In commenting on the cross section of computer science research in South Africa, I will use the classification in the table of contents of the "Summary of Awards: Fiscal Year 1989," a document published recently by the US National Science Foundation. Of the 5 categories, I will treat Numeric and Symbolic Computation as inappropriate for the discussion below. In this category I noted no research in the computer science setting in South Africa. It is also common in the US and elsewhere to place this effort in other departments, e.g., departments of mathematics or applied mathematics.

Of the remaining categories I found South Africa to be strongest in software systems and engineering, to have a substantial investment in computer systems and architecture, and to be weakest in computer and computation theory.

The coverage in software systems and engineering (SSE) was broad, topical, and similar in scope to that in US universities. Technology transfer and the corresponding relations with industry seemed to be in place or developing along promising lines. I comment in passing that this was rather surprising to me. In the US the development of SSE within university departments has lagged behind almost all other disciplines of computer science. A primary problem has been the insatiable appetite of industry for all Ph.D. graduates in the SSE field.

The investment in parallel processing, computer networks, and distributed computing appears sound, although I expected to see a greater emphasis on mathematical foundations (see my remarks below), particularly in the parallel algorithms area. Given current resources, South African institutions are doing remarkably well in computer science research. But computer science is a fundamentally important course of study, beginning at an early age and extending through graduate Ph.D. research; I take this as sufficiently obvious that I need not dwell on justifications. With this in mind, and with the necessary resources in hand, South Africa should, in my opinion, expand and consolidate its computer science research effort, increase its visibility in the international arena, and correct the rather thin distribution of graduate research among universities.

I can see much of this proceeding along present lines, but I would strongly recommend a concerted development in computer and computation theory (CCT), education and research; this is mainstream computer science and forms the basis for virtually all other fields of study within computer science. It is by no means absent in South Africa curricula, but it appears to be under-represented in advanced studies and Ph.D. level research.

At the graduate level CCT is heavily mathematical. I understand that mathematical foundations are supplied by mathematics departments in certain cases. This is not ideal, but workable and it is justified by limited resources. However, it is important that mathematics departments not regard this as a mere service; faculty will have to make a major commitment to theoretical computer science, publishing in its leading journals (e.g. SIAM Journal of Computing, Journal of the ACM, Journal of Algorithms, Algorithmica, Journal of Computer and Systems Sciences, Theoretical Computer Science, etc.), and providing the supervision of theses sponsored by computer science departments and leading to degrees in computer science. I would also encourage active participation in the international computer science "theory" societies and their meetings; two highly prestigious examples of the latter are the annual Symposium on the Theory of Computing and the Foundations of Computer Science conference.

Returning to the thin distribution of computer science research, I would make the following point. If the current situation is only a stage of development - i.e., if further resources (both human and financial) can be counted on to bring at least a few of the departments to a critical mass - then little needs to be said beyond the earlier remarks. Critical mass is hard to define, but calls for adequate, expert coverage of mainstream computer science research. In view of the breadth of this research, 8-10 Ph.D. full-time-equivalent faculty would seem to be barely adequate; with the usual clumping of faculty in specific research areas, more would be expected. South Africa has a talent base such that there is little doubt that such departments would achieve a much wider international recognition.
On the other hand, if resources remain fixed at current or even slightly retrenched levels, then I would recommend consolidation to achieve the same goals on a smaller scale. Within a university this can often be done by establishing interdisciplinary, degree-granting laboratories or institutes of computer science, which bring together the computer science efforts located in various departments other than computer science, such as electrical engineering, industrial engineering, business/management science, mathematics, and operations research. The idea is to enjoy the advantages (opportunity, synergy, awareness, etc.) to both students and faculty of reasonably large computer science programs. There are many examples of such intramural laboratories in North America and Europe.

This approach could also be considered among universities within a confined geographical area, admittedly with greater difficulty perhaps. The Institute of Discrete Mathematics and Computer Science connecting Princeton University, Rutgers University, AT&T Bell Laboratories, and Bell Communications Research is a possible model. Examples in South Africa might consist of universities and research institutions on the Reef or those in the Western Cape (just to mention those with which I'm a little familiar).

As a final comment, I should note that my impressions have been based on limited information which may not give a representative picture. I am sure that my reactions will be appropriately discounted where I have been off target.

---

Editor’s Notes

Prof John Schochot has graciously accepted to be SACJ’s subeditor for papers relating to Information Systems. Authors wishing to submit papers in this general area should please contact him directly. I look forward working with John, and to a significant increase in IS contributions in future.

