Constraint Propagation for Linear Recursive Rules

James Harland and Kotagiri Ramamohanarao
Department of Computer Science
University of Melbourne
Parkville, 3052
Australia
{jah,rao}@cs.mu.oz.au

Abstract
There are many ways in which the query answering process for deductive databases may be optimised. Many of these methods rely on applying constraints as soon as possible, to avoid the production and later rejection of facts which are not relevant to the query. This propagation of constraints is much simpler for *left-linear* programs than for many others. In this paper we show how to transform *right-linear* and *mixed-linear* programs into a left-linear form, to make constraint propagation more effective. Our technique generalises the magic set transformation for linear programs; the magic set transformation can only propagate constraints of the form $X = a$, whereas our technique can handle arbitrary goals as constraints.

1 Introduction

An important issue for deductive database systems is that of query optimization, and in particular for recursively defined relations. In recent years there has been much work done on various transformation methods which involve re-writing the program to enable a bottom-up computation to evaluate the program more efficiently. Various methods have been proposed for transforming the program in this way, including the well known technique of *magic sets* [4, 5]. Such methods usually involve re-writing the rules of the database to take advantage of some extra information, such as the adornments of the definition of the relation. This information may be thought of as specifying how "binding" constraints (i.e. those which bind variables to particular values) may be passed from one invocation of a rule to another, which in turn is used to ensure that only "relevant" work is done.

For certain classes of programs, it is possible to make such transformations more efficient than is generally the case. In particular, the class of *linear recursive rules* has been much studied ([12, 13] among others) as have transitive closure algorithms and operators ([1, 2, 3] among others), and there are refinements to the magic set transformation which may be applied in such cases [12].

Such methods are designed to make effective use of the constants in the query, and make considerable improvements to the efficiency of evaluation. However, such techniques do not consider more general constraints, such as those which bind a variable not to a unique value, but to a finite number of values, or "filtering" rather than binding constraints such as $X > 1$. In [8] it was shown how a technique known as *constraint propagation* may be used to "insert" constraints associated with a query into the place in the program where the constraint will have most effect, and hence reduce the number of tuples generated. Another approach is to use the *context transformations* [9, 11], which exploit the properties of a certain class of linear recursions in order to propagate constraints which may have more than one solution.

In this paper we study the problem of the propagation of arbitrary constraints for the same classes of programs as in [9], i.e. *left*-, *right*- and *multi-linear* programs. An important aspect of this problem is to note that constraint propagation is much simpler for left-linear programs than for right-linear ones, and that right-linear programs can be easily transformed into left-linear ones [11]. Hence the best method of propagation seems to be to transform the program into the left-linear form and then propagate the constraint. An interesting sideline to this analysis is that the magic sets transformation may be seen as performing the same transformation, but in a more direct manner. In particular, the refinements of [12] seem straightforward in this framework.

For example, consider the query ?- $t(1, Y)$ for the program below.

$t(X, Y)$:- $p(X, Y)$.
$t(X, Y)$:- $p(X, Z), t(Z, Y)$.

The magic set transformation for the adornment bf, together with the refinements of [12] produces the program and query below.

?- $ft(Y)$.
$ft(Y)$:- $m(X), p(X, Y)$.
$m(Z)$:- $m(X), p(X, Z)$.
$m(1)$.

Note that the original program is *right-linear* for the adornment bf. The left-linear version of the same transitive closure is given below.

$t(X, Y)$:- $p(X, Y)$.
$t(X, Y)$:- $t(X, Z), p(Z, Y)$.

Given the query ?- $t(1, Y)$ it is more efficient to use the program and query below.

?- $ft(Y)$.
$ft(Y)$:- $p(1, Y)$.
$ft(Y)$:- $ft(Z), p(Z, Y)$.

For these reasons we may think of left-linear recursions as a normal form for left- and right-linear programs. A similar methodology may be applied for multi-linear programs, for which the normal form is a simpler class, which we call *left-multi-linear* programs. These may be thought of as making the recursion as "left-linear" as possible, and it is shown how constraint propagation is as simple as in the left-linear case.

Thus the main contribution of this paper is to show how "normalising" the recursion makes constraint propagation simpler, and that this technique, in one form or another, seems to occur in many optimizing transformations. For example, the magic sets transformation may be seen as a particular instance of our scheme, being the case for equality constraints such as $X = a$. For this reason we believe that the methodology of this paper is in some ways more important than the technical results.

This paper is organised as follows. In Section 2 we give the necessary technical preliminaries, and in the following section we show how right- and mixed-linear programs may be transformed into left-linear or left-multi-linear ones. In Section 4 we show how the left-linear and left-multi-linear forms may be used to propagate arbitrary constraints. Finally in Section 5 we discuss some possibilities for further work.

