

Necessary and Sufficient Conditions to Linearize Doubly Recursive Programs in Logic Databases

WEINING ZHANG and CLEMENT T. YU

University of Illinois at Chicago

and

DANIEL TROY

Purdue University Calumet

Linearization of nonlinear recursive programs is an important issue in logic databases for both practical and theoretical reasons. If a nonlinear recursive program can be transformed into an equivalent linear recursive program, then it may be computed more efficiently than when the transformation is not possible. We provide a set of necessary and sufficient conditions for a simple doubly recursive program to be equivalent to a simple linear recursive program. The necessary and sufficient conditions can be verified effectively.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*query processing*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*logic programming*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Equivalence, knowledge database, linear and nonlinear recursive rules, logic database

1. INTRODUCTION

Processing recursive queries efficiently has been a central issue of research in the deductive database area. Numerous methods [1, 3–11, 13–15, 17, 19, 21, 23–25, 31, 33, 35] have been proposed to evaluate recursive queries. These methods differ from each other in the types of queries that they can evaluate and in the efficiency with which they process queries. Since whether a query can be processed by a given method is determined by the syntactical structure of the query (i.e., the structure of logic rules defining the query), two equivalent queries (in terms of the answers on all possible database states) may be processed by using different evaluation methods with significantly different efficiencies. We consider two

Authors' addresses: W. Zhang and C. Yu, Department of Electrical Engineering and Computing Science, University of Chicago, Chicago, IL. 60680; D. Troy, Department of Mathematical Science, Purdue University Calumet, Hammond, IN. 46323.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1990 ACM 0362-5915/90/0900-0459 \$01.50

types of queries, linear and nonlinear recursive queries, and study the characteristics of nonlinear recursive programs which can be transformed into equivalent linear recursive programs.

Motivations to study the linearization of nonlinear recursive programs include the following. From a practical point of view, strategies designed to solve linear recursions [3–6, 10, 11, 13–15, 26, 29, 35] are usually more efficient than those proposed for solving general recursions [4, 5, 8, 10, 23, 25, 29, 31, 33]. By transforming a nonlinear recursive program into an equivalent linear one, queries can be answered with a higher efficiency by evaluating the transformed linear program. For example, a common evaluation method for answering a general recursive query is the Naive method [4, 10]. But if a query is known to be linear recursive, we can apply the Semi-Naive [4, 10] method to evaluate it. It is well known, since the Semi-Naive method avoids duplicate computations, that it is likely to be more efficient than the Naive method. Notice that although the Semi-Naive method can also be used for processing nonlinear recursive queries, it is usually not as efficient for an equivalent linear recursive query. Another example is given in [29], where a nonlinear recursive program is used to show that when it is processed with the Generalized Magic Set method [5], one can obtain a set of nonregular magic rules whose evaluation is as complicated as that of the original program. On the other hand, this program does have an equivalent linear recursive counterpart which can be processed efficiently by one of the methods designed to solve linear recursions.

From a theoretical point of view, it is a fundamental issue to distinguish the “easy” or “simple” linearizable recursive programs from the “difficult” or “complicated” nonlinearizable recursive programs. It is important to understand the complexity of processing recursive queries in logic databases. Similar results for the distinction between “simple” (tree) and “complicated” (cyclic) relational queries have been obtained [34].

There are several papers [7, 18, 25, 28] related to our work. The problem of deciding the equivalence of two arbitrary Horn clause logic programs has been shown to be unsolvable in [28]. In [25], a method is proposed to transform a given logic program into an equivalent one by eliminating redundant predicates and redundant rules. This is a polynomial space algorithm. Our work differs from theirs. First, we transform a simple nonlinear recursive program into a linear program by using a simple, natural conversion method to study the conditions under which the two programs are equivalent. Therefore, the rules to be considered are not as arbitrary as those in [28]. Secondly, the conditions that we obtain for linearizing doubly recursive programs can be effectively verified.

Beeri et al. [7] indicate that whether a program containing only binary chain rules is equivalent to a program containing only regular (left or right) linear binary chain rules is undecidable. The programs considered in [7] can be mutually recursive, with a recursion order greater than two, and contain more than one recursive rule. The programs that we consider are directly, doubly recursive and have only two rules (one recursive and one nonrecursive). Further, we allow arbitrary arity of predicates, and do not require the chain property. They [7] require every rule to be a binary chain rule. Their result states that whether a chain program is equivalent to some arbitrary linear chain program is

undecidable. Nothing has been said about whether the program is equivalent to a given linear program. Because of these differences, they [7] obtained a negative result, and we have a positive one.

In [18], an approach to linearize nonlinear recursive programs is proposed. The method is based on algebra and the power-subassociativity property of bilinear functions. Their [18] result is more general than ours, since they consider mutually recursive predicates. They do not, however, provide a practical procedure. Thus the complexity of verifying their conditions for a given program is not clear. On the other hand, our result provides an effective way of identifying linearizable doubly recursive programs. Furthermore using our result, some doubly recursive programs are linearizable but do not satisfy their conditions. For example, based on our result, the program given in Example 3.6 is linearizable but the corresponding bilinear function is not power-subassociative, and therefore cannot be linearized by using the method in [18].

In this paper we consider a class of nonlinear recursive programs, called simple doubly recursive programs, and provide a set of necessary and sufficient conditions for such programs which are equivalent to the linear recursive programs obtained through a simple linearization method. A preliminary version of this paper [37] has been published in *ACM SIGMOD 87*. In Section 2 of this paper we provide definitions and a brief review of our previous result. In Section 3 we provide a set of necessary and sufficient conditions for a simple doubly recursive program to be linearizable and a number of examples to illustrate the result. A sketch of the proof of the conditions is given in Section 4. A proof of sufficiency is given in the Appendix.

2. DEFINITIONS AND PREVIOUS RESULT

A logic database consists of two components: the *extensional database* (EDB) containing a set of base (i.e., stored) relations (predicates); and the *intensional database* (IDB) containing a set of deductive rules in the form of Horn clauses. The EDB and IDB partition all predicates in the logic database, so that all derived predicates are in the IDB and all base predicates are in the EDB. Throughout this paper, we use the terms *relation* and *predicate* indistinguishably.

A *rule* is a function-free Horn clause of the form $B :- A_1 A_2 \cdots A_k$, where B and the A_i 's are predicates; B is the *head* of the rule and the conjunction of A_i 's is the *body* of the rule. A rule is directly *recursive* if the head predicate B also appears in the body. Under certain restrictions, any program can be converted into an equivalent one that only contains directly recursive rules [28]. Such a transformation may not always result in an efficient program. A *simple recursive program* contains exactly two rules with the same head: an exit rule, which is not recursive, and a directly recursive rule. The *order of recursion* for a rule is the number of occurrences of the head predicate B in the body of the rule. If a recursive rule has an order of recursion $R = 1$, it is a *linear recursive* rule; if $R > 1$, it is a *nonlinear recursive* rule. If $R = 2$, it is a *doubly recursive* rule. In this paper we consider simple doubly recursive programs only.

In the following we define various terms that we used to formulate programs and to present our results. We will use the following example to illustrate various concepts.

Example 2.1 (Taken from [32]). Consider a directed graph with colored arcs. Assume that there are two base relations *redarc* and *bluearc* such that *redarc(xy)/bluearc(x, y)* means that the edge from a node x to a node y is red/blue. The following program defines paths consisting of alternating red and blue arcs, beginning and ending with a red arc.

$$\begin{aligned} r_e: \text{path}(x_1x_2) &:- \text{redarc}(x_1x_2) \\ r_r: \text{path}(x_1x_2) &:- \text{path}(x_1u_1) \text{bluearc}(u_1u_2) \text{path}(u_2x_2). \end{aligned}$$

The meaning of this program is that there is a path from a node x_1 to a node x_2 when there is a red arc from x_1 to x_2 , or there exist some nodes u_1 and u_2 such that there are paths from x_1 to u_1 and from u_2 to x_2 , as well as a blue arc from u_1 to u_2 . The first rule is an exit rule and the second a recursive one.

Given a rule, all variables appearing in the head are called *distinguished* variables; all variables appearing in the body only are called *nondistinguished* variables. Let X be a vector of n distinguished variables and U a vector of p nondistinguished variables. For every predicate of arity g in the body of the rule, g variables are selected from the $(n + p)$ variables in X and U and placed at g argument positions of the predicate. Such selection and placement can be represented by a formal matrix product $(X, U)H$, where H , called a *selector*, is an $(n + p) \times g$ $(0, 1)$ matrix with exactly one 1 in each column. We usually write $(X, U)H$ as XUH . For predicates in the body of the recursive rule, a *position* is *distinguished* or *nondistinguished* when it is occupied by a distinguished or nondistinguished variable. A *position for a selector* H is the position where H places a selected variable.

Example 2.1 (Continued). Consider the recursive rule for paths. The variables x_1 and x_2 are distinguished and u_1 and u_2 are nondistinguished. So $X = (x_1x_2)$ and $U = (u_1u_2)$. Let us call the selectors associated with $\text{path}(x_1u_1)$, $\text{bluearc}(u_1u_2)$, and $\text{path}(u_2x_2)$ Z^1 , W , and Z^2 , respectively. The matrices of the selectors are the following:

$$Z^1 = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}, \quad W = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad Z^2 = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 0 & 0 \\ 1 & 0 \end{bmatrix}.$$

It is easy to see that $\text{path}(XUZ^1)$ yields $\text{path}(x_1u_1)$. We say that Z^1 places x_1 to position 1 and u_1 to position 2. The distinguished position for Z^1 is position 1 and the nondistinguished position for Z^1 is position 2. The descriptions for W and Z^2 are similar.

