On Transformations into Linear Database Logic Programs

Foto Afrati[1], **Manolis Gergatsoulis**[2], **Maria Katzouraki**[2]

[1] Dept. of Electrical and Computer Engineering,
National Technical University of Athens, 157 73 Athens, Greece,
e_mail: afrati@softlab.ece.ntua.gr
[2] Inst. of Informatics & Telecom. N.C.S.R. 'Demokritos',
153 10 A. Paraskevi Attikis, Greece

Abstract. We consider the problem of transformations of logic programs without function symbols (database logic programs) into a special subclass, namely linear logic programs. Linear logic programs are defined to be the programs whose rules have at most one intentional atom in their bodies. a) We investigate linearizability of several syntactically defined subclasses of programs and present both positive and negative results (i.e. demonstrate programs that *cannot* be transformed into a linear program by *any* transformation technique), and b) We develop an algorithm which transforms any program in a specific subclass namely the piecewise logic programs into a linear logic program.

Keywords: program transformations, Datalog programs, program optimization, deductive databases.

1 Introduction

Logic program transformation has been the object of a large amount of research activity recently. The program transformation methodology is often followed in order to improve the efficiency of the program.

In this paper, we consider logic programs without function symbols and investigate the problem of transforming them into a syntactically simple subclass, namely the linear programs. In general, it is desirable, whenever possible, to replace non-linear programs by equivalent linear programs, because there are efficient algorithms for the computation of the latter which do not extend to the former. Linear programs have been widely studied [1, 13, 4] both as concerns their efficiency and the possibility of transformation of non-linear programs into linear. A program is linear if all the rules are linear, i.e., there is at most one intentional atom in the rule's body.

As an example, consider the following program which checks if there is a path joining two nodes of a graph:

$path(X,Y) \leftarrow edge(X,Y).$
$path(X,Y) \leftarrow path(X,Z), path(Z,Y).$

This program is equivalent to:

$path(X, Y) \leftarrow edge(X, Y).$
$path(X, Y) \leftarrow edge(X, Z), path(Z, Y).$

The second is a linear program, while the first is not.

In this paper we investigate the problem of transformation into linear programs from two different perspectives: a) We discuss the expressivity of linear programs by demonstrating the fact that this subclass of programs is quite rich since it can describe some quite difficult problems and, still, on the other end, there are some very simple programs that are proven not to be linearizable (i.e., they cannot be transformed into linear programs). b) We present an algorithm that transforms a specific subclass of programs, namely the piecewise linear programs, into linear programs. This algorithm still applies in the general case (i.e., when function symbols are allowed).

The results in part (a) above are presented in a unified way to point out certain limits that program transformations cannot go beyond. They are proven after developing elaborate technical tools [1, 3, 2]. The results in part (b) are proven by presenting an algorithm that uses unfold/fold techniques in a coordinated form to arrive in a certain syntactically simpler program. On top on techniques used in [16, 17], we had to develop some more technical tools to show our results.

The results presented here are also interesting in view of the fact that they attack, the problem of program transformation from the point of view of defining a priori the subclass of programs we aim at.

The rest of this paper is organized as follows. After giving some preliminaries in section 2, we investigate linearizability of several syntactically defined subclasses of programs and present both positive and negative results in section 3. In section 4, we present our transformation algorithm, and in section 5 we see an application of this algorithm. Finally, in section 6, a conclusion is given.

2 Preliminaries

In the following, we assume familiarity with the basic terminology of first order logic and logic programming [12].

2.1 Datalog programs

Deductive database systems divide their information into two categories: The *data* which are represented by a predicate with constant arguments (all true tuples are stored in the database) and the *rules* which define new predicates in terms of existing ones. Rules are Horn clauses without function symbols. The data are often referred to as the EDB (*extensional database*) and the rules as the IDB (*intensional database*). A collection of rules is also called a *Datalog program*.

A predicate p *depends on* a predicate q iff there is a rule with p in its head and either q or a predicate r which depends on q, in its body. An atom B is *called in the body* of a clause C iff B is unifiable with an atom in the body of C.

Definition 1. The *transitive closure of a predicate p in a program P w.r.t. deduction* is a set of clauses S_P where: C is in S_P if its head predicate is p, or there is a clause C' in S_P and the head of C is called in the body of C'.

