Right-, left- and multi-linear rule transformations that maintain context information

Technical Report 90/2

David B. Kemp
Kotagiri Ramamohanarao
Zoltan Somogyi

{kemp,rao,zs}@cs.mu.OZ.AU
Department of Computer Science
University of Melbourne
Parkville, 3052
Australia

Abstract

We present an algorithm that takes recursive rules, predicates and queries belonging to a particular class and transforms them into rules, predicates and queries that allow efficient bottom-up computation of answers. This work extends the work presented in "*Efficient evaluation of right-, left-, and multi-linear rules*" by J. Naughton, R. Ramakrishnan, Y. Sagiv, and J. Ullman. Our transformation can handle calls whose input arguments are not manifest constants but are computed by other calls in the query, and it can handle predicates containing pseudo-left-linear rules together with right-linear and/or multi-linear rules. The first of those properties allows us to apply our algorithm effectively to calls to predicates that occur in the bodies of rules as well as in queries. Experimental results indicate that our algorithm can achieve considerable speedups over previous methods.

1 Introduction

The past several years have seen great advances in the field of deductive databases. Much of this progress took the form of the development of evaluation algorithms and rule rewriting techniques that can increase the speed of bottom-up computation of answers to queries by several orders of magnitude. These techniques, which include differential or semi-naive evaluation [3, 4], magic sets [1, 5, 7, 17], the Alexander method [11, 15], counting sets [16, 17], rule rectification [19], and constraint propagation [9], generally work by reducing the number and/or the size of the tuples one must generate to answer a given query.

In two other recent papers ([13] and then [12]), J. Naughton, R. Ramakrishnan, Y. Sagiv, and J. Ullman have taken further steps in this direction. They have defined a transformation that can reduce the cost of answering some queries from $O(n^2)$ to $O(n)$. This transformation, which we call the NRSU-transformation, is applicable to a fairly large class of procedures that occurs frequently in real programs: those defined by right-, left-, and multi-linear rules (see sections 2 through 5; see also [13] and chapter 15 of [18]).

In this paper we address the fundamental limitation of the NRSU-transformation: it works only when the call to the transformed procedure has constants in its input positions. We define a new transformation that does not require this restriction: it works even when the input of the transformed procedure is provided by a non-singleton relation. Since the key feature of our algorithm is its management of context information that records which outputs go with which inputs, we call our algorithm the context-transformation.

In [13], Naughton et al tentatively suggest another approach to this problem. They propose executing the NRSU-transformed program for each tuple in the input relation if the input relation is "small enough". Although we can confirm their conjecture that when the input relation contains *very* few tuples (two or at most three) their method is indeed better than magic sets, our work renders this point irrelevant. Our measurements show that the context-transformation is more efficient than the NRSU-transformation whenever the input relation has two or more tuples. When the input relation has many tuples, the difference between the two techniques becomes more than an order of magnitude.

From our experience with logic programs, we expect that many procedures in real deductive databases will belong to the class of procedures handled by the NRSU-transformation. However, only a small fraction of calls are likely to have constants as the input arguments; and if any of the input arguments are variables, then one can seldom guarantee that the input relation to the call will always be a singleton set. The true significance of our algorithm is that it imposes no such requirements on the inputs of calls; it can therefore optimize far more calls in a typical deductive database program than the NRSU-transformation can. The fact that the context-transformation can handle procedures that mix pseudo-left-linear rules with right-linear, multi-linear and basis rules is a secondary contribution.

The structure of this paper is as follows. The rest of this section contains background information. Sections 2 through 4 deal with procedures containing right-linear rules, left-linear rules, and multi-linear rules respectively; section 5 covers procedures that contain linear rules of more than one type. Each of these sections starts with a motivating example, showing how the NRSU-transformation and our context-transformation handle the same simple program; we then give the definition of the context-transformation as it applies to that class of procedures. These sections also contain estimates of the efficiency of the programs produced by the transformations under consideration, using the number of tuples generated as the cost criterion. In Section 6 we consider the optimization of calls occurring in bodies rather than queries, including the interaction of our transformation with other optimization techniques. In section 7 we validate these estimates by presenting performance results derived from experiments using Aditi, a full-scale deductive database system based on relational algebra operations that is currently being built at the University of Melbourne [20].

In this paper we consider only positive databases, although we strongly believe that our results can be easily generalized to handle stratified databases (just as magic sets have been [2, 14]); we discuss this issue further in section 6. For simplicity and without loss of generality we assume that all rules in the database are in homogeneous form, i.e. their heads contain only distinct variables. In sections 2 through 5 we also assume that the database contains only one derived predicate, all other predicates representing base relations; we lift this restriction in section 6.

We assume the reader is familiar with the terminology of deductive databases, the notion of differential or semi-naive bottom-up computation, the concepts of stratified databases and maximal stratification, and the general idea of the magic set transformation.

A final note on terminology. Most papers in this area define a rule as linear recursive if the head predicate occurs exactly once in the body. However, in [13] and [12], Naughton et al allow a rule to contain more than one occurrence of the head predicate in the body in their definition of multi-linear recursion (and in their definition of left-linear recursion as well in [12]). For the purposes of this paper, we use the definitions given in [13], which we reproduce in sections 2 through 5.