The hand of the new production editor, Riëls Smit, will be clearly evident in this issue. Those papers not prepared in camera-ready format by the authors themselves were prepared by him in TEX. He will be announcing revised guidelines for camera-ready format in a future issue. If you use TEX or one of its variations, Riëls would be happy to provide you with a styles document to SACJ format.

At last some Department of National Education committee has decided that SACJ should now be on the list of approved journals. This places it amongst the ranks of some 6800 other journals. These include not merely a number of ACM and IEEE Transactions but also such journals as Ostrich, Trivium, Crane Bag, Koers, Mosquito News, Police Chief, Connoisseur, Lion and the Unicorn, About the House and Ohio Agricultural Research and Development Center Department Series ESS. You will recall that in 1990 this same committee decided that, if judged on its own merits, SACJ did not deserve to be on the illustrious list. In the absence of other evidence, we must assume that the sole reason for its revised decision is that SACJ’s predecessor, Questions Informaticæ, was there. (I have a secret suspicion that the committee liked that name.)

It is my understanding that for official purposes, all journals on this list are regarded as equally meritorious, and all of them are more meritorious than any conference proceedings. What does all of this mean?

The momentous implication of the committee’s deliberations is that the State will not give your institution a single cent for anything that you publish in SACJ. Instead, the State and your institution will scrupulously keep a score of the annual number of publications that count - but actually don’t - because someday they might! And to encourage your enthusiastic participation in this Alice in Wonderland exercise, your institution might actually give you some of the standard subsidy funding that the State should have provided according to its own formulae, but didn’t.

You will not be allowed to use this money to buy yourself a car - not even a casual meal. You may only use it to finance activities that are provably directed towards producing more papers in approved journals. The great consolation, of course, is that you will not be required to pay income tax on this money. The only tax involved will be the VAT component when you spend it in an approved manner. As a good computer scientist who enjoys recursion, my vote would be that all such revenue collected by the State should be earmarked to be placed in the pay packets of committee members who decided that SACJ should be approved.

If you publish in these approved journals with sufficient regularity and enthusiasm you will almost deserve to be regarded as a researcher. What you additionally need to do, is to ensure that you befriend and impress at least three overseas referees. You then apply to the FRD for official recognition as a researcher, and if they are sufficiently impressed, they will give you more of the non-taxable kind of money that you need to spend on research to publish in approved journals.

Derrick Kourie
Editor

Editor's Notes

SACJ/SART, No 6, 1992
Efficient Evaluation of Regular Path Programs

P T Wood

Department of Computer Science, University of Cape Town, Rondebosch 7700

Abstract

The next generation of query languages for database systems should have the ability to express recursive queries, the efficient evaluation of which will be crucial to the success of these systems. One such query language which has been the subject of much research is Datalog. We define a class of Datalog programs, namely, the regular path programs, which can always be evaluated efficiently, in particular, when constants are present in a query. Efficient evaluation is ensured by reducing the number of arguments appearing in each predicate defined in the program. The class of regular path programs is incomparable to previous classes to which the technique of argument reduction has been applied.

Keywords: Deductive databases, logic programming, optimisation, rule rewriting.


Received October 1991, Accepted November 1991

1 Introduction

There is little doubt that the next generation of database systems should support query languages that are more powerful than those usually associated with relational databases. The language Datalog is one of those that has received much attention in this regard [7]. Datalog has a Prolog-like syntax, but, unlike Prolog, is evaluated bottom-up rather than top-down. The reasons for this are to ensure termination of queries and to exploit the efficiency of relational algebra for operations on sets of tuples [8].

Example 1.1 Consider the following example taken from [3]. Assume that the database comprises two relations:

likes(X, Y), which states that person X likes product Y, and
knows(X, Y), which states that person X knows person Y. Suppose that people buy the products that they like, or those bought by someone they know. Then the following program P defines the relation buys(X, Y) of people X and the products Y they buy.

buys(X, Y) :- likes(X, Y).
buys(X, Y) :- knows(X, Z), buys(Z, Y).

Note that this program is recursive, in that buys is computed in terms of itself. This is a capability not found in traditional query languages.

If we are interested in the entire buys relation, then we phrase the query ? - \(\text{buys}(X, Y)\) against the program. In order to evaluate this query bottom-up, the system would undertake an iterative process known as semi-naive evaluation, in which successive approximations to the complete buys relation are computed. These approximations represent larger and larger subsets of the buys relation, with the final iteration producing the complete answer. The first approximation is the likes relation itself. Successive approximations are found by joining previous approximations of buys with knows (based on the common attribute Z) until no new tuples are found.