Definition 2. A predicate p is a *recursive predicate* iff p depends on itself. Two predicates p and q are *mutually recursive* iff p depends on q and q depends on p. A predicate p is said to be a *non-recursive predicate* iff p is not recursive.

Definition 3. A Datalog program P is said to be *peicewise linear* iff for every rule in P, at most one atom with a predicate which is mutually recursive with the predicate in its head, is included in its body. P is said to be *linear* iff every rule in P has at most one intensional atom in its body.

2.2 Unfold/fold transformations

Unfold/fold transformations[15, 17, 8, 9] for definite clause programs were first formulated in [18] so as to preserve the meaning of programs. The *meaning* $M(P)$ of a logic program P is defined as: $M(P) = \{A|A$ is a ground atom which is a logical consequence of $P\}$. $M(P)$ is identical to[12] the *least Herbrand model* of P. In the system in[18], which is used in this paper, we start from an *initial program* P_0, and produce a sequence of programs by applying transformation rules:

Definition 4. An *initial program* P_0, is a logic program satisfying the following conditions:

a) P_0 is divided into two disjoint sets of clauses, P_{new} and P_{old}. The predicates defined in P_{new} are called *new predicates* while those defined in P_{old} are called *old predicates*.

b) The new predicates appear neither in P_{old} nor in the bodies of the clauses in P_{new}.

Definition 5. Let C be a clause in $P_l \ \ (l \geq 0) : A \leftarrow B, K$, where B is an atom and K a conjunction of atoms, and $C_1, ..., C_m$ all clauses in P_l, whose heads are unifiable with B by most general unifiers $\theta_1, ..., \theta_m$. The result of *unfolding C at B* is the set $\{C'_1, ..., C'_m\}$ such that if $C_j \ \ (1 \leq j \leq m)$ is $B_j \leftarrow Q_j$ and $B_j\theta_j = B\theta_j$, then C'_j is $(A \leftarrow Q_j, K)\theta_j$. Then, $P_{l+1} = (P_l - \{C\}) \cup \{C'_1, ..., C'_m\}$. C is called the *unfolded clause* and $C_1, ..., C_m$ the *unfolding clauses*. The atom A_i is called the *unfolded atom*.

Definition 6. Let C be the clause $H \leftarrow K, L$ in P_l and F the clause $A \leftarrow K'$ in P_{new}, where K,K', and L are conjunctions of atoms. Then, the clause C' : $H \leftarrow A\theta, L$ is the *result of folding C using F*, if there exists a substitution θ satisfying the following conditions:

a) $K'\theta = K$.

b) All variables in the body of F, which do not appear in the head of F are mapped through θ into distinct variables which do not occur in C'.

c) Either the head predicate of C is an old predicate, or C has been unfolded at least once in the sequence $P_0, P_1, ..., P_{l-1}$.

d) F is the only clause in P_0 whose head is unifyable with $A\theta$. Then, $P_{l+1} = (P_l - \{C\}) \cup \{C'\}$. C is called the *folded clause*, and F is called the *folding clause*. $B_0\theta$ is called the *atom introduced by folding*.

2.3 On the Complexity of Datalog programs

Because of their recursive nature, queries expressed in Datalog are harder to evaluate (from the point of view of *parallel* complexity): while first-order queries are in deterministic log-space [20] (even in AC^0), Datalog programs are sometimes log-space complete for $\mathcal{P}$:

$access(X) \leftarrow source(X).$
$access(X) \leftarrow access(Y_1), access(Y_2), triple(Y_1, Y_2, X).$

The above program encodes the well-known *path system accessibility* problem [6]: the EDB predicates *source* and *triple* represent, respectively, source nodes and accessibility conditions; $triple(y_1, y_2, x)$ means that if y_1, y_2 are accessible from the source nodes, then so is x.

A large body of recent research has addressed the problems of finding efficient evaluation methods and compile-time optimization techniques for Datalog programs (see [5] for a survey). These studies usually concentrate on *syntactically restricted* Datalog programs; two common approaches are the following: a) Restrict the *width* (number of arguments) of the IDB predicates. b) Impose a *linearity* condition on the rules (as, e.g., in [10, 13, 14]).

