Recursive Query Optimization in Deductive Databases via Proof-Tree Transformations

Karima Ashraf

A thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

April 1996

ISBN 0-612-10819-8

Abstract

Recursive Query Optimization in Deductive Databases via Proof Tree Transformations

Karima Ashraf

A central question in deductive databases is the efficient processing of recursive queries, which are harder to compute than nonrecursive queries. A powerful form of recursive query optimization is to detect whether the recursion is real, or *bounded*. A recursive query is said to be *bounded* if there is an a priori number k such that k iterations of the rule application suffice to compute the least fixpoint, regardless of the contents of the database. It is 1-bounded if the number k is 1. Detection of bounded programs results in elimination of recursion from the (parts of) given program. When a query program is known to be bounded, the sophisticated optimization theory developed for relational databases can be exploited for processing the query efficiently.

Reduction of "arity" of the recursion in a given program is also a useful form of query optimization. Indeed, linear recursion is much easier and faster to evaluate than nonlinear recursion. Linearization is an optimization technique which attempts to transform nonlinear recursive programs into equivalent linear programs. In general, linearization techniques do not preserve the stages of derived tuples with respect to a bottom up fixpoint evaluation. *Stage Preserving Linearization* is a linearization technique which results in an equivalent linear program without increasing the number of stages.

Typically, in database applications, the relations are required to satisfy integrity constraints, such as functional dependencies. Thus query optimization in the presence of integrity constraints is of considerable interest, giving rise to *Semantic Query Optimization*, an active topic of research.

In this thesis, using the well-known Proof-Tree Transformation technique as a basic tool, we have studied two different recursive query optimization problems: *1-Boundedness* and *Stage Preserving Linearizability*. The contributions of this thesis include detection of 1-boundedness for a certain class of recursive programs consisting of two occurrences of the recursive predicates in the rule bodies. Such programs are called *bilinear* sirups (for *single recursive rule programs*). In our study we also consider recursive programs whose input EDB is known to satisfy a set of functional dependencies. We develop a syntactic characterization of a substantial class of bilinear sirups that are 1-bounded, and provide a linear time algorithm for testing 1-boundedness for this class. We extend our results, algorithm, and its complexity above to the case where it is known that the input databases to the query program satisfy a given set of functional dependencies. We also show the applications of Proof Tree Transformation technique for the detection of Stage Preserving Linearization.

Detecting 1-bounded programs is of practical interest since knowing that a nonlinear program is 1-bounded can lead to a dramatic reduction in the query processing complexity. Stage Preserving Linearization results in significant improvement in efficiency of evaluation compared to other linearization techniques that do not preserve stages.

Acknowledgments

I would like to express sincere gratitude to my thesis supervisor Dr. V. S. (Laks) Lakshmanan for his dedicated supervision in the research and preparation of this thesis. It has been a pleasure working with Prof. Lakshmanan. Despite having a very busy schedule, he always had time for discussions and exchange of ideas. I thank him for sincere co-operation and collaboration.

I am grateful to my family for their unconditional support, motivation and countless sacrifices. It is my honour to dedicate this thesis to my brother Feroz Ashraf, who inspired me to join the Master's program at Concordia University and has always motivated me to do my best.

I also would like to thank my colleague Subramanian Iyer for everyday help and various discussions.

Lastly, I like to thank the Computer Science Department staff for their co-operation and help throughout my affiliation with the University.

Several research grants supported me financially during my years as a graduate student and I would like to gratefully acknowledge the following grants: Natural Sciences and Engineering Research Council of Canada (NSERC) and Fonds pour la Formation de Chercheurs et l'Aide a la Recherche (FCAR) of Quebec.

Contents

List of Figures

Chapter 1

Introduction

1.1 Deductive Databases

In recent years there has been considerable interest in the Database community to extend relational database systems and to improve the expressive power of database systems. One of the many proposals is to augment the Relational Database systems with logic-based query languages. The result is sometimes called a Deductive database, because of the ease with which implicit information can be "deduced" from the facts stored in the relations. The field of Deductive Database is concerned with developing logic query programming systems that can manipulate large quantities of data efficiently.

A Deductive Database consists of two types of relations or predicates - the *base* relations and the *derived* relations. The set of all base relations forms part of extensional database, EDB, which is equivalent to traditional relational database. The set of derived relations form the intensional database, IDB, in which, the derived relations are deduced from the base and derived relations using the sets of rules defining them.

A rule defining a derived predicate L_0 is a function free, Horn clause of the form:

L_0:- L_1, ..., L_n, where L_1, ..., L_n are either IDB or EDB predicates.

A finite set of rules forms a *Query Program*.

Recursion in Deductive Databases

Deductive Databases extend the expressive power of relational database systems by adding recursion. Indeed, conventional relational query languages like relational algebra or their commercial counterparts like SQL cannot express recursive queries. Deductive database query languages like *Datalog* allow queries to be written in logic in the form of (database) logic programs, usually expressed in a Prolog like syntax. *E.g.*, consider the following logic recursive query program from [20] consisting of the rules

$r_0 : buys(X, Y) :- likes(X, Y)$, and

$r_1 : buys(X, Y) :- trendy(X), buys(W, Y)$

These rules define the relation $buys(X, Y)$, meaning person X buys object Y, in terms of the relations $likes(X, Y)$, $trendy(X)$, and the relation $buys$ itself. The query program above is *recursive* since it defines the relation $buys$ in terms of itself and EDB predicates.

Recursive Query Optimization

The presence of recursion in the query programs is a source of increased expressive power and higher complexity of query processing. Indeed, efficient query processing and optimization is one of the most extensively researched topic in deductive databases (see Bancilhon and Ramakrishnan [3], Ceri et al. [4] for surveys).

Query Optimization can be regarded as the process of transforming a query program Q into an equivalent query program Q' which can be evaluated more efficiently. Q' should be equivalent to Q in the sense that Q and Q' have the same answer for every database instance.

Since, the source of difficulty in evaluating Datalog queries is their recursive nature, the first line of attack in trying to optimize such query is to eliminate recursion. Extensive research has shown that in some cases, the recursion employed by a query program can be redundant and it is quite possible to reduce the arity of recursivity, or even better, eliminate it. A classic example of such recursive query optimization is when, a nonlinear[1] program can be transformed into an equivalent linear recursive program or even a nonrecursive program. Optimization of the former type is called *linearization* whereas optimization of a recursive program into a nonrecursive program is referred to as *boundedness transformation*.

It is well known that recognition of boundedness and linearizability is an important optimization problem. It has been proved from various studies that evaluation of bounded

[1]A recursive program is called *nonlinear* provided there is more than one recursive predicate in the body of the recursive rule.

or even linear recursive programs is considerably more efficient than that of (nonlinear) recursive programs.

Boundedness

One form of query optimization for Datalog is the recognition of the case when an apparently recursive query is in fact equivalent to a nonrecursive query, *independent of the contents of the database*.

Recognition of bounded programs is a very powerful form of recursive query optimization in deductive database since (i) evaluation of nonrecursive queries is cheaper and (ii) it makes available the powerful optimization techniques well developed for relational databases to such queries.

Indeed, it can be shown that the apparently recursive "buys" program (rules r_0 and r_1 above) is in fact equivalent to the following non-recursive program: $\{r_0 : buys(X,Y)$:- $likes(X,Y)$, $r_1' : buys(X,Y)$:- $trendy(X), likes(W,Y)\}$. Programs in which recursion can be eliminated in this manner are said to be *bounded*.

Consider another example which defines a nonlinear recursive program P_{rec}.

Example 1.1.1

$r_0 : p(X_1, X_2, X_3)$:- $e(X_1, X_2, X_3)$.
$r_1 : p(X_1, X_2, X_3)$:- $p(V, X_2, U)$, $p(U, X_2, X_3)$, $a(X_1, X_2)$, $b(U, X_2)$. □

The above program P_{rec} in fact, is an example of a doubly recursive program as it has exactly two occurences of the recursive predicate p in the body of the rule. The subgoals e, a, b are (EDB) database predicates. On the other hand, the predicate p corresponds to an intensional database (IDB) (*derived*) predicate since it is intensionally defined in terms of other predicates.

The above program is an example of a *1-bounded* program. This means it is equivalent to the following nonrecursive program P_{nonrec} obtained by replacing both occurrences of the recursive predicate p in r_1 by the exit predicate e.

r_0: $p(X_1, X_2, X_3)$:- $e(X_1, X_2, X_3)$.
r_1': $p(X_1, X_2, X_3)$:- $e(V, X_2, U)$, $e(U, X_2, X_3)$, $a(X_1, X_2)$, $b(U, X_2$

Since optimization of nonrecursive queries is a classical problem for which powerful techniques have been developed in relational database theory, detection of (1-)bounded query programs opens up the possibility of using such techniques for any query against such programs.

1.2 Previous Work

Substantial amount of work has been done in recursive query optimization. In the sequel, we will refer only to selected works which are relevant to our study of boundedness and linearizability in recursive programs, which are two major approaches to query independent program optimization. Ceri et al. [5] surveys numerous strategies for efficient processing of recursive queries. This also was the topic of a recent PhD thesis in the department [23]. A common theme of all these strategies is that they are query dependent, and attempt to push selection in the query inside the recursion using various approaches. By contrast, the kind of query program optimization studied in this thesis is query independent program transformation, which is inherently more challenging. As can be seen below, many problems in this regard either have a high complexity or are undecidable.

Firstly, it has been proved by Gaifman et al. [7] that the general problem of detecting boundedness and linearization is undecidable. Ioannidis [10], and Naughton [20] are some early works on boundedness reporting some positive results. Various surveys and accounts of numerous decidability and undecidability results on boundedness have been reported in [30], Kanellakis and Abiteboul [12], and Hillebrand et al. [9].

On the specific problem of 1-boundedness, Kanellakis [11] has shown that detecting 1-boundedness is NP-hard even for linear sirups defining a predicate of arity four. Saraiya [27] has tightened this result to more restricted classes of linear sirups. He also gives a polynomial time algorithm for testing 1-boundedness of linear sirups with no repeating subgoals. The proposed algorithm is based on the idea that testing 1-boundedness reduces to two 2-containment tests. Wood [31] has recently given a syntactic characterization and a linear time algorithm for detecting 1-boundedness for a subclass of linear sirups with some restrictions.

For works dealing with more general problems of optimization (*e.g.*, *recursive redundancy* and proof-tree removability) see Naughton [21] and Lakshmanan and Hernandez [14]. The latter also considers the effect of functional dependencies on proof-tree removability of subgoals (and hence boundedness). Numerous studies have also been done to take into account the effect of semantic knowledge in the form of integrity constraints for some of the above optimization problems (*e.g* Sagiv [25], Lakshmanan and Hernandez [14] etc). It has been shown that the presence of different types of data dependencies can be used for more efficient query processing. Sagiv [25] has shown that notion of uniform equivalence of datalog programs can be applied to minimize datalog programs whose input databases satisfy a set of tuple generating dependencies.

Extensive work has been done on the problem of *linearization*. For certain restricted class of bilinear recursive programs, Zhang et al. [32] and subsequently Saraiya [26] have studied the problem of detecting *ZYT-linearizability*, a specific way of performing linearization, by replacing exactly one of the recursive subgoals of the sirup by the exit predicate. Ramakrishnan et al. [24] proves several proof-tree transformation theorems and shows their application to commutativity of rules and ZYT-linearizability. They show that when the EDB subgoals are allowed to repeat, ZYT linearizability is NP-hard, and may be undecidable. Saraiya [27] among other things tightens this NP-hardness result to more restricted classes of sirups. Finally, in our most recent work [17], we address the problem of detecting *stage preserving linearizability* and also show its application for the detection of *1-boundedness* for the class of bilinear sirups with distinct EDBs. The detection technique above uses a uniform approach based on the notion of homomorphic tree embeddings [17]. We also show that stage preserving linear programs obtained using our technique can be evaluated much more efficiently as compared to other linearization techniques which do not preserve stages.

1.3 Scope of this thesis

The goal of this thesis is to develop an efficient query optimization algorithm to detect redundant recursion in bilinear recursive rules. Specifically, we focused our study on the development of two powerful optimization technique : *1-Boundedness* and *Stage Preserving linearizability*.

We have considered the problem of detecting 1-boundedness for a certain class of recursive programs consisting of one recursive rule and a nonrecursive rule. Such programs are termed as *sirups* (for *si*ngle recursive *ru*le *p*rograms). Specifically, we consider sirups when the recursive rule contains, two occurences of the recursive predicates in the rule body. Such sirups are called *bilinear* sirups. In our study we also consider recursive programs whose input EDB is known to satisfy a set of functional dependencies.

Our results are based on Proof Tree Equivalences and also on identifying Syntactic Properties of such recursive programs. We develop a syntactic characterization of a substantial class of bilinear sirups that are 1-bounded, and provide a linear time algorithm for testing 1-boundedness for this class. When the input EDB to a datalog program is constrained by integrity constraints, the property of boundedness is affected, a program which in general is not bounded, may become bounded under given integrity constraints. We extend our results, algorithm, and its complexity above to the case where it is known that the input databases to the query program satisfy a given set of functional dependencies.

In addition, a *new optimization technique* called *Stage Preserving Linearizability* is also presented in this thesis. We discuss how linearization can help in query optimization. The exact contributions are enumerated in Section 3.2. In the rest of this section, we provide an informal exposition of the contributions made by this thesis.

Let P be a datalog program and D an input database (EDB) to P. Then the output of P on D is the least fixpoint of P and D, obtained by iterative applications of the rules of P on the facts in D and those generated in previous iterations, until the process saturates. When the output can be obtained in a fixed number of iterations k, *independent* of the input database $\mathcal{D}$, the program is said to be *k-bounded*. *1-boundedness* corresponds to the special case when $k = 1$. This thesis establishes a syntactic characterization of a certain class of 1-bounded datalog programs and develops a linear time algorithm for detecting them. The thesis also extends the characterization and the algorithm to the case when the input to the datalog program is constrained by a set of functional dependencies.

Previous techniques for linearizing nonlinear recursive programs have ignored the "rate" at which the output is computed by the original program and its equivalent linearized program. In this thesis, we propose a linearization technique which ensures that for any input database $\mathcal{D}$, any fact derived by the equivalent linear program in no more iterations than by the original program.

1.4 Organization of the thesis

The necessary background for the development of the thesis is presented in Chapter 2. It includes the fundamentals of deductive databases. Chapter 3 introduces the topic of the thesis and also lists the research contributions made by this thesis. Proof-tree transformations for the detection of 1-boundedness and sp-linearizability is briefly introduced. Chapter 4 is dedicated to the applications of proof-tree transformations to recognizing 1-boundedness. The main result regarding the detection of 1-boundedness as well as the syntactic characterization of 1 boundedness is also presented in this chapter. In Chapter 5 we study 1 boundedness in the presence of functional dependencies. The complete syntactic characterization of 1 boundedness in presence of FDs is also included in this chapter. In Chapter 5 we also present an algorithm based on the complete characterization of 1-boundedness. Chapter 6 deals with the applications of proof-tree transformations to sp-linearizability. Conclusions are given in Chapter 7 with suggestions for some possible future research directions.

Chapter 2

Preliminaries

This chapter contains a review of the basic concepts needed for the development of the thesis. Notions from deductive databases are described first. This includes syntax and semantics of the Datalog programs. Different types of the proof trees are also presented in this chapter. We then introduce some of the fundamental notions, conventions, and notations that we shall employ in this thesis.

2.1 Syntactic Structure of Datalog Programs

Datalog is a database query language based on the rule-based logic programming paradigm. It has been developed to describe deductive databases. Discussions of the basics of deductive databases can be found in the text-books by Ullman [28, 29]. A complete survey on Datalog appears in Ceri et al. [5] as well.

A typical Datalog program consists of Horn-clause rules. A *rule* has the form L_0, L_1, ..., L_n, where each L_i is a literal of the form $s_i(t_1, \ldots, t_k)$ such that s_i is a predicate symbol and t_j are terms. A *term* is either a constant or a variable. L_0 is called the *head* of the rule and $L_1, \ldots, L_n$ represents the *body* of the rule. Clauses with empty body are called *facts*. Intuitively, a rule may be thought of as a tool for deducing new facts.

Example 2.1.1 *The rule in datalog to represent the "grand-parent" relation can be written as* $grandpar(X, Z)$:- $parent(X, Y)$, $parent(Y, Z)$.

Here the subgoals grandpar, parent *are predicate symbols. grandpar(X, Z) is the head, whereas parent(X, Y), parent(Y, Z) represents the body of the rule. Each parent predicate represents a database relation storing the child-parent facts. An example of a fact in the parent relation could be parent(bob, john). A tuple grandpar(X, Z) would be true iff the conjunction parent(X, Y) and parent(Y, Z) are facts in the parent relation.* □

The logic database consists of two components: the *extensional database* (EDB), containing a set of ground facts (stored as relations) and the *intensional database* (IDB), containing a set of deductive rules. In the example above *parent* is the EDB relation whereas *grandpar* is an IDB predicate.

A rule is (directly) *recursive* if the head predicate appears in the body as well. The *arity* of recursivity is the number of times the head predicate appears in the body of the rule. A rule is *linear* if the arity of the recursivity in the rule is 1. *Nonlinear* rules have more than one occurence of the recursive predicate in the rule body. The rule defining the relation "grandparent" is nonrecursive. An example of a query program which defines a recursive relation is as follows:

Example 2.1.2 *The rule in datalog to represent the "ancestor" relation can be written as ancestor(X, Z):- ancestor(X, Y), parent(Y, Z).*

Here the subgoal ancestor is recursive, since in the body of the rule the predicate ancestor appears as well. In the above case, the recursive rule is linear since, the predicate ancestor appears only once in the body of the recursive rule. The predicate parent is an EDB *predicate.*

□

2.2 Class of Datalog Programs studied

We shall consider logic query programs Π consisting of one recursive rule and a nonrecursive (also called exit) rule. Such programs are termed as *sirups* (for *si*ngle *r*ecursive *ru*le *p*rograms). We shall in particular, be interested in those sirups where the arity of recursivity of the recursive rule is 2. Such sirups are called *bilinear* sirups. A general representation of such programs is as follows.

r_0: $p(X_1, \ldots, X_n)$:- $e(X_1, \ldots, X_n)$.
r_1: $p(X_1, \ldots, X_n)$:- $p(Y_1, \ldots, Y_n)$, $p(W_1, \ldots, W_n)$, $a_1(U_1, \ldots, U_s)$, ..., $a_k(V_1, \ldots, V_t)$.

The subgoals $e, a_1, \ldots, a_n$ correspond to EDB predicates, while p is an IDB predicate. The subgoal e appearing in the nonrecursive rule is known as the *exit* predicate. In the recursive rule r_1, there are two occurrences of the recursive predicate p, and such a rule is called a *bilinear* recursive rule. For convenience, we use the symbols p_l and p_r (for "left" and "right") to denote these two occurrences. The class of programs we consider consists of bilinear sirups with multiple but distinct EDB predicates in the rule bodies. It should be noted that repeating EDB subgoals are a source of high complexity for many optimization problems including (1-)boundedness, conjunctive query containment, and ZYT linearizability (see Ramakrishnan et al. [21]). Therefore the class of bilinear sirups we consider is in some sense maximal. Unless otherwise specified, our subsequent reference to *programs* in this thesis refers to programs in this class.