Owing to space restrictions, no proofs are provided in the main body of the paper; however, all proofs of the results herein may be found in [7].

2 Preliminaries

In this paper a deductive database consists of a finite set of Datalog clauses, i.e. Horn clauses without function symbols.

We consider such a set of clauses as defining a set of relations. *Base relations* are those which are those for which the body of all the clauses defining the relation are empty. *Derived relations* are those for which the body of all the clauses defining the relation are non-empty. Note that without loss of generality, we may assume that all relations are either base relations or derived relations. Furthermore, we assume that all clauses are *range-restricted*, i.e. that all variables which occur in the head of a clause must occur in the body of the clause (note that this means that the base relations must contain only ground terms).

We will also make use of the usual notion of adornment [15] in order to express mode information. For every derived relation in the database, we code the valid modes for the relation into the name of the relation. For example, the adorned version of the transitive closure relation above for the mode bf (i.e. in which the first argument is bound) is as follows:

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

We shall use the default left-to-right *sideways information-passing strategy* (sip) throughout this paper [15].

Given a goal $p(\overline{X}, \overline{Y})$ where the adornment of p is $b^m f^n$ and $|\overline{X}| = m$ and $|\overline{Y}| = n$, a *constraint* is an any atom $t(\overline{X})$. Note that this includes binding constraints, such as if there is a clause in the program such as $t(1)$ (so that the goal ?- $t(\overline{X})$ returns bindings for $\overline{X}$), and filtering constraints, such as $X > 1$. Given that the definition of $t(X)$ may be arbitrary, this class of constraints is maximally general among constraints that do not depend on the output variables.

Below we give the definitions of right-, left-, mixed- and multi-linear programs [12].

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:*

$$p(\overline{X}, \overline{Y}) \text{ :- } G_1, \ldots G_k, p(\overline{W}, \overline{Y})$$

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

1. *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.*
2. *p is not the predicate of any of the G_i's.*
3. *The variables in $\overline{Y}$ do not occur in any of G_i's, but they do occur in the same argument position in the recursive body literal as they do in the head.*
4. *Each variable that occurs in $\overline{W}$ either appears in $G_1 \ldots G_k$ or occurs in $\overline{X}$.*

Definition 2.2 *A derivation rule for a recursive predicate p is* left-linear with respect to an adornment α *if it is of the form:*

$$p(\overline{X}, \overline{Y}) \text{ :- } p(\overline{X}, \overline{Z}), G_1, \ldots G_k$$

where the following conditions hold:

1. *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.*
2. *p is not the predicate of any of the G_i's.*
3. *Every variable in $\overline{Z}$ is different from every variable in $\overline{X}$.*
4. *None of the variables in $\overline{X}$ occur in any of the G_i's.*

Definition 2.3 *A derivation rule for a recursive predicate p is* multi-linear with respect to an adornment α *if it is of the form:*

$$p(\overline{X}, \overline{Y}) \text{ :- } G_1, \ldots G_k, p(\overline{W}, \overline{Y})$$

where the following conditions hold:

1. *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.*

2. *The variables in $\overline{Y}$ do not occur in any of G_i's, but they do occur in the same argument position in the last recursive body literal as they do in the head.*

3. *All the variables in $\overline{W}$ appear in $G_1 \ldots G_k$, but none occur in $\overline{X}$.*

4. *At least one of the G_i's has p as its predicate.*

5. *If G_i has p as its predicate, then*

   (a) *Its m leftmost arguments are identical to those of the head.*

   (b) *All of its other arguments have variables which do not occur in $\overline{X}$.*

6. *If G_i does not have p as its predicate, then none of its variables occur in $\overline{X}$.*

Below we introduce the class of *left-multi-linear* programs. We shall see later how such programs are useful.

Definition 2.4 *A derivation rule for a recursive predicate p is* left-multi-linear with respect to an adornment α *if it is of the form:*

$$p(\overline{X}, \overline{Y}) \text{ :- } p(\overline{X}, \overline{X_1}), \ldots p(\overline{X}, \overline{X_n}), G_1, \ldots G_k$$

where the following conditions hold:

1. *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.*

2. *p is not the predicate of any of the G_i's.*

3. *Every variable in $\overline{X_1}, \ldots \overline{X_n}$ is different from every variable in $\overline{X}$.*

4. *None of the variables in $\overline{X}$ occur in any of the G_i's.*

Definition 2.5 *A recursive predicate p is* right-linear *(resp.* left-linear, left-multi-linear, multi-linear*)* with respect to an adornment α *if p is not mutually recursive with any other predicates, and each rule defining p is either non-recursive or right-linear (resp. left-linear, left-multi-linear, multi-linear) with respect to α.*

