cup D$ for each child fact $fp(a_i)$. Further, $left(x_0)$ holds by Lemma 4.1, and we have $right(a)$ (since a is contained in "*free*") and $m\_p^{\alpha}(x_0)$. From part (2) above, there is a derivation tree in $P^{mg} \cup D$ for $p^{\alpha}(c_2, a)$. Using these facts — plus the facts in D that are children of $m\_p^{\alpha}(c_1)$ in T — in the body of the rule corresponding to s' in P^{mg}, we obtain an instance with $p^{\alpha}(x_0, a)$ in the head.

Case 2(b) (r is a left-linear rule)
By the induction hypothesis there is a derivation tree for $p^{\alpha}(x_0, a_i)$ in $P^{mg} \cup D$ for each $fp(a_i)$ fact in the body of the instance of f that derives $fp(a)$. Further, $left(x_0)$ holds, from the restrictions on selection-pushing programs, since otherwise no p^{α}, and hence *fp*, facts can be generated from non-exit rules. (Recall that all *left* conjunctions are equivalent and that all *bound_first* conjunctions are contained in *left*. Incidentally, we cannot use Lemma 4.1 to show that $left(x_0)$ holds in this case since we have not established that there is a derivation for some $m\_p^{\alpha}(x_i)$, $x_i \neq x_0$; indeed there may not be such a derivation.) Using the fact $last(a_1, \ldots, a_n, a)$ from the body of the rule instance deriving $fp(a)$ along with $left(x_0)$, $m\_p^{\alpha}(x_0)$ and the facts $p^{\alpha}(x_0, a_i)$, we obtain an instance of (the modified rule corresponding to) rule r in P^{mg}, with head fact $p^{\alpha}(x_0, a)$. □

Example 4.3 We illustrate the intuition behind selection-pushing and show that violating any of the associated conditions could destroy this property.

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

The Magic Sets algorithm rewrites this to

$$
\begin{aligned}
m\_p^{bf}(V) &:- m\_p^{bf}(X),\ l1(X),\ p^{bf}(X,U),\ c1(U,V).\\
m\_p^{bf}(V) &:- m\_p^{bf}(X),\ l2(X),\ p^{bf}(X,U),\ c2(U,V).
\end{aligned}
$$

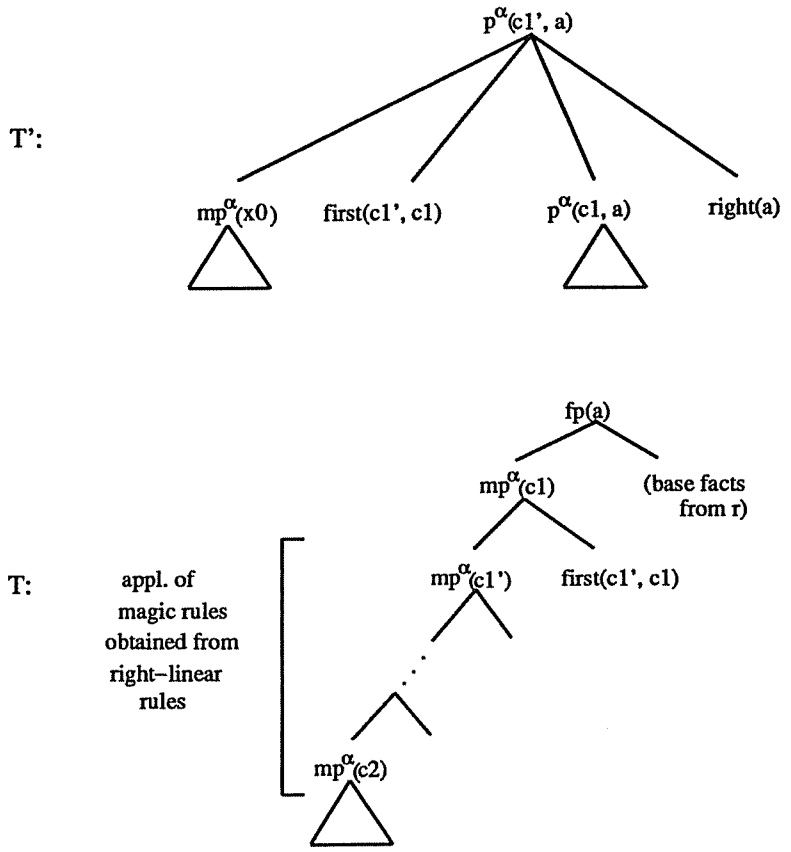

Figure 6: Corresponding derivation trees T' in $P^{mg} \cup D$ and T in $P^{fact} \cup D$, right-linear rule case.

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

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

$$
\begin{array}{lcl}
m\_p^{bf}(V) & :- & bp(X),\ l1(X),\ fp(U),\ c1(U,V). \\
m\_p^{bf}(V) & :- & bp(X),\ l2(X),\ fp(U),\ c2(U,V). \\
m\_p^{bf}(V) & :- & m\_p^{bf}(X),\ f(X,V).
\end{array}
$$

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

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, when factoring 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* conjunctions be satisfied by every potential answer tuple, that is, *free_exit* is contained in every *right* conjunction. (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* conjunction. If the initial goal does not satisfy the *left* conjunction, then we cannot apply the right-linear rule, and the condition that the *bound_first* conjunctions should be contained in the *left* conjunction ensures this.

This discussion suggests a way to strengthen the theorem — we can replace the last condition by the requirement that the bound arguments in the query should satisfy every *left* conjunction and every *bound_first* query. However, this can only be tested at run time, when the query constants are available.

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

Definition 4.7 Let P, Q be an RLC-stable program containing only combined recursive rules. Then P, Q is *symmetric* if the following conditions hold:

- Every recursive rule is a combined rule.
- For each combined rule, *free_exit* must be contained in *free*.
- For every pair of combined rules, the *middle* conjunctive queries must be equivalent.

Theorem 4.2 *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 into $bp(\overline{X})$ and $fp(\overline{Y})$ in P^{mg} with respect to the query Q.*

Proof Let us denote the factored program as P^{fact}. We will prove the following stronger claim:
For any database D

1. If there is a derivation tree for a fact $fp(a)$ in $P^{fact} \cup D$, then there is a derivation tree for the fact $p^\alpha(x_0, a)$ in $P^{mg} \cup D$; that is, relation *fp* contains exactly the set of answers to Q, and
2. If there is a derivation tree for a fact $m\_p^\alpha(c)$ in $P^{fact} \cup D$, then there is a derivation tree for $m\_p^\alpha(c)$ in $P^{mg} \cup D$.

The converses of the above two statements are easy to show from the structure of P^{fact} and P^{mg}. From the first statement and its converse, it follows that relation *fp* contains exactly the set of answers to Q, thereby establishing the theorem. The second statment is used in the induction in the proof below.

