Factoring Augmented Regular Chain Programs

Peter T. Wood
Department of Computer Science
University of Cape Town, Rondebosch 7700, South Africa

Abstract

In previous papers we have proposed a graphical query language for expressing traversal recursions in labelled, directed graphs. A fundamental feature of the language is the use of regular expressions to specify constraints on paths in these graphs. When only constants are allowed in regular expressions, it has been shown that these queries can be evaluated efficiently. In this paper, we study the inclusion of variables in regular expressions. We show that efficient evaluation algorithms still exist, and in so doing provide a translation to a class of Datalog programs, the ***augmented regular chain programs***, which can always be factored. This class of programs is incomparable to previously identified classes of factorable programs.

1 Introduction

The efficient evaluation of recursive queries on relational databases remains a topic of much interest, most of the focus having been on the query language Datalog [Ullm85]. One of the approaches has been to concentrate on subclasses of Datalog programs for which efficient evaluation can be guaranteed; this has given rise to, among others, the *separable* programs [Naug88], the *right-*, *left-* and *combined-linear* programs [NRSU89a], the *commutative* programs [Ioan89], and the *factorable* programs [NRSU89b].

Proceedings of the 16th VLDB Conference
Brisbane, Australia 1990

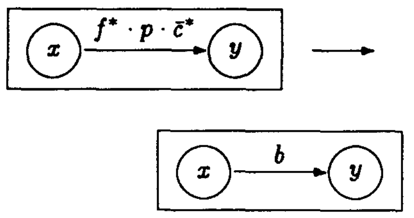

Figure 1: Query to find what products people buy.

At the same time, we have been involved in the development of a graph-based query language [CMW87,CMW88,MW89], the assumption being that most of the recursive queries that occur in practice can be modelled naturally in terms of graph traversals. Another reason for the graph-based approach is again motivated by the search for efficient evaluation algorithms.

Example 1.1 Consider the following example taken from [Naug88]. Assume there is a relation $p(X,Y)$ of people X and products Y such that Y is perfect for X, as well as relations $f(X,Y)$ of people X and their friends Y and $c(X,Y)$ of products X and Y such that X is cheaper than Y. Suppose that a person will buy a product that is perfect for them, or if their friend has bought it, or if it is cheaper than another product they will buy. Then the following program P defines a relation $b(X,Y)$ of people X and the products Y they buy.

$$\begin{aligned} b(X,Y) &\ :-\ f(X,Z), b(Z,Y). \\ b(X,Y) &\ :-\ b(X,Z), c(Y,Z). \\ b(X,Y) &\ :-\ p(X,Y). \end{aligned}$$

Program P is separable and factorable. A graph-based query Q equivalent to P is shown in Figure 1. The query Q can be interpreted as follows. The relations p, f and c can be modelled as a single directed labelled graph G in which each tuple from a relation corresponds to an edge in G labelled with the appropriate relation name. Query Q asks for pairs of nodes in G that are connected by paths whose concatenation

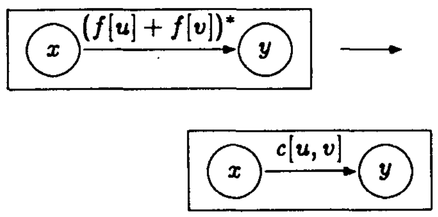

Figure 2: Query to find cities connected by sequences of flights with at most two airlines.

of edge labels is a string in the language denoted by the regular expression R appearing in Q. The symbol $\bar{c}$ in R means that edges labelled c in G are to be traversed from head to tail rather than from tail to head. □

In this paper, we show that graph-based queries can indeed be evaluated efficiently by transforming them to a class of Datalog programs which we call the *augmented regular chain programs* (*ARC-programs*). As their name implies, these programs are related to the *regular chain programs* studied in [BKBR87], for example, where it was shown that a chain program is factorable if and only if it is regular.

It turns out that ARC-programs can also always be factored—thereby admitting efficient evaluation algorithms—although the class of ARC-programs is incomparable to the class of factorable programs identified in [NRSU89b]. One of the reasons for this is that ARC-programs may contain mutually recursive rules.

Example 1.2 Suppose we have a relation $f(X,Y,Z)$ of cities X and Y such that there is a flight with airline Z between X and Y. The graph query Q depicted in Figure 2 asks for the pairs of cities that are connected by sequences of flights using at most two airlines. The symbols u and v in the regular expression R in Q are variables that are instantiated to airline names. The use of such variables adds considerable power to the query language, an issue that is discussed further in Section 3. An ARC-program P equivalent to Q—which can be generated automatically (see Section 4)—is given below.

$$\begin{array}{lll} c(X,Y,U,V) & :- & f(X,Z,U), c(Z,Y,U,V). \\ c(X,Y,U,V) & :- & f(X,Z,V), c(Z,Y,U,V). \\ c(X,Y,U,V) & :- & f(X,Y,U), f(\_,\_,V). \\ c(X,Y,U,V) & :- & f(X,Y,V), f(\_,\_,U). \end{array}$$