Due to lack of space, this paper contains no proofs for our theorems. They can be found in the technical report version of this paper [10].

2 Right-linear recursions

Before we give a general transformation, we look at a motivating example. Consider the following rules for deriving the ancestor relation:

```
anc(X, Y) ← par(X, Y).
anc(X, Y) ← par(X, Z), anc(Z, Y).
```

Suppose the query is

```
← anc(judy, Y).
```

Using the magic set transformation we get the following rules for evaluating this query:

```
anc(X, Y) ← m_anc(X), par(X, Y).
anc(X, Y) ← m_anc(X), par(X, Z), anc(Z, Y).
m_anc(Z) ← m_anc(X), par(X, Z).
m_anc(judy).
```

Suppose the relation for `par` looks something like the following set of tuples:

$\{(\mathtt{judy}, \mathtt{x}_1), (\mathtt{x}_1, \mathtt{x}_2), \ldots, (\mathtt{x}_{n-1}, \mathtt{x}_n)\}$

The answer to the query contains only n tuples, however the number of tuples generated for `anc` is $O(n^2)$.

Notice however, that in this example, the tuples generated for `m_anc` contain all the required answers, and so it is better to use the NRSU-transformation outlined in [13] to get the following rules for evaluating the query:

```
m_anc(judy).
m_anc(Y) ← m_anc(X), par(X, Y).
answers(Y) ← m_anc(X), par(X, Y).
```

This is an improvement on the magic set transformation as the number of tuples generated is only $O(n)$.

2.1 Right-linear recursions with context information

Now suppose there is a base relation called `t`, and that the query is

```
← t(C), anc(C, Y).
```

The paper [13] suggests that the same transformation can be used as for a single constant, but that separate computations could be made for each tuple in `t`. However, we have found that it is more efficient to compute the answers for all tuples in `t` together using the context-transformed program shown below.

```
mc_anc(C, C) ← t(C).
mc_anc(C, Z) ← mc_anc(C, X), par(X, Z).
ac_anc(C, Y) ← mc_anc(C, Z), par(Z, Y).
answers(C, Y) ← t(C), ac_anc(C, Y).
```

The relation defined by `mc_anc` (magic ancestor with context information) is the same as the original magic set, except that every tuple has the extra argument that gives the value from relation `t` that caused that tuple to be generated. The relation defined by `ac_anc` (ancestor answers with context information) is the set of required tuples for the `anc` relation.

If the relation `t` has m tuples, then the number of tuples generated is $O(mn)$. Hence, for the special case of $m = 1$, this transformation is almost as efficient as the NRSU-transformation, and for larger values of m the cost will never be much greater than that for magic sets (and will often be less).

2.2 General transformations for right-linear recursions

Definition 2.1 *A derivation rule for a recursive predicate* `p` *is* right-linear with respect to an adornment α *if it is of the form:*

$$\mathtt{p}(\overline{X},\ \overline{Y}) \leftarrow G_1, \ldots, G_k, \mathtt{p}(\overline{W},\ \overline{Y}).$$

where $\overline{X}$, $\overline{Y}$, and $\overline{W}$ represent vectors of variables, and the following conditions hold:

- *If the number of variables in $\overline{X}$ is m, then the m leftmost arguments of* `p` *are bound and the other arguments are free according to α.*
- `p` *is not the predicate of any of the G_i's.*
- *The variables in $\overline{Y}$ do not occur in any of the G_i's, but they do each occur in the same argument position in the recursive body literal as they do in the head.*
- *Each variable that occurs in $\overline{W}$ either appears in $G_1, \ldots, G_k$ or occurs in $\overline{X}$.* ◁

Definition 2.2 *A recursive predicate* `p` *is* right-linear with respect to an adornment α *if the following conditions hold:*

- `p` *is not mutually recursive with any other predicates.*
- *each rule defining* `p` *is either non-recursive or right-linear with respect to α.* ◁

The context-transformation for right-linear predicates consists of three separate rule transformations: one for basis rules, one for right-linear recursive rules, and one for the query. To apply the context-transformation to a right-linear predicate `p`, we apply the appropriate rule transformation to the query and to each rule defining `p`; the transformed program is the union of all the transformed rules with the transformed query.

The rule transformation for right-linear rules transforms

$\texttt{p(}\overline{X}\texttt{, }\overline{Y}\texttt{)} \leftarrow G_1, \ldots, G_k, \texttt{p(}\overline{W}\texttt{, }\overline{Y}\texttt{)}.$

into

$\texttt{mc\_p(}\overline{C}\texttt{, }\overline{W}\texttt{)} \quad \leftarrow \quad \texttt{mc\_p(}\overline{C}\texttt{, }\overline{X}\texttt{)}, G_1, \ldots, G_k.$

where $\overline{C}$ is a vector of distinct variables that do not occur in the original rule, and the number of variables in $\overline{C}$ is equal to the number in $\overline{X}$.