Let s be a derived predicate of arity n . For a given EDB, a tuple $A = (a_1a_2 \cdots a_n)$ is in s with all a 's constants if there is a finite derivation tree (as defined in [37] similar to the rule/goal tree in [31]), such that its root is labeled by $s(A)$, its internal nodes are labeled by tuples in s , and all of its leaf nodes unify with tuples in EDB. The label of any internal node is obtained from some rule defining s ; labels of the child nodes of the internal node are described as follows. First, the predicates in the body of the rule are unified with the labels of the internal node's child nodes. The resulting substitution is then propagated to

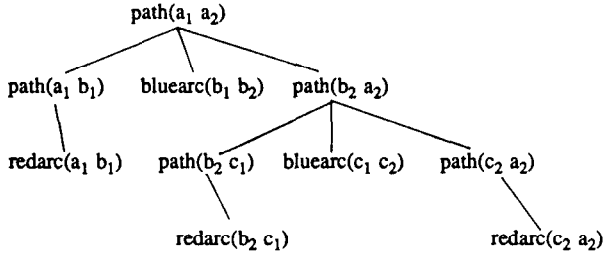


Fig. 1. Derivation tree for Example 2.1.

the head of the rule. The tuple obtained in the head is then the label of the internal node.

Example 2.1 (Continued). Assume that the EDB contains tuples $\text{redarc}(a_1 b_1)$, $\text{redarc}(b_2 c_1)$, $\text{redarc}(c_2 a_2)$, $\text{bluearc}(b_1 b_2)$, and $\text{bluearc}(c_1 c_2)$, with a 's, b 's, and c 's being nodes in the graph. Figure 1 shows a derivation tree of $\text{path}(a_1 a_2)$.

Let $s(X)$ and $t(X)$ be two predicates with the same arity, defined over the same domain. We say that $s(X)$ *implies* $t(X)$, denoted by $s(X) \Rightarrow t(X)$, if for any given EDB, every tuple in s is also in t ; and we say that $s(X)$ and $t(X)$ are *equivalent*, denoted $s(X) \Leftrightarrow t(X)$, iff $s(X) \Rightarrow t(X)$ and $t(X) \Rightarrow s(X)$.

Example 2.1 (Continued). The same type of path can also be defined by the following program.

$$\begin{aligned}
 r_e: \text{path}(x_1 x_2) &:- \text{redarc}(x_1 x_2) \\
 r_r: \text{path}(x_1 x_2) &:- \text{redarc}(x_1 u_1) \text{bluearc}(u_1 u_2) \text{path}(u_2 x_2).
 \end{aligned}$$

It is not difficult to see that the path relation here is exactly the same as that defined by the previous program. That is, they are equivalent.

It is important to notice that the first program in Example 2.1 is doubly recursive, while the second one is linear recursive. The only difference between them is that in the body of the recursive rules, the first predicate of the nonlinear rule is $\text{path}(x_1 u_1)$, while that of the linear rule is $\text{redarc}(x_1 u_1)$. This suggests a very simple and natural way to convert a simple doubly recursive program into a simple linear recursive program. That is, to unify one of the two recursive predicates in the body of the doubly recursive rule with the head of the exit rule. We are, therefore, interested in obtaining conditions under which a given simple doubly recursive program is equivalent to its converted linear one.

To study the problem, a general notion of simple doubly recursive programs and their converted linear programs is needed. For this purpose, we generalize the path programs, and consider the following general programs, which define predicates s and s_1 , respectively.

$$\begin{aligned}
 r_e: s(X) &:- f(X) \\
 r_r: s(X) &:- s(XUZ^1) r(XUW) s(XUZ^2) \quad (\text{I})
 \end{aligned}$$

$$\begin{aligned}
 r_e: s_1(X) &:- f(X) \\
 r_r: s_1(X) &:- f(XUZ^1) r(XUW) s_1(XUZ^2) \quad (\text{II})
 \end{aligned}$$

In these programs, r_e is an exit rule; r_r is a recursive rule; s and s_1 are recursive predicates; f and r are distinct base predicates; X and U are vectors of distinguished and nondistinguished variables, respectively; and Z^1 , Z^2 , and W are selectors. Notice that s and s_1 should be considered as having the same predicate name. But, for convenience, they are named differently. The recursive rule in program (II) is obtained from that in program (I) by unifying the first occurrence of the s predicate in the body with the head of the exit rule. As a result, the first occurrence of s is replaced by an f predicate. The choice of the first occurrence of s is made arbitrarily. The result of choosing the second occurrence is similar. If the variables in the base predicate f are in different order from those in s , then a new base predicate f' can be introduced such that the exit rule relates s with f' and with the variables in both predicates in the same order while another rule relates f' and f . Thus, we can place the exit rule in the form as described in (I). Notice that although the conversion described here is simple and natural, it may not be the only possible way of obtaining linear rules from nonlinear ones. In this paper, however, we consider only the simple conversion.

Consider the recursive rule in programs (I) and (II). A *variable* is **dangling** if it is nondistinguished and selected by exactly one of the selectors Z^1 , Z^2 , and W . In other words, a dangling variable is not shared by two or more predicates. A *position* is *dangling* if it is occupied by a dangling variable. A *variable* is *single (multiple) dangling* if it is dangling with exactly (more than) one occurrence(s). A *position* is *single (multiple) dangling* if it is occupied by a single (multiple) dangling variable.

Example 2.2. Consider the following rule:

$$s(x_1x_2x_3x_4) :- s(x_1u_1x_3u_2) r(u_1u_3u_3u_4) s(u_4x_2u_5x_4)$$

The variables u_2 and u_5 are single dangling and the variable u_3 is multiple dangling. The positions taken by these variables are single and multiple dangling, respectively.

In a recursive rule, let Z^j and Z^k be selectors of two predicates with the same predicate name, say s , in the body. Let the arity of s be n . Z^j is said to *dominate* Z^k if, for every position i , $1 \leq i \leq n$, the variable placed at position i by Z^k is either the same as that placed at position i by Z^j , or a dangling variable that can only take those positions taken by this variable placed at i by Z^j . Two selectors Z^j and Z^k are *equivalent* if they dominate each other.

Intuitively, if in the body of a rule there are two predicates with the same predicate name and the selector of one predicate dominates that of the other, then the former predicate will be stricter than the latter. The latter predicate is, therefore, redundant. Thus, if in the doubly recursive rule r , one of the selectors Z^1 or Z^2 dominates the other, the s predicate with the dominated Z selector can be discarded because, in any derivation tree, whenever the recursive rule is used to establish some tuple, the tuple satisfying the s node with the dominating Z always satisfies the s node with the dominated Z selector. If in rule r , Z^1 and Z^2 are equivalent, either one can be discarded, yielding an equivalent linear recursive rule. This type of doubly recursive rule is called *trivially linearizable*.

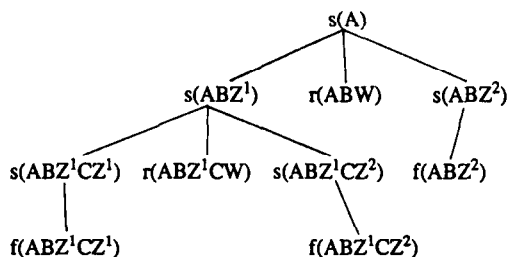


Fig. 2. Minimal derivation tree for five leaf nodes.

If in rule r , one of $s(XUZ^1)$ and $s(XUZ^2)$ is exactly the same as $s(X)$, the rule is called *endless-recursive* because, from the viewpoint of a top-down evaluation, such a rule implies an infinite recursion. The recursive rule r , in a simple recursive program is *degenerative* if it is endless-recursive or trivially linearizable.

In our previous work [30, 37], we provided necessary and sufficient conditions for a simple doubly recursive program defined by (I) to be equivalent to a simple linear recursive program defined by (II), under the following assumptions:

Assumption 1. The rules are range-restricted, i.e., in each rule every distinguished variable appears in the body. This assumption is commonly used [4] to ensure finite answers to queries defined over these rules.

Assumption 2. The recursive rule r , is of the following type. There exist two nondistinguished variables, u_i and u_j , such that u_i is shared by Z^1 and W , but not selected by Z^2 ; u_j is shared by Z^2 and W , but not selected by Z^1 . (This is a condition chaining the variables among the predicates in the body of the rule.)

Assumption 3. None of the Z selectors places any distinguished variable to two or more positions in the same predicate.

Assumption 4. The tree in Figure 2 is a minimal derivation tree (in terms of the total number of nodes) of the tuple $s(A)$ for the EDB consisting of precisely the five leaf nodes, where A , B , and C are constant vectors with distinct elements.

A tuple derived by applying the nonlinear rule once can be derived by applying the corresponding linear rule. Figure 2 gives a derivation of a tuple derived by applying the nonlinear rule twice. Assumption 4 says that there is no smaller tree to derive the tuple. In other words, the nonlinear rule is not “obviously” equivalent to a linear rule. Our intention is to identify nonlinear rules that are not obviously equivalent, but are actually equivalent to linear rules.