We make the following observations. First, all rules defining *bp* or *fp* obtained from non-exit rules are redundant, from the structure of symmetric rules. Thus, we only need to consider

the magic rules and the rules obtained from the exit rules in the factored program. Second, $left_i(x_0)$ must hold for at least one rule, say r_i, else the magic rules are also redundant, i.e., we cannot generate any magic fact other than the seed.

The proof is by induction on the height of derivation trees. As the basis, the only tree of height 1 for an m_p^α or fp fact in $P^{fact} \cup D$ is $m\_p^\alpha(x_0)$, and this fact is also in $P^{mg} \cup D$.

For the induction, assume that the claim is true for trees of height less than N.

Case 1 (magic facts)
There must be a rule, say r_i, such that $left_i(x_0)$ holds, since we have derived a magic fact, say m, that is distinct from the seed. By the induction hypothesis, if the body of the (magic) rule instance used to derive m contains the facts $fp(a_1), \ldots, fp(a_n)$, there are derivation trees for $p^\alpha(x_0, a_1), \ldots, p^\alpha(x_0, a_n)$ in $P^{mg} \cup D$. Using these facts, the seed $m\_p^\alpha(x_0)$, and the "*center*" facts from the rule instance (in P^{fact}) deriving m, we get an instance of the magic rule in P^{mg} obtained from the rule r_i, with m as the head fact.

Case 2 (fp facts)
The body of the rule instance used to derive the fact, say $fp(a)$, contains an m_p^α fact, say $m\_p^\alpha(c_1)$. By the induction hypothesis, there is a derivation for this magic fact in $P^{mg} \cup D$, and thus, using an exit rule, for the fact $p^\alpha(c_1, a)$.

Now consider the rule instance used to derive $m\_p^\alpha(c_1)$ in $P^{fact} \cup D$:

$$m\_p^\alpha(c_1) \;\text{:-}\; m\_p^\alpha(c_2), left(c_2), fp(a_1), fp(a_2), \ldots, fp(a_n), center(u, c_1).$$

By the induction hypothesis there is a derivation tree in $P^{mg} \cup D$ for $m\_p^\alpha(c_2)$ and for each of the facts $p^\alpha(x_0, a_1), \ldots, p^\alpha(x_0, a_n)$.

There must be some rule, say r_i, such that $left_i(x_0)$ holds. (If $c_2 = x_0$, then $left(x_0)$ holds in the above rule instance. If c_2 is not identical to the seed x_0, we have generated a magic fact distinct from the seed, and there must be some such rule.) Consider the modified rule in P^{mg} that corresponds to rule r_i. From the second condition on symmetric rules, $right_i(a)$ holds. Using the seed $m\_p^{alpha}(x_0)$, the facts $p^\alpha(x_0, a_1), \ldots, p^\alpha(x_0, a_n)$, the fact $center(u, c_1)$, the fact $p^\alpha(c_1, a)$, and the fact $right_i(a)$, we obtain an instance of this modified rule in P^{mg} that derives $p^\alpha(x_0, a)$. □

Example 4.4 This example illustrates symmetric programs.

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

Rewriting using Magic Sets, factoring, and then applying further transformations described in Section 4 yields:

$$m\_p^{bf}(W) \;\text{:-}\; bp(X),\ l1(X),\ fp(U),\ fp(V),\ c(U,V,W).$$

$$
\begin{aligned}
m\_p^{bf}(W) &:- bp(X),\ l2(X),\ fp(U),\ fp(V),\ c(U,V,W).\\
m\_p^{bf}(5).&\\
bp(X) &:- m\_p^{bf}(X),\ e(X,Y).\\
fp(Y) &:- m\_p^{bf}(X),\ e(X,Y).\\
query(Y) &:- fp(Y).
\end{aligned}
$$

We observe that once a *bp* tuple is generated that is also in $l1$, we can discard the second rule defining m_p^{bf}; similarly, we can delete the first rule if a *bp* tuple is also in $l2$. (We can only delete one of them, of course.) Also, if the original query, 5, is not contained in either $l1$ or $l2$, then neither the first nor the second rule will produce any facts. □

The ideas underlying selection-pushing and symmetric programs can be combined to identify a third class of programs, which we call *answer-propagating* programs.

Definition 4.8 Let P, Q be an RLC-stable program containing only combined recursive rules. Then P, Q is *answer-propagating* if the following conditions hold:

- *Left-linear rules* For each left-linear rule, *bound_exit* must be contained in *bound.*
- *Right-linear rules* For each right-linear rule, *free_exit* must be contained in *free.*
- *Combined rules* For each combined rule, *free_exit* must be contained in *free.*
- *Pairs of recursive rules*
  For every pair of combined rules, the *middle* conjunctive queries must be equivalent.
  For every pair of one left-linear and one combined rule, *bound* for the left-linear rule must be contained in *bound* for the combined rule, and *free_last* must be contained in *free.*
  For every pair of one right-linear and one combined rule, *bound_first* must be contained in *bound.*
  For every pair of one right-linear and one left-linear rule, *bound_first* must be contained in *bound*, and *free_last* must be contained in *free.*

We have the following technical lemma.

Lemma 4.2 *Let P be an answer-propagating program, $m\_p^{\alpha}(x_0)$ be a seed, and D be an EDB. If there is a derivation tree in $P^{mg} \cup D$ or $P^{fact} \cup D$ for $m\_p^{\alpha}(x_i)$, $x_0 \neq x_i$, then there is at least one combined rule in P such that x_0 is contained in the (set of answers to the) associated conjunctive query "bound".*

Proof Consider the first magic fact generated that is distinct from the seed. The magic rule used to derive this fact must be obtained from a combined or a right-linear rule, and the magic fact used in the instance of this magic rule must be the seed. If the magic rule is obtained from a combined rule, the observation clearly holds. If it is obtained from a right-linear rule, x_0 must be in the associated conjunctive query *bound_first*, and thus by the definition of answer-propagating programs, $left(x_0)$ must hold for every combined rule. □

The following theorem, which strictly generalizes Theorem 4.2, is proved using a combination of the arguments used in the proofs of Theorems 4.1 and 4.2. The structure of the proof is similar to that of Theorem 4.1, but there are some significant differences.

Theorem 4.3 *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 answer-propagating, 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.*

Proof Let us denote the factored program as P^{fact}. We will prove the following stronger claim:
For any database D

1. If there is a derivation tree for a fact $fp(a)$ in $P^{fact} \cup D$, then there is a derivation tree for the fact $p^\alpha(x_0, a)$ in $P^{mg} \cup D$; that is, relation fp contains exactly the set of answers to Q, and
2. If there is a derivation tree for a fact $m\_p^\alpha(c)$ in $P^{fact} \cup D$, then there is a derivation tree for $m\_p^\alpha(c)$ in $P^{mg} \cup D$.