The rule transformation for basis rules transforms

$\texttt{p(}\overline{X}\texttt{, }\overline{Y}\texttt{)} \leftarrow H_1, \ldots, H_j.$

into

$\texttt{ac\_p(}\overline{C}\texttt{, }\overline{Y}\texttt{)} \quad \leftarrow \texttt{mc\_p(}\overline{C}\texttt{, }\overline{X}\texttt{)}, \ H_1, \ldots, H_j.$

The rule transformation for queries transforms

$\leftarrow F_1, \ldots, F_j, \texttt{p(}\overline{C}\texttt{, }\overline{Y}\texttt{)}, F_{j+1}, \ldots, F_f.$

where each variable in $\overline{C}$ is either a constant or a term ground by $F_1, \ldots, F_j$, into the query

$\leftarrow F_1, \ldots, F_j, \texttt{ac\_p(}\overline{C}\texttt{, }\overline{Y}\texttt{)}, F_{j+1}, \ldots, F_f.$

and the rule

$\texttt{mc\_p(}\overline{C}\texttt{, }\overline{C}\texttt{)} \quad \leftarrow \quad F_1, \ldots, F_j.$

Theorem 2.1 *The context-transformation for right-linear rules preserves equivalence with respect to the query.*

2.3 Efficiency

For the special case where the query involves constants instead of bound variables, the context-transformation produces rules that are similar to the rules produced by the NRSU-transformation. The context-transformed program generates exactly the same number of tuples as the NRSU-transformed program; the difference is that the tuples generated by the context-transformed program will include the constants from the query and will therefore be larger. Hence, for this special case, the NRSU-transformation is more effective.

In other cases, however, the original NRSU-transformation does not apply, whereas the context-transformation does. In those cases, the context-transformed program will still only generate $O(n)$ tuples, compared to the $O(n^2)$ tuples of the magic set transformed program.

3 Left-linear recursions

Using the same rules for ancestor as presented in the previous section, consider the query

```
← anc(X, judy).
```

Using a suitable sideways information-passing strategy, the magic set transformation gives

```
anc(X, Y) ← m_anc(Y), par(X, Y).
anc(X, Y) ← m_anc(Y), anc(Z, Y), par(X, Z).
m_anc(Y) ← m_anc(Y).
m_anc(judy).
```

One can easily see that `m_anc` will only ever contain the tuple `judy`. Hence it is more efficient to use the NRSU-transformation whose result is

```
a_anc(X) ← par(X, judy).
a_anc(X) ← a_anc(Z), par(X, Z).
answers(X) ← a_anc(X).
```

3.1 Left-linear recursions with context information

Now suppose the query is

```
← t(C), anc(X, C).
```

It is possible to use the NRSU-transformation and compute answers corresponding to each tuple in `t`. However, a comparison of the magic set transformation with the NRSU-transformation will show that the NRSU-transformation only reduces the size of each tuple, and the number of tuples generated remains unchanged. Hence the magic set transformation is reasonably efficient for the above query.

We present a transformation which is similar to the magic set transformation. The rules produced are no more efficient than those produced by the magic set transformation, but they do allow an easy transition to the mixed-linear cases. As with the right-linear case, we define two new predicates called `mc_anc` and `ac_anc`. The `mc_anc` predicate (magic ancestor with context information) is again a magic set with each tuple containing an extra argument that gives the value from `t` for which `anc` is being evaluated. The tuples generated for the `ac_anc` predicate (ancestor answers with context information) are again the answers to the query.

Our context-transformation produces the following rules for our running example

```
mc_anc(C, C) ← t(C).
ac_anc(X, C) ← mc_anc(C, Y), par(X, Y).
ac_anc(X, C) ← ac_anc(Z, C), par(X, Z).
answers(X, C) ← t(C), ac_anc(X, C).
```

3.2 General transformations for left-linear recursions

Definition 3.1 *A derivation rule for a recursive predicate* `p` *is* left-linear with respect to an adornment α *if it is of the form*

$\mathtt{p}(\overline{X},\ \overline{Y}) \leftarrow \mathtt{p}(\overline{X},\ \overline{V}),\ G_1, \ldots, G_k.$

and the following conditions hold:

- *If the number of variables in* $\overline{X}$ *is* m*, then the* m *leftmost arguments of* `p` *are bound and the other arguments are free according to* α*.*
- `p` *is not the predicate of any of the* G_i*'s.*
- *Every variable in* $\overline{V}$ *is different from every variable in* $\overline{X}$*.*
- *None of the variables in* $\overline{X}$ *occur in any of the* G_i*'s.*

If all the conditions except the last are satisfied, then the rule is called pseudo-left-linear. ◁

Definition 3.2 *A recursive predicate* `p` *is* left-linear with respect to an adornment α *if the following conditions hold:*

- `p` *is not mutually recursive with any other predicates.*
- *each rule defining* `p` *is either non-recursive or left-linear with respect to* α*.*

If the definition of `p` *contains pseudo-left-linear rules as well as non-recursive and/or left-linear rules, then* `p` *is called* pseudo-left-linear. ◁

The context-transformation for left-linear and pseudo-left-linear predicates consists of three separate rule transformations: one for basis rules, one for left-linear and pseudo-left-linear recursive rules, and one for the query. To apply the context-transformation to a left-linear predicate p, we apply the appropriate rule transformation to the query and to each rule defining p; the transformed program is the union of all the transformed rules with the transformed query.