The previous results of [30] and [37] are summarized as follows:

PROPOSITION 2.1. *Assume that s and s_1 are defined by (I) and (II), respectively, with nondegenerative recursive rules, and that they satisfy Assumptions 1 through 4. A necessary and sufficient condition for $s \Leftrightarrow s_1$ is that in rule r , the following occur:*

- (i) each distinguished variable x_j is selected by at least one of the selectors, Z^1 or Z^2 , and that x_j is placed by the selector at position j ; and
- (ii) if x_j is selected by W , then both Z^1 and Z^2 select x_j .

In [30], we also give necessary and sufficient conditions for higher order recursive rules to be equivalent to linear recursive rules under rather restricted conditions.

3. GENERALIZED NECESSARY AND SUFFICIENT CONDITIONS

Assumptions 2 through 4 in the previous section impose some syntactic constraints on the type of doubly recursive programs to which our previous result applies. While some interesting programs do satisfy these assumptions, there are also interesting programs that do not. (For example, the well-known ancestor predicate does not satisfy Assumption 2.) In order to generalize the previous result so that it can be applied to more general types of programs, it is important to remove Assumptions 2 through 4. Thus, in this section, we only require Assumption 1 to be satisfied by programs (I) and (II), and we provide necessary and sufficient conditions for program (I) to be equivalent to linear program (II). The result is given below.

PROPOSITION 3.1. *Let s and s_1 be defined by (I) and (II), respectively, with nondegenerative recursive rules that satisfy Assumption 1. Then $s \Leftrightarrow s_1$ iff one of the following groups of conditions is satisfied by the recursive rule.*

Group 1

- (1) for every distinguished variable x_i selected by Z^1 or Z^2 , an x_i is placed at position i by the Z selector;
- (2) every x_i placed at more than one position by Z^1 or Z^2 is selected by both Z^1 and Z^2 ; and
- (3) one of the following cases holds:
 - (a) a W -selector does not exist (i.e., the rule has no r predicate); or
 - (b) W exists but selects no distinguished variable; or
 - (c) W selects some distinguished variable, shares some nondistinguished variables with Z^1 or Z^2 , and for every x_i selected by W , x_i is selected by Z^1 and Z^2 as well; or
 - (d) W selects some distinguished variable and shares no nondistinguished variable with Z^1 or Z^2 ; there is at least one distinguished variable x_j that is selected only by W and Z^2 ; and for every x_i selected by W , one of the following cases holds:
 - (i) x_i is also selected by Z^2 ;
 - (ii) each of Z^1 and Z^2 places some single dangling variable at position i ; or
 - (e) W selects some distinguished variable and shares no nondistinguished variable with Z^1 or Z^2 ; there is no distinguished variable selected by W and Z^2 only; and for every x_i selected by W , one of the following cases holds:
 - (i) both Z^1 and Z^2 place x_i at position i ;
 - (ii) both Z^1 and Z^2 place the same x_t , for some $t \neq i$, at position i ;
 - (iii) each of Z^1 and Z^2 places, at position i , some dangling variable.

In the last case, both variables must have the same number of occurrences and occupy the same set of positions; furthermore, for each such position q , x_q is selected only by W .

Group 2

- (1) Z^1 selects all distinguished variables; and
- (2) (i) for every x_i placed at some position t by Z^1 , Z^1 places x_t at position i ; or
 (ii) Z^2 selects all distinguished variables, and for every x_i placed at some position t by Z^1 , Z^2 places x_t at position i ; in either case, Z^1 places at least one x_i to a position $t \neq i$.

Group 3

- (1) for every distinguished variable x_i selected by Z^1 , an x_i is placed at position i by Z^1 ; and
- (2) one of the following cases holds:
 - (i) Z^1 does not share any nondistinguished variable with W or Z^2 ; or
 - (ii) Z^2 does not share any nondistinguished variable with W or with Z^1 , and every x_i selected by W is selected by Z^1 ; or
 - (iii) W does not share any nondistinguished variable with Z^2 or with Z^1 , and every x_i selected by Z^2 is selected by Z^1 .

Group 4

- (1) for every distinguished variable x_i selected by Z^1 and placed at some position t , there exists some position h such that Z^1 also places an x_i at h and Z^2 places an x_h at t ; and
- (2) for every nondistinguished variable u_s shared by Z^1 and W , or by Z^1 and Z^2 , Z^1 places a u_s at some position t , then there exists some position h such that Z^1 also places a u_s at h and Z^2 places an x_h at t ; and
- (3) for every dangling variable u_s placed at a set of positions t by Z^1 , either Z^2 places a variable at all positions in t , or Z^2 places at each position in t some distinguished variable x_h such that $h \in t$.

Group 5

- (1) for every distinguished variable x_i selected by Z^1 and placed at some position t , there exists a position h such that Z^1 places an x_i at h and Z^2 places an x_h at t ; and for every x_j selected by Z^2 and placed at some position p , there exists a position q such that Z^1 places an x_j at q and places an x_q at p ; and
- (2) for every dangling variable u_s placed at a set of positions t by Z^1 (respectively Z^2), Z^2 (respectively Z^1) places a variable at all positions in t ; and
- (3) for every nondistinguished variable u_s shared by Z^1 and Z^2 , let a u_s be placed at some position t_1 by Z^1 and at some position t_2 by Z^2 , then either $t_1 = t_2$, or there is another nondistinguished variable u_g such that a u_g is placed at t_2 by Z^1 and at t_1 by Z^2 ; and
- (4) W does not share any nondistinguished variable with Z^1 or with Z^2 .

Notice that the results given here are symmetric in terms of Z^1 and Z^2 if the linear recursive rule is obtained by unifying the second occurrence of s in the body of the doubly recursive rule with the head of the exit rule.

Intuitively, Group 1 requires that if a distinguished variable x appears in one of the recursive predicates in the body of the rule, it should be at its *natural* position, which is the position that it takes at the head predicate. In cases where x appears in predicate r , both Z^1 and Z^2 will make sure that the variables placed at the natural position of x are symmetric in nature. This symmetry is the key for the given program to be linearizable.

The intuitive meanings of Groups 2 through 5 can best be explained by the derivation tree in Figure 3, where only selectors are shown in the predicates.

Group 2 indicates that either $s(Z^1Z^1)$ or $s(Z^1Z^2)$ is the same as $s(X)$. Group 3 ensures that $s(Z^1Z^1)$ has the same variable pattern (in terms of which distinguished and shared nondistinguished variables are selected, where these variables are placed and how repeated variables are placed) as $s(Z^1)$. Group 4 ensures that $s(Z^1Z^2)$ has the same variable pattern as $s(Z^1)$. Group 5 implies that $s(Z^1Z^1)$ has the same variable pattern as $s(Z^1)$ and that $s(Z^1Z^2)$ has the same variable pattern as $s(Z^2)$.

Intuitively, these groups of conditions make it possible to reconstruct derivation trees in a linear fashion. This can be seen in the proof of sufficiency of the conditions.

Since the result is complicated, we demonstrate it by a number of examples.

Example 3.1. Consider the ancestor relation, defined by the following program:

$$\begin{aligned} r_e: \text{ancestor}(x_1x_2) &:- \text{parent}(x_1x_2) \\ r_r: \text{ancestor}(x_1x_2) &:- \text{ancestor}(x_1u_1) \text{ancestor}(u_1x_2). \end{aligned}$$

The converted linear program is given below:

$$\begin{aligned} r_e: \text{ancestor}(x_1x_2) &:- \text{parent}(x_1x_2) \\ r_r: \text{ancestor}(x_1x_2) &:- \text{parent}(x_1u_1) \text{ancestor}(u_1x_2). \end{aligned}$$

The recursive rules satisfy conditions 1, 2, and 3a of Group 1 in Proposition 3.1 (where condition 2 is trivially satisfied, since no x_i is placed at more than one position by Z^1 or Z^2). Thus the two programs are equivalent. This equivalence can be easily verified by the content of the programs.

Example 3.2. Consider the path programs given in Example 2.1:

$$\begin{aligned} r_e: \text{path}(x_1x_2) &:- \text{redarc}(x_1x_2) \\ r_r: \text{path}(x_1x_2) &:- \text{path}(x_1u_1) \text{bluearc}(u_1u_2) \text{path}(u_2x_2). \\ r_e: \text{path}(x_1x_2) &:- \text{redarc}(x_1x_2) \\ r_r: \text{path}(x_1x_2) &:- \text{redarc}(x_1u_1) \text{bluearc}(u_1u_2) \text{path}(u_2x_2). \end{aligned}$$

The recursive rules satisfy conditions 1, 2, and 3b of Group 1 in Proposition 3.1. Again, condition 2 is satisfied because there is no x_i taking more than one position in any predicate in the body of the recursive rules. Thus, by Proposition 3.1, the two programs are equivalent. Equivalence can also be verified by the content of the programs.

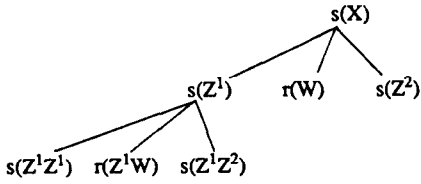


Fig. 3. Derivation tree to illustrate Groups 2-5.