Program P is not separable since, in the first two rules, the sets of argument positions of c in the body sharing variables with f are neither equal nor disjoint [Naug88]. Given query goal $c(x_0,Y,U,V)$ with the first argument bound, P is not right-linear according to the definition in [NRSU89a], since, for example, in the first rule, U appears in f as well as in both occurrences of c. However, the first two rules of P commute [Ioan89], and P along with $c(x_0,Y,U,V)$ is right-linear according to [NRSU89b], and hence factorable. Later examples introduce ARC-programs that are not in any of the classes considered in [NRSU89a,NRSU89b], yet can nevertheless be factored using the methods in [NRSU89a]. □

The outline of the rest of this paper is as follows. In the next section, we define the graph-based query language GRE and describe how it differs from our previous languages. Section 3 is devoted to the translation of GRE queries to ARC-programs. We also demonstrate that an alternative translation scheme does not lead to factorable programs. In Section 4, we show that, in the presence of constants in queries, these ARC-programs can always be factored, thus admitting efficient evaluation algorithms. Finally, conclusions and directions for future work are discussed in Section 5.

2 The GRE Query Language

In this section, we define the syntax and semantics of the GRE query language. We begin by defining the graph structures against which such queries are formulated.

Definition 2.1 Let U be a set of *constants*, V be a set of *variables*, and P be a set of *predicate symbols*. A *literal* is of the form $p[a_1,\ldots,a_n]$, $n \geq 0$, (a *forward* literal) or $\bar{p}[a_1,\ldots,a_n]$, $n \geq 0$, (a *reverse* literal), where p is an n-ary predicate symbol and each $a_i \in U \cup V$, $1 \leq i \leq n$. If each $a_i \in U$, $1 \leq i \leq n$, then the literal is *ground*. We denote the set of forward ground literals over P and U by $F(P,U)$.

A *database graph* (*db-graph*, for short) $G = (N,E,\psi,U,P,\nu,\lambda)$ is a directed, labelled graph, where N is a set of *nodes*, E is a set of *edges*, and ψ is an *incidence function* mapping E to $N \times N$. The *node labelling function* ν associates with each node $x \in N$ a distinct node label $\nu(x) \in U$, while λ is an *edge labelling function* mapping E to $F(P,U)$, the set of forward ground literals. □

There is an obvious way in which a db-graph G can be viewed as a set R of relations. For each n-ary predicate symbol r labelling an edge in G, there is an $(n+2)$-ary relation r in R. The tuple $(x,y,a_1,\ldots,a_n)$

appears in relation r if and only if there is an edge labelled $r[a_1, \ldots, a_n]$ from the node with label x to the node with label y in G. We say that the set of relations R *represents* the db-graph G.

Next, we turn our attention to the syntax of GRE queries, examples of which have already been presented in Figures 1 and 2.

Definition 2.2 A *GRE query* Q is a pair (H, S), where H is a *pattern graph* and S is a *summary graph*. Both H and S are labelled directed graphs comprising exactly two nodes connected by an edge[1]. Let the sets P, U and V be defined as above, and denote the set of literals over P, U and V by $T(P, U, V)$. Then the node labels of H are drawn from $U \cup V$, while the edge label is a *regular expression*, which is defined in the usual way [HU79], except that its alphabet is $T(P, U, V)$. The edge label of S is a forward literal subject to the restriction that every variable that appears in S must also appear in H. □

Example 2.1 Consider the GRE queries Q_1 and Q_2 shown in Figures 1 and 2, respectively. Pattern graphs appear in the left-hand boxes, while summary graphs appear in the right-hand boxes. In each case, the variable x in the pattern graph is called the *source* variable and y the *sink* variable. The predicate symbols of Q_1 are f, p, c and b, all of which are 0-ary. Literal $\bar{c}$ is a reverse literal, while f, for example, is a forward literal. In the pattern graph of Q_2, f is the only predicate symbol and it is unary, giving rise to the literals $f[u]$ and $f[v]$. □

The meaning of a GRE query applied to a db-graph is given by the following definitions.

Definition 2.3 Let $G = (N, E, \psi, U, P, \nu, \lambda)$ be a db-graph and $p = (v_1, e_1, \ldots, e_{n-1}, v_n)$, $n \geq 2$, where $v_i \in N$, $1 \leq i \leq n$, and $e_j \in E$, $1 \leq j \leq n-1$, be a path in G in which edge directions are ignored. We call such a path an *undirected* path and the string $\lambda(e_1) \cdots \lambda(e_{n-1})$ the *path label* of p, denoted by $\lambda(p)$. Let $Q = (H, S)$ be a GRE query with source label x, sink label y, regular expression R, and set of variables V. A *valuation* θ is a mapping from $U \cup V$ to U such that if $u \in U$ then $\theta(u) = u$. We denote by $\theta(R)$ and $\theta(S)$ the application of θ to all variables in R and S, respectively. We say that the path p in G *satisfies* $\theta(R)$ if

1. $\lambda(p)$ *matches* a string in $L(\theta(R))$, the language denoted by $\theta(R)$, where "matches" means "identical to" except that predicate symbol s matches $\bar{s}$, and
2. if $\lambda(e_i)$ matches a forward literal then e_i is from v_i to v_{i+1}, or if $\lambda(e_i)$ matches a reverse literal then e_i is from v_{i+1} to v_i.