It has been observed that linear Datalog programs can be evaluated in $\mathcal{NC}^2$ (cf. [7, 19]). Moreover, all the Datalog programs currently known to be $\mathcal{P}$-complete (see [7, 19, 4]) can be shown to require non-linear rules, because in each case there is a *first-order reduction* from path system accessibility. The question naturally suggested, then, is the following: are there Datalog programs *in* $\mathcal{NC}$ which are not equivalent to linear programs? This question has been answered affirmatively in [1]: There exist Datalog programs in $\mathcal{NC}^2$ which are not equivalent to any linear program (see theorems 9, 10 below). Programs in theorems 9, 10 belong to the class of *elementary chain* programs [19, 4];

3 Linearizable and non-linearizable Datalog Programs

In this section we focus on databases with only binary relations; such databases can be thought of as directed graphs with edges labelled by EDB predicates. We consider the special class of *chain* queries, which detect the existence of certain paths. Among them there are queries in $\mathcal{NC}^2$ requiring non-linear Datalog programs[1].

3.1 Chain Queries and Linear Recursion

Let $\mathcal{D} = (D, r_1, \ldots, r_n)$ be a database where each r_i is binary, and let $\Sigma = \{R_1, \ldots, R_n\}$ be an alphabet containing one letter R_i for each relation r_i (we use the same symbol, R_i, for the letter of Σ corresponding to r_i and for the EDB predicate denoting r_i). A *path spelling a word* $R_{i_1} \cdots R_{i_l} \in \Sigma^*$ is a sequence $u_1, \ldots, u_{l+1}$ of elements of D such that $(u_j, u_{j+1}) \in r_{i_j}$, for $j = 1, \ldots, l$; if $l = 0$, the path spells the empty word, ϵ.

For any language $L \subseteq \Sigma^*$, the *chain query* Q_L *obtained from* L is defined as: $Q_L(\mathcal{D}) = \{(u,v)$: there is a path of $\mathcal{D}$ from u to v, spelling a word in $L\}$.

Chain queries obtained from *context-free* languages are of particular interest: a context-free grammar G (generating a language $L(G)$) corresponds in a natural way to a Datalog program computing the chain query $Q_{L(G)}$. We illustrate this correspondence by an example:

Example 1. If G is $I \to R_1 I R_2 I \mid \epsilon$, then $Q_{L(G)}$ is computed by the program:

$I(X,Y) \leftarrow R_1(X,Z_1), I(Z_1,Z_2), R_2(Z_2,Z_3), I(Z_3,Y).$
$I(X,X).$

Datalog programs as above are called *elementary chain* programs [19]. We now turn to chain queries which are *linearizable.*

Example 2. The language $L = \{R_1^i R_2^i R_3^j R_4^j : i,j \geq 0\}$ can be shown to be not linear context free. Still there is a linear Datalog program that expresses Q_L:

$I(X,Y) \leftarrow P(X,Z,Z,Y).$
$P(X_1,Y_1,X_2,Y_2) \leftarrow P(X_1',Y_1',X_2,Y_2), R_1(X_1,X_1'), R_2(Y_1',Y_1).$
$P(X_1,Y_1,X_2,Y_2) \leftarrow P(X_1,Y_1,X_2',Y_2'), R_3(X_2,X_2'), R_4(Y_2',Y_2).$
$P(X_1,X_1,X_2,X_2).$

The same is true for the language $L = \{R_1^i R_2^i R_3^i : i \geq 0\}$:

$I(X,Y) \leftarrow P(X,Z,Z,Y).$
$P(X_1,Y_1,X_2,Y_2) \leftarrow P(X_1',Y_1',X_2',Y_2), R_1(X_1,X_1'), R_2(Y_1',Y_1), R_3(X_2,X_2').$
$P(X_1,X_1,X_2,X_2).$

It can be shown that the class of linearizable chain queries is closed under general substitutions. A *substitution* is a mapping f from Σ to subsets of Σ^*; it is extended to strings by defining $f(R_{i_1} \cdots R_{i_l}) = \{\rho_{i_1} \cdots \rho_{i_l} : \rho_{i_j} \in f(R_{i_j})\}$, and to languages by defining $f(L) = \bigcup_{\rho \in L} f(\rho)$.