Example 3.3. Let s and s_1 be defined by the following programs:

$$\begin{aligned}
 r_e: s(x_1 x_2 x_3 x_4 x_5 x_6 x_7) &:- f(x_1 x_2 x_3 x_4 x_5 x_6 x_7) \\
 r_r: s(x_1 x_2 x_3 x_4 x_5 x_6 x_7) &:- s(x_1 x_1 u_1 u_1 x_5 u_3 x_7) r(x_2 x_3 x_4 x_5) s(x_1 x_1 u_2 u_2 x_5 x_6 u_3). \\
 r_e: s_1(x_1 x_2 x_3 x_4 x_5 x_6 x_7) &:- f(x_1 x_2 x_3 x_4 x_5 x_6 x_7) \\
 r_r: s_1(x_1 x_2 x_3 x_4 x_5 x_6 x_7) &:- f(x_1 x_1 u_1 u_1 x_5 u_3 x_7) r(x_2 x_3 x_4 x_5) s_1(x_1 x_1 u_2 u_2 x_5 x_6 u_3).
 \end{aligned}$$

Rule r_r satisfies conditions 1, 2, and 3e of Group 1. Rule 3e is satisfied because W does not share any nondistinguished variable with the Z 's, and, for the distinguished variables selected by W , x_5 satisfies condition 3e(i); x_2 satisfies 3e(ii); and x_3 and x_4 satisfy 3e(iii). By Proposition 3.1, $s \Leftrightarrow s_1$.

Example 3.4. In this example we consider Ackerman's function. Given $0 \leq x \leq N_x$ and $0 \leq y \leq N_y$, Ackerman's function can be defined as follows:

$$A(x, y) = \begin{cases} 1, & \text{if } x = 0 \\ 2, & \text{if } x = 1 \text{ and } y = 0 \\ x + 2, & \text{if } x > 1 \text{ and } y = 0 \\ A(A(x - 1, y), y - 1), & \text{if } x > 0 \text{ and } y > 0 \end{cases}$$

We define two base relations f and r such that f contains only $f(0, 0, 1)$, $f(0, 1, 1), \dots, f(0, N_y, 1), f(1, 0, 2), f(2, 0, 4), \dots, f(N_x, 0, N_x + 2)$ and r contains only $r(1, 1, 0, 0), r(1, 2, 0, 1), \dots, r(N_x, N_y, N_x - 1, N_y - 1)$. Let $s(x_1 x_2 x_3)$ state that the value of Ackerman's function of inputs x_1 and x_2 is x_3 . The nonlinear program defining Ackerman's function is given below:

$$\begin{aligned}
 r_e: s(x_1 x_2 x_3) &:- f(x_1 x_2 x_3) \\
 r_r: s(x_1 x_2 x_3) &:- s(u_1 x_2 u_2) r(x_1 x_2 u_1 u_3) s(u_2 u_3 x_3).
 \end{aligned}$$

The converted linear program is

$$\begin{aligned}
 r_e: s(x_1 x_2 x_3) &:- f(x_1 x_2 x_3) \\
 r_r: s(x_1 x_2 x_3) &:- f(u_1 x_2 u_2) r(x_1 x_2 u_1 u_3) s(u_2 u_3 x_3).
 \end{aligned}$$

Notice that the recursive rules do not satisfy condition 1 of Group 1 because of the treatment of x_1 and x_2 . They do not satisfy condition 1 of Group 2 because of the placement of x_1 and x_3 . They do not satisfy condition 2 of Group 3 due to the placement of u_1 and u_3 . They do not satisfy condition 1 of Groups 4 and 5 because of the placement of x_2 and u_3 . So for every group in Proposition 3.1, some conditions are violated, and therefore the two programs are not equivalent. In fact, the tuple (2, 2, 4) can be derived for s but not for s_1 .

Example 3.5. Let s and s_1 be defined by the following programs:

$$\begin{aligned} r_e: s(x_1x_2x_3x_4) &:- f(x_1x_2x_3x_4) \\ r_r: s(x_1x_2x_3x_4) &:- s(x_3x_1x_4x_2) r(x_4x_3) s(x_2x_4x_1x_3). \\ r_e: s_1(x_1x_2x_3x_4) &:- f(x_1x_2x_3x_4) \\ r_r: s_1(x_1x_2x_3x_4) &:- f(x_3x_1x_4x_2) r(x_4x_3) s_1(x_2x_4x_1x_3). \end{aligned}$$

The recursive rule satisfies the conditions of Group 2, thus $s \Leftrightarrow s_1$ by Proposition 3.1.

Example 3.6. Let s and s_1 be defined by the following programs:

$$\begin{aligned} r_e: s(x_1x_2x_3) &:- f(x_1x_2x_3) \\ r_r: s(x_1x_2x_3) &:- s(u_1u_2x_3) r(u_3x_1) s(u_3u_4x_2). \\ r_e: s_1(x_1x_2x_3) &:- f(x_1x_2x_3) \\ r_r: s_1(x_1x_2x_3) &:- f(u_1u_2x_3) r(u_3x_1) s_1(u_3u_4x_2). \end{aligned}$$

Rule r_r satisfies the conditions of Group 3 in Proposition 3.1, so $s \Leftrightarrow s_1$.

4. SKETCH OF THE PROOF

Because the proof of Proposition 3.1 is very complicated, we will only provide a sketch. A detailed proof of sufficiency for the conditions of Group 1 is given in the Appendix, and interested readers may refer to [36] and [38] for a complete proof.

PROOF OF PROPOSITION 3.1 (Sufficiency). Intuitively, the proof consists of reconstructing derivation tree T_1 into one of the derivation trees T_2 , T_3 , or T_4 (see Figure 4). Under *Group 1* conditions, the reconstructed tree is T_4 ; under *Group 2* conditions, the reconstructed tree is T_2 ; under conditions in *Groups 3, 4, or 5*, the reconstructed tree is T_3 . Under *Group 1* conditions, the reconstructed tree, T_4 , is of the same size as the original tree, while under the conditions of the other groups, the reconstructed tree is smaller.

Condition Group 1. Assume that condition Group 1 in Proposition 3.1 is satisfied by a recursive rule. The proof consists of two steps: First, we show that for any EDB, any tuple $s(A)$, for some constant vector $A = (a_1 \dots a_n)$, derivable by the tree T_L in Figure 5a is also derivable by the tree T_R in Figure 5b. This is done by instantiating each x_i (the vector of distinguished variables $X = (x_1, \dots, x_i, \dots, x_n)$) in T_R to a_i and showing that $U = (u_1, \dots, u_p)$ and $V = (v_1, \dots, v_p)$ nondistinguished variables can be instantiated so as to create the following unifications:

$$\begin{aligned} U1: s(XUZ^1) &\text{ unifies with } s(ABZ^1CZ^1) \\ U2: s(XUZ^2VZ^1) &\text{ unifies with } s(ABZ^1CZ^2); \\ U3: s(XUZ^2VZ^2) &\text{ unifies with } s(ABZ^2). \end{aligned}$$

Depending on the conditions satisfied by the given recursive rule, these instantiations also decide for each r -node shown in T_R which of $r(ABW)$ and $r(ABZ^1CW)$ in T_L to unify with. Here, $B = (b_1 \dots b_p)$ and $C = (c_1 \dots c_p)$ are constant vectors defined by the given EDB; T_1 , T_2 , and T_3 are finite subtrees of any shape deriving $s(ABZ^1CZ^1)$, $s(ABZ^1CZ^2)$, and $s(ABZ^2)$, respectively.

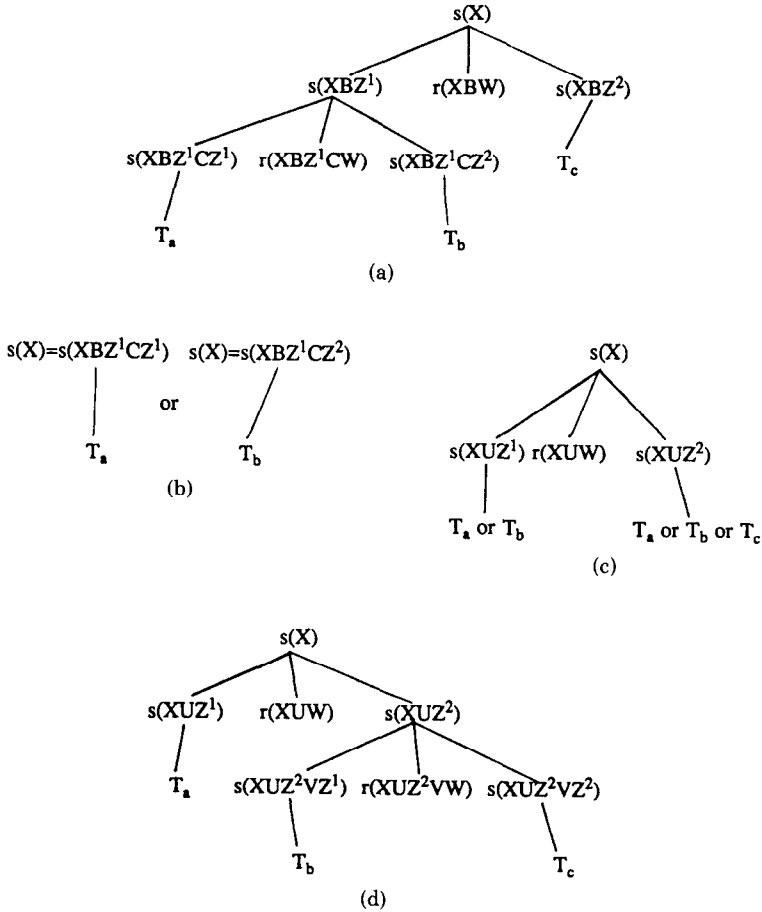
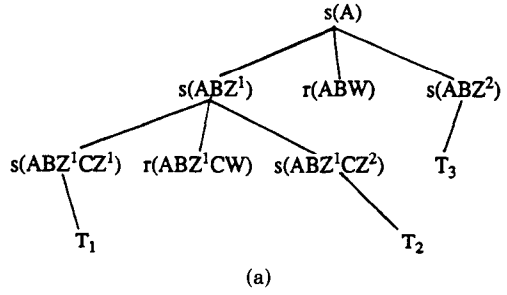
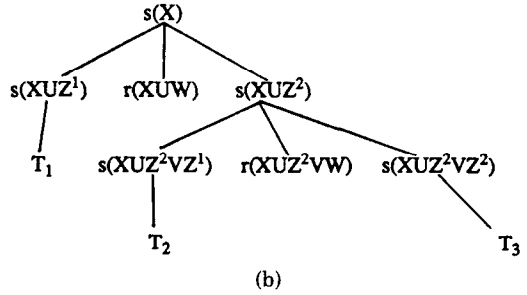


Fig. 4. Derivation trees: (a) T_1 ; (b) T_2 ; (c) T_3 ; and (d) T_4 .