The rule transformation for left-linear and pseudo-left-linear rules transforms

$\texttt{p(}\overline{X}\texttt{, }\overline{Y}\texttt{)} \leftarrow \texttt{p(}\overline{X}\texttt{, }\overline{V}\texttt{), } G_1, \ldots, G_k.$

into

$\texttt{ac\_p(}\overline{C}\texttt{, }\overline{Y}\texttt{)} \leftarrow$
$\quad \texttt{mc\_p(}\overline{C}\texttt{, }\overline{X}\texttt{), ac\_p(}\overline{C}\texttt{, }\overline{V}\texttt{), } G_1, \ldots, G_k.$

where $\overline{C}$ is a vector of distinct variables that do not occur in the original rule, and the number of variables in $\overline{C}$ is equal to the number in $\overline{X}$.

The literal whose predicate is mc_p is not needed for pure left-linear rules, which can be transformed into

$\texttt{ac\_p(}\overline{C}\texttt{, }\overline{Y}\texttt{)} \leftarrow \texttt{ac\_p(}\overline{C}\texttt{, }\overline{V}\texttt{), } G_1, \ldots, G_k.$

The rule transformations for basis rules and for queries are the same for left-linear and pseudo-left-linear predicates as for right-linear predicates.

Theorem 3.1 *The context-transformation for left-linear rules preserves equivalence with respect to the query.*

3.3 Efficiency

The rules produced by the context-transformation for left-linear rules compute approximately the same number of tuples as the magic set transformation, and so the context-transformation is not an improvement over magic sets in this case. However, our transformation can be easily combined with the context-transformation for right-linear rules to give an efficient transformation for predicates that have both left-linear and right-linear rules. This is the subject of section 5.

4 Multi-linear recursions

Our transformation can handle multi-linear recursive rules. Consider the following multi-linear rules and query for the ancestor relation:

```
anc(X, Y) ← par(X, Y).
anc(X, Y) ← anc(X, Z), anc(Z, Y).
 ← anc(judy, Y).
```

The NRSU-transformation gives the rules

```
m_anc(judy).
m_anc(Z) ← a_anc(Z), m_anc(X).
a_anc(Y) ← m_anc(X), par(X, Y).
```

where the relation a_anc contains the answers to the query. This program is more efficient than the untransformed and magic set transformed programs.

4.1 Multi-linear recursions with context information

For a query such as

```
← t(C), a(C, Y).
```

the context-transformation gives the following rules:

```
mc_anc(C, C)  ← t(C).
mc_anc(C, Z)  ← ac_anc(C, Z).
ac_anc(C, Y)  ← mc_anc(C, X), par(X, Y).
answers(C, Y)  ← t(C), ac_anc(C, Y).
```

As usual, the context magic set is initialized with values from relation `t`. The recursive derivation rule for `mc_anc` is built from the recursive rule for `anc`. As with right-linear recursion, the derivation rule for the predicate `ac_anc` is built from the basis rule for `anc`.

4.2 General transformations for multi-linear recursions

Definition 4.1 *A derivation rule for a recursive predicate* `p` *is* multi-linear with respect to an adornment α *if it is of the form*

$\mathtt{p}(\overline{X},\ \overline{Y}) \leftarrow G_1, \ldots, G_k,\ \mathtt{p}(\overline{W},\ \overline{Y})$.

and the following conditions hold:

- *If the number of variables in $\overline{X}$ is m, then the m leftmost arguments of* `p` *are bound and the other arguments are free according to α.*
- *The variables in $\overline{Y}$ do not occur in any of the G_i's, but they do each occur in the same argument position in the last recursive body literal as they do in the head.*
- *All the variables in $\overline{W}$ appear in $G_1, \ldots, G_k$, but none occur in $\overline{X}$.*
- *At least one of the G_i's has* `p` *as its predicate.*
- *If G_i $(i = 1, \ldots, k)$ has* `p` *as its predicate then,*
  - *Its m leftmost arguments are identical to those of the head, i.e. those arguments contain all the variables in $\overline{X}$ in order, and*
  - *All of its other arguments have variables that do not occur in $\overline{X}$.*
- *If G_i $(i = 1, \ldots, k)$ doesn't have* `p` *as its predicate, then none of its variables are among $\overline{X}$.* ◁

Definition 4.2 *A recursive predicate* `p` *is* multi-linear with respect to an adornment α *if the following conditions hold:*

- `p` *is not mutually recursive with any other predicates.*
- *each rule defining* `p` *is either non-recursive or multi-linear with respect to α.* ◁

The context-transformation for multi-linear predicates consists of three separate rule transformations: one for basis rules, one for multi-linear recursive rules, and one for the query. To apply the context-transformation to a multi-linear predicate `p`, we apply the appropriate rule transformation to the query and to each rule defining `p`; the transformed program is the union of all the transformed rules with the transformed query.

The rule transformation for multi-linear rules transforms p($\overline{X}$, $\overline{Y}$) $\leftarrow G_1, \ldots, G_k$, p($\overline{W}$, $\overline{Y}$).

into

mc_p($\overline{C}$, $\overline{W}$) $\leftarrow G'_1, \ldots, G'_k$.

where $\overline{C}$ is a vector of distinct variables that do not occur in the original rule, and the number of variables in $\overline{C}$ is equal to the number in $\overline{X}$. The G'_i are the same as the G_i, except for those whose predicates are `p`. These are replaced by literals whose predicates are `ac_p`, whose leftmost m variables are replaced by $\overline{C}$ and whose remaining variables are the same variables as those of the literal being replaced.