One disadvantage of bottom-up methods is their inability in certain cases to "focus" the computation. For instance, if in the above example we wanted to know what John buys, rather than what everyone buys, then we would specify the query ? - \(\text{buys}(\text{john}, Y)\) instead of ? - \(\text{buys}(X, Y)\). However, a simple-minded bottom-up evaluation would still compute the entire buys relation before selecting out those tuples whose first component is John. What is needed is a method analogous to pushing select operations into relational algebra expressions [7], but one which works for recursive programs.

Top-down methods are usually better in this respect. In our example, a top-down method would first find which products John likes, then whom John knows along with the products they like, and so on. One way to achieve such behaviour for bottom-up evaluation is to transform a given program so that its bottom-up evaluation mimics a top-down evaluation. This is the goal of a method known as Magic Sets [1, 2].

Example 1.2 Given the above program P along with query

? - \(\text{buys}(\text{john}, Y)\), Magic Sets produces the following:

\(\text{magic}(\text{john})\).
\(\text{magic}(Z) :- \text{magic}(X), \text{knows}(X, Z)\).
\(\text{buys}(X, Y) :- \text{magic}(X), \text{likes}(X, Y)\).
\(\text{buys}(X, Y) :- \text{magic}(X), \text{knows}(X, Z), \text{buys}(Z, Y)\).

? - \(\text{buys}(\text{john}, Y)\).

The purpose of the first two rules is to compute all possible bindings of the first argument of buys that would be used in a top-down evaluation. This "magic" predicate is then substituted into each of the original rules in order to restrict the bindings in a bottom-up computation.

However, the Magic Sets transformation in the above example is still not the most efficient possible. It is not hard to see that the recursive rule for buys is now redundant, since the transitive closure of the knows relation starting with John is computed by the second magic rule, from which all the products bought by John are found by the nonrecursive rule for buys. We can therefore transform the
program to the following:

\[
\begin{align*}
magic(john) &: = magic(X), knows(X, Z). \\
buys(john, Y) &: = magic(X), likes(X, Y). \\
? - buys(john, Y).
\end{align*}
\]

Notice that the number of arguments in the recursive predicate in the program (initially \(buys\), now \(magic\)) has been reduced from two to one. This technique, known as argument reduction or factoring, has been applied previously to various classes of Datalog programs [5, 6]. In this paper, we apply the technique to a different class of programs, namely, the regular path programs.

In the next section, we begin by defining the class of regular path programs. Section 3 is devoted to the application of argument reduction to regular programs when constants are specified in queries. Our claim that this leads to efficient evaluation of such programs is explored in Section 4, where it is shown that the transformed programs are often considerably less time consuming to evaluate than the original programs. Conclusions and topics for future research are discussed in Section 5.

2 Regular Path Programs

In the previous section, we saw one example of a regular path program; we now define the class of such programs exactly. The term regular path program is derived from the fact that there is a correspondence between these programs and the problem of finding paths which satisfy a given regular expression in a labelled, directed graph. This connection is explored in more detail in [9], although it should be noted that the regular path programs defined below differ from the programs defined in that paper.

Essentially, we want to generalise programs such as that given in the previous section in two ways. Firstly, we would like to have more complicated rule structures as demonstrated by the following example.

**Example 2.1** Assume we have a semantic network in which the nodes represent instances, classes or properties, while edges from instances to classes are labelled with \(isa\) (classification), edges between classes are labelled with \(ako\) (generalisation), and edges from either instances or classes to properties are labelled with \(can, has\) or \(is\). The following program finds the relationship \(i_inherit\) between instances and the properties they inherit.

\[
\begin{align*}
i_inherit(X, Y) &: = isa(X, Z), \\
c_inherit(X, Y) &: = ako(X, Z), \\
c_inherit(X, Y) &: = can(X, Y), \\
c_inherit(X, Y) &: = has(X, Y), \\
c_inherit(X, Y) &: = is(X, Y).
\end{align*}
\]

Note that the above program displays a natural correspondence to the notion of a regular grammar.

A second way in which we would like to generalise programs is by adding arguments to predicates in order to pass more information among the rules. The following example demonstrates this ability.

**Example 2.2** Suppose we have an application involving a network with values labelling the edges connecting nodes. Assume that this information is stored as an edge relation \(e(X, Y, Z)\), which states that there is an edge from \(X\) to \(Y\) labelled \(Z\). If we want to find all nodes that are connected by routes on which at most two alternating values are used, the following program will suffice.

\[
\begin{align*}
s(X, Y, U, V) &: = e(X, Z, U), t(Z, Y, U, V). \\
t(X, Y, U, V) &: = e(X, Z, V), s(Z, Y, U, V). \\
s(X, Y, U, V) &: = e(X, Z, U), e(Z, Y, V).
\end{align*}
\]

Note that the program still has a regular structure and contains mutually recursive predicates \(s\) and \(t\). The variables \(U\) and \(V\) are used to hold the pairs of edge labels, while the alternation of values is captured by the mutual recursion.