Secondly, we show that for any tuple $s(A)$ derivable for a given EDB, its derivation tree (of any shape) can be converted into an equivalent one (i.e., it still derives $s(A)$) by repeatedly replacing subtrees of the shape of T_L by the corresponding T_R . The resulting tree has the following property: for every s node, either the node itself is derived by an f tuple (via the exit rule), or its left child (which is also an s node) is derived by an f tuple. It is trivial to convert such a tree into an s_1 -tree deriving $s_1(A)$.

In the first step, we show the following:

- (1) If conditions 1, 2, and 3a are satisfied, unifications U_1 through U_3 are consistent, that is, no variable is substituted by more than one constant.
- (2) If conditions 1, 2, and 3b are satisfied, then not only U_1 through U_3 but also two more unifications are consistent, U_4 : $r(XUW)$ unifies with $r(ABZ^1CW)$ and U_5 : $r(XUZ^2VW)$ unifies with $r(ABW)$.

Fig. 5. Derivation trees: (a) T_L ; (b) T_R .

- (3) If conditions 1, 2, and 3c are satisfied, unifications U_1 through U_5 are consistent. $U_4: r(XUW)$ unifies with $r(ABZ^1CW)$ and $U_5: r(XUZ^2VW)$ unifies with $r(ABW)$.
- (4) If conditions 1, 2, and 3d are satisfied, U_1 through U_5 are consistent. $U_4: r(XUW)$ unifies with $r(ABW)$ and $U_5: r(XUZ^2VW)$ unifies with $r(ABW)$.
- (5) If conditions 1, 2, and 3e are satisfied, U_1 through U_5 are consistent. $U_4: r(XUW)$ unifies with $r(ABW)$ and $U_5: r(XUZ^2VW)$ unifies with $r(ABZ^1CW)$.

Once we show the above, we can provide a procedure to transform a given derivation tree of arbitrary shape into an equivalent linear one. The details of the proof can be found in the Appendix.

Condition Groups 2 through 5. The sufficiency of the condition Groups 2 through 5 can be proved by using a similar approach. Given any EDB, the derivation tree of any tuple $s(A)$ is of the shape shown in Figure 6, where each T_i is a subtree of any shape constructed by rules in program (I). Let the root of T_i be s^i . To show that $s_1(A)$ can also be derived from the same EDB, we show that to establish $s(A)$, the root of each subtree T_i in Figure 6 can be replaced by a node s^h derivable from a tuple f^h in the EDB using the exit rule only. Notice that when an s -node is replaced by another s -node, the subtree rooted at the former s -node is also replaced by the subtree rooted at the latter s -node. It is trivial to convert the resulting tree into an s_1 -tree.

The proof consists of two steps. First, consider any subtree T_i in Figure 6 such that its root s^i has three children. We can show that if the recursive rule satisfies conditions 1 and 2(i) in Group 2, the parent of s^i can be replaced by the left child

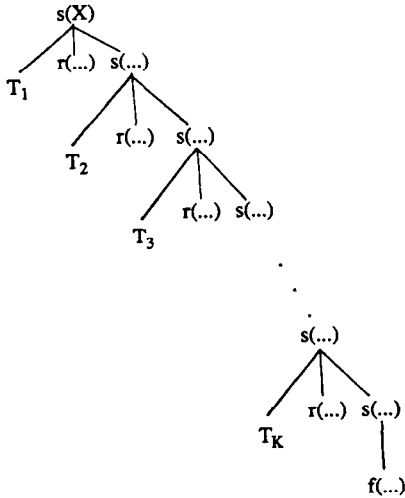


Fig. 6. General derivation tree for tuple $s(X)$.

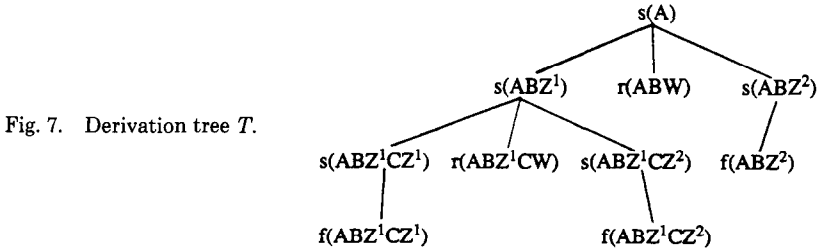
of s^i ; if the rule satisfies conditions 1 and 2(ii) in Group 2, the parent of s^i can be replaced by the right child of s^i ; if the recursive rule satisfies conditions 1 and 2(i) of the Group 3, s^i can be replaced by its left child; if the rule satisfies conditions 1 and 2(ii) in Group 3, s^i and its sibling r -node can be replaced by its left and middle children, respectively; if conditions 1 and 2(iii) of Group 3 are satisfied, s^i and its sibling s -node can be replaced by its left and right children, respectively; if condition Group 4 is satisfied, s^i can be replaced by its right child; and if condition Group 5 is satisfied, s^i and its sibling s -node can be replaced by its right and left children, respectively. In each case, the resulting tree still derives $s(A)$.

Second, we transform the tree of $s(A)$, based on the following procedure. If the root of the tree has only one child, it can be trivially converted into a tree of $s_1(A)$. Otherwise, start with the left child of the root of the tree. Let us call the node currently being considered the current node. If the current node has only one child (which must be an f -node), then repeat the procedure on the left child of the current node's sibling s -node. If the current node has three children, then make the replacement based on the description given in last paragraph (the current node corresponds to s^i). Then repeat the procedure on the current node again. The procedure repeats until no more replacement is possible.

Since in each replacement the resulting tree is smaller than the original one while still deriving the same tuple, and the initial tree is finite for any derivable tuple in s , the procedure will terminate. The resulting tree is such that the left child of any s -node is derived by a tuple in relation f . This tree can be converted into an s_1 -tree in an obvious way.

(Necessity). Despite the fact that we only consider simple programs with doubly recursive rules, the proof of the necessity of the five groups of conditions in Proposition 3.1 involves a complicated case analysis.

The main idea behind the proof is the following. By definition, if $s \Leftrightarrow s_1$, then for any EDB, a tuple derivable for s implies that the same tuple must be derivable



for s_1 . Notice that to derive a tuple for s_1 , the derivation tree must be right-skewed (or right-linear), while to derive the same tuple for s , the derivation tree may be left-skewed or in any other shape. So we can construct some special EDB such that a particular tuple can be derived for s by a tree other than a right-skewed one. By analyzing how the same tuple can be derived for s_1 in the same EDB, we can show that the recursive rule must satisfy one of the five groups of conditions.

The specific EDB, called DB1, is constructed from the derivation tree T in Figure 7. Specifically, in DB1, there exist three constant vectors: $A = (a_1 a_2 \dots a_n)$, $B = (b_1 b_2 \dots b_p)$, and $C = (c_1 c_2 \dots c_p)$ with all a 's, b 's, and c 's being distinct. All tuples in the base relations f and r in DB1 are those leaf nodes of T : $f(ABZ^1CZ^1)$, $f(ABZ^1CZ^2)$, $f(ABZ^2)$, $r(ABZ^1CW)$, and $r(ABW)$. Since any derivation tree of s is constructed by using the rules defining s , the nodes (or tuples) in both T_L and T_R can be considered as being obtained in the same systematical way. For example, in T_L , the tuple ABZ^1CZ^1 is obtained as follows. First, an n -vector ABZ^1 is obtained by applying Z^1 to select from the concatenation of the vector A of n constants for distinguished variables and the vector B of p constants for nondistinguished variables. Then the n -vector ABZ^1 is concatenated with the p -vector C , and the result is selected by Z^1 again, to produce the tuple ABZ^1CZ^1 .

It is clear that T derives the tuple $s(A)$. If $s \Leftrightarrow s_1$, one of the derivation trees shown in Figure 8 must be the minimal tree deriving $s_1(A)$ by using the tuples in DB1 only, where $X = (x_1 x_2 \dots x_n)$, $U = (u_1 u_2 \dots u_p)$, and $V = (v_1 v_2 \dots v_p)$ are vectors of variables. Notice that the derivation tree T^* in Figure 8 represents all right-linear trees that use the recursive rule at least twice.