In addition, if $\nu(v_1) = \theta(x)$ and $\nu(v_n) = \theta(y)$, then we say that p *satisfies* $\theta(H)$. The *query* Q on db-graph G is defined as

$$\bigcup \{\theta(S) \mid \theta \text{ is a valuation of } H \text{ and there is an undirected path in } G \text{ satisfing } \theta(H)\}.$$

□

The above definitions of GRE differ in a number of ways from our previous graph-based query language G$^+$ [CMW87,CMW88]. Among the most significant of these are that edges in db-graphs and graph queries can now effectively be typed, and that the semantics are defined in terms of matching arbitrary paths in db-graphs rather than *simple* paths, a restriction that makes the query evaluation problem NP-complete [MW89]. In these respects, GRE shares similarities with another modification of G$^+$ called GraphLog [CM90].

For simplicity we assume from now on that there is only a single unary predicate symbol labelling edges in any db-graph, which can therefore be represented by a single ternary relation. Because of this, we will usually drop all occurrences of the predicate symbol from any regular expression, whose alphabet will thus comprise only constants and variables.

3 From GRE-Queries to ARC-Programs

In this section, we demonstrate how to translate a GRE query into a Datalog program such that the resulting program can be evaluated efficiently. Even though this program may contain mutually recursive rules, we show in the next section that every recursive predicate in the program can be factored, a result that is not true of an alternative translation scheme also presented below.

Example 3.1 Consider the GRE query Q of Example 1.2 in which the regular expression R is $(f[u] + f[v])^*$. The translation to an ARC-program proceeds in two stages. First we construct a regular chain program[2] [BKBR87] from Q using the standard procedure for generating a regular grammar from the regular expression R [HU79].

[1]More powerful versions of the language allow more general pattern graphs.

[2]In fact, the programs do not always conform exactly to the definition of regular chain programs in that base predicates may have additional arguments.

$$\begin{aligned} c(X,Y) &\ :-\ f(X,Z,U), c(Z,Y). \\ c(X,Y) &\ :-\ f(X,Z,V), c(Z,Y). \\ c(X,Y) &\ :-\ f(X,Y,U). \\ c(X,Y) &\ :-\ f(X,Y,V). \end{aligned}$$

In the second stage, we augment the regular chain program by propagating the variables of R through the program in a bottom-up manner. If the head H of a rule corresponds to the start symbol of the regular grammar, then all the variables from R must appear in H. Since the resulting rule may not be safe, we may have to add literals to the right-hand side in order to provide bindings for all the variables of R.

$$\begin{aligned} r_1: \ c(X,Y,U,V) &\ :-\ f(X,Z,U), \\ &\qquad c(Z,Y,U,V). \\ r_2: \ c(X,Y,U,V) &\ :-\ f(X,Z,V), \\ &\qquad c(Z,Y,U,V). \\ r_3: \ c(X,Y,U,V) &\ :-\ f(X,Y,U), f(\_,\_,V). \\ r_4: \ c(X,Y,U,V) &\ :-\ f(X,Y,V), f(\_,\_,U). \end{aligned}$$

Since c corresponds to the starting symbol of the associated grammar, both U and V appear in the head of rules r_3 and r_4. In order to provide bindings for U and V in r_3 and r_4, additional f literals are included as shown. Now, U and V are propagated first to the bodies and then to the heads of r_1 and r_2. The result is a safe ARC-program. □

Perhaps an intuitively more appealing translation would be (1) to construct an expression tree from parsing R, and (2) to generate a Datalog program by assigning IDB predicates to the interior nodes of the tree and interpreting $\cdot$ as join, $+$ as union, and $*$ as transitive closure. For the above example, this would give rise to the following program.