Before we proceed, we need to introduce some standard terminology. Given a rule such as

\[
buys(X, Y) : = knows(X, Z), buys(Z, Y).
\]

the predicate to the left of : is called the head of the rule, while those to the right constitute the body of the rule. Variables in a program are denoted by strings with an initial upper case letter (e.g. \(X\)), while constants are numeric values or strings having an initial lower case letter (e.g. \(john\)). As is common with Datalog, we assume that all rules are safe, that is, every variable appearing in the head of a rule also appears somewhere in the body. We also assume that there are no constants in the rules, except possibly for the query clause. Predicates that correspond to relations stored in the database (such as \(knows\) above) are called EDB predicates (extensional database predicates); those that are defined by rules (such as \(buys\) above) are called IDB predicates (intensional databases predicates). We make the standard assumption that no EDB predicate appears in the head of any rule.

Regular path programs are defined as follows:

1. Rules must correspond to productions of a regular grammar; in other words, the body of each rule must comprise either (a) a single EDB predicate and a single IDB predicate (an internal rule), or (b) one or more EDB predicates (an exit rule).
2. All IDB predicate occurrences must have the same number of arguments.
3. Given an internal rule of the form

\[
s(X, Y, U, V) : = e(X, Z, U), t(Z, Y, U, V). \\
s(X, Y, U, V) : = e(X, Z, V), s(Z, Y, U, V). \\
s(X, Y, U, V) : = e(X, Z, U), e(Z, Y, V).
\]

where \(s\) and \(t\) are IDB predicates, \(e\) and \(t\) have exactly one variable in common that does not appear in the head (\(Z\) above). The position of this variable in \(t\) is called the linking position.

4. The position in the head \(s\) corresponding to the linking position in \(t\) is occupied by a variable (\(X\) above) which appears only in \(e\). This is the source variable.

5. The remaining positions in \(s\) must be occupied by variables appearing in the same positions in \(t\)—the persistent variables (\(Y, U\) and \(V\) above).

6. The source (or linking) position for each IDB predicate in the program must be the same.

7. For the whole program there must be exactly one persistent position such that for every IDB predicate ap-
pearing in the body of a rule the variable occupying that position appears nowhere else in the body. This position is called the sink position, and in each rule the corresponding variable is called the sink variable.

Example 2.3 Referring to the program in Example 2.1, it is easy to see that it conforms to the definition of a regular path program. In each of the internal rules, \( Z \) is the linking variable, \( X \) is the source variable, and \( Y \) is the sink variable. There are no other persistent variables.

Now turning to the first internal rule of Example 2.2, once again \( Z \) is the linking variable and \( X \) the source variable; hence, \( Y, U \) and \( V \) are persistent variables. Because \( U \) also appears in \( e(X, Z, U) \), the third argument position in IDB predicates cannot be the sink position. From the second internal rule, we establish that the fourth IDB argument position (occupied by \( V \)) also cannot be the sink position. Hence, \( Y \) is the sink variable in each of the internal rules.

The reader should not be mistaken into thinking that the EDB predicate must always precede the IDB predicate in an internal rule. The rule

\[
s(X, Y, V) : = t(X, Z, V), e(Z, Y, V).
\]

where \( e \) is the EDB predicate, does not violate the definition: \( Z \) is the linking variable, \( Y \) the source variable, and \( X \) the sink variable. Some examples violating the definition are given below.

Example 2.4 The pair of internal rules

\[
s(X, Y, V) : = e(X, Z, U), t(Z, Y, V).
\]
\[
t(X, Y, V) : = e(X, Z, Y), s(X, Y, Z).
\]
violates item (6) in the definition, since the source position is 1 in the first rule and 3 in the second. On the other hand, the pair of rules

\[
s(X, Y, V) : = e(X, Z, U), t(Z, Y, V).
\]
\[
t(X, Y, V) : = e(X, Z, Y), s(X, Y, Z).
\]
violates item (7), since the first rule implies that the sink position must be 2, while the second rule implies it must be 3.

The definition of regular path programs can be extended in a number of ways. Firstly, we can allow a chain of EDB predicates of the form

\[
e_0(W_1), e_1(W_1, W_2), \ldots, e_n(W_n, Z)
\]

rather than the single EDB predicate \( e(X, Z) \) in an internal rule. It is also possible to allow tuples of variables rather than single variables for source and sink variables. We do not use these extensions in the remainder of the paper as they tend to make the notation more difficult to follow.