The basis rules and the query are transformed in exactly the same manner as in the previous sections.

Theorem 4.1 *The context-transformation for multi-linear rules preserves equivalence with respect to the query.*

5 Mixed-linear recursions

When a predicate has a mixture of right-linear, left-linear, pseudo-left-linear and multi-linear rules, the transformations shown in the previous sections can still be applied. To illustrate this, we use the same example given in [13]. The predicate `p` is defined as follows:

r_1: p(X, Y, Z) $\leftarrow$ q(X, Y, Z).
r_2: p(X, Y, Z) $\leftarrow$ a(X, U), p(U, Y, Z).
r_3: p(X, Y, Z) $\leftarrow$ b(Y, V), p(X, V, Z).
r_4: p(X, Y, Z) $\leftarrow$ c(Z, W), p(X, Y, W).

Suppose the query is

$\leftarrow$ p(christina, Y, Z).

With respect to this query, rule r_2 is right-linear while rules r_3 and r_4 are left-linear. The rules defining the magic predicate `m_p` are built from the query and rule r_2 as follows:

m_p(christina).
m_p(U) $\leftarrow$ m_p(X), a(X, U).

These magic rules combined with the following four rules make up the rules produced by the NRSU-transformation.

a_p(Y, Z) $\leftarrow$ m_p(X), q(X, Y, Z). From r_1.
a_p(Y, Z) $\leftarrow$ a_p(V, Z), b(Y, V). From r_3.
a_p(Y, Z) $\leftarrow$ a_p(Y, W), c(Z, W). From r_4.
p(christina, Y, Z) $\leftarrow$ a_p(Y, Z). From query.

These rules are at least as efficient as those produced by the magic set transformation, and our timing results in section 7 show that they are considerably more efficient.

5.1 Mixed-linear recursions with context information

Consider a query of the form

$\leftarrow$ t(C), p(C, Y, Z).

Combining the context-transformation for left- and right-linear rules gives the following rules and query:

```
mc_p(C, C) ← t(C).
mc_p(C, U) ← mc_p(C, X), a(X, U).
ac_p(C, Y, Z) ← mc_p(C, X), q(X, Y, Z).
ac_p(C, Y, Z) ← ac_p(C, V, Z), b(Y, V).
ac_p(C, Y, Z) ← ac_p(C, Y, W), c(Z, W).
 ← t(C), ac_p(C, Y, Z).
```

5.2 General transformations for mixed-linear recursions

Definition 5.1 *A recursive predicate* `p` *is* mixed-linear with respect to an adornment α *if the following conditions hold:*

- `p` *is not mutually recursive with any other predicates.*
- *each rule defining* `p` *is either non-recursive, right-linear, left-linear, pseudo-left-linear, or multi-linear with respect to* α*.* ◁

To apply the context-transformation to a mixed-linear predicate `p`, we apply the appropriate rule transformation to the query and to each rule defining `p` as they were defined in the previous sections; the transformed program is the union of all the transformed rules with the transformed query.

Theorem 5.1 *The context-transformation for mixed-linear rules preserves equivalence with respect to the query.*

Unlike the NRSU-transformation, the context-transformation works even when a mixed-linear predicate contains pseudo-left-linear rules: the arguments that carry context values provide bindings for the variables that make the rule pseudo-left-linear as opposed to pure left-linear. This is a serendipitous side-effect of our approach.

In a recent paper [8], Jiawei Han and Ling Liu have suggested methods for processing a class of linear recursions that they call *multiple linear* recursions. This class overlaps the class of procedures we consider in this paper: mixed-linear procedures without multi-linear rules belong to both classes. For such programs, the two techniques give the same results: a bottom-up evaluation of the rules that our transformation produces results in the same computation as the evaluation of the relational algebra formulae produced by the methods given in [8]. The most important difference between the two methods is that only ours can handle multi-linear rules, whereas only their method can handle procedures such as same generation, in whose recursive rules the head and the recursive call share neither the full set of inputs nor the full set of outputs. (We intend to leave the optimization of such rules to a separate counting-sets transformation.) The other main difference is that their methods produce relational algebra formulae directly whereas our transformation yields logic programs, making further optimizations easier.

6 Calls in rule bodies

In the previous four sections we have applied our transformation only to calls occurring in the query. However, our transformation can also handle calls within program rules, provided that the predicate of the call concerned is not mutually recursive with any other predicates.

To see that our transformation works correctly in such situations, imagine the position of a to-be-transformed predicate in a maximally stratified database. (We are still talking about positive databases; we will discuss negation in a moment.) Our condition forbidding mutual recursion guarantees that this predicate is the only one on its level. Now follow the course of a level-structured bottom-up computation. Such a computation applies the rules of the predicates in the bottom stratum until it can generate no more new facts; it then repeats the process with successively higher strata. By the time we get to the level containing the to-be-transformed predicate (and nothing else), all predicates in lower levels have been completely evaluated. Such a situation is isomorphic to the case of a single recursive predicate and a collection of base relations. Our transformation therefore works as well when applied to calls in the bodies of rules as when applied to calls in queries.