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

Although the above method works well in this case, it does not always lead to factorable programs. Consider the regular expression $R = (v_1 \cdot v_2^* \cdot v_3)^*$ as part of a GRE query on a db-graph labelled with predicate symbol r. The following Datalog program P is that constructed from the expression tree of R.

$$\begin{aligned} q(X,Y,V_1,V_2,V_3) &\ :-\ p(X,Y,V_1,V_2,V_3). \\ q(X,Y,V_1,V_2,V_3) &\ :-\ p(X,Z,V_1,V_2,V_3), \\ &\qquad q(Z,Y,V_1,V_2,V_3). \\ p(X,Y,V_1,V_2,V_3) &\ :-\ s(X,Z,V_1,V_2), \\ &\qquad r(Z,Y,V_3). \\ s(X,Y,V_1,V_2) &\ :-\ r(X,Z,V_1), t(Z,Y,V_2). \\ t(X,Y,V_2) &\ :-\ r(X,Z,V_2), t(Z,Y,V_2). \\ t(X,X,V_2) &\ :-\ r(\_,\_,V_2). \end{aligned}$$

Although the last rule is not safe, this can easily be remedied and does not alter the point being made. It turns out that program P cannot be factored with respect to either of the recursive predicates q and t if X or Y is bound to a constant in query $q(X,Y,V_1,V_2,V_3)$. In contrast, our translation yields a program in which all recursive predicates *can* be factored simultaneously.

Our general translation scheme is given below. For simplicity of exposition, we assume that only variables appear in regular expressions, and that all such variables also label the edge in the summary graph. We comment on relaxing these and other restrictions at the end of the section.

Assume we are given a GRE query $Q = (H, S)$ comprising source variable x, sink variable y, and regular expression R containing variables $\Sigma = \{v_1, \ldots, v_n\}$. Furthermore, assume that Q is to be applied to a db-graph labelled with predicate symbol r, and that the literal labelling the edge in S is $b[v_1, \ldots, v_n]$. The translation from Q to an ARC-program P proceeds in three stages.

1. Construct a regular grammar $W = (\Sigma, N, D, B)$ for generating $L(R) - \{\epsilon\}$[3], where Σ is the set of variables in R, N the set of nonterminals, D the set of productions, and B the start symbol, corresponding to the predicate symbol labelling the edge in S.

2. For each production C in D, generate a Datalog rule as follows:

   (a) if C is of the form $T \rightarrow vU$, generate the rule
   $$t(X,Y)\ :-\ r(X,Z,V), u(Z,Y).$$

   (b) if C is of the form $T \rightarrow v$, generate the rule
   $$t(X,Y)\ :-\ r(X,Y,V).$$

   Call the resulting program P.

3. From P generate a new program Q by performing a bottom-up propagation of the regular expression variables as follows.

   (a) For each base rule in P of the form
   $$t(X,Y)\ :-\ r(X,Y,V_1).$$
   add the rule
   $$t(X,Y,V_1)\ :-\ r(X,Y,V_1).$$
   to Q.

[3]This is because we are interested in paths of non-zero length only.

(b) Repeat the following process until no (syntactically) new rule is added to Q. If there is a rule with head $t(X, Y, \overline{V})$ in Q and a rule of the form

$$s(X,Y) \ :- \ r(X,Z,U), t(Z,Y).$$

in P, then add the rule

$$s(X,Y,\overline{W}) \ :- \ r(X,Z,U), t(Z,Y,\overline{V}).$$

to Q, where $\overline{W}$ is a tuple including both U and all of $\overline{V}$ such that the ordering of variables is consistent with that in the summary S.

If, in either of the above cases, the head of the rule corresponds to the start symbol B, then

(a) include all the variables of R in the head, and

(b) if the rule is now unsafe because variable U does not appear in the body, add the literal $f(\_,\_,U)$ to the body of the rule.

Such a rule is called a *starting* rule.

In generating the program Q, we may use the same predicate symbol to denote relations of differing degrees. This is done to make the derivation clearer, on the understanding that unique symbols could always be used instead.

Example 3.2 Consider the regular expression $R = (v_1 \cdot v_2)^*$. We first construct a regular chain program P for R.

$$\begin{aligned} s(X,Y) &\ :- \ r(X,Z,V_1), t(Z,Y). \\ t(X,Y) &\ :- \ r(X,Z,V_2), s(Z,Y). \\ t(X,Y) &\ :- \ r(X,Y,V_2). \end{aligned}$$

Note that P contains mutually recursive predicates s and t. Next, we augment P by propagating the bindings for the regular expression variables in a bottom-up manner. The ARC-program Q generated from P is as follows.

$$\begin{aligned} s(X,Y,V_1,V_2) &\ :- \ r(X,Z,V_1), t(Z,Y,V_1,V_2). \\ t(X,Y,V_1,V_2) &\ :- \ r(X,Z,V_2), s(Z,Y,V_1,V_2). \\ s(X,Y,V_1,V_2) &\ :- \ r(X,Z,V_1), t(Z,Y,V_2). \\ t(X,Y,V_2) &\ :- \ r(X,Y,V_2). \end{aligned}$$

Note that t is of degree both three and four. Instead of the last two rules we could have generated the rule

$$t(X,Y,V_1,V_2) \ :- \ r(X,Z,V_2), r(\_,\_,V_1).$$

However, since t does not correspond to the start symbol of the grammar, it might be inefficient to do so—the corresponding tuples derived may not contribute to any answers. □

It can be shown that the above translation scheme is correct in the sense that, given a GRE query Q and db-graph G, if program P is generated from Q and relation r represents G, then $P(r)$ represents $Q(G)$.