In the next section, we consider regular path programs with queries in which constants appear, for example, \( ? - \text{buy}(\text{john}, Y) \) or \( ? - s(X, Y_0, U, V_0) \). If the constant appears in the source position, we call the program right-linear; if constants appear in one or more of the persistent positions, the program is called left-linear. These terms are consistent with those used in [5].

### 3. Argument Reduction

We now turn our attention to ways in which a left- or right-linear regular path program can be rewritten so that its evaluation can be performed more efficiently. In this section, we show that the techniques presented in [5] can be extended to apply to these programs.

#### Left-Linear Programs

If any persistent variables are bound to constants in a query to a regular path program \( P \), we simply substitute the constants for the corresponding variables in all EDB predicates in \( P \) and delete the variables wherever else they appear in \( P \).

Example 3.1 Consider the program of Example 2.2 in which argument positions 2, 3 and 4 are persistent. If we have the query \( s(X, Y_0, U, V_0) \), then the program is rewritten as follows.

\[
s(X, U) : = e(X, Z, U), t(Z, U).
\]
\[
t(X, U) : = e(X, Z, V), s(Z, U).
\]
\[
s(X, U) : = e(X, Z, U), e(Z, Y_0, V_0).
\]

All occurrences of \( Y \) and \( V \) in EDB predicates have been deleted, while all occurrences of \( Y \) and \( V \) in EDB predicates have been replaced by \( Y_0 \) and \( V_0 \), respectively. As a result, the number of arguments in each IDB predicate occurrence has been reduced from four to two.

Given a regular path program \( P \), we can always rewrite it so that (1) each rule uses the same set of variables in its head, and (2) the source variable appears in the first argument position of each IDB predicate in the head, the sink variable appears in the second position, and the remaining persistent variables appear in the same order in each rule. We will assume this standard form from now on.

Let the query to the program be given by \( ? - q(X, V) \), the tuple of persistent variables being \( V \). We assume that \( \overline{W} = W_1, \ldots, W_m \) is the subtuple of variables in \( V \) that are bound to constants \( w_i, 1 \leq i \leq m \), in the query, and that \( \theta \) is the substitution that replaces each \( W_i \) by \( w_i, 1 \leq i \leq m \). Let \( \overline{V} - \overline{W} \) denote the removal of all variables in \( \overline{W} \) from \( \overline{V} \). The general method is as follows.

1. Given an exit rule of the form

\[
t(X, \overline{V}) : = \varepsilon_1, \ldots, \varepsilon_k.
\]

where \( \varepsilon_1, \ldots, \varepsilon_k \) are \( \text{E} \) (EDB) literals, transform it to

\[
t(X, \overline{V} - \overline{W}) : = \theta(\varepsilon_1), \ldots, \theta(\varepsilon_k).
\]

2. Given an internal rule of the form

\[
t(X, \overline{V}) : = e(X, Z, U), s(Z, \overline{V}).
\]

where \( U \) appears in \( \overline{V} \), transform it to

\[
t(X, \overline{V} - \overline{W}) : = e(X, Z, \theta(U)), s(Z, \overline{V} - \overline{W}).
\]

The query \( q(X, \overline{V} - \overline{W}) \) is now applied to the transformed program.

#### Right-Linear Programs

If the source variable \( X \) in a query \( q(X, Y, V) \) is bound to a constant, we apply a transformation based on that of Magic Sets [1, 2], similar to the technique in [5]. 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 $P^\text{ad}$ [7], in which each IDB predicate $p$ has an adornment $\alpha$ indicating which arguments of $p$ are bound and which are free. For example, $p^\text{ad}$ means that the first argument of $p$ is bound while the second is free.

Example 3.2 Consider again the program $P$ of Example 2.2. If we assume that the query is $\text{?- } s(x_0, Y, V, V)$, then the adorned program $P^\text{ad}$ is as follows.

\[
\begin{align*}
\text{\textit{s}}(X, Y, V, V) & : - e(X, Z, U), \\
\text{\textit{t}}(Y, Z, U, V) & : - e(X, Z, U), \\
\text{\textit{t}}(Y, Z, U, V) & : - e(X, Z, U), \\
\text{\textit{t}}(Y, Z, U, V) & : - e(X, Z, U), \\
\text{\textit{t}}(Y, Z, U, V) & : - e(X, Z, U), \\
\text{\textit{t}}(Y, Z, U, V) & : - e(X, Z, U), \\
\text{\textit{t}}(Y, Z, U, V) & : - e(X, Z, U), \\
\text{\textit{t}}(Y, Z, U, V) & : - e(X, Z, U), \\
\text{\textit{t}}(Y, Z, U, V) & : - e(X, Z, U), \\
\text{\textit{t}}(Y, Z, U, V) & : - e(X, Z, U), \\
\text{\textit{t}}(Y, Z, U, V) & : - e(X, Z, U), \\
\end{align*}
\]