This kind of argument can show that the original NRSU-transformation could also handle calls that occur in the bodies of rules. The advantage of the context-transformation of course is that it can handle calls that have variables among their input arguments. Based on our experience with logic programs, we believe this to be a very significant advantage: constants simply do not occur anywhere near as often in rules as variables do. The inputs of most calls are provided by the outputs of other calls in the same rule or by the inputs of the predicate being defined.

In the presence of negation, the context-transformation suffers from a problem that magic sets also suffer from: the resulting program may be unstratified even when the input program has a stratification. Since unstratified databases do not have an agreed-on semantics, this is not acceptable. Fortunately, the picture is not all bleak. Since the cause of the unstratification is the same in each case (a rule defining a magic predicate may contain a negative reference to its parent predicate) we believe that the techniques that solve this problem for magic sets (labelling algorithms [2] and structured interpreters [14]) should also solve it for the context-transformation. We intend to explore this issue in more detail; we will report our results in a later paper.

The context-transformation is compatible with most other optimization techniques but with some notable exceptions. First, like most optimizations, it is not compatible with itself, in the sense that applying it twice to the same predicate will not improve performance and may in fact hurt (if the second application is possible at all). Second, it is not compatible with optimizations that derive their improvements from the same source: applying the magic set- or the NRSU-transformation to a context-transformed program will likewise reduce performance. The reason is that the second transformation adds overhead but cannot reduce the number of intermediate tuples generated (the context-transformation already computes the necessary magic sets; computing them again will not help). On the other hand, techniques such as differential evaluation [3, 4], rule rectification [19] and constraint propagation [9] are as applicable to context-transformed programs as to any others.

We have also devised an optimization technique that applies *only* to context-transformed programs. This optimization targets the overhead involved in copying contexts around by representing each context $\overline{C}$ by a single integer `I`. The optimization carries out this substitution in all of the rules defining `ac_p` and `mc_p`, and it further modifies the clauses containing calls to the transformed predicate and the basis rule of the context magic predicate. The body of a calling clause, which originally was of the form

$F_1, \ldots, F_j,$ `ac_p(`$\overline{C}$`, `$\overline{Y}$`)`$, F_{j+1}, \ldots, F_f$

now becomes

$F_1, \ldots, F_j,$ `id_p(I, `$\overline{C}$`), ac_p(I, `$\overline{Y}$`)`$, F_{j+1}, \ldots, F_f$

with the `id_p` relation storing the mapping between contexts and their identifiers. (This mapping must be established at run time; the underlying database system could easily provide special support for this.) Given this mapping, the rule for `mc_p`, which was originally

`mc_p(`$\overline{C}$`, `$\overline{C}$`)` $\leftarrow F_1, \ldots, F_j.$

(derived from the call to `p`) now becomes

Right-linear query

```
anc(X, Y) ← par(X, Y).
anc(X, Y) ← par(X, Z), anc(Z, Y).
q(X, Y) ← t(X), anc(X, Y).
```

Left-linear query

```
anc(X, Y) ← par(X, Y).
anc(X, Y) ← par(X, Z), anc(Z, Y).
q(X, Y) ← t(Y), anc(X, Y).
```

Multi-linear query

```
anc(X, Y) ← par(X, Y).
anc(X, Y) ← anc(X, Z), anc(Z, Y).
q(X, Y) ← t(X), anc(X, Y).
```

Mixed-linear query

```
p(X, Y, Z) ← v(X, Y, Z).
p(X, Y, Z) ← a(X, U), p(U, Y, Z).
p(X, Y, Z) ← b(Y, V), p(X, V, Z).
p(X, Y, Z) ← c(Z, W), p(X, Y, W).
q(X, Y, Z) ← t(X), p(X, Y, Z).
```

Figure 1: Test Programs

$\texttt{mc\_p(I, }\overline{C}\texttt{)} \;\leftarrow\; \texttt{id\_p(I, }\overline{C}\texttt{)}.$

Unless the context was already of minimal size, this will reduce the size of the magic set (in bytes, not in tuples); and it will always eliminate the redundant evaluation of $F_1, \ldots, F_j$.

7 Performance results

We have tested the effectiveness of the context-transformation using Aditi, a deductive database system currently under development at the University of Melbourne [20]. Aditi is designed to evaluate database queries using relational algebra operations like join and union, and has a compiler for transforming Prolog rules into a low-level relational language. Aditi is designed to handle large databases, and so it stores all its intermediate relations on disk. To make our measurements as realistic as possible (specifically, to include disk access times) we measure performance by the real time required to perform a given query.

We compared five transformation algorithms, the magic set algorithm, the supplementary magic set algorithm, the NRSU-transformation, the context-transformation, and the null transformation (as a control). We tried out each of these algorithms on four programs (one right-linear, one (pure) left-linear, one multi-linear and one mixed-linear). We ran each of the resulting transformed programs on five sets of data, asking two queries with different-sized input relations on each combination. (It turns out that the supplementary magic set transformation of the left-linear program gives the same rules as those given by the magic set transformation, and hence there was no need to do separate runs for this case.) Some results (those pairing the NRSU-transformation with large input relations) are missing due to the excessively large amount of time that would have been required to complete them.