Theorem 3.1 *Let Q be a GRE query and P be the ARC-program generated from Q. Then Q and P represent equivalent queries.*

Proof: The proof is similar to that given for translating graph-based queries to regular chain programs given in [Wood88]. □

It is a simple matter to extend the translation process to account for multiple predicate symbols in db-graphs and GRE queries, giving rise to ARC-programs containing more than one base predicate. Also, if certain variables in a regular expression do not appear in the summary graph of query Q, they can simply be projected out of the heads of starting rules in P.

4 Factoring ARC-Programs

We now turn our attention to properties of ARC-programs, in particular their efficient evaluation in the presence of selection constants in a query. In this section we show that for such queries the methods presented in [NRSU89a] can be extended to apply to the factoring of ARC-programs as well.

In general, factoring refers to the process of replacing a recursive predicate $p(\overline{X}, \overline{Y})$ by predicates $bp(\overline{X})$ (the *bound* part of p) and $fp(\overline{Y})$ (the *free* part of p) [NRSU89b]. Our use of the term is more specific and refers to the reduction of the degree of p by replacing it by fp, since the bound parts of predicates can always be deleted from ARC-programs. As a result, the more general factoring techniques of [NRSU89b] are not necessary for ARC-programs and those of [NRSU89a] suffice.

The programs considered in [NRSU89b] are restricted to *RLC-stable* programs: those containing only right-linear, left-linear and combined-linear rules in terms of a single IDB predicate and one exit rule[4]. We have already seen that ARC-programs can contain multiple exit rules (Example 3.1), as well as more than one IDB predicate and mutually recursive rules (Example 3.2). Thus, there are ARC-programs that are not RLC-stable. On the other hand, ARC-programs do not contain combined-linear rules, so there are RLC-stable programs that are not ARC-programs.

[4]There are other restrictions as well.

Consider a typical ARC-program starting rule such as

$$s(X,Y,\overline{V}) \quad :- \quad r(X,Z,V_1), t(Z,Y,\overline{V}).$$

Recall that X is the source variable, Y is the sink variable, and the variables in $\overline{V}$ are the regular expression variables. However, in ARC-programs what is more important is the means by which bindings are propagated in rules. In this respect, the sink variable and regular expression variables play similar roles in that they appear in the same position in both the head of each rule and the IDB predicate in the body. Borrowing terminology from [Ioan89], we will refer to these variables collectively as *persistent* variables[5]. There are two cases to consider, corresponding to whether source or persistent variables are bound in a query to an ARC-program.

4.1 Bound Persistent Variables

If any persistent variables are bound to constants in a query to an ARC-program P, we simply substitute the constants for the corresponding variables in all base predicates in P and delete the variables wherever else they appear in P.

Example 4.1 Consider the ARC-program of Example 3.2. If we assume that Y is bound to y_0 and V_2 is bound to v_0 in the query $s(X,Y,V_1,V_2)$, then the factored program is as follows.

$$\begin{array}{rcl} s(X,V_1) & :- & r(X,Z,V_1), t(Z,V_1). \\ t(X,V_1) & :- & r(X,Z,v_0), s(Z,V_1). \\ s(X,V_1) & :- & r(X,Z,V_1), t(Z). \\ t(X) & :- & r(X,y_0,v_0). \end{array}$$

All occurrences of Y and V_2 in IDB predicates have been deleted, while all occurrences of Y and V_2 in base predicates have been replaced by y_0 and v_0, respectively. □

Let the query to the ARC-program be given by $q(X,\overline{U},\overline{W})$, the tuple of persistent variables being $\overline{U},\overline{W}$, where $\overline{W} = W_1,\ldots,W_m$. Without loss of generality, assume that W_i is bound to w_i, $1 \le i \le m$, in the query, and let θ be the substitution that replaces each W_i by w_i, $1 \le i \le m$. Let $\overline{V} - \overline{W}$ denote the removal of all variables in $\overline{W}$ from $\overline{V}$, that is, the *reduction* of $\overline{V}$ by $\overline{W}$. The general method is as follows.

1. Given a base rule of the form

$$t(X,\overline{V}) \quad :- \quad \mathcal{R}_1,\ldots,\mathcal{R}_k.$$

where $\mathcal{R}_1,\ldots,\mathcal{R}_k$ are r (EDB) literals, transform it to

$$t(X,\overline{V}-\overline{W}) \quad :- \quad \theta(\mathcal{R}_1),\ldots,\theta(\mathcal{R}_k).$$

2. Given a non-base rule of the form

$$t(X,\overline{T}) \quad :- \quad r(X,Z,V), s(Z,\overline{S}).$$

where all variables in $\overline{T}$, $\overline{S}$ and V appear in $\overline{U}$ or $\overline{W}$, transform it to

$$t(X,\overline{T}-\overline{W}) \quad :- \quad \theta(r(X,Z,V)), s(Z,\overline{S}-\overline{W}).$$

The query $q(X,\overline{U})$ is now applied to the transformed program.