The converses of the above two statements are easy to show from the structure of P^{fact} and P^{mg}. From the first statement and its converse, it follows that relation fp contains exactly the set of answers to Q, thereby establishing the theorem.

The proof is by induction on the height of derivation trees. As the basis, the only tree of height 1 for an m_p^α or fp fact in $P^{fact} \cup D$ is $m\_p^\alpha(x_0)$, and this fact is also in $P^{mg} \cup D$.

For the induction, assume that the claim is true for trees of height less than N.

Case 1 (m_p^α facts):
Consider a derivation tree T of height N for $m\_p^\alpha(c)$ in $P^{fact} \cup D$. Since the rules for m_p^α obtained from left-linear rules are redundant, $m\_p^\alpha(c)$ must have been generated using a magic rule obtained from either a right-linear or combined rule, say r.

If r is a right-linear rule, the children of $m\_p^\alpha(c)$ in T are facts in D, except for one magic fact, say $m\_p^\alpha(c_1)$. By the induction hypothesis, $m\_p^\alpha(c_1)$ also has a derivation tree in $P^{mg} \cup D$. Thus, we have a derivation tree for $m\_p^\alpha(c)$ in $P^{mg} \cup D$.

If r is a combined rule, the children of $m\_p^\alpha(c)$ in T include facts in D, plus one magic fact, say $m\_p^\alpha(c_1)$, and some fp facts. By the induction hypothesis, there is a derivation tree for $p^\alpha(x_0, a_i)$ in $P^{mg} \cup D$ for each child fact $fp(a_i)$. Further, by Lemma 4.2, there must be a combined rule, say r_i, such that $left_i(x_0)$ holds, since we have derived a magic fact that is distinct from the seed. Using the above p^α facts, the seed $m\_p^\alpha(x_0)$, and the *center* facts from the rule instance (in P^{fact}) deriving $m\_p^\alpha(c_1)$, we get an instance of the magic rule in P^{mg} obtained from the rule r_i, with $m\_p^\alpha(c)$ as the head fact. (With the difference that we consider r_i instead of r, this proof in this case is identical to that of case 1 in Theorem 4.1.)

Case 2 (fp facts):
Consider a derivation tree T of height N for $fp(a)$ in $P^{fact} \cup D$. Since the rules for fp obtained

from right-linear and combined rules are redundant, $fp(a)$ must have been generated from a rule in P^{fact}, say f, obtained from either an exit or a left-linear rule, say r, in P. In either case, for each right-linear or combined rule, a is in the associated conjunctive query "$free$" by the conditions on answer-propagating programs. (Recall that queries "$free\_last$" and "$free\_exit$" are always contained in queries "$free$".) As in the proof of Theorem 4.1, there are two subcases to consider: (i) r is an exit rule, and (ii) r is a left-linear rule.

Case 2(a) (r is an exit rule)
A magic fact, say $m\_p^\alpha(c_1)$, appears as a child of $fp(a)$ in T. By the induction hypothesis, there is a derivation tree in $P^{mg} \cup D$ for $m\_p^\alpha(c_1)$. Let the magic rule from which $m\_p^\alpha(c_1)$ is generated be obtained from rule s in P. There are two subcases to consider.

Case 2(a)(i) (s is a combined rule) Let $m\_p^\alpha(c_1) = m\_p^\alpha(c_1)$. Clearly, there is a tree in $P^{mg} \cup D$, using the modified rule corresponding to r, for $p^\alpha(c_1, a)$. Consider the children of $m\_p^\alpha(c_1)$ in T. In addition to facts in D, there is one magic fact, and some fp facts. By the induction hypothesis, there is a derivation tree for $p^\alpha(x_0, a_i)$ in $P^{mg} \cup D$ for each child fact $fp(a_i)$. Further, $left(x_0)$ holds by Lemma 4.2 for some combined rule, say s'. Using these facts — plus the facts in D that are children of $m\_p^\alpha(c_1)$ in T and the facts $m\_p^\alpha(x_0)$, $right(a)$ and $p^\alpha(c_1, a)$ — in the body of the rule corresponding to s' in P^{mg}, we obtain an instance with $p^\alpha(x_0, a)$ in the head. (With the difference that we must use some rule s', instead of s, this proof is identical to the proof of Case 2(a)(i) in Theorem 4.1.

Case 2(a)(ii) (s is a right-linear rule) Let $m\_p^\alpha(c_2)$ be a magic fact that is either the seed or is generated from a magic rule obtained from a combined rule, and let the magic fact $m\_p^\alpha(c_1)$ be obtained from $m\_p^\alpha(c_2)$ by one or more applications of a magic rule obtained from a right-linear rule. (Such a fact $m\_p^\alpha(c_2)$ must clearly exist.) Let $m\_p^\alpha(c_1) = m\_p^\alpha(c_1)$ and let $m\_p^\alpha(c_2) = m\_p^\alpha(c_2)$.

As in the proof of Theorem 4.1, we show that there is a derivation tree for $p^\alpha(x_0, a)$ in $P^{mg} \cup D$ in three steps. We show, in order, that there is a derivation tree in $P^{mg} \cup D$ for: (1) $p^\alpha(c_1, a)$, (2) $p^\alpha(c_2, a)$, and (3) $p^\alpha(x_0, a)$. Parts (1) and (2) can be established by exactly the arguments used in the proof of Theorem 4.1.

For part (3), if $m\_p^\alpha(c_2)$ is the seed, the claim holds trivially. If not, $m\_p^\alpha(c_2)$ is generated from a combined rule, say s''. The argument in this case is essentially that of Case 2(a)(i), using rule s'' instead of s. Consider the children of $m\_p^\alpha(c_2)$ in T. In addition to facts in D, there is one magic fact, and some fp facts. By the induction hypothesis, there is a derivation tree for $p^\alpha(x_0, a_i)$ in $P^{mg} \cup D$ for each child fact $fp(a_i)$. ¿From part (2) above, there is a derivation tree in $P^{mg} \cup D$ for $p^\alpha(c_2, a)$. Further, $left(x_0)$ holds by Lemma 4.2 for some combined rule, say s'. Using these facts — plus the facts in D that are children of $m\_p^\alpha(c_1)$ in T and the facts $m\_p^\alpha(x_0)$, $right(a)$ and $p^\alpha(c_1, a)$ — in the body of the rule corresponding to s' in P^{mg}, we obtain an instance with $p^\alpha(x_0, a)$ in the head.