Our four test programs are shown in figure 1. In each case, `q` is the query predicate, the query being a call to `q` with all arguments free.

Our test data consists of the relations `par`, `v`, `a`, `b`, and `c`. Our relation `par` is a set of tuples that describe a full binary tree:

$$\{\texttt{par(1,2)}, \texttt{par(1,3)}, \texttt{par(2,4)}, \texttt{par(2,5)}, \ldots, \texttt{par(}n\texttt{, } 2n+1\texttt{)}\}$$

The five data sets differ in their value of n; we used the values 127, 255, 511, 1023, and 2047, corresponding to tree depths of 7, 8, 9, 10 and 11 respectively.

The relation `v` is a set of tuples that can be regarded as vectors in three dimensional space. They are arranged in a cubic grid, separated from each other by one hundred units, and contained in a cube which has opposite vertices of $(0, 0, 0)$ and $(200, 200, 200)$. In other words, `v` is the set

$$\{(x,y,z) : x = 100i,\ y = 100j,\ z = 100k,\ 0 \le i \le 2,\ 0 \le j \le 2,\ 0 \le k \le 2\}$$

In our test data, the contents of the relations `a`, `b`, and `c` are identical; each is the set

$$\{(x,y) : x = y + 1,\ y = 100i + j,\ 0 \le i \le 2,\ 0 \le j \le n\}$$

The three data sets differ in their value of n (which in this case we call the *run length*); we used the values 3, 4, 5, 6 and 7. (The derived relation `p` consists of the cube described by `v` but with every point in `v` extended into a small cube, the length of whose sides is given by n.)

Table **??** reports our results. The four sections of the table cover right-, left-, multi- and mixed-linear programs respectively. Each table section contains results in five rows, measuring the performance of the original, the magic set transformed, the supplementary magic set transformed, the NRSU-transformed, and the context-transformed programs respectively. The five major columns divide the four data sets: trees of depths 7, 8, 9, 10 and 11 for the right-, left- and multi-linear programs and cubes with run lengths of 3, 4, 5, 6 and 7 for the mixed-linear program. The two minor columns within each major column give the size of the `t` relation, i.e. the size of the input to the linear predicate. We consider two cases: the input contains one tuple or it contains several. When `t` contains just one tuple, the tuple is chosen from the middle level of the tree (for the right-, left- and multi-linear cases) or contains zero (for the mixed-linear cases). When `t` contains several tuples, the tuples were chosen randomly. For the right-, left- and multi-linear cases, the size of the `t` relation was fixed at five percent of the size of the `par` relation; for the mixed-linear cases, it was fixed at fifty percent of the possible argument values of the first argument of `p`. All computations used the differential evaluation strategy [3].

Each entry in table **??** gives the speedup achieved by the given transformation strategy on the given test data with the given query; the reference is the performance of the untransformed program on the same test data and the same query. The speedups compare real (wallclock) times on an Encore Multimax 320 with 64 megabytes of memory under UMAX 4.2 (revision 3.3.1). The test database was stored on an NEC D2362 disk drive connected via an asynchronous SCSI interface with a 1.5 Mb/s maximum transfer rate.

Every figure we report is based on the average of several runs. When `t` contained just one tuple, the repetition served only to reduce timing errors. When `t` contained several tuples, the repetition served to eliminate the influence of the large variation in performance between different sets of input values: each measurement is an average of several runs with different `t` relations. For the large times, two or three runs were enough, but in some cases we needed four or five runs to establish a reasonably firm value for the mean. The majority of the raw results lie within about five percent of the mean.

Our results confirm our expectations about the relative efficiency of the various transformations. When the relation `t` had only one tuple, the NRSU-transformed and the context-transformed programs always performed better than the magic set and the supplementary magic set transformed programs, and these in turn performed better than the untransformed programs. The NRSU-transformed program generally performed better than the context-transformed program as the tuples generated by the NRSU-transformed program were smaller. As the programs that we are considering are so simple, the supplementary magic set transformation actually adds an overhead to the magic set transformation and so it produces a less efficient program, except when applied to left-linear programs as it then produces the same rules.

Also as expected, the results for the left-linear program show that there is little difference between the NRSU-transformed, the context-transformed and the magic set transformed programs.

When the relation `t` had more than one tuple, the need to evaluate the query separately for each tuple in `t` ruined the performance of the NRSU-transformed program. The context-transformed program, on the other hand, kept its efficiency: in these cases it *always* performed better than any of the others.

One should also exercise caution in the interpretation of our results. The version of Aditi that we used for these tests is only a prototype; further tuning may help some transformations more than others. Also, the results depend critically on the precise form of the rules defining the derived relations and the precise shape of the data in the base relations. For more authoritative results, one would need to run much more extensive tests using a much wider range of programs and test data (possibly using [6] as a base).

8 Conclusions

Our measurements prove that our optimization technique can yield significant speedups, speedups that are better in most cases than those achieved by magic sets or the NRSU-transformation. It eliminates the main weakness of the NRSU-transformation: it works even when input arguments are variables, not constants, and hence it can be applied to far more calls in deductive database programs.