2.3 Basic Notions and Conventions Used

We shall now introduce some fundamental terminologies and conventions of logic programs which will be used extensively in this thesis. The variables $X_1, \ldots, X_n$ which appear in the rule heads are called *distinguished variables* (dv's); those that appear only in rule bodies are called *non-distinguished variables* (ndv's); an ndv that appears in more than one subgoal is called a *shared ndv* (sndv); finally, an ndv which occurs in only one subgoal is a *dangling variable* (dndv). Non-distinguished variables are existentially quantified over the rule body in which they appear, whereas distinguished variables are universally quantified. Two subgoals s and t are *related* provided they share an sndv. Given a subgoal s, $s : k$ will denote its kth argument, and $var(s : i)$ represents the variable appearing at position i in the predicate s. *E.g.*, $var(a_1 : 1) = U_1$ in the sirup H above. Finally, we say that a dv X_i *pivots* in p_l (p_r) if this variable occurs in the argument $p_l : i$ ($p_r : i$).

An example is provided to demonstrate the above notions of logic programs.

Example 2.3.1 *[28] The program given below defines paths consisting of alternating red and blue arcs, beginning with a red arc.*

This specific example has been chosen to illustrate a representative instance of the class of programs we have studied. The rule r_1 is the exit rule, r_2 represents the recursive rule, redarc and bluearc are the EDB *relations, redarc(X,Y) (bluearc(X,Y)) means that the edge from the node X to Y is red (blue), path is the* IDB *(recursive) predicate. In the* *rule r_2 there are two occurrences of the recursive predicate path. As described above such recursive rules are bilinear.*

r_1 : *path*(X_1, X_2):- *redarc*(X_1, X_2).
r_2 : *path*(X_1, X_2):- *path*(X_1, U), *bluearc*(U, V), *path*(V, X_2).

The variables X_1, X_2 are the dv's, all the other variables are sndv's. The predicate bluearc is related to both the occurrences of recursive predicate path. bluearc : 1 = U. □

2.4 Least Fixpoint of Datalog Programs

It is well known that the semantics of query programs in datalog follows the classical *least fixpoint semantics* employed for Horn clause logic programs (see Ullman [28], Lloyd [18]). The least fixpoint may be computed by iteratively applying the following operation until no new facts can be derived for any predicate: In each iteration, apply the rules in the program to generate (new) facts for the IDB predicates defined by the program; in applying the rules, use the relations in the database for the EDB subgoals and the relations derived so far for the IDB subgoals occurring in rule bodies.

To define the above notions more formally, consider a datalog program P and a database (EDB) D. Let $P(D)$ denote the set of facts obtained by one application of the rules in P to the facts in D. Then the fixpoint computation of P w.r.t. the database D can be expressed as follows. Let $P^0(D) = D$, $P^{m+1}(D) = P^m(D) \cup P(P^m(D))$. Finally, the least fixpoint is given by $P^*(D) = \bigcup_{i<\infty} P^i(D)$. This evaluation is called *bottom-up* because it starts with the facts in the database and uses the deductive rules to infer new tuples. The two well known bottom-up evaluation techniques are: 1) Naive evaluation and 2) Semi-naive evaluation. In the Naive evaluation technique, the tuples derived at each iteration is applied repeatedly to the deductive rules to infer the entire set of derived relation(s) again. The iterative process terminates when the two consecutive derived relations obtained in two consecutive iterations are the same. Semi-naive evaluation is also a bottom-up technique, although it is designed to eliminate redundancy in the evaluation of tuples at different iterations. This method uses a smarter iterative approach than the naive evaulation. It tries to cut down on the number of duplications of the derived tuples as follows: At each iteration, instead of computing entire relations, it only evaluates the differential of the head predicate. It is well known that both the approaches evaluate the same least fixpoint, upon termination.

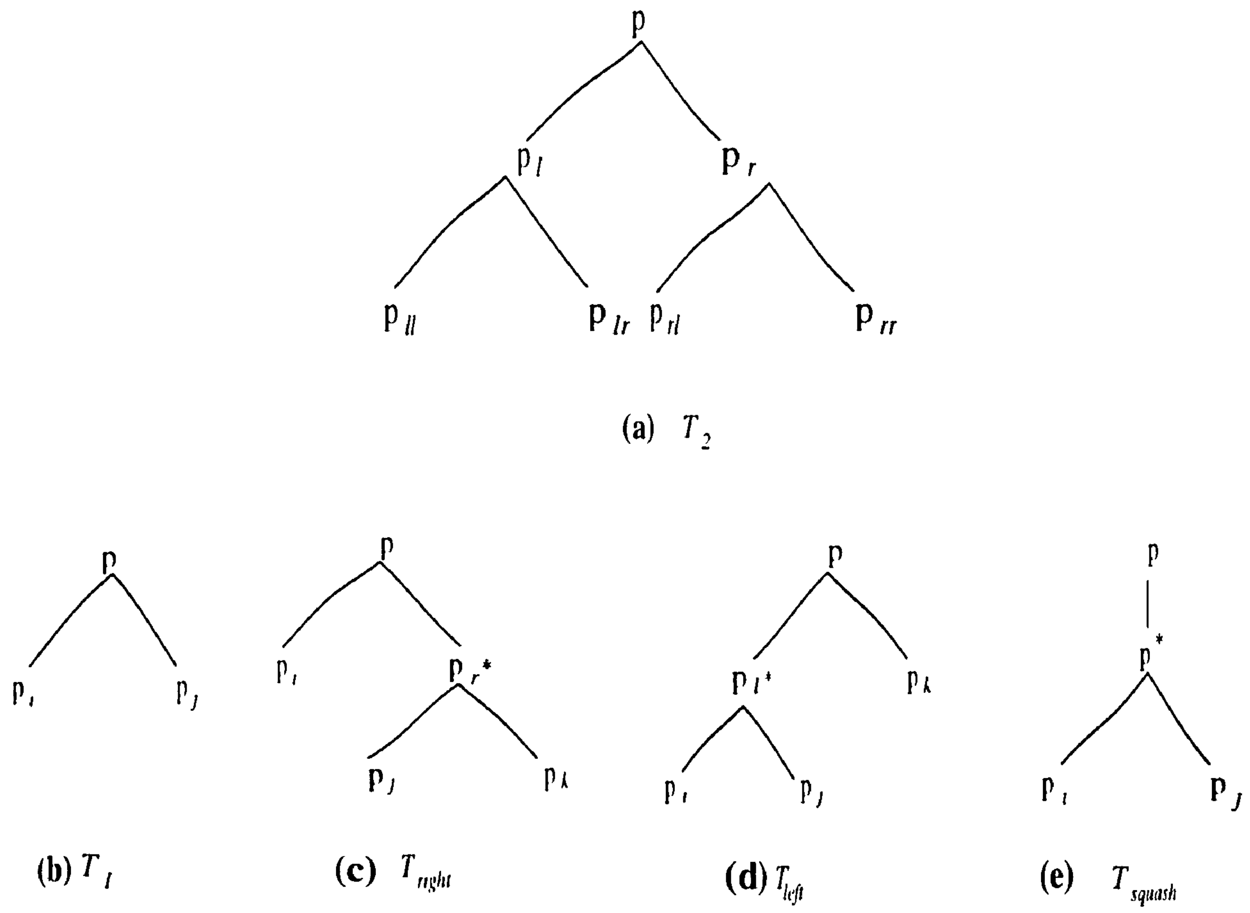
For all references purposes in this thesis, we use the semi naive evaluation technique as our standard.

2.5 Proof-Trees and containment

We define a *proof-tree* generated by a program Π to be any tree obtained by top down expansions of the goal $p(X_1, \ldots, X_n)$ using zero or more applications of the recursive rule r_1. The level in a proof-tree is counted from top down, with the root being level 0. The five fundamental types of proof-trees used extensively in our study are as follows:

- the proof-tree T_1 obtained by expanding $p(X_1, \ldots, X_n)$ using the recursive rule r_1 (See Section 2.1). In T_1, the left occurrence of the subgoal p is denoted p_ℓ while the right occurrence of p is denoted p_r.
- the tree T_{left} corresponding to an expansion of $p(X_1, \ldots, X_n)$ using r_1 and an expansion of p_ℓ using r_1. For ease of reference, each of the subgoals of T_{left} at level 2 will be identified with the subscript ℓ. The two occurrences of p at level 2 of T_{left} will be respectively referred to as $p_{\ell\ell}$ and $p_{\ell r}$.
- the proof-tree T_{right} which is symmetric to T_{left}.
- the proof-tree T_2 obtained from T_1 by expanding both p_ℓ and p_r using the recursive rule r_1. For ease of reference, each subgoal of the left (right) subtree of T_2 at level 2 will be identified with a subscript ℓ (r). The 4 occurrences of p at level 2 of T_2 will be respectively referred to as $p_{\ell\ell}, p_{\ell r}, p_{r\ell}$, and p_{rr}. These are the only leaves of T_2 involving the predicate p.
- the tree T_{squash} obtained from T_2 by (a) merging the nodes corresponding to p_ℓ and p_r into one, and (b) identifying the nodes $p_{\ell\ell}$ and $p_{r\ell}$ as well as the nodes $p_{\ell r}$ and p_{rr}, and (c) identifying the node a_ℓ and a_r, for each edb subgoal a.

Fig. 1 shows proof-trees of the form $T_1, T_2, T_{left}, T_{right}$, and T_{squash} in schematic form. In the proof-trees, we suppress the EDB subgoals, for convenience.



where $p_i, p_j, p_k \in \{ p_{ll}, p_{lr}, p_{rr}, p_{rl} \}$

Figure 1: The five fundamental proof-trees.

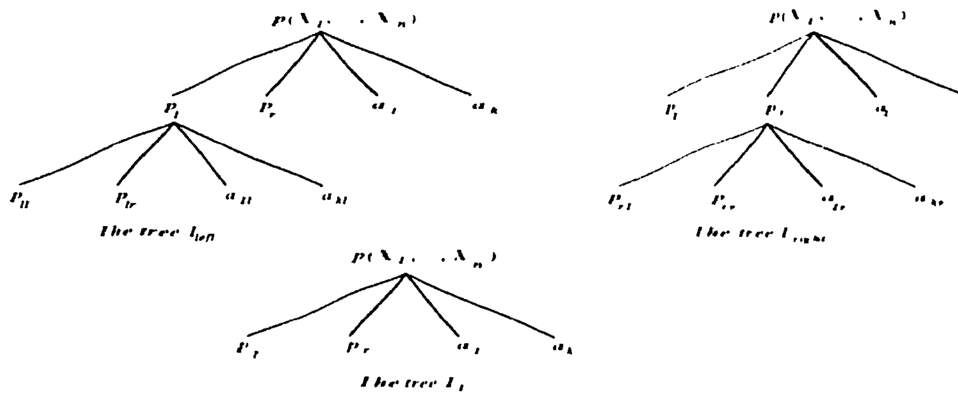


Figure 2: Proof-Trees with EDB subgoals

Conventions relevant to Proof-trees: As mentioned in the above definition we subscript the various occurences of subgoals in level 2 of T_{left} (T_{right}) with l and r respectively. Given a subgoal s, the label of s in the tree T_{left} (T_{right}) at level 2 is s_l (s_r). Specifically, if s is the subgoal p_l, then the label of the node corresponding to p_l in the left (right) subtree is p_{ll} (p_{rl}). Notice that we choose the label of the node in the right subtree to be p_{rl} rather than p_{lr} since the node in question is obtained by expanding p_r (using r_1) and then considering the node corresponding to the instance of p_l in the expansion. The labels of the nodes occurring at level 1 in T_{left} (T_{right}) have exactly the same label as the various subgoals of the rule r_1. Instances of ndv's that appear in different expansions of proof trees are distinguished using subscripts also. More precisely, let U be any ndv that appears in the body of the recursive rule r_1. Then the instance of U that appears in level 2 of T_{left} (T_{right}) is denoted U_1 (U_2). In keeping with this, we observe that if an argument position $s : i$ carries an ndv U, then when p_l is expanded we rewrite this ndv as U_1 with the result that $s_l : i$ will carry U_1. Similarly, $s_r : i$ will carry U_2. Here, s is an arbitrary subgoal.

Fig. 2 shows the proof-trees of the form T_1, T_{left}, and T_{right}, with EDB predicates in schematic form. The following example illustrates our conventions related to proof trees.

Example 2.5.1

r_0: $p(X_1, X_2, X_3)$:- $e(X_1, X_2, X_3)$.
r_1: $p(X_1, X_2, X_3)$:- $p(V, V, U), p(X_2, V, S), a(U, V), b(X_1, X_3)$.
Fig. 3 shows the proof-tree T_2 corresponding to this program as well as our conventions for the labels of the various nodes of T_2. Here, we direct the reader's attention mainly to the notations and conventions used. □

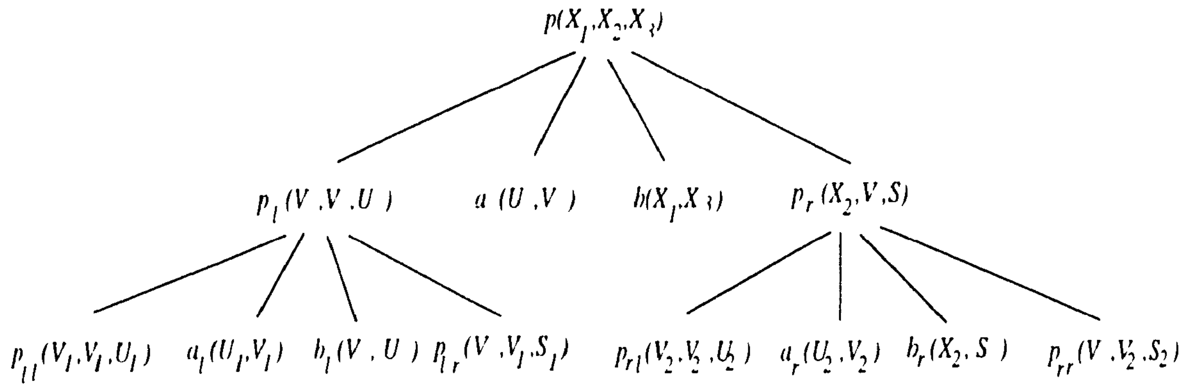


Figure 3: Proof-Tree Conventions.

Given a sirup P with the recursive rule r_1, we label the proof-tree T generated by r_1 as $r_1(T)$. For example if the proof-tree under consideration is T_{left} then we label it as $r_1(T_{left})$.

Proof-Trees as Conjunctive Queries

Intuitively, a proof-tree can be viewed as a conjunctive query as well as a scheme for computing (the set of tuples corresponding to its root). Given a proof-tree S, we can associate a *conjunctive query* with S, defined as the logical conjunction of all the subgoals occurring at the leaves of S. In the same spirit, any rule of a logic query program can also be viewed as a conjunctive query. For example, the tree T_{left} in Fig 2 can be viewed as the conjunctive query $p \leftarrow a_1 \;\&\; a_2 \;\&\; \cdots \;\&\; a_k \;\;\&\; a_{1l} \;\&\; ..\; a_{kl} \;\&\; p_{ll} \;\&\; p_{lr} \;\;\&\; p_r$. For simplification, we omit the & and understand it implicitly, and we use :- instead of $\leftarrow$ for uniformity with the prolog syntax followed for the rules.

Example 2.5.2 *For the tree T_2 given in Figure 3, the conjunctive query representation of the tree T_2 for the recursive rule is given by*

$$r_1(T_2)\text{: } p(X_1, X_2, X_3)\text{:- } p_{ll}(V_1, V_1, U_1),\ a_l(U_1, V_1),\ b_l(V, U),\ p_{lr}(V, V_1, S_1),\ a(U, V),$$
$$b(X_1, X_3),\ p_{rl}(V_2, V_2, U_2),\ a_r(U_2, V_2),\ b_r(X_2, S),\ p_{rr}(V, V_2, S_2). \quad \square$$

For a conjunctive query Q and a database D, we denote by $Q(D)$, the relation computed by Q on D. We recall the following definitions from the literature (see Ullman [29]).

Definition 2.5.1 *Let Q_1 and Q_2 be two conjunctive queries. We say that Q_1 is contained in Q_2, denoted $Q_1 \subseteq Q_2$, if for every database D consisting of relations for the predicates* *occuring in the body of* Q_1 *and* Q_2, $Q_1(D) \subseteq Q_2(D)$. Q_1 *is equivalent to* Q_2, *written* $Q_1 \equiv Q_2$, *if* $Q_1 \subseteq Q_2$ *and* $Q_2 \subseteq Q_1$.

Recall the least fixpoint semantics of datalog programs. Two datalog programs Π and Σ are *equivalent* iff for every input database D, they produce the same output, i.e. $\Pi^{\omega}(D) = \Sigma^{\omega}(D)$.

Definition 2.5.2 *[28] Consider any two conjunctive queries* Q_1 *and* Q_2*:*

Q_1*:* I :- $J_1, \ldots, J_l$.
Q_2*:* H :- $G_1, \ldots, G_k$.

We define a symbol mapping m*, from* Q_2 *to* Q_1 *as a mapping from the variables in* Q_2 *to those in* Q_1 *such that for each dv* X_i, $m(X_i) = X_i$. *A symbol mapping* m *is said to be a containment mapping (c.m.) if* $m(H) = I$, *and for* $i = 1, 2, \ldots, k$, *there is some* j *such that* $m(G_i) = J_j$.

A symbol mapping m can be extended to subgoals and hence to rules in obvious manner. We follow the notation $m : s \rightarrow t$ (or $m(s) = t$) to indicate that the c.m. m maps the subgoal s to the subgoal t. Similarly, we extend this notation to proof trees also, *e.g.* $T_1 \rightarrow T_{l,f_l}$ would mean there exists a c.m. which maps the tree T_1 to T_{l,f_l}. We recall the following classical result from [6].

Theorem 2.5.1 *For conjunctive queries* Q_1, Q_2, $Q_1 \subseteq Q_2$ *if and only if there is a containment mapping from* Q_2 *to* Q_1. □

Containment mappings play an important role in the study of many optimization problems for logic queries. A special case of containment arises when one of the subgoals in a conjunctive query is redundant. More precisely, in a query $Q = q$:- $q_1, \ldots, q_m$, a subgoal q_i is redundant provided it is equivalent to the query $Q' \equiv q$:- $q_1, \ldots, q_{i-1}, q_{i+1}, \ldots, q_m$ obtained from Q by removing the subgoal q_i. This semantic notion is captured using the notion of a subsumption mapping as follows.

Definition 2.5.3 *Let* Q *be a conjunctive query as defined above. Then a subgoal* q_i *of* Q *is* subsumed *by a subgoal* q_j *provided there is a symbol mapping (called* subsumption mapping*)* m *satisfying the following conditions:*

(i) m is an identity on all dv's as well as on all variables not occurring in q_i; (ii) there is a fixed q_j such that for each argument position $q_i : k$, $m(var(q_i : k)) = var(q_j : k)$[1].

The following result is straightforward.

Proposition 2.5.1 *Let Q be a conjunctive query as defined above. Then a subgoal q_i of Q is redundant iff there is some subgoal q_j, $j \neq i$ such that q_i is subsumed by q_j.*

Proof. Follows from the definitions of subgoal redundancy and subsumption mapping. □

We can use subsumption to eliminate some degenerate cases with respect to testing 1-boundedness, as suggested by the following fact.

Fact 2.5.1 *Suppose in a bilinear sirup one of the recursive subgoals p_l, p_r is subsumed by the other. Then the sirup is equivalent to a linear sirup.*

The significance of this fact is that 1-boundedness for linear sirups is well studied and efficiently testable characterizations already exist for them (*e.g.* see Wood [31]). Our overall objective being a linear time test for 1-boundedness for given bilinear sirups, a more important issue is whether we can test 1-boundedness of *linear* sirups (obtained as above) in *linear time*. It will be shown in Theorem 4.3.2 that our technique for testing 1-boundedness for bilinear sirups can be directly applied for linear sirups (keeping the same complexity of linear time). In view of this, it suffices to consider only bilinear sirups where neither p_r nor p_l subsumes the other, in the sequel.