By default, we say a recursive predicate p is right-linear *(resp.* left-linear, left-multi-linear, multi-linear*) if it is* right-linear *(resp.* left-linear, left-multi-linear, multi-linear*) with respect to some adornment of the form $b^n f^m$ for $n, m \geq 0$.*

A recursive predicate p is mixed-linear with respect to an adornment α *if p is not mutually recursive with any other predicates, and each rule defining p is either non-recursive, right-linear, left-linear, or left-multi-linear, with respect to α.*

By default, we say a recursive predicate p is mixed-linear *if it is mixed-linear with respect to some adornment of the form $b^n f^m$ for $n, m \geq 0$.*

A mixed-linear predicate p is a simple mixed-linear *predicate if it contains only right-linear or left-linear rules.*

Note that we allow a mixed-linear recursion to contain left-multi-linear recursions but not multi-linear ones. Thus our mixed-linear rules correspond to those of [12] rather than those of [9], in which a mixed-linear recursion may contain a multi-linear rule. It should be clear that any mixed-linear predicate which is not multi-linear may be divided into a simple mixed-linear program and multi-linear rules.

We shall see the significance of this class of programs in the next section.

We will also need to make use of the well known T_P operator. For completeness, we give the definition below; further details may be found in [10].

Definition 2.6 *Let P be a program. We define an operator T_P mapping sets of atoms to sets of atoms as follows:*

$$T_P(I) = \{A \mid \textit{there is an instance } A \text{ :- } B_1, \ldots B_n \textit{ of a clause in } P \textit{ such that } B_i \in I \textit{ for } 1 \leq i \leq n\}$$

Note that T_P is monotonic, i.e. that if $I \subseteq J$, then $T_P(I) \subseteq T_P(J)$, and that if $P \subseteq P'$, then $T_P(I) \subseteq T_{P'}(I)$. The reason that this operator is of interest is that it allows us to easily define the atomic consequences of a program.

Definition 2.7 *Let P be a program. We define the powers of T_P as follows:*

$$\begin{aligned} T_P \uparrow 1 &= T_P(\emptyset) \\ T_P \uparrow n+1 &= T_P(T_P \uparrow n) \\ T_P \uparrow \omega &= \bigcup_{i=1}^{\infty} T_P \uparrow i \end{aligned}$$

An *answer* for a query $p(\overline{X})$ is a tuple $\overline{t}$ such that $p(\overline{t}) \in T_P \uparrow \omega$. For further information on this point, see [10].

3 Transformations to "Left-Linear" Programs

Consider the query ?- $t(X), anc(X, Y)$ and the *anc* relation defined below.

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

Note that for the adornment bf, this is right-linear. Now we may re-write this definition so that it is left-linear for the same adornment. This gives us the program below.

$anc(X,Y)$:- $q(X,Y)$.
$anc(X,Y)$:- $m(X,Z), q(Z,Y)$.
$m(X,Z)$:- $p(X,Z)$.
$m(X,Z)$:- $m(X,W), p(W,Z)$.

The same tuples for anc are computed by this program as by the right-linear version. This may be seen by considering that answers to the query computed by the original program are of the form p^*q, and that the original program computes these by starting with q, and then deriving pq, and then p^2q and so forth. In this way we may consider the program to compute "rightmost". The transformed program performs the same operations in a difference sequence, commencing with p, and then deriving p^2, and then p^3 and so forth, and then finally adding q at the end, to get p^*q. For this reason we think of the transformed version as computing "leftmost". This is similar to the difference between left- and right-linear grammars. This may be clearly seen in the two grammars below, each of which may be used to derive words of the regular expression p^*q.

$A ::= q \mid pA$	$A ::= q \mid Mq$
	$M ::= p \mid Mp$

The left-linear form makes it simpler to propagate a constraint $t(X)$, for which we get the program below.

$anc\_t(X,Y)$:- $t(X), q(X,Y)$.
$anc\_t(X,Y)$:- $m\_t(X,Z), q(Z,Y)$.
$m\_t(X,Z)$:- $t(X), p(X,Z)$.
$m\_t(X,Z)$:- $m\_t(X,W), p(W,Z)$.

Now for purposes of comparison, if we assume that $t(X)$ is a (single-valued) binding constraint such as $X = 1$, then the magic set transformation of [12] is applicable. Under this transformation, for the query $anc(1,Y)$ we arrive at the following program:

$anc_1(Y)$:- $m(X), q(X,Y)$.
$m(Z)$:- $m(X), p(X,Z)$.
$m(1)$.

Note that this requires that the constraint have a single solution. For constraints which have a finite number of solutions, the context transformation of [9] may be used. Our transformation does not have this restriction, so that constraints with an infinite number of solutions, such as $X > 1$, may be used.