In the future, we intend to perform further experiments to evaluate the effectiveness of the transformation under various conditions, including the cases of calls occurring in bodies, calls with more than one bound argument, and base relations of different shapes. We also intend to find out whether our transformation warrants the application of unfolding to convert a set of mutually recursive predicates into a single self-recursive predicate, and if so, under what conditions (beside the constraint that the references between the mutually recursive predicates must form a simple loop so that unfolding will terminate).

We would like to thank Jayen Vaghani for hacking Aditi "above and beyond the call of duty". We would also like to thank Isaac Balbin for his comments on previous drafts of this paper.

References

[1] Balbin, I., Port, G., and Ramamohanarao, K. Magic set computation for stratified databases. Tech. Rep. 87/3 (revised), Department of Computer Science, University of Melbourne, Australia, 1987. To appear in the Journal of Logic Programming.

[2] Balbin, I., Port, G., Ramamohanarao, K., and Meenaskshi, K. Efficient bottom-up computation of queries on stratified databases. *Journal of Logic Programming* (1990). To appear.

[3] Balbin, I., and Ramamohanarao, K. A generalization of the differential approach to recursive query evaluation. *Journal of Logic Programming 4*, 3 (September 1987), 259–262.

[4] Bancilhon, F. Naive evaluation of recursively defined relations. In *Proceedings of the Islamadora Conference on Databases and AI* (1985).

[5] Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J. Magic sets and other strange ways to implement logic programs. In *Proceedings of the Fifth ACM Symposium on Principles of Database Systems* (Washington, DC, 1986), pp. 1–15.

[6] Bancilhon, F., and Ramakrishnan, R. Performance evaluation of data intensive logic programs. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. Morgan Kaufmann Publishers, Los Altos, California, 1988, pp. 439–518.

[7] Beeri, C., and Ramakrishnan, R. On the power of magic. In *Proceedings of the Sixth ACM Symposium on Principles of Database Systems* (San Diego, California, 1987), pp. 269–283.

[8] Han, J., and Liu, L. Processing multiple linear recursions. In *Proceedings of the First North American Conference on Logic Programming* (Cleveland, Ohio, October 1989), E. L. Lusk and R. A. Overbeek, Eds., MIT Press, pp. 816–830.

[9] Kemp, D. B., Ramamohanarao, K., Balbin, I., and Meenakshi, K. Propagating constraints in recursive deductive databases. In *Proceedings of the First North American Conference on Logic Programming* (Cleveland, Ohio, October 1989), E. L. Lusk and R. A. Overbeek, Eds., MIT Press, pp. 981–998.

[10] Kemp, D. B., Ramamohanarao, K., and Somogyi, Z. Right-, left-, and multi-linear rule transformations that maintain context information. Tech. Rep. 90/2, Department of Computer Science, University of Melbourne, Australia, 1990.

[11] Kerisit, J.-M., and Pugin, J.-M. Efficient query answering on stratified databases. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988* (November 1988), pp. 719–725.

[12] Naughton, J. F., Ramakrishnan, R., Sagiv, Y., and Ullman, J. D. Argument reduction by factoring. In *Proceedings of the Fifteenth Conference on Very Large Data Bases* (Amsterdam, The Netherlands, 1989), P. M. G. Apers and G. Wiederhold, Eds., pp. 173–182.

[13] Naughton, J. F., Ramakrishnan, R., Sagiv, Y., and Ullman, J. D. Efficient evaluation of right-, left-, and multi-linear rules. In *Proceedings of ACM SIGMOD '89* (1989), pp. 235–242.

[14] Port, G., Balbin, I., and Ramamohanarao, K. A new approach to supplementary magic optimisation. Tech. Rep. 90/5, Department of Computer Science, University of Melbourne, Australia, 1990. Submitted for publication.

[15] Rohmer, J., Lescoeur, R., and Kerisit, J. The Alexander method - a technique for the processing of recursive axioms in deductive databases. *New Generation Computing 4*, 3 (1986), 273–286.

[16] Sacca, D., and Zaniolo, C. The generalized counting method for recursive logic queries. In *Proceedings of the International Conference on Database Theory* (Rome, Italy, 1986), G. Ausiello and P. Atzeni, Eds., pp. 31–53.

[17] Sacca, D., and Zaniolo, C. Magic counting methods. In *Proceedings of ACM SIGMOD '87* (San Francisco, California, May 1987), U. Dayal and I. Traiger, Eds., pp. 49–59.

[18] Ullman, J. *Principles of database and knowledge-base systems, vol. II: The new technologies.* Computer Science Press, New York, 1989.

[19] Ullman, J. D. Bottom-up beats top-down for Datalog. In *Proceedings of the Eighth ACM Symposium on the Principles of Database Systems* (Philadelphia, Pennsylvania, March 1989), pp. 140–149.

[20] Vaghani, J., Ramamohanarao, K., and Kemp, D. B. Design overview of Aditi: a deductive database system. In *Proceedings of the Far-East Workshop on Future Database Systems* (Melbourne, Australia, April 1990). Proceedings published as Technical Report #20, Key Centre for Knowledge Based Systems, RMIT and The University of Melbourne.