A *functional dependency* (FD) over a relation r is a statement of the form $\mathbf{X} \to \mathbf{Y}$, where $\mathbf{X}$ and $\mathbf{Y}$ are set(s) of attributes. A relation r whose attributes include $\mathbf{X} \cup \mathbf{Y}$ *satisfies* the FD $\mathbf{X} \to \mathbf{Y}$ provided $\forall\, t_i, t_j \in r$, $t_i[\mathbf{X}] = t_j[\mathbf{X}] \Rightarrow t_i[\mathbf{Y}] = t_j[\mathbf{Y}]$. In this thesis, we will find it convenient to use the notation $a : \{i_1, \ldots, i_k\} \to j$ for FDs. Here, $i_1, \ldots, i_k, j$ denote the arguments of the predicate a. Given a set of FDs F, a database D *satisfies* F, provided for each FD $r : \mathbf{X} \to Y$ in F, the relation r in D satisfies $\mathbf{X} \to Y$. Here, $\mathbf{X}$ and Y denote a set of indices and an index respectively. We let $SAT(F)$ denote the set of all EDBs that satisfy F.

[1] Thus, a subsumption mapping is a special kind of containment mapping

2.6 Assumptions

Throughout this thesis, we shall consider bilinear sirups of the form given in Section 2.1. For simplicity, we will often suppress the arguments of the predicates in programs and will represent them schematically. Unless otherwise specified, we assume that the sirups we consider satisfy the following conditions.

(i) The exit predicate does not appear in the recursive rule and that the (EDB) subgoals are distinct.

(ii) The arguments occuring in the head predicate are distinct variables.

(iii) The rules are *range restricted, i.e.* all variables appearing in the head appear in the body as well.

(iv) p_l and p_r do not subsume each other.

For studying 1-boundedness, we assume the sirups satisfy the conditions above. For sp linearizability, we make the following additional assumption.

(v) None of the subgoals $p_l, p_r, p_{ll}, p_{lr}, p_{rl}, p_{rr}$ is identical to the head p.

<u>Remarks</u>: (1) Assumption (i) is a common assumption made with many optimization problems. Indeed, in many of these problems, relaxing this assumption makes the problem NP-hard. In this sense, it seems to represent the boundary between polynomial time complexity and NP-hardness (on these problems). Examples of such problems include ZYT linearizability (see Zhang et al. [32], Saraiya [26], and Ramakrishnan et al. [24]), boundedness (*e.g.* see Kanellakis and Abiteboul [12] and Hillebrand et al. [9]), and k boundedness (see Kanellakis [11], Saraiya [27]). Indeed, the classical problem of conjunctive query containment, which is polynomial when the subgoals are distinct[2], becomes NP complete when subgoals are allowed to repeat. Thus, the class of bilinear sirups we focus on is in some sense maximal. (2) Assumption (ii) is a standard one in studies of Query Optimization. (3) Assumption (iii) ensures that the program computes a finite least fixpoint when all the EDB relations are finite. (4) Assumption (iv) is made to weed out degenerate cases: indeed, in this case, the program is *sp-linearizable* and is equivalent to the linear program $\{r1' : \ p$:- $p\#, a1, \ldots, ak;\ r0 : \ p$:- $e\}$ where $p\#$ is p_l or p_r, depending on which one subsumes the other. (See Section 1.3). As discussed before, 1-boundedness of linear sirups is already well

[2] It is sufficient for the subgoals in the contained query to be distinct

studied and we will show our linear time algorithm for testing 1-boundedness developed in this thesis can be applied for linear sirups too. (5) If Assumption (v) were violated, the sirup would trivially be sp-linearizable. Notice that this type of redundancy can be checked by a (linear time) inspection of the sirup, so there is no loss of generality in making this assumption.

Chapter 3

The Problem Studied

This chapter introduces the topic of our thesis and also defines the problems we have studied. This is followed by a brief section to introduce l boundedness and proof tree transformations. In addition, a detailed list of contributions made by this thesis has also been included.

Motivation

Deductive database query languages like Datalog extend the traditional relational database language by allowing recursive queries to be expressed as well. Although the presence of recursion truly makes the language quite powerful, query processing now becomes a challenging task. And therefore, it comes as no surprise that query optimization is one of the most researched topic in deductive databases.

In recent years, a lot of interesting query optimization techniques have been proposed. It has been shown that, in many cases, it is quite possible to reduce the arity of recursivity Two powerful forms of recursive query optimization in deductive databases are tranforming a non-linear program into an equivalent linear program and transforming a recursive program into an equivalent non-recursive program. Optimization of the former type is called *linearization* whereas the latter one is referred to as *boundedness* transformation.

3.1 Research Topic

Our area of research relates to recursive query optimization. Specifically, we focused our study on the development of two powerful optimization techniques. For the class of bilinear sirups identified in Section 2.2 we have studied the problem of detecting 1-Boundedness and Stage Preserving Linearizability. As informally explained in Section 1.3, Stage Preserving linearizability is a new concept of linearization introduced by us. Compared to existing forms of linearization, it has the advantage of improved efficiency.

3.2 Contribution made by this Thesis

A brief summary of our contributions is as follows:

Research Contributions

The contributions made by this thesis is in the area of recursive query optimization techniques. Specifically, we focused our study on the development of two powerful optimization techniques: 1-boundedness and sp-linearizablity.

Our results can be listed as follows:

- Syntactic Characterization for the detection of 1-boundedness in bilinear sirups.
- A linear time algorithm for the detection of 1-boundedness.
- A preprocessing algorithm for chasing FDs.
- Characterization of 1-boundedness in presence of functional dependencies.
- An almost linear time algorithm for the detection of 1-boundedness in presence of functional dependencies.
- Development of a new Linearization technique called Stage Preserving Linearization.
- The identification of 4 unique proof-tree transformations rules with respect to sp-linearizability.

Publications Resulting From This Thesis

1) Our work on 1-boundedness appears in the paper [16].

2) Our work on Sp-linearizability was first published in the prestigious IEEE LICS Conference [17].

3) A full version of above work appears in [15].

3.3 Boundedness - An introduction

The notion of boundedness will be formally defined in this section.

Definition 3.3.1 *A program P is bounded if there exists a number k such that for all input* EDBs D, $P^{\infty}(D) = P^{k}(D)$. *We say that P is* 1-bounded *exactly when $P^{\infty}(D) = P^{1}(D)$, for* all *input* EDB D.

In terms of the bottom-up fixpoint computation techniques of query evaluation (see Ullman [29]) *boundedness* means that the query can be computed by a fixed number of iterations independent of the contents of the EDB. The special case of *1-boundedness* arises when one iteration of the fixpoint procedure is sufficient to find all answers to the given query.

Consider for instance, the following bilinear sirup P_{ex}, repeated from Example 1.1.1.

Example 3.3.1

$r_0 : p(X_1, X_2, X_3)$:- $e(X_1, X_2, X_3)$.
$r_1 : p(X_1, X_2, X_3)$:- $p(V, X_2, U)$, $p(U, X_2, X_3)$, $a(X_1, X_2)$, $b(U, X_2)$.

It can be shown using the technique developed in this thesis that the above program is 1-bounded. This means it is equivalent to the following nonrecursive program P_{nonrec} obtained by replacing both occurrences of the recursive predicate p in r_1 by the exit predicate e.

r_0: $p(X_1, X_2, X_3)$:- $e(X_1, X_2, X_3)$.
$r_{1'}$: $p(X_1, X_2, X_3)$:- $e(V, X_2, U)$, $e(U, X_2, X_3)$, $a(X_1, X_2)$, $b(U, X_2)$. $\Box$

In our study we also consider recursive programs whose input EDB is known to satisfy a set of functional dependencies. To see the effect of FDs on *1-boundedness*, indeed a program that is not 1-bounded can become 1-bounded in the presence of FDs.

Example 3.3.2

Given a program P, and the FD $F = \{a : 1 \rightarrow 2\}$.

$r_0 : p(X_1, X_2, X_3) :- e(X_1, X_2, X_3).$
$r_1 : p(X_1, X_2, X_3) :- p(X_1, W, X_3), p(X_1, X_2, W), a(X_1, W, X_2).$

It can be verified that in the absence of the given FD the program is not 1-bounded. However, we will show in a later section that the presence of additional knowledge provided by the FD F indeed makes this program 1-bounded. □

3.4 On Stage Preserving Linearizability

Since the concept of sp-linearizability requires many advanced notions and since this concept will not be used before Chapter 6, we defer a formal definition of this concept until that chapter. Here, we give an intuitive description of the underlying ideas and explain them with an example.

As remarked in Chapter 1, previously studied techniques for linearizing bilinear sirups did not take into account the "rate" at which facts in the output of a datalog program are computed. As a simple example, consider the bilinear transitive closure.

Example 3.4.1 *Consider the program P.*

$r_0 : p(X_1, X_2) :- e(X_1, X_2).$
$r_1 : p(X_1, X_2) :- p(X_1, Z), p(Z, X_2).$

It is well known that P is equivalent to the following program Q, the familiar linear definition of transitive closure.

$r_0 : p(X_1, X_2) :- e(X_1, X_2).$
$r_1' : p(X_1, X_2) :- e(X_1, Z), p(Z, X_2).$

The technique that linearizes P into a program Q is called ZYT-linearizability [32], [26]. Consider an EDB *D, consisting of a relation for e. In general, for a fact p(a, b) computed by*

Q on D in n iterations, P computes the same fact in $\lceil \log_2 n \rceil$ iterations. Thus considering bottom-up evaluations of P and Q, while less work may be done by Q per iteration (because of linearity and applicability of several efficient strategies), the total number of iterations needed in Q's evaluation is significantly more than that for P.

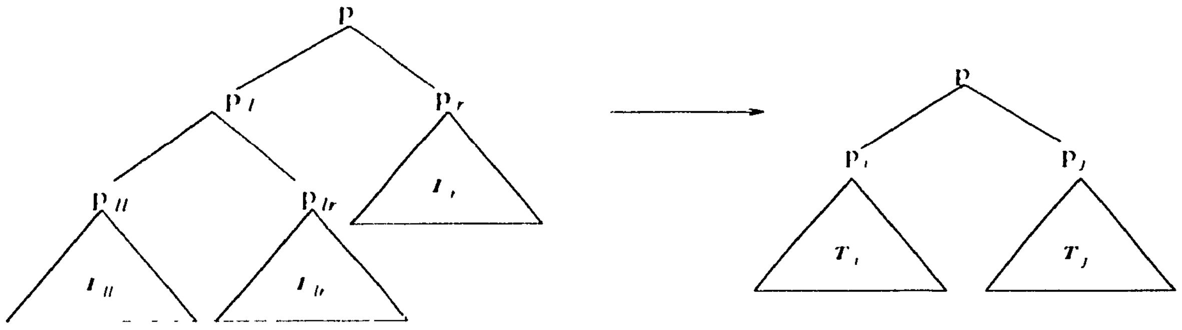
$\square$

The central idea in sp-linearizability is to linearize nonlinear recursion in a manner that does not increase the number of iterations needed to compute the output. See Example 6.1.1 and 6.1.2 to see how sp-linearization works.

3.5 Proof-Tree Transformations

This section briefly introduces the 6 different types of proof tree transformations that we have studied in detail with respect to the detection of 1-boundedness and sp linearizability. These proof-tree transformations play a central role in the study and proof of our result. The general idea of proof-tree transformations have been studied earlier by Ramakrishnan et al. [24]. We have used this knowledge as a basis in our study of 1-boundedness. Besides, in order to attack sp-linearizability, we have introduced 4 unique transformation rules.

The idea behind proof-tree transformation is the following. Suppose we want to show a program P is equivalent to another program Q. One way to show this is by showing that each of the infinitely many proof-trees generated by P (which are essentially conjunctive queries) is contained in some proof-tree generated by Q. This in turn can be done by showing that every proof-tree generated by P can be transformed into a proof tree generated by Q which contains the former. How can we have a uniform technique for transforming infinitely many proof-trees? Rather than consider each of them in turn, can we have few general pattern driven rules for transforming trees such that the undesirable trees (those generated by P) can be transformed into desirable ones (such as those generated by Q) by repeated applications of these rules? This problem is addressed by *Proof-Tree Transformation rules*. In the next two subsections, we identify the transformation rules appropriate for 1 boundedness and sp-linearizability. These will be used and explored in greater detail in subsequent chapters.



$P_i = P_{ll}$, $P_j = P_r$ OR P_{lr} **OR** $P_i = P_{lr}$, $P_j = P_r$ OR P_{ll}

Figure 4: Left Transformation Rule Corresponding to 1-boundedness.

3.5.1 Proof-tree transformations associated with 1-boundedness

There are exactly two transformation rules associated with 1-boundedness: $T_{left} \rightarrow T_1$ and $T_{right} \rightarrow T_1$. We refer to these two types of transformations as *right* and *left* transformations respectively. Fig 4 illustrates left transformation whereas the case for the right transformation is symmetrical. In addition, for each type of above transformation it can be shown that there is a c.m. $m_l : T_1 \rightarrow T_{left}$, $m_r : T_1 \rightarrow T_{right}$ respectively.

The left and right transformations play a significant role in the sufficiency proof of 1-boundedness. We can show that using these transformations, any arbitrary proof-tree can be transformed to a 1-level tree, which in fact represents a 1-bounded tree.

3.5.2 Proof-tree transformations associated with Sp-linearizability

There are four additional types of proof-tree transformations associated with sp-linearizability: $T_2 \rightarrow T_{right}$, $T_2 \rightarrow T_{left}$, $T_2 \rightarrow T_1$ and $T_2 \rightarrow T_{squash}$. Figure 5 illustrates these transformations and also indicates the mapping relationship between the (subtrees of) the original trees (and those of) the transformed trees.

In order to transform any arbitrary tree to a sp-linear tree, it is sufficient to use any one of these 4 proof-tree transformations.

The next chapter is dedicated to explain the details associated with the application of Proof-Tree transformations to 1-boundedness. Chapter 6 provides additional details for the application of Proof-Tree transformation for sp-linearization.

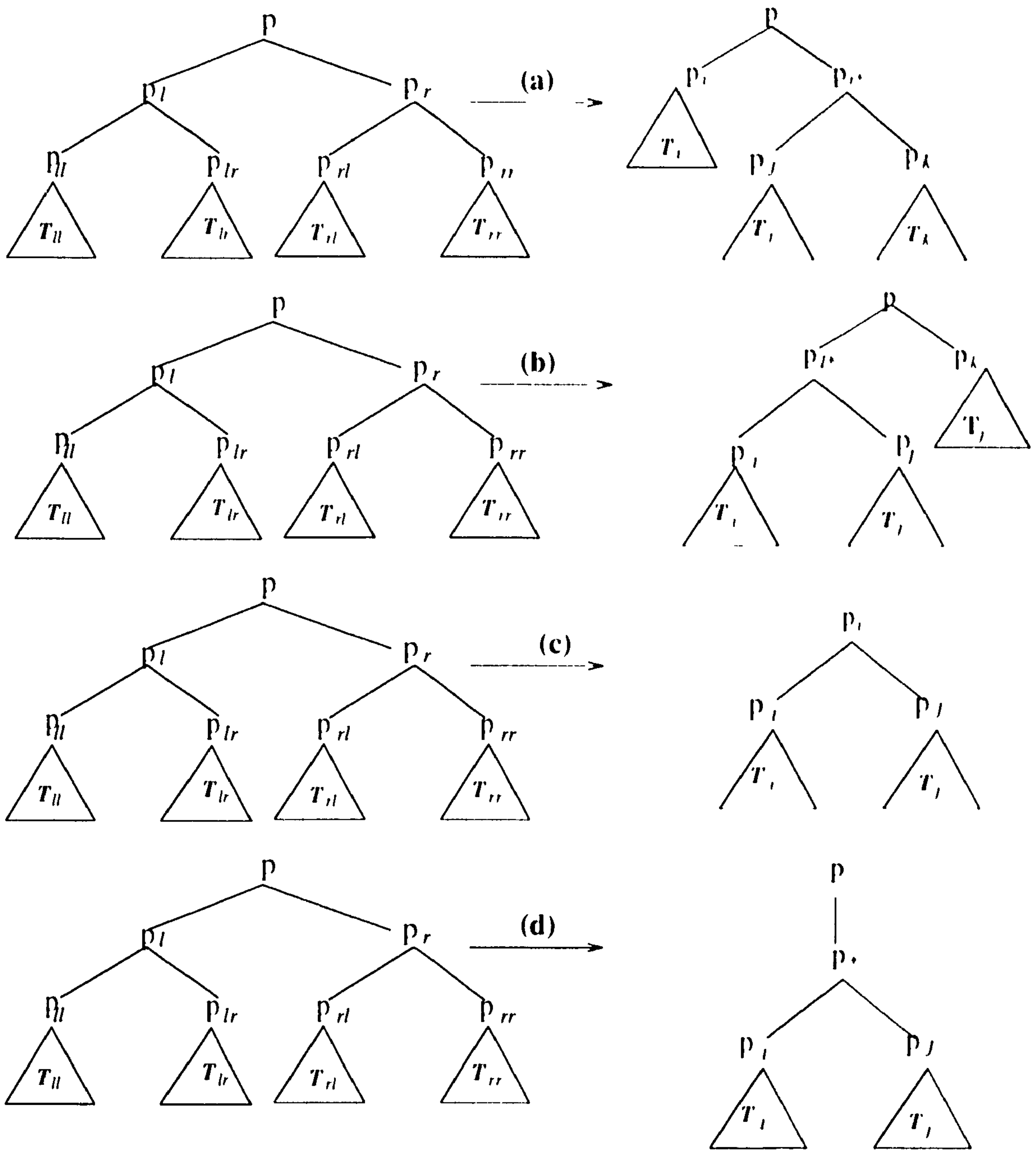


Figure 5: The Fundamental Proof-Tree Transformation Rules Sp linearizability.

Chapter 4

Applications of Proof-tree Transformations to 1-boundedness

Our goal in this chapter is to develop an efficient technique for detecting 1-boundedness of bilinear sirups. For the class of bilinear programs we develop a structural characterization of 1-boundedness. This result is based on containment mappings between proof-trees. Specifically, we show that a bilinear program is 1-bounded exactly when the proof-trees T_{left}, T_{right} corresponding to the program (see Section 2.5 for the definitions) are both contained in the proof-tree T_1 of the program.

An important issue that we came across while developing the technique based on proof-trees, was to show the equivalence between an arbitrary proof-tree and a proof-tree of height 1. To solve this problem we introduced the proof-tree transformation technique. As mentioned in the previous chapter, one of the major application of this technique is in the tranformation of an arbitrary proof-tree to 1-bounded tree. The exact details of the algorithm is present in the sufficiency proof of the main result. While the above result in itself is complete, it does not lead to an efficient algorithm. To overcome this restriction, we develop an elegant syntactic characterization of 1-boundedness for the case when there are no functional dependencies. The significance of the syntactic characterization is that it leads to a linear time algorithm. Secondly, we will show in the next chapter that the characterization can easily be adapted for the case when functional dependencies are known.

4.1 A Structural Characterization of 1-Boundedness using proof-trees

In this section we develop a characterization of 1 boundedness of bilinear sirups in terms of their proof-trees. The significance of this result is that it reduces the semantic notion of 1-boundedness to an easily testable characterization in terms of conjunctive query containment. The latter problem is completely captured using containment mappings. It will follow from the proof of the main theorem present in this section that in connection with each of the containments $T_{left} \subseteq T_1$ and $T_{right} \subseteq T_1$, only three fundamental types of containment mappings arise. We shall exploit this knowledge in the next section to develop a syntactic characterization of when these containments will hold (*without* explicitly considering any containment mappings).