An important point to note is that the latter version of the program is left-linear (in the predicate m). Hence it would seem that this is a particular instance of a more general scheme, i.e. that programs be transformed into left-linear forms whenever possible, and then propagate constraints. It is shown in [11] how right-linear and left-linear programs may be considered duals of each other. We show how a right-linear definition may be re-written as a left-linear one, which is done in order to facilitate the propagation of constraints; in particular, the propagation of a constraint into a left-linear program is simpler than the case for a right-linear program, and so we convert the right-linear program into a left linear one before propagating the constraint. The reason that propagation is simpler for left-linear programs is that all the occurrences of the recursive predicate in the body of the definition have precisely the same input variables as the head. This means that constraints on the input variables of the head are precisely those on the input variables of the recursive calls, and so the propagated constraints need only appear in the exit rules. This provides us with a new method of constraint propagation, and, as we shall see, arbitrary constraints may be propagated in this way (i.e. constraints which are not necessarily binding constraints).

For example, consider another version of the *anc* relation below, which is left-linear with respect to the adornment bf.

$$anc(X,Y) \text{ :- } p(X,Y).$$
$$anc(X,Y) \text{ :- } anc(X,Z), q(Z,Y).$$

Clearly given a query ?- $t(X), anc(X,Y)$ it is simple to propagate the constraint $t(X)$ into $anc(X,Y)$, giving the program below.

$$anc\_t(X,Y) \text{ :- } t(X), p(X,Y).$$
$$anc\_t(X,Y) \text{ :- } anc\_t(X,Z), q(Z,Y).$$

Similar techniques may be used for mixed-linear programs as well.

An important property to note is that we may re-write a linear recursion into a canonical form. This leads us to the definitions of *canonical* right-, left-, mixed- and multi-linear programs.

Definition 3.1 *A* canonical right-linear predicate *is a right-linear predicate of the form*

$$p(\overline{X},\overline{Y}) \text{ :- } e(\overline{X},\overline{Y})$$
$$p(\overline{X},\overline{Y}) \text{ :- } f(\overline{X},\overline{W}), p(\overline{W},\overline{Y})$$

A canonical left-linear predicate *is a left-linear predicate of the form*

$$p(\overline{X},\overline{Y}) \text{ :- } e(\overline{X},\overline{Y})$$
$$p(\overline{X},\overline{Y}) \text{ :- } p(\overline{X},\overline{W}), f(\overline{W},\overline{Y})$$

A canonical mixed-linear predicate *is a mixed-linear predicate of the form*

$$p(\overline{X},\overline{Y}) \text{ :- } e(\overline{X},\overline{Y})$$
$$p(\overline{X},\overline{Y}) \text{ :- } f(\overline{X},\overline{W}), p(\overline{W},\overline{Y})$$

together with a number of rules of the form

$$p(\overline{X},\overline{Y}) \text{ :- } p(\overline{X},\overline{X_1}), \ldots p(\overline{X},\overline{X_n}), g(\overline{X_1}, \ldots \overline{X_n}, \overline{Y})$$

A canonical multi-linear predicate *is a multi-linear predicate of the form*

$$p(\overline{X},\overline{Y}) \text{ :- } e(\overline{X},\overline{Y})$$

together with a number of rules of the form

$$p(\overline{X},\overline{Y}) \text{ :- } p(\overline{X},\overline{X_1}), \ldots, p(\overline{X},\overline{X_n}), f(\overline{X_1}, \ldots, \overline{X_n}, \overline{W}), p(\overline{W},\overline{Y})$$

In each case we assume that $\overline{X}$ and $\overline{Y}$ are disjoint sets of variables.

It is not hard to see that for any right-linear program, there is a canonical right-linear program which is equivalent with respect to queries on p, and similarly for the other cases.

We are now in a position to define the translations between the various types of recursive rules. First we consider right-linear and left-linear recursions.

Definition 3.2 *Let p be a canonical right-linear predicate, so that p is of the form*

$$p(\overline{X},\overline{Y}) \text{ :- } e(\overline{X},\overline{Y})$$
$$p(\overline{X},\overline{Y}) \text{ :- } f(\overline{X},\overline{W}), p(\overline{W},\overline{Y})$$

We define the left-linear *version of p, p_ll as follows:*

$$p\_ll(\overline{X},\overline{Y}) \text{ :- } e(\overline{X},\overline{Y})$$
$$p\_ll(\overline{X},\overline{Y}) \text{ :- } m\_p(\overline{X},\overline{W}), e(\overline{W},\overline{Y})$$
$$m\_p(\overline{X},\overline{Z}) \text{ :- } f(\overline{X},\overline{Z})$$
$$m\_p(\overline{X},\overline{Z}) \text{ :- } m\_p(\overline{X},\overline{W}), f(\overline{W},\overline{Z})$$

Note that it may be possible to simplify the left-linear version of p in some cases, such as when the number of variables in $\overline{W}$ is equal to the number of variables in $\overline{Y}$ and f and e are the same relation, in which case we may merely replace p with

$$p\_ll(\overline{X},\overline{Y}) \text{ :- } e(\overline{X},\overline{Y})$$
$$p\_ll(\overline{X},\overline{Y}) \text{ :- } p\_ll(\overline{X},\overline{W}), e(\overline{W},\overline{Y})$$

For example, the ancestor relation may be written in either of the forms below.