For DB1, the establishment of a tuple in s_1 implies that there is a finite right-linear derivation tree and a set of unifications that unify each leaf node in the s_1 -tree with a tuple in the EDB such that every distinct variable is substituted by (or, equivalently, takes on) exactly one value from the constant vectors. If the tuple derived is A , then it is further required that each x_i take on the value a_i . To find such a set of unifications, we consider the leaf nodes in the s_1 -tree, one after another. For each leaf node, we find a tuple in the EDB such that to unify the leaf node with the tuple does not introduce any inconsistency. A unification introduces inconsistency if it substitutes a variable by using a value different from that used by some previous unification (of other leaf nodes) or it causes some leaf node to have no tuple to unify with.

In the proof, we need to consider every tree in Figure 8. If it is the minimal tree deriving $s_1(A)$ in DB1, then we need to find all possible sets of consistent

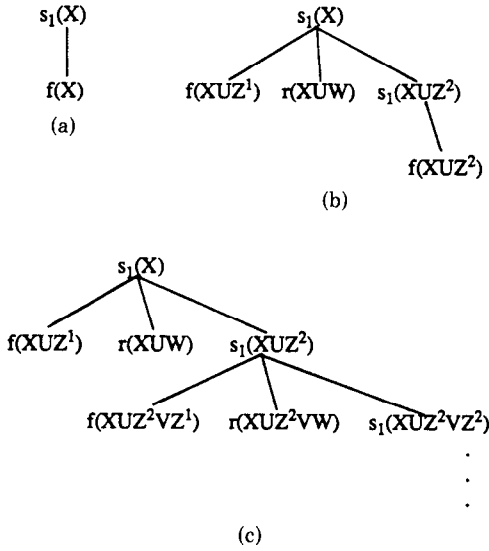


Fig. 8. Derivation tree T^* .

unifications for the leaf nodes in the tree and specify what conditions the given rule has to satisfy in order to apply each set of unifications. There are two possible situations: T^* is or is not a minimal derivation tree of $s_1(A)$.

In the first situation, T^* is the minimal derivation tree of $s_1(A)$. By showing that inconsistency will be introduced by various incorrect unifications, we eventually show that the node $f(XUZ^1)$ in T^* can only unify with the tuple $f(ABZ^1CZ^1)$, and thus are able to claim that in DB1 we do not need to consider T^* with height deeper than 2. The unification determined for $f(XUZ^1)$ indicates that many properties are satisfied by the recursive rule. These properties can be used to determine unifications of the remaining nodes in T^* . Eventually, we can determine a unification for each node in T^* and derive more properties. These properties result in the condition Group 1 of Proposition 3.1.

The analysis of the first situation is the most difficult part of the proof. We proved 10 lemmas and 19 sublemmas. For details, please refer to [36]. Lemma 1 indicates that if $s \Leftrightarrow s_1$ and T^* is the minimal tree deriving $s_1(A)$ in DB1, then the recursive rule must have certain structure. Lemma 2 through Lemma 6 show various cases where inconsistency of certain unifications can be detected. Lemmas 7 and 8 show that when a given recursive rule satisfies certain properties, s will not be equivalent to s_1 . These two lemmas depend on Sublemma 1. Based on Lemmas 1 through 8, Lemmas 9 and 10 show that $f(XUZ^1)$ does not unify with $f(ABZ^2)$ or $f(ABZ^1CZ^2)$. Sublemmas 2 through 19 show various properties of the recursive rule, assuming that T^* is the minimal tree for $s_1(A)$ in DB1 and that certain unifications have been made (among which we have $f(XUZ^1)$ unified with $f(ABZ^1CZ^1)$).

In the second situation, derivation tree T^* in Figure 8 is not the minimal tree for deriving $s_1(A)$. Thus a minimal tree of $S_1(A)$ must be one of the two trees (a) or (b) shown in Figure 8.

To find the necessary conditions, we need to make consistent unifications of the leaf nodes of tree (a) or (b) in Figure 8 with the leaf nodes of tree T . If tree

(a) is a minimal tree of $s_1(A)$, we can easily show that the Group 2 conditions must be satisfied.

If tree (b) is a minimal tree of $s_1(A)$ in Figure 8, then because in tree (b) there are two f -nodes and one r -node, while in tree T (that is in DB1) there are three tuples in relation f and two tuples in relation r , there are altogether 18 possible ways to associate leaf nodes in tree (b) with leaf nodes in tree T . Since s is not degenerative, it can be shown that 12 out of 18 cases are impossible. The six possible (and consistent) cases are the following:

- Case 1.* $f(XUZ^2)$ unifies with $f(ABZ^2)$, $r(XUW)$ unifies with $r(ABW)$, and $f(XUZ^1)$ unifies with $f(ABZ^1CZ^1)$.
- Case 2.* $f(XUZ^2)$ unifies with $f(ABZ^2)$, $r(XUW)$ unifies with $r(ABW)$, and $f(XUZ^1)$ unifies with $f(ABZ^1CZ^2)$.
- Case 3.* $f(XUZ^2)$ unifies with $f(ABZ^2)$, $r(XUW)$ unifies with $r(ABZ^1CW)$, and $f(XUZ^1)$ unifies with $f(ABZ^1CZ^1)$.
- Case 4.* $f(XUZ^2)$ unifies with $f(ABZ^2)$, $r(XUW)$ unifies with $r(ABZ^1CW)$, and $f(XUZ^1)$ unifies with $f(ABZ^1CZ^2)$.
- Case 5.* $f(XUZ^2)$ unifies with $f(ABZ^1CZ^2)$, $r(XUW)$ unifies with $r(ABW)$, and $f(XUZ^1)$ unifies with $f(ABZ^1CZ^1)$.
- Case 6.* $f(XUZ^2)$ unifies with $f(ABZ^1CZ^1)$, $r(XUW)$ unifies with $r(ABW)$, and $f(XUZ^1)$ unifies with $f(ABZ^1CZ^2)$.

For each of the six cases, a set of properties can be obtained by using techniques similar to those used in the first situation. The conditions in Group 3 are obtained by combining properties associated with Cases 1, 3, and 5; those in Group 4 are obtained from properties associated with Cases 2 and 4; and those in Group 5 are from properties associated with Case 6. The full proof is in [38]. \square

6. CONCLUDING REMARKS

In this paper we extended our previous work [30, 37] by providing a set of necessary and sufficient conditions for transforming simple doubly recursive programs into their equivalent linear programs by using a simple conversion method. The results not only provide a complete characterization for a class of linearizable doubly recursive programs, but also allow for effective verification of linearizability. We believe that the linearization of nonlinear recursive programs has both practical and theoretical importance. Further research in this area will lead to fruitful results. The syntactic characterization of nonlinear rules indicates how the placement of variables in predicates affects the linearizability of a recursive rule and, in turn, how the linearizability of a recursive rule affects the complexity of recursive query processing. Such a full understanding is a foundation for efficient query processing and optimization in deductive databases. We plan to generalize our analysis to more general recursive programs. We note that the analysis is rather complicated. The complication is due to the elimination of Assumptions 2 through 4.

Saraiya indicates that he has a generalization of our result to allow multiple distinct subgoals, though no detailed proof has been provided. We have also

obtained generalizations of our results in different directions. All these are beyond the scope of this paper.

APPENDIX

In this appendix we provide a detailed proof of sufficiency for Group 1 of Proposition 3.1. As mentioned in Section 4, the proof intuitively consists of reconstructing derivation tree T_1 into one of the derivation trees T_2 , T_3 , or T_4 (see Figure 4). Under *Group 1* conditions, the reconstructed tree is T_4 and the reconstructed tree is the same size as the original tree.

Assume that condition Group 1 in Proposition 3.1 is satisfied by a recursive rule. The proof consists of two steps. First, we show that for any EDB, any tuple $s(A)$, for some constant vector $A = (a_1 \dots a_n)$, derivable by the tree T_L in Figure 5a is also derivable by the tree T_R in Figure 5b. We do this by instantiating each x_i (the vector of distinguished variables $X = (x_1, \dots, x_i, \dots, x_n)$) in T_R to a_i and showing that $U = (u_1, \dots, u_p)$ and $V = (v_1, \dots, v_p)$ nondistinguished variables can be instantiated so as to create the following unifications:

$$\begin{aligned} U1: & s(XUZ^1) \quad \text{with } s(ABZ^1CZ^1); \\ U2: & s(XUZ^2VZ^1) \quad \text{with } s(ABZ^1CZ^2); \\ U3: & s(XUZ^2VZ^2) \quad \text{with } s(ABZ^2). \end{aligned}$$

These instantiations will also unify each of the r -nodes shown in T_R with one of the r tuples $r(ABW)$ or $r(ABZ^1CW)$ in T_L . $B = (b_1 \dots b_p)$ and $C = (c_1 \dots c_p)$ are constant vectors defined by the given EDB; T_1 , T_2 , and T_3 are finite subtrees of any shape deriving $s(ABZ^1CZ^1)$, $s(ABZ^1CZ^2)$, or $s(ABZ^2)$, respectively.

Second, we show that for any tuple $s(A)$ derivable in a given EDB, its derivation tree (of any shape) can be converted into an equivalent one (i.e., it still derives $s(A)$) by repeatedly replacing subtrees of the shape of T_L by the corresponding T_R . The resulting tree has the following property: for every s node, either the node itself is derived by an f tuple (via the exit rule) or its left child (which is also an s node) is derived by an f tuple. It is trivial to convert such a tree into an s_1 -tree deriving $s_1(A)$.