Theorem 4.1 *For a given query, the factored ARC-programs produce the same answer as the original programs and are no less efficient.*

In fact, as shown in [NRSU89a,NRSU89b], factored programs can lead to an order of magnitude improvement in terms of evaluation efficiency.

4.2 Bound Source Variables

If the source variable X in a query $q(X,Y,\overline{V})$ is bound, we apply a transformation based on that of Magic Sets [BMSU86,BR87], similar to the technique in [NRSU89a]. The first step in such a transformation is the top-down propagation of the binding patterns through a program P, leading to an *adorned* program P^{ad} [Ullm85], in which each IDB predicate p has an adornment α indicating which arguments of p are bound and which are free. For example, p^{bf} means that the first argument of p is bound while the second is free. In certain circumstances, the adorned programs that we derive differ from the classical versions in that they exploit properties of ARC-programs.

The second step in the transformation is to derive the set of magic rules for P^{ad}. In the final step, magic predicates are introduced into the base rules (those containing only EDB predicates) of P^{ad}. Only the magic rules and these modified base rules are used to answer the original query. We present an example before describing the general method.

Example 4.2 Consider again the program of Example 3.1. If we assume that X is bound to x_0 in the query $c(X,Y,U,V)$, then the adorned program P^{ad} contains the rules

[5]Strictly speaking, the variable V_1 is semi-persistent.

$$\begin{aligned}
c^{bfa_1a_2}(X,Y,U,V) &\ :-\ f(X,Z,U),\\
&\qquad c^{bfba_2}(Z,Y,U,V).\\
c^{bfa_1a_2}(X,Y,U,V) &\ :-\ f(X,Z,V),\\
&\qquad c^{bfa_1b}(Z,Y,U,V).\\
c^{bfa_1a_2}(X,Y,U,V) &\ :-\ f(X,Y,U), f(\_,\_,V).\\
c^{bfa_1a_2}(X,Y,U,V) &\ :-\ f(X,Y,V), f(\_,\_,U).
\end{aligned}$$

for $a_1 = b$ or f, and $a_2 = b$ or f. In other words, there are 16 rules in P^{ad} representing all possible binding patterns for the variables U and V. Now the Magic Sets transformation yields the following set of magic rules (where m rather than m_c is the magic predicate for c).

$$\begin{aligned}
m^{bfff}(x_0)\\
m^{bfbf}(Z,U) &\ :-\ m^{bfff}(X), f(X,Z,U).\\
m^{bffb}(Z,V) &\ :-\ m^{bfff}(X), f(X,Z,V).\\
m^{bfbf}(Z,U) &\ :-\ m^{bfbf}(X,U), f(X,Z,U).\\
m^{bffb}(Z,V) &\ :-\ m^{bffb}(X,V), f(X,Z,V).\\
m^{bfbb}(Z,U,V) &\ :-\ m^{bfbf}(X,U), f(X,Z,V).\\
m^{bfbb}(Z,U,V) &\ :-\ m^{bffb}(X,V), f(X,Z,U).\\
m^{bfbb}(Z,U,V) &\ :-\ m^{bfbb}(X,U,V), f(X,Z,U).\\
m^{bfbb}(Z,U,V) &\ :-\ m^{bfbb}(X,U,V), f(X,Z,V).
\end{aligned}$$

Once again, we have permitted the same predicate symbol to denote relations of differing degree. Finally, for each of the eight base rules in P^{ad} with head predicate c^{α}, we substitute the appropriate magic predicate m^{α} into the body of the rule, and drop the adornments for c in order to complete the program.

$$\begin{aligned}
c(Y,U,V) &\ :-\ m^{bfff}(X), f(X,Y,U),\\
&\qquad f(\_,\_,V).\\
c(Y,U,V) &\ :-\ m^{bfff}(X), f(X,Y,V),\\
&\qquad f(\_,\_,U).\\
c(Y,U,V) &\ :-\ m^{bfbf}(X,U), f(X,Y,U),\\
&\qquad f(\_,\_,V).\\
c(Y,U,V) &\ :-\ m^{bfbf}(X,U), f(X,Y,V),\\
&\qquad f(\_,\_,U).\\
c(Y,U,V) &\ :-\ m^{bffb}(X,V), f(X,Y,U),\\
&\qquad f(\_,\_,V).\\
c(Y,U,V) &\ :-\ m^{bffb}(X,V), f(X,Y,V),\\
&\qquad f(\_,\_,U).\\
c(Y,U,V) &\ :-\ m^{bfbb}(X,U,V), f(X,Y,U),\\
&\qquad f(\_,\_,V).\\
c(Y,U,V) &\ :-\ m^{bfbb}(X,U,V), f(X,Y,V),\\
&\qquad f(\_,\_,U).
\end{aligned}$$