$anc(X,Y)$:- $p(X,Y)$.
$anc(X,Y)$:- $anc(X,Z), p(Z,Y)$.

$anc(X,Y)$:- $p(X,Y)$.
$anc(X,Y)$:- $p(X,Z), anc(Z,Y)$.

However, such simple transformations are only possible in certain cases; in general we need to use *p_ll*.

It is not difficult to show the equivalence of a canonical right-linear program and its left-linear version.

Proposition 3.1 *Let P be a canonical right-linear program and let P_ll be the left-linear version of it. Then $p(\bar{t},\bar{u}) \in T_P \uparrow \omega \Leftrightarrow p\_ll(\bar{t},\bar{u}) \in T_{P\_ll} \uparrow \omega$.*

The proof of this result relies on the "unwinding" the recursion in a canonical right-linear programs into a left-linear one. A similar technique can be used to unwind the recursion in a canonical left-linear program into a right-linear one.

Hence, for any right-linear program there is a left-linear program for which equivalence is preserved with respect to the query. Thus we may transform right-linear programs into left-linear ones. In fact, it is also possible to transform left-linear programs into right-linear ones. Such a transformation is useful for Prolog programs, for which such a transformation may ensure termination. However, for bottom-up methods it is the right-linear to left-linear transformation that is of most interest.

Next we turn to the mixed-linear case.

Definition 3.3 *Let p be a canonical mixed-linear predicate, so that p is of the form*

$$p(\overline{X},\overline{Y}) \text{ :- } e(\overline{X},\overline{Y})$$
$$p(\overline{X},\overline{Y}) \text{ :- } f(\overline{X},\overline{W}), p(\overline{W},\overline{Y})$$

together with a number of rules of the form

$$p(\overline{X},\overline{Y}) \text{ :- } p(\overline{X},\overline{X_1}), \ldots p(\overline{X},\overline{X_n}), g(\overline{X_1}, \ldots \overline{X_n}, \overline{Y})$$

We define the left-multi-linear *version of p, p_l as follows:*

$$r\_p(\overline{X},\overline{Y}) \text{ :- } e(\overline{X},\overline{Y})$$
$$r\_p(\overline{X},\overline{Y}) \text{ :- } m\_p(\overline{X},\overline{Z}), e(\overline{Z},\overline{Y})$$
$$m\_p(\overline{X},\overline{Z}) \text{ :- } f(\overline{X},\overline{Z})$$
$$m\_p(\overline{X},\overline{Z}) \text{ :- } m\_p(\overline{X},\overline{W}), f(\overline{W},\overline{Z})$$
$$p\_l(\overline{X},\overline{Y}) \text{ :- } r\_p(\overline{X},\overline{Y})$$

together with

$$p\_l(\overline{X},\overline{Y}) \text{ :- } p\_l(\overline{X},\overline{X_1}), \ldots p\_l(\overline{X},\overline{X_n}), g(\overline{X_1}, \ldots \overline{X_n}, \overline{Y})$$

for each multi-linear rule in the original program.

Note that when the predicate is a simple mixed-linear one (i.e. there are only left- or right-linear rules), the left-multi-linear program is in fact left-linear.

Again, it is not hard to show that this transformation preserves all answers to a given query.

Proposition 3.2 *Let P be a canonical mixed-linear program, and P_l be the left-multi-linear version of it. Then $p(\overline{t},\overline{u}) \in T_P \uparrow \omega$ iff $p\_l(\overline{t},\overline{u}) \in T_{P\_l} \uparrow \omega$.*

The proof also relies on the ability to "unwind" certain recursions; in particular, rewriting the "right-linear" part as a left-linear recursion.

Hence, for any mixed-linear program there is a left-(multi-)linear program for which equivalence is preserved with respect to the query.

Finally we consider multi-linear recursions.