Theorem 7. *If Q_L is linearizable, and $Q_{f(R_i)}$ is linearizable, $i = 1, \ldots, n$, then $Q_{f(L)}$ is linearizable.*

Corollary 8. *If $Q_L, Q_{L'}$ are linearizable, then $Q_{L \cup L'}$, $Q_{LL'}$, Q_{L^*} are linearizable.*

3.2 Non-linearizable Chain Queries in $\mathcal{NC}^2$

Consider the following context-free language $L^0 \subseteq \{R_1, R_2\}^*$:

$L^0 = \{\rho : \rho$ has the same number of occurrences of R_1 and $R_2\}$.

By the results in [4, 19] the chain query Q_{L^0} is in $\mathcal{NC}^2$:

Theorem 9. *The chain query Q_{L^0} is not linearizable.*

Theorem 10. *If L is generated by one of the context-free grammars below, then the chain query Q_L is not linearizable:*
a) $I \rightarrow IR_1I(R_2IR_1I)^j \mid \epsilon$, *where* $j \geq 1$.
b) $I \rightarrow (IR_1)^iIR_2I(R_1I)^j \mid \epsilon$, *where* $i, j \geq 1$.

In [4] it is shown that the context-free languages in Theorem 10 can be accepted by pushdown automata satisfying the polynomial stack property, and therefore the corresponding chain queries are in $\mathcal{NC}^2$.

4 Transforming piecewise linear to linear programs

In this section we show that a piecewise linear Datalog program can always be transformed into an equivalent linear program using unfold/fold transformations.

Example 3. Let $P = \{C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8, C_9\}$ be the following piecewise linear Datalog program:

$C_1: \ a(X,Y) \leftarrow edb1(X,Y).$
$C_2: \ a(X,Y) \leftarrow b(X,Z), a(Z,Y).$
$C_3: \ b(X,Y) \leftarrow edb2(X,Y).$
$C_4: \ b(X,Y) \leftarrow edb3(X,Z,W), c(Z,W,F), b(F,Y).$
$C_5: \ c(X,Y,Z) \leftarrow edb4(X,Y,W), d(W,Z).$
$C_6: \ d(X,Y) \leftarrow edb5(X,Y).$
$C_7: \ d(X,Y) \leftarrow edb6(X,Z), e(Z,Y).$
$C_8: \ e(X,Y) \leftarrow edb7(X,Y).$
$C_9: \ e(X,Y) \leftarrow edb8(X,Z), d(Z,Y).$

P is not linear due to the non-linear clauses C_2 and C_4. We will replace C_4 with linear clauses. For this, we unfold C_4 at '$c(Z,W,F)$'. We obtain:

$C_{10}: \ b(X,Y) \leftarrow edb3(X,Z,W), edb4(Z,W,W1), d(W1,F), b(F,Y).$

Now we introduce the following Eureka definition.

$D_1: \ new1(X,Y) \leftarrow d(X,Z), b(Z,Y).$

Then, we fold C_{10} using D_1. We take:

$C_{11}: \ b(X,Y) \leftarrow edb3(X,Z,W), edb4(Z,W,W1), new1(W1,Y).$

Now, we try to find a (linear) recursive definition for the predicate '$new1$'. For this, we unfold D_1 at '$d(X,Z)$' using the clauses C_6 and C_7. We obtain:

$C_{12}: \ new1(X,Y) \leftarrow edb5(X,Z), b(Z,Y).$
$C_{13}: \ new1(X,Y) \leftarrow edb6(X,W), e(W,Z), b(Z,Y).$

Unfolding C_{13} at '$e(W,Z)$' using C_8 and C_9 we get:

$C_{14}: \ new1(X,Y) \leftarrow edb6(X,W), edb7(W,Z), b(Z,Y).$
$C_{15}: \ new1(X,Y) \leftarrow edb6(X,W), edb8(W,Q), d(Q,Z), b(Z,Y).$

Folding C_{15} using D_1 we obtain:

$C_{16}: new1(X,Y) \leftarrow edb6(X,W), edb8(W,Q), new1(Q,Y)$.