We now proceed with the first step. Consider Figure 5. As described in Section 4, the tuples are obtained in a systematic way. Thus, to show that any tuple $s(A)$ derivable by T_L is also derivable by T_R with the intended unifications, we need to show that the unifications $U1$ through $U3$ and those for the two r nodes in T_R (to be specified later) can always be made consistently for deriving $s(A)$ (i.e., the substitution resulting from these unifications will substitute every x_i by a_i and each nondistinguished variable (e.g., u_j or v_j) by exactly one value from the vectors A , B , and C). Based on condition 3, there are five cases to consider.

Case 1. Conditions 1, 2, and 3a are satisfied.

Since there is no r -node in T_R , we only need to show that $U1$, $U2$, and $U3$ are consistent.

First consider the distinguished variables in T_R . For every x_i placed at some position t by Z^1 , an x_i is at t in $s(XUZ^1)$ in T_R and, by condition 1, an a_i is at t in $s(ABZ^1CZ^1)$ in T_L . So $U1$ is consistent for distinguished variables. To see that $U2$ is consistent on distinguished variables, notice that for every x_i at some

position t in $s(XUZ^2VZ^1)$, according to the way T_R is constructed, Z^2 must select x_i . Z^2 may or may not place x_i at positions other than i . If Z^2 does not place x_i at any other position, then having x_i at position t in $s(XUZ^2VZ^1)$ means that Z^1 selects x_i and places it at position t . Notice that t may or may not be the same as i . If $t \neq i$ (therefore $x_t \neq x_i$), by Assumption 1 and conditions 1 and 2, x_i is selected by Z^2 and is placed only at position t . Therefore a_i is at position t in $s(ABZ^1CZ^2)$. The same is true if $t = i$. If Z^2 places x_i at some other position p , by Assumption 1 and condition 1, Z^1 places x_p at position p . By condition 2, Z^1 does not place x_p at any other position. In this case there are only two possible ways to get x_i at position t in $s(XUZ^2VZ^1)$: either Z^1 places x_i at t or $p = t$ and Z^1 places x_p at t . In both cases, a_i is at t in $s(ABZ^1CZ^2)$. Thus, U_2 is also consistent for distinguished variables. By a similar argument, U_3 is also consistent for distinguished variables.

Consider nondistinguished variables in T_R . For every nondistinguished position p for Z^1 (respectively, Z^2), by condition 1 and Assumption 1, x_p is selected only by Z^2 (respectively, Z^1), and by conditions 1 and 2, x_p is placed only at position p . Thus, for every nondistinguished variable u_g placed at some position t_1, \dots, t_k , for some $k \geq 1$ by Z^1 (respectively Z^2), we have u_g at t_1, \dots, t_k in $s(XUZ^1)$ (respectively, $s(XUZ^2VZ^1)$), v_g at t_1, \dots, t_k in $s(XUZ^2VZ^1)$ (respectively, $s(XUZ^2VZ^2)$) in T_R and b_g at t_1, \dots, t_k in $s(ABZ^1CZ^2)$ (respectively, $s(ABZ^2)$), c_g at positions t_1, \dots, t_k in $s(ABZ^1CZ^1)$ (respectively, $s(ABZ^1CZ^2)$). Thus U_1 through U_3 substitute u_g by c_g and v_g by b_g . So U_1 through U_3 are consistent for nondistinguished variables.

Case 2. Conditions 1, 2, and 3b are satisfied.

In this case the intended unification of r -nodes in T_R unify $r(XUW)$ with $r(ABZ^1CW)$ (denoted by U_4) and $r(XUZ^2VW)$ with $r(ABW)$ (denoted by U_5).

Consider distinguished variables in T_R . By the same argument as in Case 1, U_1 and U_3 are consistent for distinguished variables. Since W does not select distinguished variables, U_4 and U_5 are consistent for distinguished variables and, by an argument similar to that given in Case 1, U_2 is consistent for distinguished variables.

Consider nondistinguished variables in T_R . Notice that for every nondistinguished position p for Z^1 (respectively, Z^2), by Assumption 1, x_p is selected by at least one of Z^1 , Z^2 , and W . Since p is nondistinguished for Z^1 (respectively, Z^2), by condition 1, Z^1 (respectively, Z^2) does not select x_p , by condition 3b, W does not select x_p either. So x_p is selected by Z^2 (respectively, Z^1) only, and, by conditions 1 and 2, is placed at position p only. Also notice that for every nondistinguished variable u_g placed at some position t by W , u_g is at t in $r(XUW)$ and v_g is at t in $r(XUZ^2VW)$ in T_R , while b_g is at t in $r(ABW)$ and c_g is at t in $r(ABZ^1CW)$. By arguments similar to those given in Case 1, U_1 through U_5 are consistent on nondistinguished variables.

Case 3. Conditions 1, 2 and 3c are satisfied.

In this case, the intended unifications of the r nodes in T_R are to unify $r(XUW)$ with $r(ABZ^1CW)$ (denoted by U_4) and $r(XUZ^2VW)$ with $r(ABW)$ (denoted by U_5). We show that unifications U_1 through U_5 can be made consistently.

Consider distinguished variables in T_R . By an argument similar to that given in Case 1, $U1$ through $U3$ are consistent on distinguished variables. To see that $U4$ and $U5$ are also consistent on distinguished variables, notice that by conditions 1 and 3c, in T_R , if x_i is at a position t in $r(XUW)$ then it is also at i in $s(XUZ^2)$, and therefore is also at t in $r(XUZ^2VW)$. This also means that a_i is at t in both r tuples in T_L .

Now consider nondistinguished variables in T_R . Since in this case, for every nondistinguished position t for Z^2 (respectively, Z^1), by Assumption 1 and conditions 1, 2, and 3c, x_t is selected by Z^1 (respectively, Z^2) only and placed at position t only. Thus a similar argument as given in Case 2 can be used to show that $U1$ through $U5$ are consistent on nondistinguished variables, too.

Case 4. Conditions 1, 2, and 3d are satisfied.

In this case, the intended unifications for r nodes in T_R are to unify both $r(XUW)$ and $r(XUZ^2VW)$ with $r(ABW)$ (denoted by $U4$ and $U5$, respectively).

Consider distinguished variables in T_R . By using the same arguments as used in Case 1, we can show that conditions 1 and 2 assure that $U1$, $U2$, and $U3$ are consistent for distinguished variables. It is trivial to see that $U4$ is also consistent for distinguished variables. Let us consider $r(XUZ^2W)$. For every x_i placed at some position t by W , a_i is at t in $r(ABW)$ in T_L . So $U5$ implies that a_i will substitute for the variable at position i in $s(XUZ^2)$ in T_R . By condition 3d, the variable at position i in $s(XUZ^2)$ is either x_i or a single dangling variable. In either case, $U5$ is consistent for distinguished variables.

Consider nondistinguished variables in T_R . Because W does not share a nondistinguished variable with Z^1 or Z^2 , for every nondistinguished variable u_g selected by W , $U4$ and $U5$ are consistent on u_g and v_g in T_R . For every nondistinguished position p for Z^1 (respectively, Z^2), by Assumption 1, conditions 1, 2, and 3d, x_p is (1) selected by Z^2 (respectively, Z^1) only and placed at position p only; or (2) selected by W only and p is a single dangling position for both Z^1 and Z^2 . Notice that (1) and (2) cannot both be true. For those nondistinguished variables which are placed at positions where (1) is true, arguments similar to those given in Case 2 can be used to show that $U1$ through $U5$ are consistent. For any nondistinguished position p where (2) is true, let some u_g be at p in $s(XUZ^2)$ and some u_s be at p in $s(XUZ^1)$ where both u -variables are single dangling. Since x_p is selected by W only, let W place x_p at some position q . Only $r(XUZ^2VW)$ needs consideration for this type of nondistinguished variable. Then, u_g is at position q in $r(XUZ^2VW)$ while a_p is at q in $r(ABW)$ and b_s is at q in $r(ABZ^1CW)$. $U5$ assures consistency on u_g . Thus $U1$ through $U5$ are consistent.

Case 5. Conditions 1, 2, and 3e are satisfied.

The intended unifications for the r nodes in T_R are to unify $r(XUW)$ with $r(ABW)$ (denoted by $U4$) and $r(XUZ^2VW)$ with $r(ABZ^1CW)$ (denoted by $U5$).

Consider distinguished variables in T_R . By the same arguments as in Case 4, $U1$ through $U4$ are consistent for distinguished variables. Consider $r(XUZ^2VW)$ in T_R . For every x_i at some position t in $r(XUZ^2VW)$, there must be some position p such that Z^2 places x_i at p and W places x_p at t . As

implied by condition 3e, Z^1 also places x_i at position p (notice that i and p may or may not be equal). So in T_L , a_i is at p in $s(ABZ^1)$ and therefore at t in $r(ABZ^1CW)$. This implies that $U5$ is also consistent for distinguished variables.

Consider nondistinguished variables in T_R . Again, for every u_g selected by W , $U4$ and $U5$ are consistent on u_g and v_g in T_R . For every nondistinguished position t_1 for Z^1 (respectively, Z^2), by Assumption 1, conditions 1, 2 and 3e, x_{t_1} is (1) selected by Z^2 (respectively, Z^1) only and placed at position t_1 only; or (2) selected by W only and for both Z^1 and Z^2 , p is dangling; furthermore, if the dangling variable u_g is placed by Z^1 (respectively, Z^2) at some positions t_1, \dots, t_k , the other dangling variable u_s must be placed by Z^2 (respectively, Z^1) at t_1, \dots, t_k and x_{t_1}, \dots, x_{t_k} are selected by W only. Notice that (1) and (2) cannot both be true. For nondistinguished variables which are placed at positions where (1) is true, $U1$ through $U5$ can be shown to be consistent by arguments similar to those given for Case 2. For nondistinguished variables which are placed at positions where (2) is true, similar to Case 4, only $r(XUZ^2VW)$ needs to be considered, and $U5$ assures consistency. Thus, $U1$ and $U5$ are consistent.