Our main theorem of this section provides necessary and sufficient conditions for a bilinear sirup to be 1-bounded.

Theorem 4.1.1 *Let Π be a bilinear program as defined earlier. Then Π is 1-bounded if and only if the proof-trees T_{left} and T_{right} are both contained in T_1.*

Proof. There are two parts for the proof of the above theorem. In the necessity proof, we show that a bilinear sirup is 1-bounded provided the proof-trees T_{left} and T_{right} are both contained in T_1. For this direction of proof, we directly apply the definition of 1 boundedness to support our argument. For the proof of sufficiency we first show that we have to consider only 4 different cases of containment: Normal Case i) and ii), Partial Case, and Total Switch case. For convenience, we indicate each category of c.m. by specifying it only for p_l and p_r. We follow the notation $s \rightarrow t$ ($m(s) = t$) to indicate that the containement mapping m maps the subgoal s to the subgoal t. Similarly, we extend this notation to proof trees also. E.g. $T_1 \rightarrow T_{left}$ would mean there exists a c.m. which maps the tree T_1 to T_{left}. We then proceed by showing the applications of the proof-tree transformations to illustrate the transformation of an arbitrary proof-tree to a 1-bounded tree.

($\Rightarrow$): Suppose Π is 1-bounded. By definition, this implies that $\Pi^{\infty}(D) = \Pi^1(D)$, $\forall$ EDB D. Let $\mathcal{I}$ be any database for the predicates corresponding to the leaves of T_{left} (or T_1). Consider the EDB $\mathcal{D}$ obtained from $\mathcal{I}$ by replacing the input relation corresponding to p by the corresponding input relation for e. It is not hard to see that (i) $\Pi^1(D) = T_1(I)$ and (ii)

$T_{left}(I) \subseteq \Pi^*(D)$. By 1-boundedness, $\Pi^*(D) \subseteq \Pi^1(D)$, which implies $T_{left}(I) \subseteq T_1(\mathcal{I})$. Since I is arbitrary, we have $T_{left} \subseteq T_1$. The argument for showing $T_{right} \subseteq T_1$ is similar.

($\Leftarrow$): Let $T_{left} \subseteq T_1$. Then there must be a c.m. from T_1 to T_{left}. Indeed, we can show that there exists a c.m. $m_l : T_1 \rightarrow T_{left}$ satisfying one of the following conditions.

1. <u>Normal Case : (i)</u> $m_l(p_l) = p_{ll}$, $m_l(p_r) = p_r$

   <u>Normal Case : (ii)</u> $m_l(p_l) = p_{ll}$, $m_l(p_r) = p_{lr}$

2. <u>Partial Switch Case :</u> $m_l(p_l) = p_{lr}$, $m_l(p_r) = p_r$

3. <u>Total Switch Case :</u> $m_l(p_l) = p_{lr}$, $m_l(p_r) = p_{ll}$

We will next show that it is sufficient to consider the above 3 cases of c.m. to capture the containment $T_{left} \subseteq T_1$. For all other types of c.m. from T_1 to T_{left}, we can show formally that either i) the mapping implies a subsumption between the recursive predicates or ii) we can always construct another c.m. satisfying one of the 3 conditions above.

Consider any c.m. $m : T_1 \rightarrow T_{left}$ which does not correspond to any of the 4 types of c.m. listed above. The following cases arise with respect to patterns of possible containment mappings from T_1 to T_{left}.

<u>Case 1.</u> $p_l \rightarrow p_r, p_r \rightarrow p_x$ where $p_x \in \{p_{lr}, p_r, p_{ll}\}$.

For each c.m. m of this category, we shall show that there is a subsumption mapping m' showing that p_r subsumes p_l.

Consider the mapping m' which coincides with m on the variables in p_l and is an identity on all other variables. To show that m' is a subsumption mapping, we need to show (i) it is an identity on all dv's, and (ii) $m'(p_l) = p_r$, and $m'(s) = s$, for every subgoal other than p_l of the sirup. The only non-trivial case for (i) is an ocurrence of a dv X_i in some argument $p_l : j$. Suppose $m'(p_l : j) = p_r : j \neq X_i$. This however would imply m is not an identity on the dv X_i, a contradiction. For (ii), first observe that $m'(p_l) = p_r$ by construction. It is also easy to see that for every subgoal s unrelated to p_l, $m'(s) = s$. Consider a subgoal s related to p_l. Let $p_l : i = s : j = U$, where U is an ndv. The following cases arise.

<u>Case 1.1.</u> s is an EDB subgoal.

We show that $p_r : i = U$. Suppose to the contrary $p_r : i = Z \neq U$. We shall derive a contradiction. We first show $m(s) = s$. Suppose to the contrary $m(s) = s_t$. Then $m(s) : j = s_t : j = U_1$, whereas $m(p_r) : i = p_r : i \neq U_1$, a contradiction. Thus, $m(s) = s$. However, in this case, $m(p_r) : i = p_r : i = Z \neq U = m(s) : j = s : j$, a contradiction.

In view of the above, we conclude that whenever an ndv U occurs in $p_r : i$ and $a : j$ for some EDB subgoal a, $p_r : i = U$, which shows that for all EDB subgoals a related to p_r, $m'(a) = a$. It only remains to show that $m'(p_r) = p_r$. This is addressed by the next case.

<u>Case 1.2.</u> s is the IDB subgoal p_t.

In this case $p_t : i = p_r : j = U$. Suppose $m(p_t) = p_{tt}$. Then $m(p_t) : j = p_{tt} : j = U_1 \neq m(p_r) : i = p_r : i$, a contradiction, since m is a c.m. Suppose then $m(p_r) = p_{tt}$. Since m is a c.m., $m(p_r) : i = p_r : i = m(p_t) : j = p_{tt} : j$. Let $p_r : i = Z$. This implies $\exists k$ such that $p_r : k = Z$ and $p_t : j = X_k$. However, since $m(p_t) = p_t$, this implies $m(X_k) = m(p_t) : j = p_t : j = U$, a contradiction, since m is a c.m. Thus, we must have $m(p_r) = p_r$. This implies m is an identity on the variables in p_r. This by the construction of m' implies that $m'(p_r) = p_r$ as well. This completes the proof of Case 1.

<u>Case 2.</u> p_t—p_{tt} and p_r—p_{tt}.

Let m be the c.m. conforming to the above pattern. We shall show that there exists a c.m. m_t conforming to Normal Case (i) or (ii). Let a be any EDB subgoal related to p_t, say $p_t : i = a : j =$ an ndv U. Then since $m(p_t) = p_{tt}$, it is trivial to see that $m(a) = a_t$. Now, consider p_r. Since $m(p_r) = p_{tt}$, we have that whenever $p_r : i = X_j$, $\exists k$ such that $p_t : k = X_j$ and $p_t : i = X_k$. However, since $m(p_t) = p_{tt}$, we must have $p_{tt} : i = X_j$, which implies $p_t : k = X_k$, and hence $k = j$.

(*) That is, whenever $p_r : i = X_j$, we must have $p_t : j = X_j$.

(**) Besides, for every ndv U occurring in p_t, $m(U) = U_1$, since $m(p_t) = p_{tt}$.

Now consider the mapping m_t such that m_t coincides with m on variables occurring in p_t (thus it maps p_t to p_{tt}) and it maps a variable occurring at an argument of a subgoal s to the variable occurring at the corresponding argument of s_t, for every subgoal of r_1 that is related to p_t; m_t is an identity on all other variables. To complete the proof for this case, we need to show that m_t is a c.m. Notice that m_t is certainly of type Normal Case (ii) or (i) depending on whether p_r is related to p_t or not

First, observe that for all EDB subgoals a which are unrelated to p_l, $m_l(a) = a$, by construction. To show m_l is a c.m., We need to show (i) m_l is an identity on all dv's and (ii) whenever $s : i = t : j$, then $m_l(s) : i = m_l(t) : j$.

For (i), let X_i be a dv occurring in any subgoal argument $s : j$. If s is p_l, $m_l(p_l) : j = m(p_l) : j = p_{ll} : j = X_i$. If s is p_r, $m_l(p_r) : j = p_{rl} : j = X_i$, in view of (*). The case where s is an EDB subgoal unrelated to p_l is trivial since in this case $m_l(s) = s$. This leaves the case where s is an EDB subgoal related to p_l. In this case, (**) implies $m(s) \neq s$ and hence $m(s) = s_l$. Since m is a c.m., $m_l(s) : j = m(s) : j = s_l : j = X_i$. This completes the proof of (i).

For (ii), suppose $s : i = t : j$, where s and t are any subgoals of the sirup. By the proof of (i), the only non-trivial case is when the shared variable is an ndv. If both are EDB subgoals, either $m_l(s) = s$ and $m_l(t) = t$, or $m_l(s) = s_l$ and $m_l(t) = t_l$, depending on whether they are related to p_l or not. In either case, $m_l(s) : i = m_l(t) : j$. Suppose s is p_l. If t is an EDB subgoal, by construction, we have $m_l(p_l) = p_{ll}$ and $m_l(t) = t_l$, which will gurantee that $m_l(s) : i = m_l(t) : j$. Suppose t is p_r. Again, the construction ensures $m_l(p_l) = p_{ll}$ and $m_l(p_r) = p_{rl}$, which will preserve the euality $m_l(s) : i = m_l(t) : j$. Finally, suppose s is p_r and t is an EDB subgoal. In this case, either $m_l(p_r) = p_r$ and $m_l(t) = t$, or $m_l(p_r) = p_{rl}$ and $m_l(t) = t_l$, depending on whether s (and hence t) is related to p_l or not. Again, in either event, the equality $m_l(s) : i = m_l(t) : j$ is preserved. This was to be shown.

<u>Case 3.</u> p_l—p_{rr} and p_r—p_{rr}.

Let m be the c.m. conforming to the above pattern. First, let us establish some properties of m. (1) Since $m(p_r) = p_{rr}$, it is trivial to see that for each EDB subgoal s related to p_r, $m(s) = s_r$. (2) Let t be a subgoal related to p_l, say $p_l : i = t : j =$ an ndv U. Suppose $m(t) = t$. Since $m(p_l) = p_{rr}$, we must have $p_{rr} : i = U$. This implies U must occur in p_r, making t related to p_r. We have already shown in this case $m(t) = t_r$. (3) Since $m(p_l) = p_{rr} = m(p_r)$, we have that (a) each dv X_k occurring in p_r must occur in the position $p_l : k$ and (b) whenever a dv X_i occurs in a position $p_l : j$, $\exists k$ such that $p_r : k = X_i$, and $p_l : j = X_k$. By (a), we have that in this case $p_l : i = X_i$. Thus, whenever $p_l : j = X_i$, we must have $p_l : i = X_i$, implying $p_{ll} : j = X_i$.

Consider the mapping m_l which sends p_l to p_{ll} and all subgoals s related to p_l to s_l and all subgoals t unrelated to p_l to themselves. m_l corresponds to Normal case (ii) or (i) depending on whether p_r is related to p_l or not. We need only show that m_l is indeed a

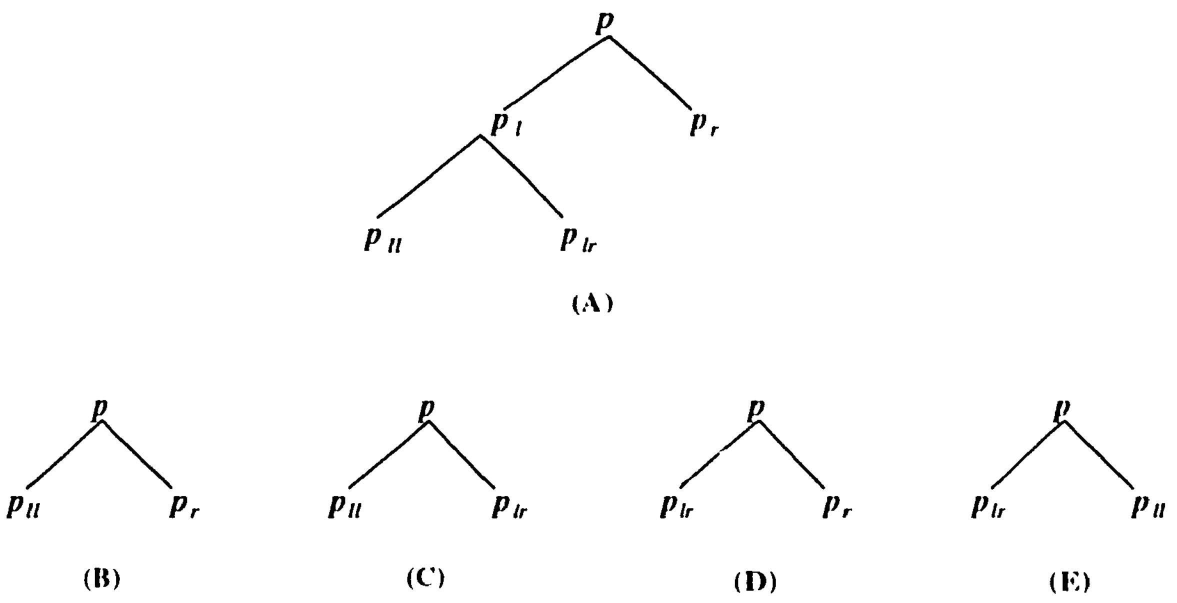


Figure 6: (A) Left linear tree (B)...(E) Four possible trees obtained by transformation from the tree of (A), w.r.t. to 1-boundedness.

c.m.

(i) Let $s : i = X_j$. If s is an EDB subgoal unrelated to p_l, $m_t(s) : i = s : i = X_j$. If s is an EDB subgoal related to p_l, then $m_t(s) : i = s_l : i = m(s) : i$. Since m is a c.m., $s_l : i = X_j$. Suppose s is p_l. This case readily follows from (3) above. The last is where s is p_r. Now, $m_t(p_r) = p_r$ or p_{lr}, depending on whether p_r is related to p_l or not. The former case is trivial, while the latter one follows from the fact that $m_t(p_r) = m(p_r) = p_{lr}$.

(ii) Let $s : i = t : j =$ an ndv U. If both are subgoals, either $m_t(s) = s$ and $m_t(t) = t$ or $m_t(s) = s_l$ and $m_t(t) = t_l$. In both cases, the equality $m_t(s) : i = m_t(t) : j$ is preserved. Suppose s is p_l. If t is an EDB subgoal, by (2), $m(t) : j = t_l : j = U_1$. Clearly, $p_{ll} : i = U_1$. This implies $m_t(s) : i = m_t(t) : j$. If t is the IDB subgoal p_r, then $m_t(p_l) : i = p_{ll} : i = U_1 = p_{lr} : j = m_t(p_r) : j$. Lastly, let s be p_r and t be an EDB subgoal. This is trivial, since $m_t(p_r) = p_{lr} = m(p_r)$ and $m_t(t) = t_l = m(t)$, and m is a c.m.

To summarize the arguments, to test whether $T_{left} \subseteq T_1$, it is sufficient to test for the existence of a c.m. conforming one of the mapping patterns Normal Case (i) or (ii), Partial Switch, or Total Switch. Whenever a c.m. not conforming to these patterns holds, either one of p_l, p_r subsumes the other, or another c.m. conforming to one of the above patterns exists as well. This completes the first part of sufficiency proof.

Proof Tree Transformations Technique

Fig. 6 depicts the proof-tree transformations induced by the c.m. m_l corresponding to each of the above patterns of subgoal mapping. Dually, it can be shown that there is a c.m. m_r : $T_1 \to T_{right}$ with similar properties as above, and that there are tree transformations induced by m_r which are symmetrical to the ones induced by m_l. As introduced in Chapter 3 we refer to these two types of transformations as *left* and *right* transformations respectively.

To complete the proof, we will prove that using these proof-tree transformations repeatedly we can show that an arbitrary proof-tree generated by the sirup r_1 can be reduced to a proof tree of height 1, while preserving equivalence. This is made precise by the following algorithm.

Algorithm 4.1.1 (An Algorithm for Proof Tree Transformations)

[Input :] *An arbitrary proof-tree S generated by the sirup* Π, *with the root goal* $p(X_1, \ldots, X_n)$.

[Output:] *A Proof tree of height 1.*

[Method:] *Repeatedly apply left and right transformations to S in a 'bottom-up' manner until no longer possible.*

```
repeat
    repeat
        Apply left transformations:
        For all subtrees $S_i$ of $S$ of the form $T_{left}$ where $p_{lf}$ and $p_{lr}$ are leaves.
        ($p_i$ can be the root of any subtree)
        transform $S_i$ into a subtree of the form $T_1$, using
        the appropriate left transformation from Fig. 2
    until $S$ has no subtrees of the above form.
    repeat
        Similarly, apply Right transformations in a symmetrical manner:
        For all subtrees $S_i$ of $S$ of the form $T_{right}$ where $p_{rf}$ and $p_{rr}$ are leaves,
        ($p_i$ can be the root of any subtree)
        transform $S_i$ into a subtree of the form $T_1$, using
```

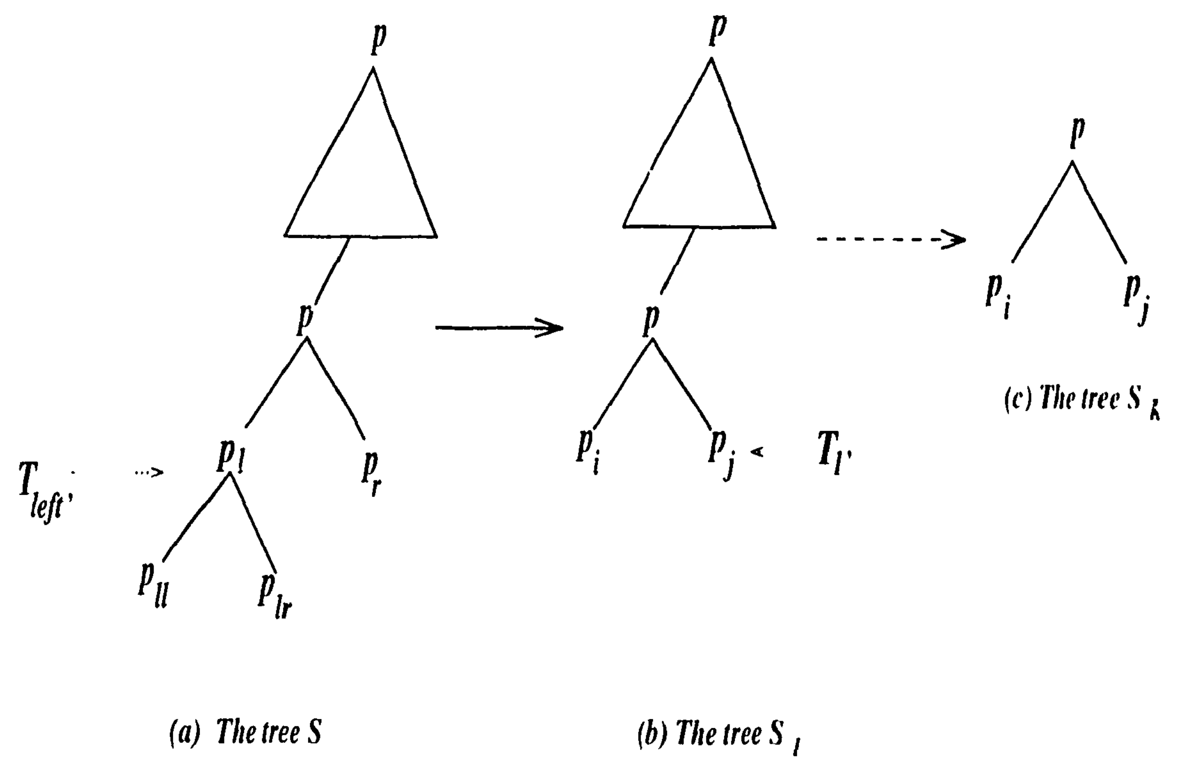


Figure 7: Showing Proof Tree transformation for 1-Boundedness

```
            the appropriate right transformation.
        until $S$ has no subtrees of the above form.
    until $S$ has no subtrees $S_i$ of the form $T_{left}$, $T_{right}$.
```

Applying the above proof-tree transformation procedure to the proof tree S, we can show that (i) the procedure successfully transforms S into a proof-tree S_k of the form T_1 (with root $p(X_1, \ldots, X_n)$), and (ii) S is contained in the resulting tree S_k.