Definition 3.4 *Let p be a canonical multi-linear predicate, so that p is of the form*

$$p(\overline{X},\overline{Y}) \text{ :- } e(\overline{X},\overline{Y})$$

together with a number of rules of the form

$$p(\overline{X},\overline{Y}) \text{ :- } p(\overline{X},\overline{X_1}),\ldots,p(\overline{X},\overline{X_n}),f(\overline{X_1},\ldots,\overline{X_n},\overline{W}),p(\overline{W},\overline{Y}).$$

We define the left-multi-linear *version of p, p_lm as follows:*

$$p\_lm(\overline{X},\overline{Y}) \text{ :- } e(\overline{X},\overline{Y})$$

together with

$$p\_lm(\overline{X},\overline{Y}) \text{ :- } p\_lm(\overline{X},\overline{X_1}),\ldots,p\_lm(\overline{X},\overline{X_n}),f(\overline{X_1},\ldots,\overline{X_n},\overline{Z}),e(\overline{Z},\overline{Y}).$$

for each rule in the original program.

This may seem too simple; however, this particular form may be derived by considering p as a right-linear program, transforming it accordingly, and then performing some optimizations on the ensuing mutual recursion, to arrive at the rules above. Hence, what may be considered the main problem for multi-linear rules turns out to lead to a significant simplification.

In fact, this transformation is inspired by the equivalence of the two grammars below.

$$P ::= e \mid PPfP \qquad\qquad P ::= e \mid PPfe$$

The correctness of the transformation follows from the result below.

Proposition 3.3 *Let P be a multi-linear program, and let P_lm be the left-multi-linear version of it. Then $p(\overline{t},\overline{u}) \in T_P \uparrow \omega$ iff $p\_lm(\overline{t},\overline{u}) \in T_{P\_lm} \uparrow \omega$.*

Hence, for any multi-linear program there is a left-multi-linear program for which equivalence is preserved with respect to the query. Thus we may transform any right-, mixed- or multi-linear program into one which is either left-linear or left-multi-linear, and clearly for the latter programs, the propagation of constraints is significantly easier.

4 Propagation in Left-Linear Programs

We saw in the previous section how a right-linear program may be transformed into a left-linear one. The importance of such a transformation is that we may propagate constraints more easily in the left-linear version than in the right-linear one. In the *anc* case above, the left-linear form is

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

Given the query ?- $t(X), anc(X,Y)$ we clearly may propagate the constraint $t(X)$ into $anc(X,Y)$, resulting in the program below.

$anc\_t(X,Y)$:- $t(X), p(X,Y)$.
$anc\_t(X,Y)$:- $t(X), m(X,Z), p(Z,Y)$.
$m(X,Y)$:- $m(X,Z), q(Z,Y)$.
$m(X,Y)$:- $q(X,Y)$.

We may then further propagate the constraint into $m(X,Z)$ resulting in the program

$anc\_t(X,Y)$:- $t(X), p(X,Y)$.
$anc\_t(X,Y)$:- $m\_t(X,Z), p(Z,Y)$.
$m\_t(X,Y)$:- $m\_t(X,Z), q(Z,Y)$.
$m\_t(X,Y)$:- $t(X), q(X,Y)$.

Note that we have not assumed anything about the nature of the constraint $t(X)$. It may be a "binding" constraint, such as $X = 1$, and hence binds X to a specific value or values, or it may be a "filtering" constraint, such as $X > 1$. It may also be defined by rules elsewhere in the program. In any case, propagating the constraint in this manner is profitable, in that it is used to restrict the number of tuples that are generated.

Note also that we may further optimise the program if certain properties are known. For example, if the query is ?- $anc(1,Y)$, which we may represent by adding $t(1)$ to the program and asking the query ?- $t(X), anc(X,Y)$, then we are not interested in the values of X, i.e. the first argument to *anc*, and so we may rewrite the program as

$anc\_t2(Y)$:- $t(X), p(X,Y)$.
$anc\_t2(Y)$:- $m(Z), p(Z,Y)$.
$m(Y)$:- $m(Z), q(Z,Y)$.
$m(Y)$:- $t(X), q(X,Y)$.

Now, by exploiting the similarity between the first two rules we may further rewrite the program as

$anc\_t2(Y)$:- $m(X), p(X,Y)$.
$m(Y)$:- $m(Z), q(Z,Y)$.
$m(Y)$:- $t(X), q(X,Y)$.
$m(X)$:- $t(X)$.

Then by eliminating the third rule, which is redundant, and utilizing the fact that $t(X)$ is just $X = 1$, we arrive at

$anc\_t2(Y)$:- $m(X), p(X,Y)$.
$m(Y)$:- $m(Z), q(Z,Y)$.
$m(1)$.

which is the transformation of [12] for this program. Hence, the use of left-linear programs to propagate constraints provides us with a powerful and general method of constraint propagation.

Another point to note is that our transformation gives a similar result to the context transformation of [9]. The relevant context-transformed program is below.