$\{C_{12}, C_{14}, C_{16}\}$ is a linear program for '$new1$'. The new program is $P_1 = \{C_1, C_2, C_3, C_5, C_6, C_7, C_8, C_9, C_{11}, C_{12}, C_{14}, C_{16}\}$. P_1 is equivalent to $P \cup \{D_1\}$. Similarly, starting from P_1, we can replace C_2 by an equivalent set of linear clauses. In this way we obtain a linear program.

In the following, we present an algorithm based on unfold/fold transformation rules, which transforms a piecewise linear Datalog program into a linear program, building on top on techniques used in [17].

Definition 11. An *unfolding selection rule* (or *U-rule*) is a (partial) function from clauses to atoms. The value of the function for a clause is a body atom called the *selected atom*.

Definition 12. Let P be a program, C a clause and S a U-rule. An *unfolding tree* (or *U-tree* for short) T for $< P, C >$ via S is a tree labelled with clauses, such that:

a) C is the root label of T, and

b) If M be a node labelled by$A \leftarrow B, K$, and B the atom selected by S. Then, for each clause $B' \leftarrow L$ in P for which a most general unifier θ of B and B' exists, there is a child node N of M labelled by: $(A \leftarrow L, K)\theta$.

We suppose that an unfolding selection rule S is uniquely determined by the set of IDB atoms in the bodies of the clauses.

Definition 13. A nonempty tree T' is called an *upper portion* of a tree T iff the following hold:

a) The set of nodes of T' is contained in the set of nodes of T,

b) If N is a node of T' then every ancestor of N in T is also an ancestor of N in T' and,

c) If N is in T' then any brother of N in T is also a brother of N in T'.

An upper portion of T consisting of a single node is called a *trivial upper portion*.

For any program P and a clause C, if L is the set of leaves of an upper portion of a U-tree for $< P, C >$ via S, then[17] $M(P \cup \{C\}) = M(P \cup L)$.

Definition 14. Let P be a program, C a clause, and S a U-rule. A clause D in a node of a U-tree T for $< P, C >$ via S is said to be *foldable* iff there is an ancestor F of D in T and a tuple I of IDB atoms such that the tuples of all IDB atoms in the bodies of both C and F are instances of I. F is called a *folding ancestor* of D.

Definition 15. Let P be a Datalog program, C a clause, and S a U-rule. The U-tree T for $< P, C >$ via S is said to be *linearizable* iff there is a finite upper portion U of T such that each leaf clause of U is either a linear clause or a foldable clause or a failing clause[3]. U is said to be a *linearizable upper portion* of T.

Definition 16. A non-linear clause C in a piecewise linear program P is *minimally non-linear* iff the transitive closure w.r.t. deduction, of any body atom B of C, which is not mutually recursive with the head of C, is a linear program.

We can easily show that, for any piecewise linear Datalog program P, either P is linear or there is (at least one) minimally non-linear clause C in P.

Definition 17. A *linear unfolding selection rule* is an unfolding selection rule S such that S always selects an IDB body atom (if any) of C with a predicate p whose transitive closure is a linear program, otherwise $S(C)$ is undefined.

It is easy to see that if a clause C in a program P is minimally non-linear then, there is always a body atom of C whose transitive closure in P w.r.t. deduction is a linear program. Therefore, a linear U-rule is always defined for any minimally non-linear clause.

Lemma 18. *Let C be a minimally non-linear clause in $P \cup \{C\}$, S a linear U-rule and U a U-tree for $< P, C >$ via S. Then all non-linear clauses in the set of leaves L of an upper portion of U are also minimally non-linear clauses in $P \cup L$.*

An immediate consequence of lemma 18 is that when we unfold a minimally non-linear clause C in a program P via an unfolding selection rule S, then S is also defined for all non-linear clauses (if any) produced by this unfolding (as these clauses are minimally non-linear).

Procedure 4.1 (*Clause Linearization procedure (CLP)*).
Input : a piecewise linear program P, a minimally non-linear clause C in P and a linear U-rule S.
Output : a set of linear clauses L and a set of new predicate definitions ED.