Note that the degree of recursive predicates has been reduced from four to at most three. □

Before defining the method of transformation in detail, we describe how our adorned program can differ from the classical one. Consider the rule

$$s(X,Y,V_1,V_2) \ :-\ r(X,Z,V_1), t(Z,Y,V_2).$$

and the binding pattern *bfff* for s. Since, in a top-down evaluation, variable V_1 is bound by the time t is considered, we add V_1 as an argument of t to yield the adorned rule

$$\begin{aligned}
s^{bfff}(X,Y,V_1,V_2) &\ :-\ r(X,Z,V_1),\\
&\qquad t^{bfbf}(Z,Y,V_1,V_2).
\end{aligned}$$

In this way, the adorned program mirrors the way the ARC-program was constructed from the corresponding GRE query, except that bound variables are now propagated top-down rather than bottom-up. A consequence of this is that all regular expression variables appear in the head of any base rule. As a result, some base rules in the adorned program may not be safe for bottom-up evaluation, but they will always be safe for top-down evaluation.

Given an ARC-program P and query $q(X,Y,\overline{V})$ with X bound to x_0, the general method is as follows.

1. Generate the adorned program P^{ad} from P and the query, adding bound regular expression variables to IDB predicates where applicable.

2. From each non-base rule in P^{ad} of the form

   $$t^{\alpha_1}(X,Y,\overline{U}) \ :-\ r(X,Z,V), s^{\alpha_2}(Z,Y,\overline{W}).$$

   generate its magic rule by (i) prefixing both s and t with $m\_$, (ii) deleting all free variables in s and t, and (iii) exchanging m_s and m_t.

3. Generate the rule

   $$m\_q(x_0).$$

4. For each base rule in P^{ad} of the form

   $$p^{\alpha}(X,Y,\overline{V}) \ :-\ \mathcal{R}_1,\ldots,\mathcal{R}_k.$$

   where $\mathcal{R}_1,\ldots,\mathcal{R}_k$ are r (EDB) literals, generate the rule

   $$q(Y,\overline{V}) \ :-\ m\_p^{\alpha}(X,\overline{V}), \mathcal{R}_1,\ldots,\mathcal{R}_k.$$

The query $q(Y,\overline{V})$ is now applied to the generated program.

In common with [NRSU89a], the above method has the advantage over Magic Sets that magic predicates are substituted into base rules alone, the remaining rules of the original program being discarded. The method generalizes naturally to handle multiple EDB predicates.

Example 4.3 Consider again the program P of Example 3.2. If we assume that X is bound to x_0 in the query $s(X,Y,V_1,V_2)$, then the adorned program P^{ad} is as follows.

$$\begin{array}{lcl}
s^{bfff}(X,Y,V_1,V_2) & :- & r(X,Z,V_1), \\
 & & t^{bfbf}(Z,Y,V_1,V_2). \\
t^{bfbf}(X,Y,V_1,V_2) & :- & r(X,Z,V_2), \\
 & & s^{bfbb}(Z,Y,V_1,V_2). \\
s^{bfbb}(X,Y,V_1,V_2) & :- & r(X,Z,V_1), \\
 & & t^{bfbb}(Z,Y,V_1,V_2). \\
t^{bfbb}(X,Y,V_1,V_2) & :- & r(X,Z,V_2), \\
 & & s^{bfbb}(Z,Y,V_1,V_2). \\
s^{bfff}(X,Y,V_1,V_2) & :- & r(X,Z,V_1), \\
 & & t^{bfbf}(Z,Y,V_1,V_2). \\
s^{bfbb}(X,Y,V_1,V_2) & :- & r(X,Z,V_1), \\
 & & t^{bfbb}(Z,Y,V_1,V_2). \\
t^{bfbf}(X,Y,V_1,V_2) & :- & r(X,Y,V_2). \\
t^{bfbb}(X,Y,V_1,V_2) & :- & r(X,Y,V_2).
\end{array}$$

Now the magic rules and base rules of P^{ad} yield the following program.

$$\begin{array}{rcl}
m\_s^{bfff}(x_0) & & \\
m\_t^{bfbf}(Z,V_1) & :- & m\_s^{bfff}(X), r(X,Z,V_1). \\
m\_s^{bfbb}(Z,V_1,V_2) & :- & m\_t^{bfbf}(X,V_1), \\
 & & r(X,Z,V_2). \\
m\_t^{bfbb}(Z,V_1,V_2) & :- & m\_s^{bfbb}(X,V_1,V_2), \\
 & & r(X,Z,V_1). \\
m\_s^{bfbb}(Z,V_1,V_2) & :- & m\_t^{bfbb}(X,V_1,V_2), \\
 & & r(X,Z,V_2). \\
s(Y,V_1,V_2) & :- & m\_t^{bfbf}(X,V_1), \\
 & & r(X,Y,V_2). \\
s(Y,V_1,V_2) & :- & m\_t^{bfbb}(X,V_1,V_2), \\
 & & r(X,Y,V_2).
\end{array}$$

Note that *both* s and t have been factored. □

Theorem 4.2 *For a given query, the factored ARC-programs produce the same answer as the original programs and are no less efficient.*

Once again, factored programs can generally be evaluated far more efficiently than their unfactored counterparts, and can result in an order of magnitude improvement over Magic Sets.