In the query, only $X$ is bound so we start with the adornment $b\textit{iff}$ for $s$. Given that $X$ is bound in the first rule, by the time $t$ is evaluated in a top-down evaluation, $Z$ and $U$ will also be bound; hence the adornment $b\textit{iff}$ for $t$. This process continues until all adornments which are generated in a top-down manner for all IDB predicates have been considered.

The second step in the transformation is to use the standard technique to derive the set of magic rules for $P^\text{ad}$ [1]. The following example demonstrates the method.

Example 3.3 We use the adorned program $P^\text{ad}$ from the previous example. The magic rules are used to compute bottom-up those bindings for variables that would have been used in a top-down computation. These bindings are represented by so-called magic predicates, one for each adorned version of an IDB predicate in $P^\text{ad}$. The magic predicates are formed by prefixing $m_.$ to the IDB predicates. We first generate a rule for the constant in the query:

\[
m_{s\text{\textit{iff}}}(x_0)
\]

For each internal rule in $P^\text{ad}$ of the form

\[
t^\text{ad}(X, Y, V) : - e(X, Z, U), s^\text{ad}(Z, Y, V).
\]

we 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 $m_.$ and $m_!$. The reason for step (ii) is that we are interested only in bound variables, while that for step (iii) is that we want to simulate top-down evaluation by bottom-up evaluation. This gives rise to the following set of magic rules:

\[
\begin{align*}
m_{s\text{\textit{iff}}}(Z, U) & : - m_{s\text{\textit{iff}}}(X), \\
m_{t\text{\textit{iff}}}(Z, U) & : - m_{t\text{\textit{iff}}}(X), \\
m_{t\text{\textit{iff}}}(Z, U) & : - m_{t\text{\textit{iff}}}(X), \\
m_{t\text{\textit{iff}}}(Z, U) & : - m_{t\text{\textit{iff}}}(X), \\
m_{t\text{\textit{iff}}}(Z, U) & : - m_{t\text{\textit{iff}}}(X), \\
m_{t\text{\textit{iff}}}(Z, U) & : - m_{t\text{\textit{iff}}}(X), \\
m_{t\text{\textit{iff}}}(Z, U) & : - m_{t\text{\textit{iff}}}(X), \\
m_{t\text{\textit{iff}}}(Z, U) & : - m_{t\text{\textit{iff}}}(X), \\
m_{t\text{\textit{iff}}}(Z, U) & : - m_{t\text{\textit{iff}}}(X), \\
m_{t\text{\textit{iff}}}(Z, U) & : - m_{t\text{\textit{iff}}}(X), \\
\end{align*}
\]

In the final step of the transformation, magic predicates are introduced into the exit rules of $P^\text{ad}$. Unlike in the usual Magic Sets algorithm, only the magic rules and these modified exit rules are used to answer the original query.

Example 3.4 Continuing with our running example, we must substitute each of $m_{s\text{\textit{iff}}}$ and $m_{s\text{\textit{bb}}}$ into the exit rule of the program, giving:

\[
s(Y, U, V) : - m_{s\text{\textit{iff}}}(X), e(X, Z, U), e(Z, Y, V).
\]

\[
s(Y, U, V) : - m_{s\text{\textit{bb}}}(X, U, V), e(X, Z, U), e(Z, Y, V).
\]

Note once again that the number of arguments in all IDB predicates has been reduced by the transformation.

We now summarise the general method. Given a program $P$ and query $q(x_0, Y, V)$, $P$ is rewritten as follows.

1. Generate the adorned program $P^\text{ad}$ from $P$ and the query.
2. Generate the magic rules from $P^\text{ad}$ according to the algorithm in [1].
3. For each exit rule in $P^\text{ad}$ of the form

\[
p^\text{ad}(X, Y, V) : - E_1, \ldots, E_k.
\]

where $E_1, \ldots, E_k$ are (EDB) literals, generate the rule

\[
q(Y, V) : - m_{p^\text{ad}}(X(U)), E_1, \ldots, E_k.
\]

where $U$ is the tuple of persistent variables bound according to $\alpha$.

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

In common with [5], the above method has the advantage over Magic Sets that magic predicates are substituted into exit rules alone, the remaining rules of the original program being discarded.