Case 2(b) (r is a left-linear rule)
By the induction hypothesis there is a derivation tree for $p^\alpha(x_0, a_i)$ in $P^{mg} \cup D$ for each $fp(a_i)$ fact in the body of the instance of f that derives $fp(a)$. Further, $left(x_0)$ holds, since *bound* for any left-linear rule must be contained in *bound* for any combined-linear rule, and from

Lemma 4.2, there is at least one combined-linear rule such that $left(x_0)$ holds. Using the fact $last(a_1, \ldots, a_n, a)$ from the body of the rule instance deriving $fp(a)$ along with $left(x_0)$, $m\_p^{\alpha}(x_0)$ and the facts $p^{\alpha}(x_0, a_i)$, we obtain an instance of (the modified rule corresponding to) rule r in P^{mg}, with head fact $p^{\alpha}(x_0, a)$. □

Example 4.5 This example illustrates answer-propagating programs.

$$
\begin{aligned}
p(X,Y) &:\!- \; l1(X),\ p(X,U),\ p(X,V),\ c(U,V,W),\ p(W,Y),\ r1(Y).\\
p(X,Y) &:\!- \; l2(X),\ p(X,U),\ p(X,V),\ c(U,V,W),\ p(W,Y),\ r2(Y).\\
p(X,Y) &:\!- \; l3(X),\ p(X,U),\ la(U,Y).\\
p(X,Y) &:\!- \; f(X,U),\ p(U,Y),\ r3(Y).\\
p(X,Y) &:\!- \; e(X,Y).\\
query(Y) &:\!- \; p(5,Y).
\end{aligned}
$$

Rewriting using Magic Sets, factoring, and then applying further transformations described in Section 4 yields:

$$
\begin{aligned}
m\_p^{bf}(W) &:\!- \; bp(X),\ l1(X),\ fp(U),\ fp(V),\ c(U,V,W).\\
m\_p^{bf}(W) &:\!- \; bp(X),\ l2(X),\ fp(U),\ fp(V),\ c(U,V,W).\\
m\_p^{bf}(U) &:\!- \; bp(X),\ f(X,U).\\
m\_p^{bf}(5).&\\
bp(X) &:\!- \; m\_p^{bf}(X),\ e(X,Y).\\
bp(X) &:\!- \; m\_p^{bf}(X),\ f(X,U),\ bp(U),\ fp(Y),\ r3(Y).\\
fp(Y) &:\!- \; m\_p^{bf}(X),\ e(X,Y).\\
fp(Y) &:\!- \; m\_p^{bf}(X),\ l3(X),\ bp(X),\ fp(U),\ la(U,Y).\\
query(Y) &:\!- \; fp(Y).
\end{aligned}
$$

□

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 three things:

1. In the magic program, every answer to a subgoal is an answer to the original query.
2. In the factored program, the "subgoals" generated are exactly those generated in evaluating the original magic program.
3. In the factored program, let S be the subgoals generated, and let A be the answers generated. Then for every a in A, there is some s in S such that a is an answer to S in the magic program.

Because testing for these factorable 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 conjunctions will be empty, in which case polynomial time algorithms for testing if a recursion satisfies Theorems 4.1, 4.2, and 4.3 exist.

We conclude this section with an example involving a program with function symbols. The example illustrates that applying bottom-up evaluation to a factored program can give order of magnitude speedups over Prolog on the original program.

Example 4.6 We return to Example 1.2 from the introduction. Recall that we wish to compute the set of all members of a given list that satisfy some predicate p by augmenting the standard member procedure

$$\begin{aligned} pmem(X,[X|T]) &:- p(X). \\ pmem(X,[H|T]) &:- pmem(X,T). \end{aligned}$$

and the following query

$$q(X) :- pmem(X,[x_1,x_2,\ldots,x_n]).$$

where $[x_1,x_2,\ldots,x_n]$ is the given list. As noted in the introduction, on this program and query, if all members of the given list satisfy the predicate p, Prolog will compute the $O(n^2)$ facts $pmem(x_i,[x_j,\ldots,x_n])$ for $1 \leq j \leq n$ and $j \leq i \leq n$.

Now consider first rewriting the program in standard form:

$$\begin{aligned} pmem(X,L) &:- list(X,T,L),\ p(X). \\ pmem(X,L) &:- pmem(X,T),\ list(H,T,L). \end{aligned}$$

This program, with the query $q(X)$:-$pmem(X,[x_1,x_2,\ldots,x_n])$, generates the adorned program

$$\begin{aligned} pmem(X,L)^{fb} &:- list(X,T,L),\ p(X). \\ pmem(X,L)^{fb} &:- pmem(X,T)^{fb},\ list(H,T,L). \end{aligned}$$

This program is *selection pushing*, so the corresponding magic program can be factored. At this point we revert to the original notation (recall that the "standard" notation with lists represented by the EDB relation *list* is used only for compile-time testing for factorability) to get the magic program

$$m\_pmem([x_1,x_2,\ldots,x_n]).$$

$$m\_pmem(T) \;\text{:-}\; m\_pmem([H|T]).$$

$$pmem(X,[X|T])^{fb} \;\text{:-}\; m\_pmem([X|T]),\ p(X).$$
$$pmem(X,[H|T])^{fb} \;\text{:-}\; m\_pmem(X),\ pmem(X,T)^{fb}.$$

$$query(X) \;\text{:-}\; pmem(X,[x_1,x_2,\ldots,x_n])^{fb}$$

Factoring, we get

$$m\_pmem([x_1,x_2,\ldots,x_n]).$$
$$m\_pmem(T) \;\text{:-}\; m\_pmem([H|T]).$$

$$bpmem([X|T]) \;\text{:-}\; m\_pmem([X|T]),\ p(X).$$
$$bpmem([H|T]) \;\text{:-}\; m\_pmem(X),\ fpmem(X),\ bpmem(T).$$

$$fpmem(X) \;\text{:-}\; m\_pmem([X|T]),\ p(X).$$
$$fpmem(X) \;\text{:-}\; m\_pmem(X),\ fpmem(X),\ bpmem(T).$$

$$query(X) \;\text{:-}\; fpmem(X),\ bpmem([x_1,x_2,\ldots,x_n]).$$

Applying optimizations from Section 5 gives

$$m\_pmem([x_1,x_2,\ldots,x_n]).$$
$$m\_pmem(T) \;\text{:-}\; m\_pmem([H|T]).$$

$$fpmem(X) \;\text{:-}\; m\_pmem([X|T]),\ p(X).$$

$$query(X) \;\text{:-}\; fpmem(X).$$

Evaluating the resulting program bottom-up produces the *m_pmem* relation

$$m\_pmem([x_n])$$
$$m\_pmem([x_{n-1},x_n]$$
$$\vdots$$
$$m\_pmem[x_1,\ldots,x_n])$$

A standard structure-sharing implementation of lists stores these n facts in $O(n)$ space and avoids a list copy on each application of the recursive rule for *m_pmem*. Assuming such a structure-sharing implementation of lists, the factored program computes the answer in linear time. □

5 Some Additional Optimizations

We use the following definitions.