$mc\_anc(C,C)$:- $t(C)$.
$mc\_anc(C,Z)$:- $mc\_anc(C,X), q(X,Z)$.
$ac\_anc(C,Y)$:- $mc\_anc(C,Z), p(Z,Y)$.
$answers(C,Y)$:- $t(C), ac\_anc(C,Y)$.

As noted above, this transformation requires that $t(X)$ be a binding constraint (as $t(C)$ is used to generate tuples), whereas our transformation does not.

Another case considered in [12] is that of multi-linear rules. For example, given the query $anc(1,Y)$ and the program

$anc(X,Y)$:- $p(X,Y)$.
$anc(X,Y)$:- $anc(X,Z), anc(Z,Y)$.

the transformation of [12] gives the program

$anc\_1(Y)$:- $m(X), p(X,Y)$.
$m(Z)$:- $anc\_1(Z)$.
$m(1)$.

Moreover, it is not hard to see that this program is equivalent to the one below.

$anc\_1(Y)$:- $m(X), p(X, Y)$.
$m(Z)$:- $m(X), p(X, Z)$.
$m(1)$.

This is just the transformation of [12] for the left-linear case. In our case, the transformation gives

$anc\_t(X, Y)$:- $t(X), p(X, Y)$.
$anc\_t(X, Y)$:- $anc\_t(X, Z), p(Z, Y)$.

which also is just the same as our transformation for the left-linear version of the *anc* predicate. Then we may exploit the nature of $t(X)$ to obtain

$anc\_t1(Y)$:- $p(1, Y)$.
$anc\_t1(Y)$:- $anc\_t1(Z), p(Z, Y)$.

5 Experimental Results

In this section we give some empirical comparisons between our method and the context transformations of [9]. The program and query used are given below.

?- $t(X), anc(X, Y)$.
$anc(X, Y)$:- $p(X, Y)$.
$anc(X, Y)$:- $p(X, Z), anc(Z, Y)$.

where in one experiment t contains one tuple, and in a second experiment, t contains three tuples. In the latter case, the NRSU methods are not directly applicable, but may be used "serially", i.e., evaluating the answers to one value of the constraint, then the next, and then the third. For completeness, we have included in our comparison the (naive) magic set transformation and the NRSU transformation for this program.

Three sets of data were used; one was a chain of length 200 (i.e., a tree that has degenerated into a list), a full binary tree containing 2^{13} elements, and a "cylinder", with 20 elements at either end, and a total size of 2,000 edges. In answering the queries on this data, sometimes intermediate relations of the order of 250,000 tuples were obtained.

The context transformed program is given below.

?- $ac\_anc(X, Y)$.
$ac\_anc(C, Y)$:- $mc\_anc(C, Z), p(Z, Y)$.
$mc\_anc(C, C)$:- $t(C)$.
$mc\_anc(C, Z)$:- $mc\_anc(C, X), p(X, Z)$.

Under our transformation the program becomes

Data	Tuples	Magic	NRSU	Context	LC
Tree	1	63.86	22.11	23.56	21.05
	3	80.93	58.99	23.29	20.98
Chain	1	69.63	11.75	12.41	11.69
	3	114.32	31.08	12.79	11.39
Cylinder	1	809	16.34	15.42	14.77
	3	>1000	36.92	18.03	16.90

Table 1: Empirical comparisons of the various transformations

?- $anc\_t(X, Y)$.
$anc\_t(X, Y)$:- $t(X), p(X, Y)$.
$anc\_t(X, Y)$:- $m\_t(X, Z), p(Z, Y)$.
$m\_t(X, Y)$:- $t(X), p(X, Y)$.
$m\_t(X, Y)$:- $m\_t(X, Z), p(Z, Y)$.

Now clearly the m_t and anc_t relations are the same, and so we may omit the first two lines above, giving us

?- $m\_tt(X, Y)$.
$m\_tt(X, Y)$:- $t(X), p(X, Y)$.
$m\_tt(X, Y)$:- $m\_tt(X, Z), p(Z, Y)$.

We will refer to this transformation as the LC version (for linearised constraint).

The results are given in Table 1. These were obtained using the Aditi deductive database system.

Thus it appears that when the constraint has three answers that magic sets and the NRSU transformation are much less efficient than either of the other two, and that our transformation is more efficient than the context transformation. Note also that for constraints for which there is not a finite number of answers (such as $X > 1$) our transformation is still applicable.

6 Conclusion

We have seen how converting right-, mixed- and multi-linear programs to left-linear or left-multi-linear ones simplifies the propagation of constraints. Moreover, we have seen how this methodology may be used for *any* form of constraint on the bound variables of the adornment. In particular, the magic set transformation for such programs may be seen as an instance of our scheme.