4 Efficient Evaluation

The technique of argument reduction has been shown to speed up the evaluation of a range of Datalog programs. The programs to which the technique is applied in [6] generalise the one-sided programs of [3], the separable programs of [4], and the right-, left- and combined-linear programs of [5]. These programs are the so-called 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\(^1\). For these programs it has been proved that the evaluation of a program $P$ transformed by argument reduction is never less efficient than the evaluation of $P$ transformed by the Magic Sets algorithm, which is the best general purpose technique for speeding up evaluation of Datalog programs. In fact, as shown in [6], programs transformed by argument reduction can lead to an order of magnitude improvement over Magic Sets in terms of evaluation efficiency.

We have already seen that regular path programs can contain multiple exit rules (Example 2.1), as well as more than one IDB predicate and mutually recursive rules (Example 2.2). Thus, there are regular path programs that are not RLC-stable. On the other hand, regular path programs do not contain combined-linear rules, so there are RLC-stable programs that are not regular path programs.

We show below by means of an example that, given a regular path program that is not RLC-stable, the transformations described in this paper can also lead to an order

\(^1\)There are other restrictions as well.

70 SACJ/SART, No 6, 1992
of magnitude improvement over Magic Sets in terms of evaluation efficiency.

Example 4.1 Once again, we consider the program $P$ of Example 2.2, excluding the persistent variables $U$ and $V$ which are not needed for the present purpose:

$s(X, Y) : = (X, Z, t(Z, Y)).$
$t(X, Y) : = (X, Z), s(Z, Y)).$
$s(X, Y) : = (X, Z), e(Z, Y)).$

Assume that the query is $-s(x_0, Y)$ and that the relation $e$ contains the $2n+1$ tuples $\{(x_i, x_{i+1}) | 0 \leq i \leq 2n-1\}$. The program $P^{sd}$ is as follows:

$\mathcal{M}(X, Y) : = (X, Z), \mathcal{M}(Z, Y)$.
$\mathcal{M}(X, Y) : = (X, Z), s(Z, Y)$.
$s(Z, Y) : = (X, Z), e(Z, Y)$.

From $P^{sd}$ we get the following magic rules:

$m_s(x_0) = m_s(x_0)$.
$m_t(Z) : = m_s(x_0), e(x, Z)$.
$m_s(Z) : = m_t(x_0), e(x, Z)$.

The relation $m_s$ contains the $n$ values $\{x_1, x_2, \ldots, x_{2n-1}\}$, while $m_s$ contains the $n+1$ values $\{x_0, x_2, \ldots, x_{2n}\}$. Substituting the magic predicate into the exit rule according to the method of Section 3.2 yields the following:

$s(Y) : = m_s(x_0), e(x, Z), e(Z, Y)$.

The relation $s$ contains the values $\{x_2, x_4, \ldots, x_{2n}\}$, which is the answer to the query. On the other hand, the Magic Sets algorithm produces

$s(M, Y) : = m_s(M), e(x, Z), e(Z, Y)$.

where $s$ contains the tuples $\{(x_{2i}, x_{2i+2}) | 1 \leq i \leq n\}$. So in both cases the relation for $s'$ contains $n$ tuples. But now Magic Sets goes on to substitute magic predicates into the internal rules of the program as well, yielding:

$s'(X, Y) : = m_s(X), e(x, Z), t(Z, Y)$.

Here $s'$ contains the tuples $\{(x_{2i}, x_{2j}) | 0 \leq i < j \leq n\}$, while $t'$ contains the tuples $\{(x_{2i-1}, x_{2j-1}) | 1 \leq i < j \leq n\}$. These two relations together contain exactly $n^2$ tuples, an order of magnitude larger than the relations in the program transformed by argument reduction. □

5 Conclusion

We have defined a class of Datalog programs, the regular path programs, whose efficiency of evaluation can often be improved significantly by applying the technique of argument reduction. This class of programs is incomparable to previous classes to which this technique has been applied.

It has been shown that bottom-up evaluation using the Magic Sets transformation followed by semi-naive evaluation is never less efficient than top-down evaluation [8]. On the other hand, Magic Sets does not perform as well as it might for certain classes of programs and it is highly unlikely that any single method will prove to be the most efficient for all programs. This has led to the search for subclasses of programs which are amenable to specialised techniques for improving evaluation efficiency. One such class of programs are the RLC-stable programs [6]. The regular path programs defined in this paper now provide another such class.

One obvious area of future research is to attempt to establish whether the class of regular path programs can be integrated with the RLC-stable programs, thereby producing a strictly broader class of programs to which argument reduction can be applied.