In order to prove (i), notice that any proof-tree of height > 1 should have either a subtree of the form T_{left} with p_{ll} and p_{lr} as leaves, or a subtree of the form T_{right} with p_{rl} and p_{rr} as leaves. Every time a left / right transformation is applied, the number of subtrees of the form T_{right} or T_{left} is reduced by one, and the transformed tree is always a proof tree for $p(X_1, \ldots, X_n)$. It follows that the procedure will indeed terminate, leaving behind a resulting proof-tree of height 1 i.e. it is of the form T_1.

For (ii), we have the following inductive argument. Suppose that the tree S_1 is obtained from S by a left transformations as suggested in Fig. 7. Let $m_\ell : T_1 \rightarrow T_{left}$ be the associated c.m. Let $T_{lt_{ft}}'$ be the subtree of S to which the left transformation is applied converting $T_{lt_{ft}}'$ into T_1' (and hence S to S_1). We say a variable Z in $T_{lt_{ft}}$ (T_1) corresponds to a variable W in $T_{lt_{ft}}'$ (T_1') provided whenever Z occurs in an argument of a predicate a, W occurs in the same argument of the corresponding predicate occurence in $T_{lt_{ft}}'$ (T_1'). For a tree T (which is essentially a conjunctive query), let $vars(T)$ denote the set of all variables appearing in T. Since $T_{lt_{ft}}'$ (T_1') is an instance of $T_{lt_{ft}}$ (T_1), there exists a substitution θ : $vars(T_{lt_{ft}}) \rightarrow vars(T_{lt_{ft}}')$ such that θ maps each variable in $vars(T_{lt_{ft}})$ to the corresponding variable in $vars(T_{lt_{ft}}')$. Now, consider the binary relation m_t' defined as $m_t' = \theta^{-1} \circ m_t \circ \theta$. We claim that m_t' is a c.m. from T_1' to $T_{lt_{ft}}'$ showing $T_{lt_{ft}}'$ is contained in T_1'. If this is true, then m_t' can be easily extended into a mapping, say m : $S_1 \rightarrow S$ such that m coincides with m_t' on all variables appearing in T_1' and is an identity on all other variables in S_1. It is straightforward to see that m is then a c.m. from S_1 to S, as required. Thus, to complete the proof of (ii), it suffices to show that m_t' defined above is a c.m. Notice that to be a c.m., first of all, m_t' must be a function. We directly prove below that m_t' is a c.m.

(a) On every dv X_i, m_t' is an identity, *i.e.*, $m_t'(X_i) = X_i$.

Let X_i occur at any argument in T_1', say $s:j = X_i$, where s is any (not necessarily EDB) predicate. This implies the argument $s:i$ in the body of the sirup must contain an occurrence of some dv, say X_k, implying that the pair (X_i, X_k) is in θ^{-1}. Now, since m_t is an identity on all dv's, $m_t(X_k) = X_k$. Finally, by the definition of inverse, $\theta(X_k) = X_i$. This shows (a). Notice that the argument above implies that m_t' maps each dv to itself, and to no other variable.

(b) For every ndv V occurring in T_1', m_t' maps it to a unique variable in $T_{lt_{ft}}'$.

Suppose without loss of generality that the ndv V occurs in two (not necessarily distinct) argument positions, say $s:i$ and $t:j$. Two cases arise, depending on how $s:i$ and $t:j$ happen to share a common ndv.

<u>Case 1</u>. The arguments $s:i$ and $t:j$ in the sirup body carry some dv's, say $s:i = X_k$ and $t:j = X_m$, and the occurrence of the recursive predicate p at the root of the subtree T_1' carries the same ndv V at the positions $p:k$ and $p:m$.

In this case, the pairs (V, X_k) and (V, X_m) are both in θ^{-1}. As before, these pairs also belong to the composition $\theta^{-1} \circ m_t$. Finally, θ contains the pairs (X_k, V) and (X_m, V) by the definition of inverse, implying that $m_t'(V) = V$. This proves claim (b).

Case 2. The arguments $s : i$ and $t : j$ in the sirup body contain the same ndv, say U.

In this case, $(V, U) \in \theta^{-1}$. In fact, it is not hard to see that this is the only pair in θ^{-1} with the first component V. Now, since m_i and θ are functions, it is straightforward to see that $m_i{}'$ maps V to a unique variable, again implying claim (b).

From claims (a) and (b), it follows that $m_i{}'$ is a function and is an identity on all dv's. By the construction of $m_i{}'$ it is obvious that for every leaf subgoal s in $T_1{}'$, $m_i{}'(s)$ is a leaf subgoal of $T_{left}{}'$. This implies $m_i{}' : T_1{}' \rightarrow T_{left}{}'$ is indeed a c.m. This was to be shown. The proof of sufficiency is complete. □

It follows from the tree transformations above that any fact $p(X_1, \ldots, X_n)$ which can be derived using more than 1 application of the recursive rule in II (preceded by an application of the exit rule) can be obtained (using 1 application of the exit rule followed by) one application of the recursive rule. Thus, II is 1-bounded.

Discussion: Theorem 4.1.1 gives a direct method to test 1 boundedness of bilinear sirup. We however emphasize that a direct application of the above theorem will *not* yield a polynomial time test because of the exponentially many possible ways of mapping the EDB nodes in one tree (T_1) to the other (T_{left} or T_{right}). Thus an efficiently testable characterization is necessary.

To conclude this section, we give a few examples to illustrate the idea behind Theorem 4.1.1. In order to demonstrate the different cases of containment mappings, we have chosen 3 specific examples which illustrates the Normal, Partial, and Total Switch, identified in Thereom 4.1.1.

Example 4.1.1 *(An example of normal case)*

Given a bilinear program Σ, consisting of the rules:

$r_0 : p(X_1, X_2, X_3)$:- $e(X_1, X_2, X_3)$. *and*
$r_1 : p(X_1, X_2, X_3)$:- $p(X_1, V, X_1)$, $p(U, X_2, X_3)$, $a(X_1, V)$, $b(U, X_2)$.

Consider the proof-tree T_1 and T_{left} generated from Σ:

$r_1(T_1) : p(X_1, X_2, X_3)$:- $p(X_1, V, X_1)$, $p(U, X_2, X_3)$, $a(X_1, V)$, $b(U, X_2)$.
$r_1(T_{left}) : p(X_1, X_2, X_3)$:- $p_{ll}(X_1, V_1, X_1)$, $p_{lr}(U_1, X_2, X_3)$, $a_l(X_1, V_1)$, $b_l(U_1, V)$,
$p_r(U, X_2, X_3)$, $a(X_1, V)$, $b(U, X_2)$.

In this example, we will use the structural characterization of 1-boundedness as given in Theorem 4.1.1 to prove that the given program Σ is 1-bounded. We will first illustrate the containment mapping from T_{left} to T_1. More specifically, we will show that Σ satisfies the containment mapping for Normal Case i) from T_{left} to T_1. We have already seen from the proof of Theorem 4.1.1 that the subgoal mapping corresponding to Normal Case i) maps p_l of the tree T_1 to p_{ll} of the tree T_{left}. In addition, all subgoals related to p_l also map to the corresponding subgoals present in the level 2 of tree T_{left}. The rest of the subgoal follow identity mapping. Notice that in this example the subgoal a is related to p_l, therefore the EDB *subgoal a must map to a_l in the tree T_{left}. To show the existence of such a containment mapping, we enlist the resulting variable mapping m_l from T_1 to T_{left}: $m_l : (X_1) = X_1$, $m_l : (X_2) = X_2$, $m_l : (X_3) = X_3$, $m_l(V) = V_1$, and $m_l(U) = U$.*

Next, we will show that T_{right} is also contained in T_1. For this particular program, the containment $T_{right} \subseteq T_1$ is quite similar to $T_{left} \subseteq T_1$. It follows the normal case i) also. The resulting variable mapping from T_1 to T_{right} is as follows: $m_r : (X_1) = X_1$, $m_r : (X_2) = X_2$, $m_r : (X_3) = X_3$, $m_r(V) = V$, and $m_r(U) = U_1$.

Since, both the containments hold, using Theorem 4.1.1 we can conclude Σ is 1-bounded. □

The next example will illustrate : i) that $T_{left} \subseteq T_1$ by satisfying partial switch case.
ii) the proof-tree transformation technique by the construction of the c.m. m_l' showing T_{left}' $\subseteq T_1'$ for a tree of height 3.

Example 4.1.2 *Let us revisit Example 3.3.1. Consider the proof-trees T_1 and T_{left} generated by* Π.

$r_1(T_1) : p(X_1, X_2, X_3)$:- $p(V, X_2, U)$, $p(U, X_2, X_3)$, $a(X_1, X_2)$, $b(U, X_2)$.
$r_1(T_{left}) : p(X_1, X_2, X_3)$:- $p_{ll}(V_1, X_2, U_1)$, $p_{lr}(U_1, X_2, U)$, $a_l(V, X_2)$, $b_l(U_1, X_2)$,
$p(U, X_2, X_3)$, $a(X_1, X_2)$, $b(U, X_2)$.

i) It can be verified that there exists a containment mapping satisfying partial switch case from T_{left} to T_1 such that p_l — p_{ll}, p_r — p_r, a — a, and b — b. The symbol mapping m_l from T_1 to T_{left} is as follows: $m_l(X_1) = X_1$, $m_l(X_2) = X_2$, $m_l(X_3) = X_3$, $m_l(V) = U_1$ and $m_l(U) = U$. Hence we conclude $T_{left} \subseteq T_1$.

ii) Consider a left linear proof-tree S obtained as a result of expanding the recursive predicate p_l of the given rule r_1 3 times. Let T_{left}' be the left linear tree to which the proof-tree *transformations will be applied. Γ_1' is the 1-level tree corresponding to T_{left}'. In this example we will construct the mapping m_t' to show that the tree S of height 3 can be reduced to S_1, a proof-tree of height 2.*

$r_1(T_{left}')$: $p(V, X_2, U)$:- $p_{ll}(V_2, X_2, U_2)$, $p_{ll}(U_2, X_2, U_1)$, $a_l((V_1, X_2)$, $b_l(U_2, X_2)$, $p_r(U_1, X_2, U)$, $a(V, X_2)$, $b(U_1, X_2)$.

$r_1(\Gamma_1')$: $p(V, X_2, U)$:- $p_l(V_1, X_2, U_1)$, $p_l(U_1, X_2, U\ )$, $a(V, X_2)$, $b(U_1, X_2)$.

As proved in the sufficiency proof of the Theorem 4.1.1, $m_t' = \theta^{-1} \circ m \circ \theta$.

Step 1) We first construct θ^{-1} to represent the 1-1 mapping from the variables of Γ_1' to the corresponding variables of Γ_1: $\theta^{-1} = \{V_1/V, V/X_1, X_2/X_2, U/X_3, U_1/U\}$.

Step 2) The containment mapping $m: \Gamma_1 \rightarrow T_{left}$ has already been given above.

Step 3) θ: $T_{left} \rightarrow T_{left}' = \{U_1/U_2, X_1/V, X_2/X_2, X_3/U, U/U\}$.

Step 4) $m' = \{X_2/X_2, V_1/U_2, V/V, U/U, U_1/U_1\}$ is the required mapping to show that $\Gamma_1' \subseteq T_{left}'$. □

<u>Remark</u>: The proof of Theorem 4.1.1 can be used further to construct the mapping corresponding to $S \subseteq S_1$, where S_1 is the tree obtained as a result of performing 1 left transformation on S.

We will demonstrate a more complicated example of proof tree transformation technique. The next example has been chosen specifically to demonstrate the case when θ^{-1} is not a function.

Example 4.1.3 *Consider the sirup*

$r_1 : p(X_1, X_2, X_3, X_4, X_5)$:- $p(X_2, X_1, X_2, X_3, V)$, $p(X_1, X_2, X_3, X_4, W)$, $a(X_5)$.

The corresponding trees T_1 and T_{left} are as follows:

$r_1(T_1) : p(X_1, X_2, X_3, X_4, X_5)$:- $p(X_2, X_1, X_2, X_3, V)$, $p(X_1, X_2, X_3, X_4, W)$, $a(X_5)$.
$r_1(T_{left}) : p(X_1, X_2, X_3, X_4, X_5)$:- $p_{ll}(X_1, X_2, X_1, X_2, V_1)$, $p_{lr}(X_2, X_1, X_2, X_3, W_1)$,
$a_l(V)$, $p_r(X_1, X_2, X_3, X_4, W)$, $a(X_5)$.

In order to demonstrate the proof-tree transformation technique with respect to this example, *we will expand the recursive rule 3 times to obtain a proof-tree of height 3. Now consider the tree $T_{l,ft}'$ and T_1', where $T_{l,ft}'$ is the bottom-most left linear tree, whereas T_1' is the corresponding 1-level tree with respect to $T_{l,ft}'$.*

$r_1(T_{l,ft}')$: $p(X_2, X_1, X_2, X_3, V)$:- $p_{ll}(X_2, X_1, X_2, X_1, V2)$, $p_{lr}(X_1, X_2, X_1, X_2, V2)$, $a_l(V_1)$,
$p_l(X_2, X_1, X_2, X_3, W_1)$, $a(V)$.

$r_1(T_1')$: $p(X_1, X_2, X_1, X_2, V_1)$, $p(X_2, X_1, X_2, X_3, W_1)$, $a(V)$.

We next construct the c.m. m_t' showing $T_{l,ft}' \subseteq T_1'$. θ^{-1} corresponding to mapping T_1' to T_1 is: $\{X_1/X_2, X_2/(X_1, X_3), V_1/V, X_3/X_4, W_1/W, V/X_5\}$. The c.m. $m_t : T_1 - T_{l,ft}$ is as follows: $\{V - W_1$, the rest of the variables $(X_1, \ldots, X_5, W)$ follow an identity mapping$\}$. The mapping θ from $T_{l,ft}$ to $T_{l,ft}'$ sends $\{X_2/X_1, X_1/X_2, X_3/X_2, W_1/W_2, X_1/X_3, W/W_1, X_5/V$ $\}$. Finally the c.m. from T_1' to $T_{l,ft}'$ is given by $m_t' = \theta^{-1} \circ m \circ \theta$. Therefore, $m_t' = \{V_1 - W_2$, and identity on $(X_1, X_2, X_3, W_1, V)\}$. □

Example 4.1.4 *Illustrates the c.m. for Total Switch case:*

Given a bilinear program 3, with the following rules:

$r_0 : p(X_1, X_2, X_3, X_4)$:- $e(X_1, X_2, X_3, X_4)$.
$r_1 : p(X_1, X_2, X_3, X_4)$:- $p(X_3, V, X_1, U)$, $p(X_1, U, X_3, V)$, $a(X_2, X_4)$.
The corresponding trees T_1 and $T_{l,ft}$ generated by 3 are:

$r_1(T_1) : p(X_1, X_2, X_3, X_4)$:- $p_l(X_3, V, X_1, U)$, $p_r(X_1, U, X_3, V)$, $a(X_2, X_4)$.
$r_1(T_{l,ft}) : p(X_1, X_2, X_3, X_4)$:- $p_{ll}(X_1, V_1, X_3, U_1)$, $p_{lr}(X_3, U_1, X_1, V_1)$, $a(V, U)$,
$p_r(X_1, U, X_3, V)$, $a(X_2, X_4)$.

This example satisfies Total Switch case since the containment mapping m_f maps the subgoal p_l to p_{lr} and p_r to p_{ll}. The corresponding symbol mapping sends $V - U_1$, $U - V_1$ and all the dv's to identify mapping. □

Thus using various examples, we have illustrated the technique based on proof trees and containment mappings to detect 1-boundedness. Our next objective is to capture the notion of proof tree containment (and hence 1-boundedness) by identifying the syntactic characteristics of 1-bounded sirups. In the next section we develop some of the syntactic properties which are necessary before presenting our next result.

4.2 Syntactic Properties of Bilinear sirups

Some of the fundamental syntactic properties developed follows next. We will also introduce a lemma which will be helpful in the proof of main results.

Definition 4.2.1 *We say p_ℓ* saves *a variable Z occurring in $\varphi : i$, where φ is p_ℓ or p_r, if $\exists$ k such that $\varphi : k$ also carries Z and $p_\ell : i = X_k$. p_ℓ is* dv saving *if it saves every dv occurring in itself. We say that p_ℓ* preserves *a subgoal s related to it if every dv occurring in s also occurs in p_ℓ and is saved by p_ℓ. Intuitively, if p_ℓ saves a variable Z occurring in $p_\ell : i$, e.g., then it means $p_{\ell r} : i$ will carry Z. When p_ℓ is dv-saving, it implies $\forall i, j$, $p_\ell : i = X_j \Rightarrow p_{\ell\ell} : i = X_j$. p_ℓ* copies and saves *a dv X_j occuring in $p_r : i$, provided $\exists k$ such that $p_\ell : k = X_j$ and $p_\ell : i = X_k$. We say p_ℓ* projects *an argument of p_r in position i, provided $p_\ell : i$ is a dv. Further, p_ℓ* projects *equal arguments of p_r in positions i and j, provided $p_\ell : i = X_k$ and $p_\ell : j = X_m$, for some k and m, and $p_r : k = p_r : m$. Intuitively, this implies, $p_{\ell r} : i = p_{\ell r} : j$. All the notions above are defined for p_r in a similar manner.*

□

The following example makes these definitions concrete.

Example 4.2.1 *Consider the following recursive rule r_1. We will use r_1 to illustrate some of the above syntactic properties.*

$r_1 : p(X_1, X_2, X_3, X_4)$:- $p(X_1, X_2, X_3, U)$, $p(X_1, V, V, X_4)$, $a(U, X_2, X_1)$.
The recursive predicate p_ℓ is dv-saving in this example, since it saves every dv occuring in itself. Notice that every dv which occurs in p_ℓ occupies the ith position. p_ℓ also preserves the related EDB *a, as the dv's $\{X_2, X_1\}$ both appear in p_ℓ and they are saved as well. We say p_ℓ projects the argument V present in $p_r : 2$, since $p_\ell : 2 = X_2$. Similar argument can be made for $p_\ell : 3$. In addition, p_ℓ* projects *equal arguments of p_r in positions 2 and 3, since $p_\ell : 2 = X_2, p_\ell : 3 = X_3$, and $p_r : 2 = p_r : 3$. This will imply $p_{\ell r} : 2$ and $p_{\ell r} : 3$ will have the same argument V_1.* □

Definition 4.2.2 *Let $\alpha \in \{\ell, r\}$. Then by $\overline{\alpha}$ we denote its complement, defined as $\alpha = r$ iff $\alpha = \ell$ and vice versa. In a bilinear sirup Π, we say that the subgoals p_ℓ and p_r in the body (of the recursive rule) have matching ndv-patterns provided the following conditions are satisfied.*

1. *an argument $p_l : i$ of p_l carries a dude if and only if the corresponding argument $p_r : i$ of p_r carries a dude.*

2. *whenever an argument $p_l : i$ carries an ndv U, there exists an argument $p_r : j$ such that it carries an ndv V (here we may possibly have $i = j$, in which case, we must have $U = V$), and further $\forall k$, U appears in $p_\alpha : k$ iff V appears in $p_{\bar{\alpha}} : k$, for $\alpha = l, r$. In addition, if $U \neq V$ above, then neither of U, V appears in any edb subgoal.*

3. *the converse of the previous condition holds.*

We next present the technical lemma.