Definition 5.1 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 5.2 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 5.1 *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.*

Example 5.1 As an illustration of the above lemma, consider the following program and query:

$$\begin{aligned} p^{bbf}(X,Y,Z) &:\!- \; a(X),\ p^{bbf}(X,Y,W),\ d(W,U),\ p^{bbf}(X,U,Z). \\ p^{bbf}(X,Y,Z) &:\!- \; exit(X,Y,Z). \\ query(U) &:\!- \; p^{bbf}(5,6,U)? \end{aligned}$$

None of the theorems in Section 3 is applicable since X, a variable that appears in a bound argument position in the head of the first rule, also appears in the right linear literal. The reduced program with respect to the first argument position, which is a static argument, is:

$$\begin{aligned} s^{bf}(Y,Z) &:\!- \; a(5),\ s^{bf}(Y,W), d(W,U),\ s^{bf}(U,Z). \\ s^{bf}(Y,Z) &:\!- \; exit(5,Y,Z). \\ query(U) &:\!- \; s^{bf}(6,U)? \end{aligned}$$

The theorems in Section 3 are applicable to the reduced program. □

A special class of rules was defined in [NRSU89].

Definition 5.3 A *pseudo-left-linear* rule is a rule that satisfies all the criteria for a left-linear rule, except that the conjunctions *left* and *last* may share a variable.

Example 5.2 A second example of the use of the reduction lemma is a program that contains a pseudo-left-linear rule.

$$\begin{aligned} p^{bbf}(X,Y,Z) &:- p^{bbf}(X,Y,W), d(W,X,Z). \\ p^{bbf}(X,Y,Z) &:- exit(X,Y,Z). \\ query(U) &:- p^{bbf}(5,6,U)? \end{aligned}$$

As in the previous example, the theorems in Section 3 do not apply to this program because X is connected to the variable in the free argument, Z. Reducing the program with respect to the first argument, which is a static argument, gives us

$$\begin{aligned} s^{bf}(Y,Z) &:- s^{bf}(Y,W), d(W,5,Z). \\ s^{bf}(Y,Z) &:- exit(5,Y,Z). \\ query(U) &:- s^{bf}(6,U)? \end{aligned}$$

This program contains only a left-linear recursive rule, and can be factored using the theorems of Section 3. □

If every occurrence of the recursive predicate in a unit program is left-linear, we observe that every bound argument is a static argument. Using this observation, the previous example can be generalized to show the following lemma.

Lemma 5.2 *Reducing a unit program containing only pseudo-left-linear rules with respect to the bound arguments that violate left-linearity yields a program in which every recursive rule is left-linear.*

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 5.1 *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 5.2 *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 5.3 *Let the original query correspond to the fact $m\_p^{\alpha}(\overline{c})$. If a rule contains the literal $bp(\overline{c})$ and also an fp literal, then the literal $bp(\overline{c})$ can be deleted.*

Some additional simple observations that are useful are mentioned below.

Proposition 5.4 *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 [Sag88].

Proposition 5.5 *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.[2]

Example 5.3 Consider again the factored version of P^{mg} from the three-rule transitive closure (Figure 2.) 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:

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

[2] Consider a program predicate p, and let p' be a new predicate defined by the single rule $p'(\overline{X}) :- p(\overline{Y})$. where $\overline{X}$ is identical to the vector $\overline{Y}$ except that the ith argument is deleted. We say that the ith argument position of p in a given occurrence of p is *existential* if replacing this p literal by the corresponding p' literal leaves the set of answers to the query unchanged.

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:

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

□

6 A Unifying Overview

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

6.1 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 6.1 *(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

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

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 6.2 *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.*

Proof We are given that P is a simple one-sided recursion that has been expanded to the form

$$\begin{aligned} p(\overline{A},\overline{B}) &:- p(\overline{A},\overline{C}),c(\overline{C},\overline{D},\overline{B}). \\ p(\overline{A},\overline{B}) &:- exit(\overline{A},\overline{B}) \end{aligned}$$

where $\overline{A}$, $\overline{B}$, $\overline{C}$, and $\overline{D}$ are vectors of disjoint variables. There are two possible full-selection queries on this program: $p(\overline{a},\overline{B})$? and $p(\overline{A},\overline{b})$?. We consider each in turn.

If the query is $p(\overline{a},\overline{B})$?, then the adorned program P^{ad} corresponding to P is

$$\begin{aligned} p^{bf}(\overline{A},\overline{B}) &:- p^{bf}(\overline{A},\overline{C}),c(\overline{C},\overline{D},\overline{B}). \\ p^{bf}(\overline{A},\overline{B}) &:- exit(\overline{A},\overline{B}) \end{aligned}$$

This is an RLC-stable program, since it consists of one left-linear rule and one exit rule, and the only adorned version of p in the program is p^{bf}. Furthermore, this adorned program is selection-pushing, since by definition of selection-pushing recursions, any single recursive rule, RLC-stable program with a left-linear recursive rule is selection pushing. Then by Theorem 4.1, P^{mg} factors with respect to the query $Q = p(\overline{a},\overline{B})$?.

Next consider the query $p(\overline{A},\overline{b})$?. In this case, P^{ad} is

$$\begin{aligned} p^{fb}(\overline{A},\overline{B}) &:- p^{fb}(\overline{A},\overline{C}),c(\overline{C},\overline{D},\overline{B}). \\ p^{fb}(\overline{A},\overline{B}) &:- exit(\overline{A},\overline{B}) \end{aligned}$$

Now the recursive rule is a right-linear rule. This can be seen most clearly if we reverse the order of arguments in all predicates of the recursion, and reverse the order of predicates within the recursive rule, giving

$$\begin{aligned} p_r^{bf}(\overline{B},\overline{A}) &:- c_r(\overline{B},\overline{D},\overline{C}),p_r^{fb}(\overline{C},\overline{A}). \\ p_r^{bf}(\overline{B},\overline{A}) &:- exit_r(\overline{B},\overline{A}) \end{aligned}$$

(Here we have used the notation that p_r is p with the order of arguments reversed.)

The auxiliary predicate $free\_exit$ is defined by

$$free\_exit(\overline{A}) :- exit(\overline{A},\overline{B}).$$

while the predicate $free$ is defined by

$$free(\overline{A}).$$

That is, since *right* is empty in the recursive rule, $free(\overline{A})$ is true for any instantiation of $\overline{A}$. This in turn implies that $free\_exit$ is contained in $free$, so again, the program is selection-pushing. Then again by Theorem 4.1, P^{mg} factors with respect to the query $Q = p(\overline{A},\overline{b})$?. □

6.2 Separable Programs

Separable programs, defined in [Nau88a], were defined to be class of recursions for which selection queries have efficient evaluation algorithms. Essentially, [Nau88a] 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.