5 Conclusions

We believe that a higher-level query language than Datalog is desirable for expressing recursive queries on relational databases. One problem with Datalog is the number of equivalent programs for expressing a query, for example, the three canonical forms for transitive closure, for which a uniform efficient evaluation algorithm has only recently been proposed [NRSU89a]. Another problem is that, although Datalog programs may be read declaratively, they often seem to contain unnecessary verbosity, in particular with respect to the number of variables in rules.

It is our contention that the GRE query language overcomes some of these limitations of Datalog. While it is not a general purpose query language, GRE is most suitable when the query can be viewed naturally as a graph traversal, a common feature of recursive queries. In addition, we have shown that an efficient evaluation algorithm exists for GRE queries by providing a translation to a subclass of Datalog programs, the ARC-programs. These programs are interesting in their own right since we have shown that they can always be factored in the presence of single-selection queries, yet are incomparable to previously identified classes of factorable programs.

ARC-programs deserve further study. On the one hand, because of their close relationship with regular expressions there are obvious strategies for optimizing them. For example, an ARC-program corresponding to the regular expression $(u^* \cdot v^*)^*$ could be transformed into an equivalent program corresponding to the expression $(u + v)^*$. Although such transformations cannot in general be done efficiently, the potential payoff in reducing the number of recursive rules is large.

Another area for research is to try to establish whether the class of ARC-programs can be integrated in any way with previously discovered classes of factorable programs, thereby identifying a strictly broader class of factorable programs.

Acknowledgements

This work was supported by an award from the Foundation for Research Development. The comments of one referee in particular helped improve the paper significantly.

References

[BMSU86] F. Bancilhon, D. Maier, Y. Sagiv and J.D. Ullman, "Magic Sets and Other Strange Ways To Implement Logic Programs," *Proc. 5th ACM Symp. on Principles of Database Systems*, 1986, pp. 1–15.

[BKBR87] C. Beeri, P. Kanellakis, F. Bancilhon and R. Ramakrishnan, "Bounds on the Propagation of Selection into Logic Programs," *Proc. 6th ACM Symp. on Principles of Database Systems*, 1987, pp. 214–226.

[BR87] C. Beeri and R. Ramakrishnan, "On the Power of Magic," *Proc. 6th ACM Symp.*

on Principles of Database Systems, 1987, pp. 269–283.

[CM90] M.P. Consens and A.O. Mendelzon, "GraphLog: A Visual Formalism for Real Life Recursion," *Proc. 9th ACM Symp. on Principles of Database Systems*, 1990.

[CMW87] I.F. Cruz, A.O. Mendelzon, and P.T. Wood, "A Graphical Query Language Supporting Recursion," *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1987, pp. 323–330.

[CMW88] I.F. Cruz, A.O. Mendelzon, and P.T. Wood, "G^+: Recursive Queries Without Recursion," *Proc. 2nd Int. Conf. on Expert Database Systems*, 1988, pp. 355–368.

[HU79] J.E. Hopcroft and J.D. Ullman, "*Introduction to Automata Theory, Languages, and Computation*," Addison-Wesley, 1979.

[Ioan89] Y.E. Ioannidis, "Commutativity and its Role in the Processing of Linear Recursion," *Proc. 15th Int. Conf. on Very Large Data Bases*, 1989, pp. 155–163.

[MW89] A.O. Mendelzon and P.T. Wood, "Finding Regular Simple Paths in Graph Databases," *Proc. 15th Int. Conf. on Very Large Data Bases*, 1989, pp. 185–193.

[Naug88] J.F. Naughton, "Compiling Separable Recursions," *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1988, pp. 312–319.

[NRSU89a] J.F. Naughton, R. Ramakrishnan, Y. Sagiv and J.D. Ullman, "Efficient Evaluation of right-, left-, and combined-linear rules," *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1989, pp. 235–242.

[NRSU89b] J.F. Naughton, R. Ramakrishnan, Y. Sagiv and J.D. Ullman, "Argument Reduction by Factoring," *Proc. 15th Int. Conf. on Very Large Data Bases*, 1989, pp. 173–182.

[Ullm85] J.D. Ullman, "Implementation of Logical Query Languages for Databases," *ACM Trans. on Database Syst. 10*, 3 (Sept. 1985), pp. 289–321.

[Wood88] P.T. Wood, "Queries on Graphs," Ph.D. thesis, Tech. Report CSRI-223, Univ. of Toronto, Toronto, Ont. Canada, 1988.