Lemma 4.2.1 *Suppose s, t are any two related* EDB *subgoals of a bilinear sirup* Π *and* $m : T_1 \to T_{l,fl}$ *is any c.m. Then either* $m(s) = s$ *and* $m(t) = t$ *or* $m(s) = s_l$ *and* $m(t) = t_l$.

Proof. Follows directly from the definition of a c.m. □

As a consequence of Lemma 4.2.1, any two related EDB subgoal present in the sirup will always map together to the respective subgoals of level 1(2). In addition, it is obvious that EDB subgoals which are related to p_l will always map to the corresponding subgoals in level 2 of the proof tree $T_{l,fl}$ whereas for subgoals which are not related, there (always) exists an identity mapping from T_1 to $T_{l,fl}$.

In the next section, we shall develop an elegant characterization of 1-boundedness leading to an efficient algorithm.

4.3 Syntactic Characterization for 1-boundedness

The objective of this section is to develop a syntactic characterization of 1-bounded bilinear sirups. We develop a characterization of 1-boundedness and extend it in such a way that 1-boundedness in the presence of functional dependencies to be satisfied by the EDB can also be captured. In a subsequent chapter, we shall develop a linear time algorithm for testing 1-boundedness, based on our characterization.

We next present the main theorem which gives a complete characterization of 1-boundedness for bilinear sirups. We use various syntactic properties of bilinear recursive sirups developed earlier in the previous section to present our result.

Theorem 4.3.1 (Syntactic Characterization of 1-boundedness) *Let Π be a bi-linear sirup, and let T_{left} and T_1 be the proof trees generated by Π as defined earlier. Then $T_{left} \subseteq T_1$ iff Π satisfies one of the following conditions.*

1. *p_l is dv-saving and p_l preserves every subgoal which is related to p_l.*
2. *whenever p_l has a dndv V occurring in arguments $p_l : i_1, \ldots, p_l : i_k$, either p_r projects equal arguments of p_l in positions $i_1, \ldots, i_k$, or p_r has some ndv V at the positions $p_r : i_1, \ldots, p_r : i_k$; whenever $p_l : i$ carries either a dv or an sndv, p_r saves that argument.*
3. *p_l and p_r satisfy the following conditions:*
   (a) p_r saves every dv occurring in p_l, and p_l copies and saves every dv occurring in p_r.
   (b) p_l preserves every edb subgoal related to p_l, and
   (c) p_l and p_r have matching ndv-patterns.

Proof:

It will be shown in the proof of this theorem that the semantic notion of 1-boundedness (as given in Theorem 4.1.1) is exactly captured by the syntactic conditions given in Theorem 4.3.1. Moreover, in the sufficiency proof, it will be shown that the 3 conditions given in the theorem corresponds to the 3 types of containment mapping: *Normal Case, Partial Switch, and Total Switch.* However, in the proof of necessity we will show that each of the conditions given in Theorem 4.3.1 is necessary for a consistent c.m.

($\Leftarrow$) :
We shall show that the 3 conditions respectively imply the existence of a c.m. from T_1 to T_{left} corresponding to *Normal Case, Partial Switch, and Total Switch*, showing $T_{left} \subseteq T_1$.

<u>Condition 1.</u> Consider the symbol mapping $m : T_1 \rightarrow T_{left}$ corresponding to the following assignment of subgoals: $m(p_l) = p_{ll}$; $m(s) = s_l$, for all subgoals s which are related to p_l; and $m(s) = s$, for all subgoals s unrelated to p_l. Clearly, m conforms to one of the patterns *Normal Case* (ii) or (i), depending on whether p_r is related to p_l or not. To complete the proof, we need to show that m is indeed a consistent c.m.

(i) Let X_i be a dv occurring in any subgoal argument position, say $s : j$. If s is a subgoal unrelated to p_l, $m(X_i) = m(s : i) = m(s) : i = X_i$, trivially. On the other hand, if s is either p_l or a subgoal related to p_l, it follows from condition 1 that $s_l : i = X_i$ as well. In other words, $m(X_i) = m(s : i) = m(s) : i = s_l : i = X_i$, as required of a c.m.

(ii) Let s, t be any two (not necessarily distinct) subgoals sharing an ndv, say $s : i = t : j = U$. By Lemma 4.2.1, we may assume at least one of s, t is an IDB subgoal. In this case, either $m(s) = s_l$ and $m(t) = t_l$, or $m(s) = s$ and $m(t) = t$, depending on whether s, t are related to p_l or not. In this case, we trivially have $m(s) : i = m(t) : j$, as required of a consistent c.m.

This was required to be shown.

<u>Condition 2.</u> Consider the symbol mapping $m : T_1$—T_{left} corresponding to the following assignment of subgoals: $m(p_l) = p_{lr}$, and $m(s) = s$, for all other subgoals s. Clearly, m conforms to the pattern *Partial Switch*. We need to show the symbol mapping m is indeed a consistent c.m.

(i) Consider a dv X_i occurring in any subgoal argument, say $s : j = X_i$. The only non-trivial case is when s is p_l. In this case, condition 2 implies $p_{lr} : j = X_i$. Thus, m is an identity on all dv's.

(ii) Consider any two (not necessarily distinct) subgoals s, t sharing an ndv, say $s : i = t : j = U$. By Lemma 4.2.1, we may assume at least one of s, t is an idb subgoal. Suppose s is p_l. If t is different from s, then U is an sndv, and by condition 2, $p_l : i$ is saved by p_r, implying $p_{lr} : i = U$. Since t is different from p_l, $m(t) = t$, and hence we have $m(p_l : i) = m(p_l) : i = p_{lr} : i = U = m(t : j) = m(t) : j$. If t is also p_l, then the same ndv U appears in positions $p_l : i$ and $p_l : j$. By condition 2, either p_r saves these arguments or carries the same ndv, say V in its corresponding positions $p_r : i, p_r : j$. In either case, $m(p_l : i) = m(p_l) : i = p_{lr} : i = p_{lr} : j = m(p_l) : j = m(p_l : j)$. Finally, the case where neither of s, t is p_l is trivial.

We have shown that in this case there is a consistent c.m. conforming to the *Partial Switch* pattern.

<u>Condition 3.</u> Consider the symbol mapping $m : T_1$—T_{left} corresponding to the following assignment of subgoals: $m(p_l) = p_{lr}$, $m(p_r) = p_{rl}$; $m(s) = s_l$, for every subgoal s which is related to p_l or p_r; and $m(s) = s$, for all remaining suboals s. Clearly, m conforms to the *Total Switch* mapping pattern. We need to show m is indeed a consistent c.m.

(i) Consider an occurrence of any dv, say $s : i = X_j$. For the sake of non triviality, assume s is either p_l, p_r, or a subgoal related to one of these. If s is p_l (p_r), then condition 3 implies $p_{lr} : i = X_j$ ($p_{ll} : i = X_j$). If s is an EDB subgoal related to p_l, since p_l preserves s, $s_l : i = X_j$. Finally, if s is related to p_r, then by the condition of matching ndv patterns between p_l and p_r, s must be related to p_l as well, in which case, again $s_l : i = X_j$.

(ii) Let $s : i = t : j =$ an ndv U. (s and t are not necessrily distinct.)

<u>Case 1.</u> s and t are the same subgoal. If s is an EDB subgoal, it is trivial to see that $m(s) : i = m(t) : j$. If s is p_l or p_r, then by the condition of matching ndv patterns, we have that $m(s) : i = m(t) : j$.

<u>Case 2.</u> s and t are distinct subgoals. By Lemma 1.2.1, we may assume at least one of them is an IDB subgoal. Suppose s is p_l. If t is an EDB sugboal, then it is easy to see $m(p_l : i) = m(p_l) : i = p_{lr} : i = t_l : j = m(t) : j = m(t : j)$. If t is p_r, then by the condition of matching ndv-patterns, it is clear that $m(p_l : i) = m(p_l) : i = p_{lr} : i = p_{rr} : j = m(p_r) : j = m(p_r : j)$. Similar arguments apply when t is p_l (and s is any subgoal).

It follows that in all cases, m is indeed a consistent c.m. showing $T_{left} \subseteq T_1$. This completes the proof of sufficiency.

($\Rightarrow$): Suppose $T_{left} \subseteq T_1$. By Theorem 4.1.1, we know that there is a c.m. $m : T_1 \to T_{left}$ conforming to one of the mapping patterns *Normal Case* (i) or (ii), *Partial Switch*, or *Total Switch*. We shall show that each mapping pattern implies one of the three conditions in the hypothesis of the theorem.

<u>Normal Case (i) or (ii).</u> (1) Assume p_l is not dv-saving. This implies $\exists j$ such that $p_l : j = X_i$, but $p_{ll} : j \neq X_i$. Since $m(p_l) = p_{ll}$, this implies $m(X_i) \neq X_i$, a contradiction.

(2) Suppose now that p_l does not preserve some subgoal s related to it, and let $s : k = p_l : m = U$. This means $\exists j$ such that $s : j = X_i$, but $s_l : j \neq X_i$. If $m(s) = s$, $m(U) = U$, but also, $m(U) = m(p_l : m) = m(p_l) : m = p_{ll} : m = U_1 \neq U$, a contradiction. If $m(s) = s_l$, $m(X_i) = m(s : j) = m(s) : j = s_l : j \neq X_i$, again a contradiction.

<u>Partial Switch.</u> (1) Suppose a dndv U appears in the positions $p_l : i, p_l : j, i \neq j$, and (i) p_l does not project equal arguments of p_l in positions i and j, and (ii) p_r does not carry the same ndv in the positions $p_r : i$ and $p_r : j$. Then $p_l : i = p_l : j$, while $m(p_l : i) = m(p_l) : i = p_{lr} : i \neq p_{lr} : j = m(p_l) : j = m(p_l : j)$, making m inconsistent as a c.m.

(2) Suppose p_l carries a dv, say $p_l : j = X_i$, which is not saved by p_r. In this case, $p_{lr} : j \neq X_i$. This will make m inconsistent as a c.m. since $m(p_l) = p_{lr}$.

(3) Suppose p_l shares an ndv with another subgoal s, say $p_l : i = s : j = U$, and p_r does not save this argument. This implies $p_{lr} : i \neq U$. If $m(s) = s$, then $m(U) = m(s . j) = U$, and also $m(U) = m(p_l : i) = p_{lr} : i \neq U$, making m inconsistent. If $m(s) = s_r$, then $m(U) = U_1$. In this case, for consistency, we must have $p_{lr} : i = U_1$, which implies $p_r : i = U$. However, since $m(p_l) = p_{lr}$, $m(U) = U \neq U_1$, a contradiction

Total Switch (1) Suppose p_r does not preserve some subgoal related to it. Let $p_l : i = s : j = U$. If $m(s) = s$, we must have $m(U) = U$, which is possible only if $p_r : i$ is a dv. In this case, since $m(p_l) = p_{lr}$, $m(p_l : i) = p_{lr} . i = U_1$, and so m is not an identity on some dv, a contradiction. So $m(s) = s_r$, and since p_r does not preserve s, $\exists k$ such that $s : k = X_i$ and $s_r : k \neq X_i$, again leading to a contradiction tot he fact that m is a c m.

(2) Suppose p_r does save some dv occurrence in p_l. This implies $\exists j$ such that $p_l : j = X_i$, and $p_{lr} : j \neq X_i$. This will force m to be a non-identity on the dv X_i, which is impossible. Similarly, if p_r does not copy and save some dv occurrence in p_r, then $\exists j$ such that $p_r : j = X_i$, but $p_{lr} : j \neq X_i$. Again, a contradiction will result, since $m(p_l) = p_{lr}$.

(3) Suppose now that p_l and p_r do not have matching ndv-patterns. We need to consider the three conditions in the definition of matching ndv-patterns.

(i) Suppose $p_l : i$ carries a dndv, say D, while $p_r : i$ does not. Then $p_r : i$ is either a dv or an s ndv. If $p_r : i$ is a dv, then since $m(p_l) = p_{lr}$, it is easy to see inconsistency will result in the mapping m. So suppose $p_r : i = s : j = U$, where s is any other subgoal. As argued under (1) above, it is easy to show that $m(s) = s_r$, which implies $m(U) = U_1$. However, since $m(p_l) = p_{lr}$, we have that $m(U) = D_1 \neq U_1$, a contradiction. Similarly, we can show that whenever $p_r : i$ carries a dndv, so must $p_l : i$.

(ii) Suppose condition 2 of matching ndv-patterns concerning sndv's is violated. Let $p_l : i = U$ be an sndv. If U is shared with an edb subgoal s, say at position $s : j$, then condition 2 of matching ndv patterns implies that $p_r : i$ must also carry this sndv U. Suppose $p_r : i \neq U$. In this case, $m(p_l) = p_{lr}$, implying $m(U) \neq U_1$. Then $m(s) \neq s_r$, and hence $m(s) = s$, implying $m(U) = U$. This is possible only if $p_r : i$ is a dv, which in turn will make m inconsistent since $p_{lr} : i$ is not a dv. A similar argument would show that whenever $p_r : i = s : j = U$, $p_l : i = U$ as well. Suppose now $p_l : i = p_r : j = U$. If $p_l : j \neq p_r : i$, then an inconsistency will result since $m(p_l) : i = p_{lr} : i \neq p_{lr} : j = m(p_r) : j$. Further, if $p_l : j = p_r : i$ is a dv, then an inconsistency will result because of the total switch mapping pattern. Consequently, $p_l : j = p_r : i =$ some ndv, say V. A straightforward extension of this argument will show that whenever $p_l : k = U$, we must have $p_r : k = V$, and vice versa. Finally, suppose U appears in an edb subgoal, say at $s : m$. Then as argued above, we can show $m(s) = s_r$. This implies $m(U) = U_1$. However, since $m(p_l) = p_r$, and $m(p_r) = p_{rr}$, we must have $m(U) = V_1 \neq U_1$, a contradiction. The above argument can be symmetrically applied from p_r to p_l just as above. This shows the necessity of all conditions in the definition of matching ndv-patterns. This completes the proof of the theorem. □

Discussion:

We will revisit Example 4.1.1, 4.1.2 and 4.1.4 once again to verify $T_{left} \subseteq T_1$ using the syntactic conditions given in Theorem 4.3.1. Recall that in Section 4.1, we have already demonstrated $T_{left} \subseteq T_1$ for each of the above example (using structural characterization as presented in Theorem 4.1.1). Using the syntactic conditions this time, we will show the same conclusion.

Example 4.3.1 *(Normal Case)*
We will revisit Example 4.1.1 to illustrate the syntactic conditions satisfied by the sirup. It can be verified that Normal Case i) is satisfied with respect to p_l, since p_l is dv-saving and it also preserves the related subgoal a. This implies $T_{left} \subseteq T_1$. □

Example 4.3.2 *(Total Switch Case)*
We will revisit Example 4.1.4. The syntactic conditions for Total Switch case can be verified as follows: a) p_l saves the dv's (X_3, X_1) occuring in p_l (b) p_l copies and saves every dv occuring in p_r. (c) p_r and p_l have matching ndv-patterns, since the ndvs U, V are involved in an ndv-cycle of lenght 2. Since, p_l and p_r satisfy all the necessary and sufficient conditions for Total Switch, we conclude $T_{left} \subseteq T_1$. □

For the next example we will use the syntactic conditions of Theorem 4.3.1 to test the containment for both directions: $T_{left} \subseteq T_1$ and $T_{right} \subseteq T_1$. The purpose of this exercise is to illustrate that the sirup P introduced in Example 3.3.1 is 1-bounded.

Example 4.3.3 *(Partial Switch Case) Let us revisit Example 4.1.2.*

In this example, the subgoals p_r, b are related to p_l since they share the ndv U with p_l. The EDB *a is not related to any predicate. Applying the syntactic condition with respect to p_l*

first, it can be verified that the condition for normal case fails since p_l does not preserve p_r. However, the condition for partial switch case is satisfied since p_r saves all the sides and des of p_l. As seen in example 4.1.2, the corresponding symbol mapping m_l from T_1 to T_{left} is as follows: $m_l(X_1) = X_1$, $m_l(X_2) = X_2$, $m_l(X_3) = X_3$, $m_l(V) = U_1$ and $m_l(U) = U$. In view of this, $T_{left} \subseteq T_1$.

In the other direction of containment testing, normal case is satisfied since p_l is dv-saving and it preserves the related subgoals p_l, b. Hence, $T_{right} \subseteq T_1$ and the containment mapping m_r from T_1 to T_{right} maps the variables as follows: $m_r(X_1) = X_1$, $m_r(X_2) = X_2$, $m_r(X_3) = X_3$, $m_r(V) = V_2$ and $m_r(U) = U_2$. Since, $T_{right} \subseteq T_1$ and $T_{left} \subseteq T_1$, we conclude that the given sirup is 1-bounded. □

Before closing this section, we shall show that our characterization for bilinear sirups can just as well be used for testing 1-boundedness of *linear* sirups.

Theorem 4.3.2 *Let Π be the linear sirup defining a predicate p. Then Π is 1-bounded iff the recursive subgoal p in the body of the sirup is dv-saving and it preserves all subgoals related to it.*

Proof. First, notice that Π is 1-bounded iff the proof-tree T_2 obtained by expanding p twice is contained in the proof-tree T_1 with one expansion of p. Now, it is clear that the only mapping pattern possible for a potential c.m. from T_1 to T_2 is for leaf corresponding to p in T_1 to be mapped to the leaf in T_2 corresponding to p. Further, whenever a is an EDB subgoal related to p, the subgoal a in T_1 must be mapped to a_l in T_2, which occurs at level 2. The rest of the proof follows identical lines to those adopted for *Normal Case* of Theorem 4.3.1. □

In view of the linear time algorithm given in Section 5.1.2, Theorem 4.3.2 implies we can also test 1-boundedness of *linear* sirups without repeated edb subgoals in *linear time*. Throughout this thesis, we assume that in the bilinear sirups we consider, neither of p_l, p_r subsumes the other. The above observation shows that this is not a restrictive assumption.

To conclude this section an example is given demonstrating the technique presented in Theorem 4.3.2.

Example 4.3.4 *Consider the following linear sirup α_{lin}.*

r_0: $p(X_1, X_2, X_3)$:- $e(X_1, X_2, X_3)$.
r_1 : $p(X_1, X_2, X_3)$:- $p(X_1, V, X_3)$, $a(X_1, V)$, $b(V, X_2)$.

We will show that α_{lin} is 1-bounded. To prove our results, we will apply the syntactic conditions for Normal case given in Theorem 4.3.1 and follow a similar technique as used in the detection of 1-boundedness for bilinear sirups.

It is quite easy to check that Normal Case i) is satisfied, since p is dv-saving and it also preserves the related subgoal a. Thus $\Gamma_{left} \subseteq \Gamma_1 \Rightarrow \alpha_{lin}$ is 1-bounded. □

4.4 Summary

The problem of detecting 1-boundedness for bilinear sirups was considered in this chapter. Our first result on 1-boundedness was based on containment mappings and proof trees. We identified the 3 different cases of containment mappings which arise with respect to 1-boundedness. Proof tree transformation technique was described which reduces an arbitrary tree to a tree of height 1. For the reason of efficiency, we also justified the need for syntactic conditions for capturing 1-boundedness. An elegant characterization based on syntactic conditions was developed for bilinear sirups. It will be shown in the Section 5.1.2 that the syntactic characterization leads to a linear time algorithm. To conclude this chapter, we also showed that our syntactic characterization can be used to detect 1-boundedness of linear sirups as well.