In order to define the separable recursions, we need some auxiliary definitions.

Definition 6.1 Let r be a linear recursive rule and let t be the recursive predicate in r. Then r contains *shifting variables* if there is some variable X such that X appears in position p_1 in the instance of t in the head of r and in position p_2 in the instance of t in the body of r, where $p_1 \neq p_2$.

Definition 6.2 A predicate instance p_1 is *connected* to a predicate instance p_2 if p_1 shares a variable with p_2, or shares a variable with a predicate instance connected to p_2.

Definition 6.3 A subset of predicate instances C is a *maximal connected set* if

1. For every pair of predicate instances p_1 and p_2 in C, p_1 and p_2 are connected, and
2. No predicate instance in C shares a variable with any predicate instance not in C.

Definition 6.4 (Separable Recursions) Let t be defined by n linear recursive rules r_1 through r_n. Furthermore, let t_i^h be the argument positions of t such that in the instance of t at the head of rule r_i, each argument position in t_i^h shares a variable with a nonrecursive predicate in the body of r_i. Similarly, let t_i^b be the argument positions of t such that in the instance of t in the body of rule r_i, each argument position in t_i^b shares a variable with a nonrecursive predicate in the body of r_i. Then the definition of t is a *separable recursion* if

1. For $1 \leq i \leq n$, r_i has no shifting variables, and
2. For $1 \leq i \leq n$, $t_i^h = t_i^b$, and
3. For $1 \leq i \leq n$ and $i < j \leq n$, either $t_i^h = t_j^h$ or t_i^h and t_j^h are disjoint, and
4. For $1 \leq i \leq n$, removing the instance of t from the body of r_i leaves a maximal connected set.

Conditions 1 and 2, together with the fact that all rules in separable recursions must be linear, imply that any given recursive rule of a separable recursion can be written in the form (up to consistent reordering of arguments within predicates and predicates within rules)

$$t(\overline{X}, \overline{Y}) \quad :- \quad A(\overline{X}), t(\overline{X}, \overline{W}), B(\overline{W}, \overline{Y}).$$

where

- $\overline{X}$, $\overline{Y}$, and $\overline{Z}$ are disjoint vectors of variables, and
- A or B could be empty (if B is empty, then so is $\overline{Y}$).

If A is nonempty, then any full selection on such a rule must bind both $\overline{X}$ and $\overline{Y}$, since both are in t^h for that rule. On such a selection, the separable recursion evaluation algorithm does not reduce the arity of the recursion (see [Nau88b] for details.) A more interesting class of recursions are those in which A is empty. A formal definition of this class follows; first we need one auxiliary definition.

Definition 6.5 Let r be a linear recursive rule and let t be the recursive predicate in r. Then a variable X is a *fixed* variable if X appears in the same position in the instances of t in the head and body of r.

Definition 6.6 A separable recursion consisting of n recursive rules $r_1 \ldots r_n$ is *reducible* if for $1 \leq i \leq n$, no fixed variable appears in t_i^h.

Theorem 6.3 *Let P be a reducible 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.*

Proof For the purposes of this proof, it is sufficient to group the equivalence classes of arguments of t into two classes: the one bound by the query, and those not bound by the query. Using this grouping, by the definition of reducible separable recursions, the adorned program resulting from any full selection on any separable recursion consists only of 1) left-linear rules with no *left* predicate, and 2) right-linear rules with no *right* predicate. This implies that the recursion is selection-pushing; hence by Theorem 4.1, the pair P^{mg}, Q is factorable. □

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.

6.3 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.

6.4 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:

$$\begin{aligned} t^{bf}(X,Y) &:- t^{bf}(X,Z), e(Z,Y). \\ t^{bf}(X,Y) &:- e(Z,Y). \end{aligned}$$

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.

We restrict ourselves in the remainder of this section to programs in which every recursive rule is right-linear. Consider the following example:

$$\begin{aligned} p^{bf}(X,Y) &:- first1(X,U), p^{bf}(U,Y), right1(Y). \\ p^{bf}(X,Y) &:- first2(X,U), p^{bf}(U,Y), right2(Y). \\ p^{bf}(X,Y) &:- exit(X,Y). \end{aligned}$$

For either of the factoring theorems in Section 3 to be applicable, if we assume that $right1$, $right2$ and $exit$ are EDB relations, then every value in the second column of the relation $exit$ must also be in $right1$ and $right2$.

Let us consider the corresponding magic program, for the query $p^{bf}(5, X)$?:

$$
\begin{aligned}
& magic\_p^{bf}(5). \\
& magic\_p^{bf}(U) \;\text{:-}\; magic\_p^{bf}(X), first1(X,U). \\
& magic\_p^{bf}(U) \;\text{:-}\; magic\_p^{bf}(X), first2(X,U).
\end{aligned}
$$

$$
\begin{aligned}
p^{bf}(X,Y) \;\text{:-}\; & magic\_p^{bf}(X), first1(X,U), p^{bf}(U,Y), right1(Y). \\
p^{bf}(X,Y) \;\text{:-}\; & magic\_p^{bf}(X), first2(X,U), p^{bf}(U,Y), right2(Y). \\
p^{bf}(X,Y) \;\text{:-}\; & magic\_p^{bf}(X), exit(X,Y).
\end{aligned}
$$

When factored, this yields:

$$
\begin{aligned}
& magic\_p^{bf}(5). \\
& magic\_p^{bf}(U) \;\text{:-}\; magic\_p^{bf}(X), first1(X,U). \\
& magic\_p^{bf}(U) \;\text{:-}\; magic\_p^{bf}(X), first2(X,U).
\end{aligned}
$$

$$
\begin{aligned}
bp(X) \;\text{:-}\; & magic\_p^{bf}(X), first1(X,U), bp(U), fp(Y), right1(Y). \\
fp(Y) \;\text{:-}\; & magic\_p^{bf}(X), first1(X,U), bp(U), fp(Y), right1(Y). \\
bp(X) \;\text{:-}\; & magic\_p^{bf}(X), first2(X,U), bp(U), fp(Y), right2(Y). \\
fp(Y) \;\text{:-}\; & magic\_p^{bf}(X), first2(X,U), bp(U), fp(Y), right2(Y). \\
bp(X) \;\text{:-}\; & magic\_p^{bf}(X), exit(X,Y). \\
fp(Y) \;\text{:-}\; & magic\_p^{bf}(X), exit(X,Y).
\end{aligned}
$$