1. Construct a linearizable U-tree T for $< P, C >$ via S and select a minimal linearizable upper portion U of T.
2. For every foldable leaf clause D of U construct a clause E with a fresh predicate symbol in its head, and a tuple I of IDB atoms in its body such that both, the tuple I_D of the IDB atoms in the body of D and the tuple I_F of the IDB atoms in the body of the folding ancestor F of D, are instances of[4] I. The head arguments of E is the minimal subset of the variables in the body of E such that both D and F can be folded using E. Put E in ED unless E differs from a clause in ED only in the names of their head predicates or/and in the order of the arguments of their heads.

[3] A *failing clause* is a clause with a body atom that does not unify with the head of any clause in the program. A failing clause can be removed from the program.

[4] The best choice is to use as I the *most specific generalization*[11] of ID and IF. In [11], an algorithm to compute the most specific generalization of a set of expressions is given.

3. Select the (possibly trivial) minimal upper portion MU of U so as each leaf clause of MU is either a failing clause or a linear clause or it can be folded using a clause in ED. Collect the set of leaves of MU and perform all possible foldings using the clauses in ED obtaining a set LC of clauses.
4. For each clause E_i in ED compute a corresponding linear definition L_{E_i} as follows: Construct a (non-trivial) minimal U-tree U_{E_i} for $< P, E_i >$ via S such that each leaf clause of U_{E_i} is either a failing clause or a linear clause or it can be folded using a clause in ED. Collect the set of leaves of U_{E_i} and perform all possible foldings using the clauses in ED obtaining L_{E_i}.
5. Let $L = LC \cup L_{E_1} \cup .... \cup L_{E_n}$.

All clauses in ED are by construction, non-linear clauses (see step 3). Moreover, it is easy to see that the linear selection rule S (used in step 1) is always defined for the clauses in ED in the program $P \cup ED$. This is due to the fact that all clauses in ED are minimally non-linear in $P \cup ED$.

Theorem 19. *The clause linearization procedure (CLP) applied to a minimally non-linear clause C of a piecewise linear Datalog program P always terminates and returns a set of linear clauses L and a set of new definitions ED such that $P \cup ED$ is equivalent to $(P - \{C\}) \cup L$.*

Proof. (Sketch) a) *Termination:* It suffices to prove that i) For every linear U-rule S there exists a *finite* minimal linearizable upper portion U (see step 1) and, ii) for every definition E_i in ED, there exists a finite minimal (non-trivial) tree U_{E_i} whose leaf nodes are failing clauses, linear clauses or they can be folded using clauses in ED (see step 4(a)).

i) Since a linear unfolding selection rule always selects an atom whose transitive closure is a linear program, the number of the IDB atoms of each clause resulting from an unfolding step is less than or equal to the number of IDB atoms of the unfolded clause. Moreover, since the number of IBD predicates is finite then so is the tuples of predicates of the IDB atoms in the bodies of these clauses. Therefore, there is a finite minimal linearizable upper portion of U.

ii) Since in the construction of U_{E_i} we use the same U-selection rule S, and S is uniquely determined by the set of IDB atoms in the body of that clause, we have that the tree U_{E_i} will also be constructed in a finite number of unfolding steps.

b) *Correctness:* It is easy to see that all clauses in L are linear clauses. It is sufficient to show that $M(P \cup ED) = M((P - \{C\}) \cup L)$. Since all the leaf clauses of MU (see step 3) are old clauses the folding operations performed this step are correct and thus $M(P \cup ED) = M((P - \{C\}) \cup LC \cup ED)$. Moreover, since the folded clauses in step 4b are new clauses and they all have been unfolded at least once (as U_{E_i} is non-trivial (step 4a)), all folding operations are again correct. Thus, $P \cup ED$ is equivalent to $(P - \{C\}) \cup L$.

Procedure 4.2 (*Program Linearization procedure (PLP)*).
Input : a piecewise linear program P and, a linear U-rule S.

Output : a set of Eureka definitions ED and a set L of linear clauses.

Let $i = 0$ and $P_i = P$.

Let NL be the subset of all non-linear clauses in P.

while NL is non-empty **do**

- Select a minimally non-linear clause C from NL.
- Apply CLP with input P_i, C and S and output L_i and ED_i.
- Let $P_{i+1} = \{P_i - \{C\}) \cup L_i$.
- Let $NL = NL - \{C\}$, and i = i + 1.