Chapter 5

1-Boundedness with Functional Dependencies

It has been known that the presence of different types of data dependencies can be used for efficient query processing. Various earlier studies have shown that queries can be further optimized using the semantic knowledge of the given programs. Functional dependencies defined on the base relations provide crucial information which can be exploited to increase the efficiency of the programs. Our objective in this chapter is to study the influence of FDs on 1-boundedness.

We have already discussed the problem of detecting 1-boundedness in the previous chapter. In this chapter, we will address the problem of detecting bilinear sirups which are 1-bounded on all input databases which satisfy a given set of functional dependencies (FDs). We have already illustrated in Example 3.3.2 that datalog programs that are not (1-)bounded could become 1-bounded in the presence of FD's. We develop a methodology for detecting such bilinear sirups. The technique is based on preprocessing the bilinear sirup with respect to the given Functional dependencies and identifying the minimal extensions necessary to the basic syntactic characterization of Section 4.3 in order to account for the FD's.

Since our overall objective is not only to characterize 1-boundedness, but also to develop a linear time test for it, we must address the following questions. (1) How to preprocess the sirup so as to incorporate the effect of FDs on 1-boundedness. (2) How to do the above preprocessing *in linear time*? (3) What are the modifications to the syntactic characterization of Section 4.3 to take account of FDs?

Our investigation reveals that we can use the *chase procedure* developed in classical database theory by Maier et al. [19] to preprocess and hence to incorporate the effects of FDs. We then show that the syntactic characterization developed in the preceding section can be used with minor modifications to detect 1-boundedness.

The organization of this chapter is as follows. In Section 5.1 we first review the basic *chase* algorithm to incorporate the FDs into the sirups. Some relevant results on FDs and chase are also discussed in this chapter. In Section 5.2 we develop an efficient algorithm for chasing proof-trees. The overall time complexity for this algorithm is almost linear in the size of the proof-tree and the input FDs. The extended syntactic characterization of 1-boundedness is defined in Section 5.3. Finally, in Section 5.4.2 the main result of this thesis, testing of 1-boundedness in linear time is presented.

Throughout this chapter, we assume that F is a set of FDs on the EDB predicates of the given program. Since FDs on the exit predicate will not influence the chase of proof trees (see below), we only consider FDs on the other predicates.

5.1 Incorporating Functional Dependencies

Our primary goal in this section is to develop a procedure to incorporate the effects of FDs into the given sirup. This issue has been investigated several times in the earlier literature. We settle this issue by making use of the *chase procedure*. Indeed, the chase procedure has been extensively used in dependency theory as well in the context of several query optimization problems (*e.g.*, see Ullman [28], [29] and Maier [19]). Our idea is to chase the proof-tree $T_{l,ft}$ generated by the sirup and then consider the problem of testing whether the chased proof-tree $T_{l,ft}$ is contained in the proof tree of height 1, *viz.*, T_1.

We now describe the idea behind the chase, as applied to proof trees. The following algorithm describes the basic method, without regard to its time complexity.

Algorithm 5.1.1 (Algorithm for chasing Proof trees)

Input: *A proof-tree of the form $T_{l,fl}$ generated by a bilinear sirup, and a set of FDs on the* EDB *predicates of the sirup.*

Output: *The chased proof-tree $T_{l,fl}$* .

Notes: *(1) We let Args and arg denote a set of arguments and an argument respectively. (2) Since we consider bilinear sirups with distinct* EDB *subgoals and consider only FDs on* EDB *predicates, subgoals s, t below will necessarily be* EDB, *with one of them at level 1 and the other at level 2.*

```
begin
repeat
    Whenever there are subgoals $s$ and $t$ corresponding to the same predicate, $s$ at level
    1 and $t$ at level 2, such that $Args \rightarrow arg$ is an given FD on this predicate, and
    $s[Args] = t[Args]$, but $s[arg] \neq t[arg]$, equate these two arguments, replacing all
    occurrences of the variable appearing in $t[arg]$ by the variable appearing in $s[arg]$.
until no change to the proof-tree
end
```

The next example shows an illustration of the chase procedure as described in Algorithm 5.1.1.

Example 5.1.1 *(Revisit Example 3.3.2)*

Consider the following bilinear recursive sirup P together with the FD $F = \{a : 1 \rightarrow 2\}$.

$r_0 : p(X_1, X_2, X_3)$:- $e(X_1, X_2, X_3)$.
$r_1 : p(X_1, X_2, X_3)$:- $p(X_1, W, X_3)$, $p(X_1, X_2, W)$, $a(X_1, W, X_2)$.

It can be verified using the tree $T_{l,ft}$ and T_1 (given below) that in the absence of the given FD the program is not 1-bounded.

$r_1(T_1):\ p(X_1, X_2, X_3)$:- $p(X_1, W, X_3),\ p(X_1, X_2, W)\ a(X_1, W, X_2).$
$r_1(T_{l,ft}):\ p(X_1, X_2, X_3)$:- $p_e(X_1, W_1, X_3),\ p_e(X_1, W, W_1),\ a_e(X_1, W_1, W),$
$p_r(X_1, X_2, W),\ a(X_1, W, X_2).$

Therefore, we proceed with the preprocessing of the tree $T_{l,ft}$ with respect to the FD F. Applying the Algorithm 5.1.1 with respect to $T_{l,ft}$ and the FD F, we obtain the chased proof-tree $T_{l,ft}{}^c$.

$r_1(T_{l,ft}{}^c):\ p(X_1, X_2, X_3)$:- $p_e(X_1, W, X_3),\ p_e(X_1, W, W),\ a_e(X_1, W, W),$
$p_r(X_1, X_2, W),\ a(X_1, W, X_2).$

Notice that in the resulting chased proof-tree, all the occurrences of the variable W_1 get equated to W because of the FD $F = \{a : 1 \rightarrow 2\}$. □

The extended definition of containment in presence of FDs follows next

Definition 5.1.1 *Let Q_1, Q_2 be any conjunctive queries and let F be a set of integrity constraints on the* EDB *inputs to Q_1, Q_2. Then we say that Q_1 is contained in Q_2 relative to the integrity constraints F, denoted $Q_1 \subseteq_F Q_2$, provided, for all* EDB *inputs $\mathcal{D}$ satisfying the integrity constraints F, $Q_1(\mathcal{D}) \subseteq Q_2(\mathcal{D})$.*

A direct application of the above definition to the proof-trees $T_{l,ft}$ and T_1 leads us to the following result.

Theorem 5.1.1 *Let $\mathcal{H}$ be a bilinear sirup, F a set of FDs on the* EDB *predicates of $\mathcal{H}$, and $T_{l,ft}{}^c$ the proof-tree obtained by applying Algorithm 5.1.1 to $T_{l,ft}$. Then $T_{l,ft} \subseteq_F T_1$ iff $T_{l,ft}{}^c \subseteq T_1$.*

Proof. Consider any EDB $\mathcal{D}$, input to $T_{l,ft}{}^c$. It follows from the chase that whenever two atoms s, t in $\mathcal{D}$ corresponding to the same EDB predicate are used in deriving an output tuple using $T_{l,ft}{}^c$, the atoms s, t necessarily satisfy all FDs in F applicable to this EDB predicate. For otherwise, they would not unify with the corresponding subgoals in $T_{l,ft}{}^c$, as the chase procedure implies. Let $\mathcal{D}'$ be the maximal subset of $\mathcal{D}$ such that $\mathcal{D}'$ satisfies the FDs F. It is straightforward to show from the above that $T_{left}(\mathcal{D}') = T_{left}'(\mathcal{D})$.

($\Leftarrow$): Suppose $\mathcal{E}$ is any EDB satisfying the FDs F. Then $T_{left}(\mathcal{E}) = T_{left}'(\mathcal{E}) \subseteq T_1(\mathcal{E})$.

($\Rightarrow$): Suppose $\mathcal{E}$ is any EDB and $\mathcal{E}'$ be its maximal subset which satisfies the FDs F. Then as shown above $T_{left}'(\mathcal{E}) = T_{left}(\mathcal{E}')$. However, $T_{left}(\mathcal{E}') \subseteq T_1(\mathcal{E}') \subseteq T_1(\mathcal{E})$. □

Discussion: Theorem 5.1.1 gives a direct approach to test for the containment of proof-trees in presence of functional dependencies. It shows that containment testing between the proof-trees T_{left} and T_1 in presence of functional dependencies can be reduced to that of testing whether the chased proof-tree T_{left}' is contained in T_1 or not. Recall that the simple chase algorithm given earlier in this section can be used to generated the proof-tree T_{left}^c, however it would not be efficient if implemented as such. Since our main objective in this chapter is to develop a simple yet efficient technique for testing containment in presence of FDs, therefore Theorem 5.1.1 clearly defines the necessity for an efficient chase algorithm.

To conclude this section an example is given demonstrating the technique introduced in Theorem 5.1.1.

Example 5.1.2 *Consider the bilinear recursive program P and the FD $F = a : 1 \rightarrow 2$.*

$r_1 : p(X_1, X_2, X_3)$:- $p(X_1, W, X_3)$, $p(X_1, X_2, W)$, $a(X_1, W, X_2)$.

For the sirup P we have already seen the construction of the proof-tree T_{left}^c using the chase algorithm in Example 5.1.1. In this example we will illustrate using the result given in Theorem 5.1.1 that $T_{left} \subseteq_F T_1$.

$r_1(T_1) : p(X_1, X_2, X_3)$:- $p(X_1, W, X_3)$, $p(X_1, X_2, W)$ $a(X_1, W, X_2)$

$r_1(T_{left}^c)$: $p(X_1, X_2, X_3)$:- $p_{ll}(X_1, W, X_3)$, $p_{lr}(X_1, W, W)$, $a_l(X_1, W, W)$,
$p_r(X_1, X_2, W)$, $a(X_1, W, X_2)$.

Indeed, it can be verified that the containment $T_{left}^c \subseteq T_1$ goes through by mapping all the variables of T_1 to identity. This implies $T_{left} \subseteq_F T_1$. Similar preprocessing is required for showing $T_{right}^c \subseteq T_1$.

Overall, the symbol mapping for both the directions of containments sends W → W, X_1 → X_1, X_2 → X_2, X_3 → X_3. Hence, we have shown that the given sirup is 1-bounded in presence of the given FD F. □

We will move on to next section where we will develop an efficient algorithm for chasing the proof-tree T_{left} to incorporate the given functional dependencies F.

5.2 An efficient Algorithm to Chase Proof trees

In this section, we develop an algorithm for chasing proof-trees of the form T_{left} (or T_{right}) in time almost linear[1] in the size of the tree and the input FDs. This result in itself is quite significant. Since, the fastest known algorithm for chasing FDs in the literature (prior to our result) has a exponential time complexity.

Our algorithm for chasing proof-trees of the form T_{left} or T_{right} follows next.

Algorithm 5.2.1 (Chase(H, F))

Input: A bilinear sirup P, and a set of extensional FD's F.
Output: The proof-tree T_{left}', which is T_{left} chased with respect to F, and H' which is H incorporating the effect of any dv's that get equated during the chase of T_{left}.

Assumption: $|RHS\_of\_FD| = 1$.

Data structure:

CHASABLE, CHASED : store sets of arguments currently "chasable", and those chased so far, respectively.
TREE : An array that stores the proof-tree T_{left} (or T_{right}).
COUNT : An array of integer, ranging over the FDs in F;
For an FD $f \equiv a{:}Args \rightarrow arg$, $COUNT[f] = |Args\text{-}CHASED|$ = the number of arguments in the LHS of the FD which have not been chased so far.
LIST : An array of lists, ranging over the distinct variables in the sirup;
For a variable V, $LIST[V]$ = a list of FDs whose LHS includes V. We say the LHS of a

[1]The exact time complexity is $O(m\beta(m))$ where m is the input size of the problem and β is an extremely slow growing function.

FD a:$\{i_1, \ldots, i_k\} \rightarrow j$ includes a variable V provided for some $1 \leq \ell \leq k$, a:i_ℓ carries the varaible V in the sirup body.

Method: The key idea in the algorithm is that to carry out the chase correctly (and efficiently) the only information that is required at any stage is which sets of arguments carry an identical variable as a result of the chase so far. We represent the effect of chase on T_{left} by storing the arguments of subgoals in the form of equivalence classes. Two arguments are in the same class exactly when they are supposed to carry an identical variable as a result of the chase. We use the well-known Disjoint-Set-Union-Find algorithm to maintain the progression of the chase (see [2]). We use the following instructions from that algorithm: $UNION(S_1, S_2, S_3)$ and $FIND(x)$. In the algorithm below, we assume without loss of generality that T_{left} is to be chased with respect to F.

```
Begin
    (1) Construct $T_{left}$ and store it in TREE;
    (2) if $F = \phi$ then return ( TREE) and exit.
    (3) Initialization :
        • (a) initialize CHASABLE,CHASED to $\phi$;
        • (b) foreach argument $a : i$ do
                  LIST[a:i] := nil;
              end {for};
        • (c) for each FD $f \equiv a$:Args $\rightarrow$ arg in F do
                  COUNT[f] := |Args|;
                  for each i in Args do
                      add f to LIST[a:i];
                      if var(a:i) = var($a_\ell$:i) then
                          CHASABLE := CHASABLE $\bigcup$ {a:i};
                  end {for};
              end {for};
        • (d) for each distinct variable V in TREE d o
                  construct a tree with one node (also its root) labelled V.
              end {for};
    (4) Chasing :
        while CHASABLE$\neq \phi$ do
              choose any argument a:i in CHASABLE;
```

```
                delete a:i from CHASABLE;
                add a:i to CHASED;
                for each FD $f \equiv a{:}Args \rightarrow arg$ in LIST[(var( a:i))] do
                    COUNT[f]:= COUNT[f] -1;
                if COUNT[f]=0 then
                      if FIND(var($a_l$:arg)) $\neq$ FIND(var(a:arg) ) then
                        begin
                            UNION(var($a_l$:arg),var(a:arg),X);
                            add a:arg to CHASABLE;
                        end
                end {for}
            end while
        (5) Identification of usndv's
            for each argument s:i of the sirup do
                if (s:i has an ndv) and (FIND(var(s:i)) = FIND(var($s_l$:i))) then
                    conclude s:i carries a usndv;
            end {for}
        (6) Identification of dv's that got equated :
            for each equivalence class of arguments S do
                enumerate the variables (in the original $T_{left}$) corresponding to S.
                conclude that two dv's $X_i$ and $X_j$ get equated during the chase
                exactly when both appear in the enumeration corresponding
                to the same class S.
            end {for}
end.
```

We remark that from the point of view of testing 1-boundedness, the most important outputs expected from Algorithm 5.2.1 concern the following information: (i) which arguments of the original sirup carry a usndv; (ii) which dv's got equated during the chase of T_{left} (or T_{right} as appropriate). Using item (ii), we can easily rewrite the original sirup Π into the modified sirup Π' as follows. Suppose S is a set of distinct dv's that got equated during the chase. Pick an arbitrary representative of S, say the dv X_i. Then uniformly replace all occurrences in the sirup (both head and body) of any dv in S by the dv X_i. Repeat this procedure for each maximal set of dv's that got equated in the chase. The resulting sirup is the required program Π'. The main idea at this point is that (the algorithm based on) Theorem 4.3.1 can be applied to test if the original sirup is 1-bounded w.r.t. the FDs F. We now prove

Theorem 5.2.1 *Algorithm 5.2.1 correctly chases the proof-tree T_{left} (or T_{right} as appropriate). In particular, (i) an argument $s : i$ in the original sirup Π carries a usndv w.r.t. the FDs F iff Algorithm 5.2.1 says so; (ii) two dv's X_i, X_j get equated during the chase iff they both belong to the same equivalence class at the end of Algorithm 5.2.1.*

Proof. We first remark that the termination of Algorithm 5.2.1 follows from the facts that (a) The number of arguments is finite, (b) no argument enters CHASABLE more than once, and (c) each argument in CHASABLE is chosen (and processed upon, and deleted) at most once.

The correctness (pertaining to (i) and (ii) in the theorem) can be proved by observing the following. Initially, each equivalence class contains just one (distinct) variable which indicates the state where no variable has been equated to any other. That is, T_{left} is not chased at all. It is clear from the algorithm that two variables are put in the same equivalence class (Step (4) – see the UNION instruction) exactly when these variables appear in a pair of arguments which get equated in the chase. From this, (i) and (ii) readily follow. □

We make use of the following function to characterize the time complexity of Algorithm 5.2.1. Let α be the function defined by $\alpha(0) = 1$ and $\alpha(n) = 2^{\alpha(n-1)}$, for $n > 0$. As pointed out in Aho et. al. [2] α is extremely fast growing function. The "inverse" β of α is defined by setting $\beta(n)$ to be the smallest number k such that $\alpha(k) \geq n$. Clearly, β is an extremely slow growing function. In the following theorem, the input size m refers to the size of Π and F as measured by the space needed to write down the sirup Π and the FDs F.

Theorem 5.2.2 *Algorithm 5.2.1 takes time $O(m\beta(m))$ where m is the size of the input to the algorithm.*

Proof. It is not hard to see Steps (1)-(3) take time $O(m)$. Specifically, in Step (3), each argument in each FD is visited once and operations requiring constant time are performed after each such visit. This only leaves Steps (4)-(6). For Step (4), notice that no argument of the sirup enters CHASABLE more than once. Furthermore, at any stage, for each argument in CHASABLE, the COUNT of each FD whose LHS includes this argument is decremented. The overall time needed for this step, from start until CHASABLE becomes empty, is proportional to the size of description of the FDs in F. By using the DISJOINT-SET UNION-FIND algorithm with path compression (see [2]), the FIND and UNION instructions can be efficiently implemented. Notice that the number of times these instructions are invoked is proportional to the number of arguments in the sirup H. This number is no more than m and the above UNION-FIND algorithm, we can guarantee that a sequence of cm FIND and UNION instructions can be performed in time $c'm\beta(m)$, where c, c' are both constants, and c' depends on c. By maintaining a count for each argument of the sirup carrying an ndv, we can decide if the ndv is an sndv. Then Step (5) can be performed in time proportional to the number of arguments of the sirup. It is easy to see Step (6) takes time no more than the number of distinct variables in the sirup. Thus, the overall time complexity is $O(m\beta(m))$. □

In this section, we have assumed the exit predicate e does not satisfy any non-trivial FDs. When e does satisfy some FDs, some of them could be induced on the IDB predicates. Determination of the induced FDs is orthogonal to the problem studied in this paper. Whenever induced FDs are known, they could be incorporated in the chase (Algorithm 5.1.1). Unfortunately, determination of induced FDs is undecidable in general (Abiteboul and Hull [1]). Positive results for sirups are proved in Hernandez et al. [8].

5.3 Syntactic Characterization of 1-Boundedness in Presence of FD's

In this section we will present the main result regarding the detection of 1-boundedness in presence of FDs. The strategy we employ makes use of the syntactic characterization developed in Section 4.3 with the following modifications. Firstly, we need to strengthen the notion of related subgoals to reflect the idea that certain sndv's do not change between levels 1 and 2 in T_{left}. This is because of FDs. (See Example 5.1.1.) Secondly, when T_{left} is chased, it is possible that two dv's get equated. We then need to test the containment of such a tree in T_1.

Definition 5.3.1 *We call an sndv U occurring in a subgoal argument $s : i$ unchanging (usndv) provided $s : i = s_\ell : i = U$, in the chased tree $T_{left}{}^c$. A similar definition applies w.r.t. $T_{right}{}^c$.*

We finally modify the notion of relatedness as follows.