Both of the recursive rules defining fp can be deleted, since the head literal appears in the body in each. Since we are only interested in fp, and the exit rule defining fp does not refer to bp, bp is not reachable and all rules defining it can be discarded, leaving us with:

$$
\begin{aligned}
& magic\_p^{bf}(5). \\
& magic\_p^{bf}(U) \;\text{:-}\; magic\_p^{bf}(X), first1(X,U). \\
& magic\_p^{bf}(U) \;\text{:-}\; magic\_p^{bf}(X), first2(X,U). \\
& fp(Y) \;\text{:-}\; magic\_p^{bf}(X), exit(X,Y).
\end{aligned}
$$

Now consider the counting program (which can be obtained by adding index fields to the magic program and then deleting some arguments, literals and rules [BR87]):

$$
\begin{aligned}
& cnt\_p^{bf}(5,0). \\
& cnt\_p^{bf}(U, I+1) \;\text{:-}\; cnt\_p^{bf}(X,I), first1(X,U). \\
& cnt\_p^{bf}(U, I+1) \;\text{:-}\; cnt\_p^{bf}(X,I), first2(X,U).
\end{aligned}
$$

$$
\begin{aligned}
p^{bf}(Y,I) &:- p^{bf}(Y,I+1), right1(Y).\\
p^{bf}(Y,I) &:- p^{bf}(Y,I+1), right2(Y).\\
p^{bf}(Y,I) &:- cnt\_p^{bf}(X,I), exit(X,Y).
\end{aligned}
$$

If we delete the index fields, we obtain:

$$
\begin{aligned}
cnt\_p^{bf}(5,0).&\\
cnt\_p^{bf}(U) &:- cnt\_p^{bf}(X), first1(X,U).\\
cnt\_p^{bf}(U) &:- cnt\_p^{bf}(X), first2(X,U).\\
p^{bf}(Y) &:- p^{bf}(Y), right1(Y).\\
p^{bf}(Y) &:- p^{bf}(Y), right2(Y).\\
p^{bf}(Y) &:- cnt\_p^{bf}(X), exit(X,Y).
\end{aligned}
$$

The two recursive rules defining p^{bf} can be dropped since the head literal appears in the body, and this leaves us with the same program obtained via factoring (except that some predicates are named differently). Intuitively, the index fields are unnecessary because every value that appears in the free argument of p^{bf}, which must be from the second column of $exit$, is also in both $right1$ and $right2$, and so any value that appears in p^bf with any index value also appears in p^bf with index value 0. Thus, we can drop the indices without affecting the set of answers (which is the set of values that have index value 0).

The above example can be generalized to establish the following theorem.

Theorem 6.4 *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.*

Proof If a program-query pair contains no left-linear literals, then it must consist only of right linear rules, which have the form

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

For each rule of this form, the Magic program will have the magic rule

$$m\_p^{\alpha}(\overline{V}) \;:-\; m\_p^{\alpha}(\overline{X}), first(\overline{X},\overline{V}).$$

and a modified original rule

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

Factoring will give the rules

$$
\begin{aligned}
m\_p^{\alpha}(\overline{V}) &:- m\_p^{\alpha}(\overline{X}), first(\overline{X},\overline{V}).\\
bp(\overline{X}) &:- m\_p^{\alpha}(\overline{X}), first(\overline{X},\overline{V}), bp(\overline{V}), fp(\overline{Y}), right(\overline{Y}).\\
fp(\overline{Y}) &:- m\_p(\overline{X}), first(\overline{X},\overline{V}), bp(\overline{V}), fp\overline{Y}), right(\overline{Y}). \qquad (2)
\end{aligned}
$$

Rule 2 is trivially redundant, and can be deleted.

Corresponding to the exit rule of the original recursion,

$$p^{\alpha}(\overline{X},\overline{Y}) \;\; :- \;\; exit(\overline{X},\overline{Y}).$$

the Magic program will have the rule

$$p^{\alpha}(\overline{X},\overline{Y}) \;\; :- \;\; m\_p^{\alpha}(\overline{X}), exit(\overline{X},\overline{Y}).$$

which factors into the two rules

$$\begin{aligned} bp(\overline{X}) \;\; &:- \;\; m\_p^{\alpha}(\overline{X}), exit(\overline{X},\overline{Y}). \\ fp(\overline{Y}) \;\; &:- \;\; m\_p^{\alpha}(\overline{X}), exit(\overline{X},\overline{Y}). \end{aligned} \tag{3}$$

Finally, the query

$$q(Y) \;\; :- \;\; p^{\alpha}(\overline{x},\overline{Y})?$$

generates the following initialization rule:

$$m\_p^{\alpha}(\overline{x}).$$

Finally, the query is defined by

$$q(Y) \;\; :- \;\; fp^{\alpha}(\overline{Y}).$$

This definition can be further optimized by another transformation from Section 5: since the literal *bp* is not reachable from the query, all rules defining *bp* can be deleted.

To summarize, there are four types of rules generated. Each recursive rule generates a rule of the form

$$m\_p^{\alpha}(\overline{V}) \;\; :- \;\; m\_p^{\alpha}(\overline{X}), first(\overline{X},\overline{V}).$$

while the initialization generates

$$m\_p^{\alpha}(\overline{x}).$$

The exit rule generates

$$fp(\overline{Y}) \;\; :- \;\; m\_p^{\alpha}(\overline{X}), exit(\overline{X},\overline{Y}).$$

and the query gives

$$q(Y) \;\; :- \;\; fp^{\alpha}(\overline{Y})$$

We now turn to the Counting transformation on P.

If there are k recursive rules in P, then the ith recursive rule will generate the rules

$$\begin{aligned} cnt\_p^{\alpha}(\overline{V}, I+1, k*i+J) &:- cnt\_p^{\alpha}(\overline{X}, I, J), first(\overline{X}, \overline{V}). \\ p\_cnt^{\alpha}(\overline{Y}, I, J) &:- p\_cnt^{\alpha}(\overline{Y}, I+1, k*i+J), right(\overline{Y}). \end{aligned}$$