Let $ED = \bigcup_i ED_i$ for all i, and $L = \bigcup_i L_i$ for all i.

Theorem 20. *The program linearization procedure (PLP) applied to a piecewise linear Datalog program P always terminates and returns a set of linear clauses L and a set of new definitions ED such that if NL is the set of all non-linear clauses in P, then $P \cup ED$ is equivalent to $(P - NL) \cup L$.*

Proof. (Sketch) *Termination:* Procedure always terminates since there is a finite number of clauses in NL and in each iteration of PLP exactly one clause in NL is replaced by a set of linear clauses and the clause linearization procedure always terminates.

Correctness: It is an immediate consequence if the correctness of the clause linearization procedure.

5 The application of the algorithm

In fig. 1 and fig. 2 we can see the application of the procedure 4.1 on the clause C_4 of the program of example 3. The result of the application of the procedure is the replacement of the clause C_4 by the set of linear clauses $\{C_{11}, C_{12}, C_{14}, C_{16}\}$. The underlined atoms in non leaf nodes are the atoms selected by the U-rule. Fig. 1, corresponds to step 1 of procedure 4.1. The loop found is used to introduce the definition D_1 (step 2). D_1 is used to fold C_{10} (step 3) as it is shown in fig. 2. Finally, fig. 3 corresponds to step 4 (discovery of a linear definition for $new1$).

6 Conclusions

The problem of transforming database logic programs (Datalog programs) into equivalent linear programs, is investigated in this paper. We present both positive and negative results about linearizability of several syntactically defined subclasses of programs and develop an algorithm, based on unfold/fold transformations, which transforms any piecewise linear logic program into an equivalent linear program.

[htb]

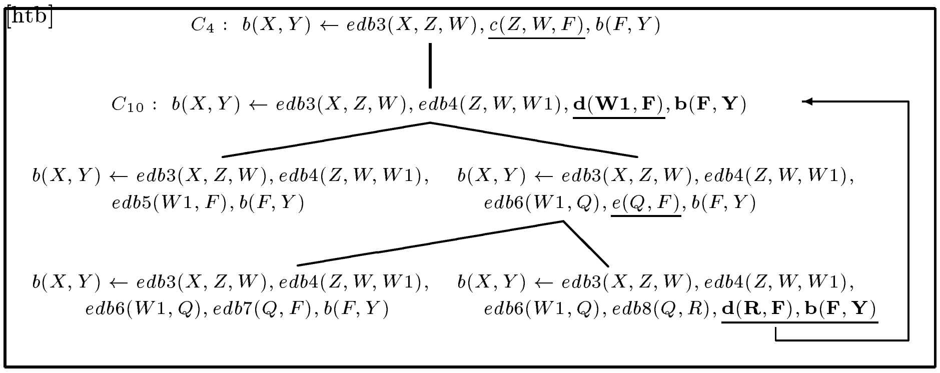

Fig. 1. A minimal linearizable upper portion of a U-tree for $< P, C_4 >$.

[htb]

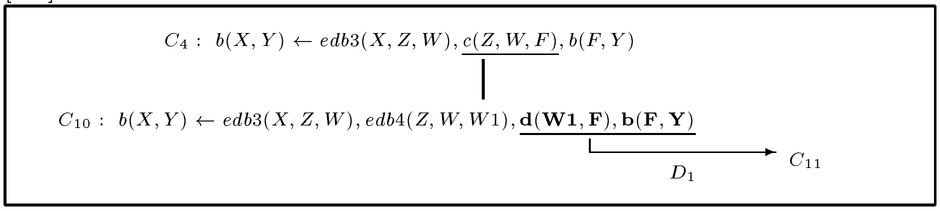

Fig. 2. A minimal upper portion of the U-tree in fig. 1, which can be folded using D_1.