As noted above, there are cases in which certain specialisations may be made. For this reason it seems best to have at hand an assortment of techniques which may be used in such cases, such as fold/unfold transformations, redundancy tests, and so forth. Thus a general technique, such as the one given in this paper, would be supplemented by certain specialised methods of constraint propagation.

A particular case in which the fold/unfold transformations [14] may be of use is when the constraints are not explicitly given. For example, consider the goal ?- $t_1(X, Y), t_2(Y, Z)$ where t_1 and t_2 are two different transitive closure programs. Clearly there is a constraint involved, in that the answers to the sub-query $t_1(X, Y)$ need only generate those Y values which are in the domain of the first argument of t_2. If these domains are large and the intersection of them small, then there will be a significant benefit in propagating such a constraint. A method for doing this, based on fold/unfold transformations was given in [6]; essentially the "domain" constraint is propagated into t_1. Clearly such methods will be useful in a general constraint propagation package.

It is important to note that the particular transformations used here make depend upon the definitions being *regular*, i.e. right-linear, left-linear etc. However, linearity alone is not enough. For example, consider the linear same generation program given below.

$$
\begin{array}{l}
sg(X, Y) \text{ :- } flat(X, Y). \\
sg(X, Y) \text{ :- } up(X, U), sg(U, V), down(V, Y).
\end{array}
$$

Our transformations will not work on this program, essentially as the expressions derivable from the corresponding grammar are of the form $u^n f d^n$, which is not regular, and hence are more difficult to optimize.

7 Acknowledgments

The authors would like to thank Isaac Balbin, David Kemp and Zoltan Somogyi for valuable discussions of this material. The support of the Australian Research Council through the Machine Intelligence Project and the Centre for Intelligent Decision Systems is gratefully acknowledged.

References

[1] R. Agrawal, Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries, *Proceedings of the 3th International Conference on Data Engineering* 580-590, Los Angeles, February, 1987. Also in *IEEE Transactions on Software Engineering* 14, 7, 879-885, 1988.

[2] R. Agrawal and H. Jagadish, Direct Algorithms for Computing the Transitive Closure of Database Relations, *Proceedings of the 13th Conference on Very Large Database Systems*, Brighton, September, 1987.

[3] R. Agrawal and H. Jagadish, Hybrid Transitive Closure Algorithms, *Proceedings of the 16th Conference on Very Large Database Systems* 326-334, Brisbane, August, 1990.

[4] F. Bancilhon, D. Maier, Y. Sagiv and J. Ullman, Magic Sets and Other Strange Ways to Implement Logic Programs, *Proceedings of the 5th ACM Symposium on Principles of Database Systems*, 1986.

[5] C. Beeri and R. Ramakrishnan, On the Power of Magic, *Proceedings of the 6th ACM Symposium on Principles of Database Systems* 269-283, 1987.

[6] J. Harland and K. Ramamohanarao, Constraints for Query Optimization in Deductive Databases, Proceedings of the Second Far East Workshop on Future Database Systems, Kyoto, April, 1992.

[7] J. Harland and K. Ramamohanarao, Constraint Propagation for Linear Recursive Rules, Technical Report 92/8, Department of Computer Science, University of Melbourne, June, 1992.

[8] D. Kemp, K. Ramamohanarao, I. Balbin and K. Meenakshi, Propagating Constraints in Recursive Deductive Databases, *Proceedings of the North American Conference on Logic Programming* 981-998, Cleveland, October, 1989.

[9] D. Kemp, K. Ramamohanarao and Z. Somogyi, Right-, Left- and Multi-linear Rule Transformations which Maintain Context Information, *Proceedings of the Sixteenth International Conference on Very Large Databases* 380-391, Brisbane, August, 1990.

[10] J.W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, Berlin, 1984.

[11] I. Mumick and H. Pirahesh, Overbound and Right-Linear Queries, *Proceedings of the ACM Symposium on Principles of Database Systems* 127-141, March, 1991.

[12] J. Naughton, R. Ramakrishnan, Y. Sagiv and J. Ullman, Efficient Evaluation of Right-, Left-, and Multi-Linear Rules, *Proceedings of the SIGMOD International Symposium on Management of Data* 235-242, Portland, May, 1989.

[13] J. Naughton, R. Ramakrishnan, Y. Sagiv and J. Ullman, Argument Reduction by Factoring, *Proceedings of the Fifteenth International Conference on Very Large Data Bases* 173-182, Amsterdam, August, 1989.

[14] H. Tamaki and T. Sato, Unfold/Fold Transformation of Logic Programs, *Proceedings of the Second International Conference on Logic Programming* 127-138, Uppsala, July, 1984.

[15] J. Ullman, Implementation of Logical Query Languages for Databases, *ACM Transactions on Database Systems* 10:3:289-321, 1985.