References

The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems, as well as shorter technical research papers. However, non-refereed review and exploratory articles of interest to the journal's readers will be considered for publication under sections marked as a Communications or Viewpoints. While English is the preferred language of the journal papers in Afrikaans will also be accepted. Typed manuscripts for review should be submitted in triplicate to the editor.

Form of Manuscript
Manuscripts for review should be prepared according to the following guidelines:

- Use double-space typing on one side only of A4 paper, and provide wide margins.
- The first page should include:
  - title (as brief as possible);
  - author's initials and surname;
  - author's affiliation and address;
  - an abstract of less than 200 words;
  - an appropriate keyword list;
  - a list of relevant Computing Review Categories.
- Tables and figures should be on separate sheets of A4 paper, and should be numbered and titled. Figures should be submitted as original line drawings, and not photocopies.
- Mathematical and other symbols may be either handwritten or typed. Greek letters and unusual symbols should be identified in the margin, if they are not clear in the text.
- References should be listed at the end of the text in alphabetic order of the (first) author's surname, and should be cited in the text in square brackets. References should thus take the following form:

Manuscripts accepted for publication should comply with the above guidelines, and may be provided in one of the following formats:

- in a typed form (i.e. suitable for scanning);
- as an ASCII file on diskette; or
- as a WordPerfect, \text{T\text{E}X} or L\text{A\text{T}\text{E}X} or file; or
- in camera-ready format.

A page specification is available on request from the editor, for authors wishing to provide camera-ready copies. A styles file is available from the editor for Wordperfect, \text{T\text{E}X} or L\text{A\text{T}\text{E}X} documents.

Charges
Charges per final page will be levied on papers accepted for publication. They will be scaled to reflect scanning, typesetting, reproduction and other costs. Currently, the minimum rate is R20-00 per final page for camera-ready contributions and the maximum is R100-00 per page for contributions in typed format.

These charges may be waived upon request of the author and at the discretion of the editor.

Proofs
Proofs of accepted papers will be sent to the author to ensure that typesetting is correct, and not for addition of new material or major amendments to the text. Corrected proofs should be returned to the production editor within three days.

Note that, in the case of camera-ready submissions, it is the author's responsibility to ensure that such submissions are error-free. However, the editor may recommend minor typesetting changes to be made before publication.

Letters and Communications
Letters to the editor are welcomed. They should be signed, and should be limited to about 500 words.

Announcements and communications of interest to the readership will be considered for publication in a separate section of the journal. Communications may also reflect minor research contributions. However, such communications will not be refereed and will not be deemed as fully-fledged publications for state subsidy purposes.

Book reviews
Contributions in this regard will be welcomed. Views and opinions expressed in such reviews should, however, be regarded as those of the reviewer alone.

Advertisement
Placement of advertisements at R1000-00 per full page per issue and R500-00 per half page per issue will be considered. These charges exclude specialized production costs which will be borne by the advertiser. Enquiries should be directed to the editor.
Contents

GUEST CONTRIBUTION

Impressions of Computer Science Research In South Africa
E.G. Coffman, Jr. .................................................. 1

RESEARCH ARTICLES

An Implementation of the Linda Tuple Space under the Helios Operating System
PC Clayton, EP Wentworth, GC Wells & FK de-Heer-Menlah ............................. 3

Modelling the Algebra of Weakest Preconditions
C Brink and I Rewitzky ........................................... 11

The Design and Analysis of Distributed Virtual Memory Consistency Protocols in an Object Oriented OS
KJ McGregor and RH Cambell .................................. 21

An Object Oriented Framework for Optimistic Parallel Simulation on Shared-Memory Computers
P Machanik .......................................................... 27

Analysing Routing Strategies in Sporadic Networks
SW Melville ......................................................... 37

Using Statecharts to Design and Specify a Direct-Manipulation User Interface
L Van Zijl & D Mitton ............................................. 44

Extending Local Recovery Techniques for Distributed Databases
HL Viktor & MH Rennhackkamp ................................. 59

Efficient Evaluation of Regular Path Programs
PT Wood ............................................................. 67

Integrating Similarity-Based and Explanation-Based Learning
GD Oosthuizen & C Avenant .................................... 72

Evaluating the Motivating Environment For IS Personnel in SA Compared to the USA. (Part I)
JD Cougar & DC Smith ........................................... 79

TECHNICAL NOTE

An Implementation of the Parallel Conditional
U Jayasekera and NCK Philips .................................. 85

COMMUNICATIONS AND REPORTS

Book Review ...................................................... 87

The CSP Notation and its Application to Parallel Processing
PG Clayton ......................................................... 90