References

1. F. Afrati and S. Cosmadakis. Expressiveness of restricted recursive queries. In *Proc. 21st ACM Symp. on Theory of Computing*, pages 113–126, 1989.
2. F. Afrati, S. Cosmadakis, and M. Yannakakis. On datalog vs. polynomial time. In *Proc. 10th ACM Symp. on Principles of Database Systems*, pages 113–126, 1991.
3. F. Afrati, S. Cosmadakis, and M. Yannakakis. On datalog vs. polynomial time. *J. Computer and Systems Sciences*, 51(2):117–196, 1995.
4. F. Afrati and C. H. Papadimitriou. The parallel complexity of simple chain queries. In *Proc. 6th ACM Symp. on Principles of Database Systems*, pages 210–213, 1987.
5. F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proc. ACM Conf. on Management of Data*, pages 16–52, 1986.
6. S. A. Cook. An observation on time-storage trade off. *J. Computer and System Sciences*, 9:308–316, 1974.
7. S. S. Cosmadakis and P. C. Kanellakis. Parallel evaluation of recursive rule queries. In *Proc. 5th ACM Symp. on Principles of Database Systems*, pages 280–293, 1986.
8. M. Gergatsoulis. *Logic program transformations: Rules and application strategies.* PhD thesis, Dept. of Computer Science, University of Athens, 1994. (In Greek).
9. M. Gergatsoulis and M. Katzouraki. Unfold/fold transformations for definite clause programs. In *Programming Language Implementation and Logic Programming (PLILP'94), LNCS 844*, pages 340–354. Spinger-Verlang, 1994.

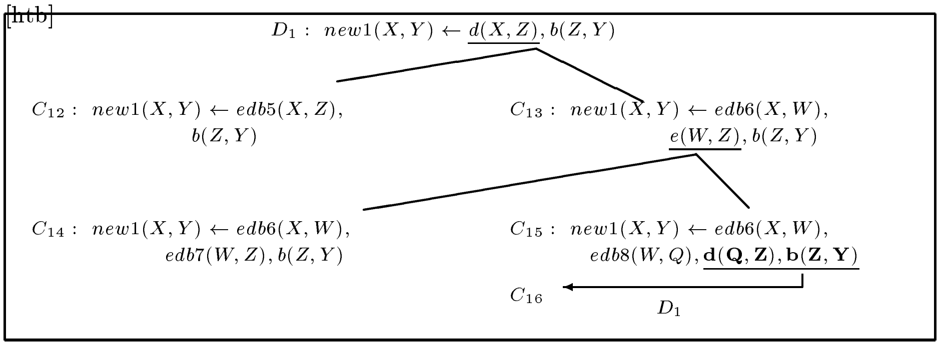

Fig. 3. A minimal (non trivial) upper portion of a U-tree for $< P, D_1 >$ whose non linear leaf clauses are foldable using ED.

10. Y. E. Ioannidis. A time bound on the materialization of some recursively defined views. In *Proc. 11th Int'l Conf. on Very Large Data Bases*, pages 219–226, 1985.
11. J-L. Lasser, M. J. Maher, and K. Marriott. Unification revisited. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann Publishers,Inc., 1988.
12. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
13. J. F. Naughton. Data independent recursion in deductive databases. In *Proc. 5th ACM Symp. on Principles of Database Systems*, pages 267–279, 1986.
14. J. F. Naughton and Y. Sagiv. A decidable class of bounded recursions. In *Proc. 6th ACM Symp. on Principles of Database Systems*, pages 227–236, 1987.
15. A. Pettorossi and M. Proietti. Transformation of logic programs : Foundations and techniques. *The Journal of Logic Programming*, 19/20:261–320, May/July 1994.
16. M. Proietti and A. Pettorossi. Synthesis of eureka predicates for developing logic programs. In *LNCS no. 432, Proc. of the 3rd European Symposium on Programming*, pages 306–325. Springer-Verlag, 1990.
17. M. Proietti and A. Pettorossi. The loop absorption and the generalization strategies for the development of logic programs and partial deduction. *The Journal of Logic Programming*, 16(1 & 2):123–162, May 1993.
18. H. Tamaki and T. Sato. Unfold/fold transformations of logic programs. In *Second International Conference on Logic Programming*, pages 127–138, 1984.
19. J. D. Ullman and A. Van Gelder. Parallel complexity of logical query programs. In *Proc. 27th IEEE Symp. on Foundations of Comp. Sci.*, pages 438–454, 1986.
20. M. Y. Vardi. The complexity of relational query languages. In *Proc. 14th ACM Symp. on Theory of Computing*, pages 137–146, 1982.