Now let us consider the second step. For any derivation tree of any tuple $s(A)$ in a given EDB , we can transform it based on the following procedure:

Start at the root of the given derivation tree of $s(A)$. If the root has only one child, which must be an f -node, the tree can be trivially converted into a tree of $s_1(A)$. Otherwise, for every s -node, say s' , which has three children, if the left child of s' is established by an f tuple (via the exit rule), then repeat the procedure on the subtree rooted at the right child of s' . If the left child of s' also has three children, then the subtree rooted at s' is of the shape of T_L and can be replaced by the corresponding tree T_R . After replacing the subtree of s' , the procedure is repeated on the new subtree rooted at s' . The procedure repeats until no more replacements are possible.

Since each time the replacement occurs the root of the subtree (i.e., s') remains unchanged and the subtree with the shape of T_R consists of the same subtrees T_1 , T_2 , and T_3 of corresponding T_L , the leaf nodes (i.e., tuples in f and r) in the two trees are the same. Since for any derivable tuple the initial tree is finite, the above procedure always terminates. It is obvious that when the procedure terminates, the resulting tree derives $s(A)$, and is such that for every s -node in the tree, it is established for an f tuple or its left child is derived by an f tuple. \square

ACKNOWLEDGMENTS

We are grateful to Jeffrey Ullman for his comments on an earlier draft of this paper, which led to the discovery of conditions stated in Groups 2 through 5 in Proposition 3.1. We also thank the referees for their comments and suggestions. We are grateful to Won Kim for his suggestions, which led to improvements in the readability of this paper. This research was supported in part by NSF under grant IRI-8901789.

REFERENCES

1. AGRAWAL, R., AND DEVANBU, P. Moving selections into linear least fixpoint queries. In *Proceedings of the IEEE Conference Data Engineers* (Los Angeles, Calif., 1988), 452–461.

2. AGRAWAL, R., AND JAGADISH, H. V. Direct algorithms for computing the transitive closure of database relations. In *Proceedings of Thirteenth International Conference on Very Large Data Bases* (Brighton, England, 1987). 255–266.
3. BANCILHON, F. Naive evaluation of recursively defined relations. In *On Knowledge Base Management Systems—Integrating Database and AI Systems*, Brodie and Mylopoulos, Eds., Springer-Verlag, New York, 1985.
4. BANCILHON, F., AND RAMAKRISHNAN, R. An amateur's introduction to recursive query processing strategies. In *Proceedings of ACM SIGMOD* (Washington, D.C., 1986). ACM, New York, 1986, 16–52.
5. BEERI, C., AND RAMAKRISHNAN, R. On the power of Magic. In *Proceedings of ACM Symposium on Principles of Database Systems* (San Diego, Calif., 1987). ACM, New York, 1987, 269–283.
6. BANCILHON, F., MAIER, D., SAGIV, Y., AND ULLMAN, J. Magic sets and other strange ways to implement logic programs. In *ACM Symposium on Principles of Database Systems* (1986). ACM, New York, 1986.
7. BEERI, C., KANELAKIS, P., BANCILHON, F., AND RAMAKRISHNAN, R. Bounds on the propagation of selection into logic programs. In *Proceedings of Principles of Data Base Conference* (Cambridge, Mass., 1987), 214–226.
8. CERI, S., GOTTLOB, G., AND LAVAZZA, L. Translation and optimization of logic queries: The algebraic approach. In *Proceedings of Twelfth International Conference on Very Large Data Bases* (Kyoto, Aug. 1986). 395–402.
9. CHAKRAVARTHY, U., GRANT, J., AND MINKER, J. Foundations of semantic query optimization for deductive databases. In *Proceedings of The Workshop on Foundations of Deductive Databases and Logic Programming* (Washington, D.C., 1986).
10. CHANG, C. On the evaluation of queries containing derived relations in relational databases. In *Advances in Data Base Theory, Vol. 1*, H. Gallaire, J. Minker, and J. Nicholas, Eds., Plenum Press, New York, 235–260, 1979.
11. GARDARIN, G. Evaluation of database recursive logic programs as recurrent function series. In *Proceedings of ACM SIGMOD* (Washington, D.C., 1986). ACM, New York, 1986, 117–186.
12. GARDARIN, G. Magic functions: A technique to optimize extended datalog recursive programs. In *Proceedings of Thirteenth International Conference on Very Large Data Bases* (Brighton, England, 1987), 21–30.
13. HAN, J., AND HENSCHEN, L. Handling redundancy in the processing of recursive database queries. In *Proceedings of ACM SIGMOD* (1987). ACM, New York, 1987, 73–81.
14. HAN, J., AND LU, H. Some performance result on recursive query processing in relational database systems. In *Proceedings of IEEE Data Eng.* (Los Angeles, Calif., 1986), 533–539.
15. HENSCHEN, L., AND NAQVI, S. On compiling queries in recursive first-order databases. *J. ACM* 31, 1 (Jan. 1984), 47–85.
16. IOANNIDIS, Y. A time bound on the materialization of some recursively defined views. In *Proceedings of Conference on Very Large Data Bases* (Stockholm, 1985), 219–226.
17. IOANNIDIS, Y., AND WONG, E. An algebraic approach to recursive inference. In *Proceedings of Conference on Expert Database Systems* (Charleston, S.C., 1986), 209–223.
18. IOANNIDIS, Y., AND WONG, E. Transforming nonlinear recursion into linear recursion. In *Proceedings 2nd International Conference on Expert Database Systems* (Vienna, Va., 1988), 187–207.
19. JAGADISH, H. V., AGRAWAL, R., AND NESS, L. A study of transitive closure as a recursive mechanism. In *Proceedings of ACM SIGMOD* (1987). ACM, New York, 1987.
20. MAIER, D. *The Theory of Relational Databases*. Computer Science Press, Rockville, Md., 1983.
21. NAUGHTON, J. Data independent recursion in deductive databases. In *Proceedings of Conference on Principles of Database Syst.* (Cambridge, Mass., 1986), 267–279.
22. NAUGHTON, J. One-sided recursion. In *Proceedings of Conference on Principle of Database Syst.* (San Diego, Calif., 1987), 340–346.
23. NEJDL, W. Recursive strategies for answering recursive queries—The RQA/FQI strategy. In *Proceedings of Thirteenth International Conference on Very Large Data Bases* (Brighton, England, 1987), 43–50.
24. RASCHID, L., AND SU, S. Y. W. A parallel processing strategy for evaluating recursive queries. In *Proceedings of Conference on Very Large Data Bases* (Kyoto, 1986), 412–419.

25. SAGIV, Y. Optimizing datalog programs. Extended abstract. In *Proceedings 6th ACM Symposium on Principles of Database Systems* (1987). ACM, New York, 1987, 349–362.
26. SACCA, D., AND ZANIOLO, C. On the implementation of a simple class of logic queries for databases. In *ACM Symposium on Principles of Database Systems* (Cambridge, Mass., 1986). ACM, New York, 1986, 16–23.
27. SACCA, D., AND ZANIOLO, C. Magic counting methods. In *Proceedings of ACM SIGMOD* (San Francisco, 1987). ACM, New York, 1987, 49–59.
28. SHMUELI, O. Decidability and expressiveness aspects of logic queries. In *ACM Symposium on Principles of Database Systems* (1987). ACM, New York, 1987, 237–249.
29. SIPPY, S., AND SOISALON-SOININEN, E. An optimization strategy for recursive queries in logic databases. Extended abstract. *IEEE Data Eng.* (1988), 470–477.
30. TROY, D., YU, C., AND ZHANG, W. Linearization of nonlinear recursive rules. *IEEE Trans. Softw. Eng.* 15 (Sept. 1989), 1109–1119.
31. ULLMAN, J. Implementation of logical query languages for databases. *ACM Trans. Database Syst.* 10, 3 (Sept. 1985), 289–321.
32. ULLMAN, J. *Principles of Database and Knowledge-Base Systems, Vol. 2*, Computer Science Press, Rockville, Md., 1989.
33. VIELLE, L. Recursive axioms in deductive databases: The query/subquery approach. *Expert Database Syst.* (1986), 179–193.
34. YU, C., AND OZSOYOGU, M. An algorithm for tree-query membership of a distributed query. *IEEE COMPSAC* (Chicago, Ill., 1979). IEEE, New York, 1979.
35. YU, C., AND ZHANG, W. Efficient recursive query processing using wavefront methods. *IEEE Data Eng.* (Los Angeles, Calif., 1987), 652–657.
36. ZHANG, W., YU, C., AND TROY, D. Necessary and sufficient conditions to linearize doubly recursive programs in logic databases. Tech. Rep., Dept. of EECS, Univ. of Illinois, Chicago, 1988.
37. ZHANG, W., AND YU, C. A necessary condition for a doubly recursive rule to be equivalent to a linear recursive rule. In *Proceedings of ACM SIGMOD* (1987). ACM, New York, 1987, 345–356.
38. ZHANG, W. A linearization of nonlinear recursive programs in logic databases. Ph.D. thesis, Dept. of EECS, Univ. of Illinois, Chicago, 1988.

Received October 1988; revised January 1989 and May 1989; accepted August 1989