The exit rule will generate

$$p\_cnt^{\alpha}(\overline{Y}, I, J) :- cnt\_p^{\alpha}(\overline{X}, I, J), exit(\overline{X}, \overline{Y}).$$

while the initialization of cnt_p will be

$$cnt\_p^{\alpha}(\overline{x}, 0, 0).$$

and the query is defined by

$$q(\overline{Y}) :- p\_cnt^{\alpha}(\overline{Y}, 0, 0).$$

Deleting index fields gives

$$\begin{aligned} cnt\_p^{\alpha}(\overline{V}) &:- cnt\_p^{\alpha}(\overline{X}), first(\overline{X}, \overline{V}). \\ p\_cnt^{\alpha}(\overline{Y}) &:- p\_cnt^{\alpha}(\overline{Y}). \end{aligned}$$

from each recursive rule. The second rule is trivially redundant, and can be deleted. The exit rule generates

$$p\_cnt^{\alpha}(\overline{Y}) :- cnt\_p^{\alpha}(\overline{X}), exit(\overline{X}, \overline{Y}).$$

and finally the initialization gives

$$cnt\_p^{\alpha}(\overline{x}).$$

and the query

$$q(\overline{Y}) :- p\_cnt^{\alpha}(\overline{Y}, 0, 0).$$

To summarize, there are four types of rules in the recursion after the index fields have been deleted: for each recursive rule, we get the rule

$$cnt\_p^{\alpha}(\overline{V}) :- cnt\_p^{\alpha}(\overline{X}), first(\overline{X}, \overline{V}).$$

while the initialization generates

$$cnt\_p^{\alpha}(\overline{x}).$$

The exit rule generates

$$p\_cnt^{\alpha}(\overline{Y}) :- cnt\_p^{\alpha}(\overline{X}), exit(\overline{X}, \overline{Y}).$$

and the query generates

$$q(\overline{Y}) \quad :- \quad p\_cnt^{\alpha}(\overline{Y}).$$

Substituting *cnt_p* for *m_p*, and *p_cnt* for *fp*, we see that this recursion is identical to that produced by factoring the magic program. □

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.

7 Directions for Future Work

The results presented in this paper motivate several interesting problems, and we describe them briefly in this section.

7.1 New Classes of Factorable Programs

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. We present some examples of programs that can be factored and that are not Magic programs. It is interesting that these factorable programs arise as a result of factoring Magic programs. This suggests that even if programmers do not write factorable programs, optimization strategies might produce programs that can be factored. Identifying broader classes of factorable programs is therefore an interesting research direction.

Example 7.1 Consider the following program.

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

Rewriting using Magic Sets, factoring, and optimizing gives us

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

This program can also be factored with respect to the predicate ft, although we cannot establish this using the results presented in this paper. Factoring and optimizing along the lines of Section

4 yields

$$
\begin{aligned}
m\_t^{bf}(5). & \\
ft1(Y) &:- ft1(U), b(U,Y), d(\_). \\
ft1(Y) &:- m\_t^{bf}(X), e(X,Y,Z). \\
ft2(Z) &:- ft1(U), b(U,Y), d(Z). \\
ft2(Z) &:- m\_t^{bf}(X), e(X,Y,Z). \\
ft(Y,Z) &:- ft1(Y), ft2(Z). \\
query(Y,Z) &:- ft(Y,Z).
\end{aligned}
$$

If the second argument is bound in the original program, the factored Magic program can again be factored with respect to the only binary predicate, ft, to yield a unary program.

Finally, consider the same initial program with a query that binds the last argument to 5.

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

This illustrates an interesting point: If we wish to treat the last argument of the body literal t as a bound argument, we must allow non-ground tuples [Ram88]. We can then factor the program. □

7.2 Relationship to Other Optimizations

We showed that for the classes of factorable Magic programs identified in this paper, the indices in Counting were unnecessary. In fact, the factored program could be optimized to essentially the Counting program with all index fields deleted. The index fields therefore represent an unnecessary overhead, and in programs with left-linear occurrences, this causes non-termination. Can such a result be established for any Magic program (corresponding to a unit program) in which the recursive predicate can be factored into its bound and free parts? That is, can we show that the Counting indices are unnecessary in factorable Magic programs, independently of the sufficient conditions that we use to ensure factorability?

Another interesting question concerns one-sided recursions. 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?

7.3 Non-Unit Programs

Suppose the program for p^α is factorable, but this predicate is not the query predicate. Then, we can still use the factored program for p^α by generating calls to this program for each p^α

goal. Can we identify cases where we need not distinguish between answers to different calls to p^α? That is, can we identify cases in which p^α can be factored even if it is not the top-level query?

Example 7.2 Consider the following two programs. The first is P_1:

$$\begin{aligned} p(X,Y) &:- b(X,U), p(U,Y). \\ p(X,Y) &:- e(X,Y). \end{aligned}$$

The second is P_2:

$$\begin{aligned} p(X,Y) &:- l(X), p(X,U), c(U,V), p(V,Y). \\ p(X,Y) &:- d(X,Y). \end{aligned}$$

For both programs, the Magic program corresponding to a selection query that binds the first argument, that is, query form p^{bf}, can be factored.

Let P be the following single-rule program:

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

Consider the program $P \cup P_1$, and the query $q(1)$?. The predicate p^{bf} can be factored in the corresponding Magic program. However, this is not the case if P is the program

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

and the query is $q(X,Y)$?.

Further, p^{bf} cannot be factored in $P \cup P_2$, regardless of which rule is chosen for P, and which of the two query forms we consider.

An important problem is to develop sufficient conditions that allow us to factor p^bf in programs where it is not the top-level goal.

Another interesting question is to identify classes of programs defining p^{bf} for which the factorability of p^{bf} in (other) programs can be decided without examining the definition of p^{bf}. As an example, consider the program P_1 as the definition of p. Let P' be any (Magic) program in which p^{bf} appears. Let the following be the only rule defining p^{bf} in P', where h is a new EDB predicate:

$$p^{bf}(X,Y) :- h(X,Y).$$

We conjecture that if p^{bf} is factorable in P', then it is also factorable in $P' \cup P_1$. What programs (defining p^{bf}) have this property? For instance, it is clear that P_2 does not. □

7.4 Order of Deleting Rules and Literals

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?

8 Conclusions

We have presented a technique for logic program optimization called *factoring.* It is a simple transformation that, when applicable, essentially allows us to compute projections of a relation instead of the actual relation. We have identified sufficient conditions under which this transformation can be used in conjunction with the Magic Sets transformation. One contribution of this work is that it allows a unified treatment of several previously proposed optimization techniques for particular classes of programs; we have shown that these earlier results can be understood as special cases of the Magic Sets / factoring approach. Our results also offer new insight into the relationship between Magic Sets and Counting. There are several promising directions in which this research can be extended, and we expect the results to be of both practical and theoretical interest.

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, Cambridge, 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.

[Llo87] John W. Lloyd. *Foundations of Logic Programming, 2nd Edition.* Springer-Verlag, 1987.

[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.

[Nau88a] Jeffrey F. Naughton. Compiling separable recursions. In *Proceedings of the SIGMOD International Symposium on Management of Data*, pages 312–319, Chicago, Illinois, May 1988.

[Nau88b] Jeffrey F. Naughton. Compiling separable recursions. Technical Report CS-TR-140-88, Princeton University, 1988.

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

[Ram88] 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.

[Sag88] Yehoshua Sagiv. Optimizing datalog programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 659–698, Los Altos, California, 94022, 1988. Morgan Kaufmann.

[Shm87] Oded Shmueli. Decidability and expressiveness aspects of logic queries. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 237–249, San Diego, California, 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.

[vEK76] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, October 1976.