Definition 5.3.2 *We say that two subgoals s, t of a sirup are* related *provided they share an ndv which is not unchanging in the context of $T_{left}{}^{c}$ ($T_{right}{}^{c}$).*

Notice that depending on the containment that needs to be tested (*i.e.*$T_{left}{}^{c} \subseteq T_1$ or $T_{right}{}^{c} \subseteq T_1$), the appropriate notion of relatedness should be used.

The next example clarifies the notion of usndvs and relatedness.

Example 5.3.1 *Consider again the bilinear sirup given in Example 5.1.1 In the sirup r_1, the variable W is an usndv since in the resulting chased proof-tree $T_{left}{}^{c}$, $a:2 = a_{\ell}:2 = W$. Notice that, as a result of the chase algorithm, the FD $F = \{a:1 \rightarrow 2\}$ makes (all) the occurences of W in level 1 and 2 equal i.e $p_{\ell}:2 = p_{\ell\ell}:2 = p_r:3 = p_{\ell r}:3 = a:2 = a_{\ell}:2 = W$. This can be verified using the tree $T_{left}{}^{c}$ given in Example 5.1.1.*

In addition, none of the subgoals in the sirup is related to either p_{ℓ} or p_r (in the sense of Definition 5.3.2), since the only sndv contributing to relatedness with p_{ℓ} and p_r is an usndv.

□

The next example illustrates another complication that can arise when we want to test if $T_{left}{}^{c} \subseteq_F T_1$.

Example 5.3.2 *(Demonstrates the case where two dv's get equated).*

Consider the following sirup Q and the FD $F = \{a:1 \rightarrow 2\}$.

$r_1 : p(X_1, X_2, X_3)$:- $e(X_1, X_2, X_3)$.
$r_2 : p(X_1, X_2, X_3)$:- $p(X_1, X_3, V)$, $p(X_1, X_2, V)$, $a(X_1, X_2, V)$.

For this example also, it can be verified using the syntactic characterization of 1-boundedness presented in Section 4.3.1 that the sirup is not 1-bounded in the absence of the FD F. The respective proof-trees T_1 and T_{left} are given below for verifications.

$r_2(T_1) : p(X_1, X_2, X_3)$:- $p(X_1, X_3, V)$, $p(X_1, X_2, V)$, $a(X_1, X_2, V)$.
$r_2(T_{left}) : p(X_1, X_2, X_3)$:- $p_{\ell\ell}(X_1, V, V_1)$, $p_{\ell r}(X_1, X_3, V_1)$, $a_{\ell}(X_1, X_3, V_1)$,
$p_r(X_1, X_2, V)$, $a(X_1, X_2, V)$.

We apply the chase procedure given in Algorithm 5.1.1 which results in T_{left}^c with the dv X_3 equated to X_2.

$$r_2(T_{left}^c) : p(X_1, X_2, X_2) :- p_l(X_1, V, V_1),\ p_{lr}(X_1, X_2, V_1),\ a_l(X_1, X_2, V_1),$$
$$p_r(X_1, X_2, V),\ a(X_1, X_2, V).$$

Since, in this example two dv's get equated, we have to show that $r_2(T_{left}^c)$ (with the equated dv's) $\subseteq r_2(T_1)$. □

The next proposition suggests how such containments can be tested.

Proposition 5.3.1 *Let Q_1 and Q_2 be two conjunctive queries defining a predicate p $(X_1, \ldots, X_n)$, such that the body of Q_2 contains an equality of the form $X_i = X_j$, $i \neq j$. Then $Q_1 \subseteq Q_2$ iff $\sigma_{X_i = X_j}(Q_1) \subseteq Q_2$.*

Proof. Follows from the observation that for any input EDB D, and any tuple $t \in Q_2(D)$, $t[i] = t[j]$. □

Our final result outlines the test for 1-boundedness in the presence of FDs. Suppose that when the proof-tree T_{left} generated by a bilinear sirup Π is chased w.r.t. the given FDs F, some pair of distinct dv's X_i, X_j of Π get equated. We can rewrite Π into another sirup Π' by replacing all occurrences of X_j by X_i throughout the program (both the recursive rule and the exit rule). We say that Π' incorporates the effect of the dv's equated during the chase of T_{left}.

Theorem 5.3.1 (Syntactic Characterization of 1-boundedness with FDs) *Let Π be a bilinear sirup and F a set of FDs on the EDB predicates of Π. Let Π' be the sirup obtained from Π by incorporating the effect of any distinct dv's that were equated during the chase of T_{left}. Then $T_{left} \subseteq_F T_1$ iff Π' satisfies one of the following syntactic conditions.*

1. *p_l is dv-saving and p_l preserves every subgoal which is related to p_r.*

2. *whenever p_l has a dndv U occurring in arguments $p_l : i_1, \ldots, p_l : i_k$, either p_r projects equal arguments of p_l in positions $i_1, \ldots, i_k$, or p_r has some ndv V at the positions $p_r : i_1, \ldots, p_r : i_k$; whenever $p_l : i$ carries either a dv or an sndv (which is not a usndv), p_r saves that argument; whenever $p_l : i$ carries a usndv, either $p_l : i = p_r : i$ or $p_r : i$ saves that argument.*

3. p_l and p_r satisfy the following conditions:

(a) p_r saves every dv occurring in p_l, and p_l copies and saves every dv occurring in p_r.

(b) p_l preserves every EDB *subgoal related to p_l, and*

(c) p_l and p_r have matching ndv-patterns.

Remark: Notice that the syntactic conditions in Theorem 5.3.1 are identical to those in Theorem 4.3.1 except for a minor modification to the conditions associated with the Partial switch case. The following example demonstrates the need for this modification.

Example 5.3.3 *Consider a bilinear program Σ and the FD $F = \{a : 1 \to 2\}$.*

$r_0 : p(X_1, X_2, X_3)$:- $e(X_1, X_2, X_3)$, *and*
$r_1 : p(X_1, X_2, X_3)$:- $p(X_1, U, X_2)$, $p(X_1, U, X_3)$, $a(X_1, U)$.

It can be verified that in the presence of the FD F $T_{left}{}^c \subseteq T_1$. The c.m. (which is the identity mapping !) corresponding to this containment falls in the Partial switch case. Notice that the usndv U in $p_l : 2$ is not saved by p_r. Instead, U also occurs in $p_r : 2$. □

Proof of Theorem 5.3.1. Consider first the case where no pair of distinct dv's gets equated during the chase of T_{left}. In this case, the only changes to the characterization of containment are (i) the way the notion of relatedness is defined and (ii) the modifications to the conditions corresponding to the partial switch case.. We can justify these changes as follows. (i) Suppose s, t are any subgoals of the sirup. Suppose the only ndv's shared by them are unchanging. Then in any c.m. from T_1 to T_{left}, s can map to s or s_f independently of whether t maps to t or t_f. More precisely, mapping s to s (or s_f) will not exert any constraint on where t must be mapped, and vice versa. (ii) It is straightforward to see that every c.m. must be an identity on usndv's. For a c.m. corresponding to the partial switch case, a usndv U occuring in $p_l : i$ will also occur in $p_r : i$ iff the condition 2 in the theorem is satisfied. Also notice that the only change needed to account for the effect of usndv's on c.m.'s associated with the Normal cases and the Total switch case is completely captured by the notion of relatedness, given in Definition 5.3.2. From these observations, it is straightforward to show that $T_{left} \subseteq_F T_1$ iff $T_{left}{}^c \subseteq T_1$ (by Theorem 5.1.1) iff Π' (which is Π in this case) satisfies one of the syntactic conditions (1) - (3) in Theorem 5.3.1.

Now, suppose the dv's X_i and X_j get equated during the chase of T_{left}. By Proposition 5.3.1, $T_{left}' \subseteq T_1$ iff $T_{left}' \subseteq \sigma_{X_i=X_j}(T_1)$ iff T_1 generated by Π' is contained in T_{left}' (also generated by Π' and chased w.r.t. F). The replacement of X_j by X_i will affect the conditions of whether (i) p_l is dv-saving, (ii) p_l saves the dv's in p_r, (iii) p_r copies and saves the dv's in p_r, and (iv) p_l preserves the subgoals related to it. Notice that in Π', p_r (p_l) saves an occurrence of X_j iff it saves an(y) occurrence of the dv X_i. From this it follows that the proof-tree T_{left}' corresponding to Π' (or equivalently to Π) is contained in the proof-tree T_1 corresponding to Π' iff Π' satisfies one of the syntactic conditions in Theorem 5.3.1. □

Finally, we will complete the example 5.3.2 by showing containment $r_2(\sigma_{X_3=X_2}(T_{left}')) \subseteq r_2(T_1)$ for the sirup Q.

Example 5.3.4 *Continuation of Example 5.3.2*

To show containment in presence of equated dv's.
Recall the proof-trees T_1 and T_{left}' generated by the sirup Q.

$r_2(T_1) : p(X_1, X_2, X_3)$:- $p(X_1, X_3, V)$, $p(X_1, X_2, V)$, $a(X_1, X_2, V)$.
$r_2(T_{left}') : p(X_1, X_2, X_2)$:- $p_{ll}(X_1, V, V_1)$, $p_{lt}(X_1, X_2, V_1)$, $a_l(X_1, X_2, V_1)$,
$p_r(X_1, X_2, V)$, $a(X_1, X_2, V)$.

Using Proposition 5.3.1, we show that T_{left}' is contained in T_1 provided the containment mapping $m_t : T_1 \rightarrow T_{left}'$ sends X_3 to X_2, V to V_1 and the dv's X_1, X_2 to identity.

The containment $T_{left}' \subseteq T_1$ can also be verified using the syntactic conditions of Theorem 5.3.1. The result shows that the sirup Q satisfies the conditions for Partial switch case from T_{left} to T_1. □

5.4 Testing 1-Boundedness

In this section, we shall provide an algorithm for testing 1-boundedness of bilinear sirups. The algorithm is general enough to be applicable to the case where the EDB input to the program is known to satisfy a set of functional dependencies. The algorithm makes use of the characterization of Theorem 5.3.1 and 5.1.1. It should be remarked the conditions in these theorems are always sufficient for bilinear sirups to be 1-bounded. They are both necessary and sufficient when the sirup has no repeated EDB subgoals. We shall show that

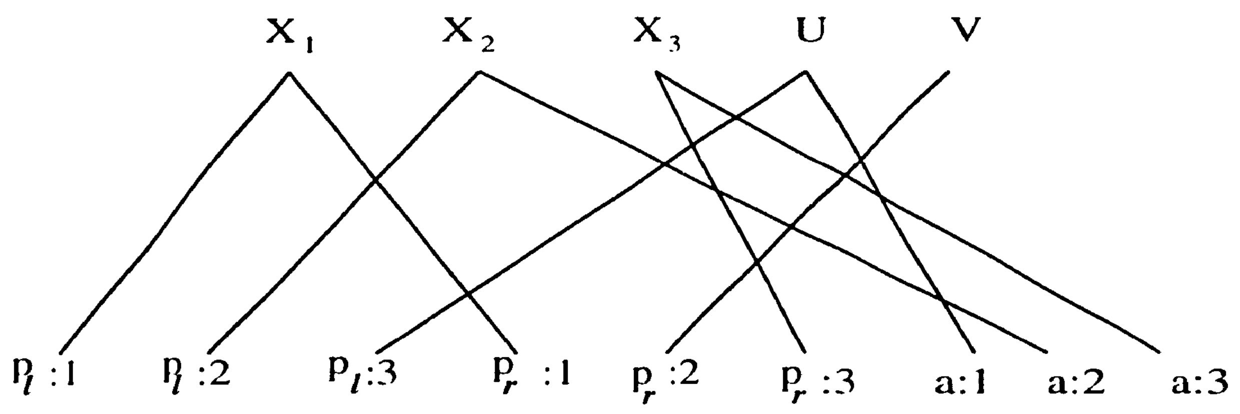


Figure 8: AV-graph representation for the bilinear sirup e

the algorithm takes time linear in the size of description of the program. Our result is in some sense optimal because when the EDB subgoals are allowed to repeat the complexity becomes NP-hard even for linear sirups [11].

The remainder of the section is organized as follows. The *AV-graph* representation of the sirup is described in Section 5.4.1. Some basic notations used in the algorithm are also introduced in this section. Section 5.4.2 describes the linear time algorithm for the detection of 1-boundedness.

5.4.1 AV-graph Representation of the sirups

Our algorithm makes use of a representation of the sirup in the form of an *argument / variable graph* (*AV-graph*). The notion of AV-graph used here is a minor variant of the one introduced by Naughton [20].

Construction of an AV-graph for bilinear sirup : An *AV-graph* of a sirup r is a (undirected) graph $G(r)$ whose nodes are the arguments (of predicates) and variables appearing in the body of r. $G(r)$ contains an edge $(s : i, Z)$. For an argument $s : i$ and a variable Z just in case Z appears in the argument $s : i$ is r's body.

It follows from the construction of the AV-graph that it takes no more than linear time in the size of the sirup.

Example 5.4.1 *Consider the following sirup e, consisting of the rules*

$r_0 : p(X_1, X_2, X_3)$:- $e(X_1, X_2, X_3)$. *and*

$r_1 : p(X_1, X_2, X_3)$:- $p(X_1, X_2, U)$, $p(X_1, V, X_3)$, $a(U, X_2, X_3)$.

The AV-graph representation of e is shown by Fig. 8.

We will introduce some simple notations which will be used in the algorithm. We denote by $sndvs(s)$ the set of sndv's (which are not usndv's) appearing in the subgoal s. The set containing all the sndvs present in the sirup is labelled by $sndv-set$. We represent the set of subgoals related to p_i as $rel(p_i)$.

5.4.2 Linear time Algorithm for Testing 1-boundedness

In this section we shall give an algorithm which will test if a given nonlinear sirup is 1 bounded in time *linear* in the size of the sirup. This is quite significant since in general testing 1-boundedness is NP-hard even for linear sirups. To our knowledge ours is the first positive (*i.e.*linear time) result on 1-boundedness of nonlinear sirups.

Before presenting the algorithm, a note concerning its use. If a bilinear sirup is to be tested for 1-boundedness, then Algorithm 5.4.1 can be applied directly. In this case, the set of usndv's (considered throughout the algorithm) is taken to be empty. If the objective is to test 1-boundedness relative to a set of FDs on the EDB predicates of the sirup, then we first chase the sirup as well as the proof trees T_{left} and T_{right} using Algorithm 5.2. Notice that Algorithm 5.2 identifies the usndv's in the sirup body w.r.t. each of the proof trees T_{left} and T_{right}.

We next present the algorithm in schematic form.

Algorithm 5.4.1 (Testing bilinear sirups for 1-boundedness)

Input: *A bilinear sirup* **r** *of the form*

$p(X_1, \ldots, X_n)$:- $a_1(U_1, \ldots, U_m)$, $\ldots$, $a_k(V_1, \ldots, V_l)$, $p(Y_1, \ldots, Y_n)$, $p(W_1, \ldots, W_n)$ *together with an exit rule* $p(X_1, \ldots, X_n)$:- $e(X_1, \ldots, X_n)$.

The sirup r is represented in the form of an AV-graph $G(r)$.

Output: *A decision as to whether the above program is 1-bounded.*

Method: *Test each of the conditions in the Theorem 5.3.1 w.r.t. the containment $T_{left} \subseteq T_1$. Similarly test the conditions corresponding to $T_{right} \subseteq T_1$. The program is 1-bounded iff both tests succeed.*

```
begin

"First test if $T_{left} \subseteq T_1$" :

Step (0) Initialize the flags: normal := true; partial_switch := true; total_switch := true;
    /* These flags keep track of which of the conditions in Theorem 5.3.1 if any, apply
    for the containment $T_{left} \subseteq T_1$. They will be updated and inspected eventually. */

Step (1)
        1.1) Identify dndv's and sndv's
                for each ndv V do
                    if (V,s:i),(V,t:j) $\in$ G(r) and $s \neq t$
                        /* i.e V appears in two different subgoal arguments */
                            then V is a sndv
                            else it is a dndv.
                end {for}
                Exclude from the sndv-set any ndv classified as usndv by Algorithm 5.2
                (if appplicable).
        1.2) Identify subgoals related to $p_\ell$ and $p_r$ :
                Initialize rel($p_\ell$) to be empty;
                Initialize S to sndvs($p_\ell$);
                Initially all nodes and edges of G(r) are unmarked;
                repeat
                    while $S \neq \phi$ do
                            pick any sndv V in S;
                            for each unmarked edge (V,s:i) $\in$ G(r) do
                              mark this edge;
                              rel($p_\ell$) := rel($p_\ell$) $\cup$ {s};
                              S  :=  S$\cup$ sndv's(s)
                            end {for};
                            delete V from S;
                    end {while};
                until (no change to rel($p_\ell$) or ( S = $\phi$);
        Similarly, compute rel($p_r$), the set of subgoals related to $p_r$;
```

Step (2) Test the conditions for the Normal Case (i) and (ii):

```
for each $dv$ $X_i$ do
    for each edge $(X_i, s{:}j) \in G(r)$ do
        if $s = p_i$ or $s \in rel(p_i)$ then
            if $p_i : i \neq X_i$ then
                normal := false;
    end {for}
end {for}
if normal then goto Step(6);
```

Step (3) Test the conditions for the Partial Switch Case:

```
for each argument $p_i : i$ of $p_f$ do
    Let $p_i : i$ carry the variable $Z$;
    if $Z$ is a dv or sndv then
        begin
            if $p_i : i \neq$ a dv then
                partial_switch := false;
            else if $p_r : i = X_k$ and $p_f : k \neq Z$ then
                partial_switch := false;
        end
    else if $Z$ is a usndv then
        begin
            if $p_r : i \neq Z$ then
                if $p_r : i \neq$ a dv then
                    partial_switch := false;
                else if $p_i : i = X_k$ and $p_i : k \neq Z$ then
                    partial_switch := false;
        end
    else /* in this case, Z is a dndv */
        begin
            Let $S_f = \{p_f : i_1, \ldots, p_i : i_k\}$ be the set of all arguments
            of $p_f$ where $Z$ appears;
            Test if either
              (a) an identical ndv, say $W$ appears in all corresponding
                  arguments of $p_r$ or
                (b) for each $i_j \in \{i_1, \ldots, i_k\}$,
                $p_r : i_j$ carries a dv $X_\ell$, where $\ell \in \{i_1, \ldots, i_k\}$.
              if neither is true then partial_switch := false;
            end
      end {for}
      if partial_switch then goto Step(6)
```

Step (4) **Test the conditions for the Total Switch Case:**

```
      4.1) Test if $p_r$ saves the dv's in $p_\ell$ :
              for each dv $X_i$ do
                  if $\exists$ an edge of the form $(X_i, p_\ell : j) \in G(r)$ then
                     if $p_r : j \neq$ a dv then
                        total_switch := false;
                     else if $p_r : j = X_k$ and $p_\ell : k \neq X_i$ then
                        total_switch := false;
              end {for}
      4.2) Test if $p_\ell$ copies and saves every dv in $p_r$ :
              for each dv $X_i$ do
                  if $\exists$ an edge of the form $(X_i, p_r : j) \in G(r)$ then
                     if [$\forall$ edge of the form $(X_i, p_\ell : j) \in G(r)$ ; $p_\ell : j \neq X_k$] then
                        total_switch := false;
              end for
      4.3) Test whether $p_\ell$ preserves every subgoal related to it;
              Similar to Step (2).
      4.4) Test if $p_\ell$ and $p_r$ have matching ndv-patterns:
              (a) Test if $p_r$ matches $p_\ell$'s dv-patterns:
                  for each ndv $U$ appearing in $p_\ell : i$ do
                     if $U$ is a dndv then
                        begin
                           Let $V$ be the variable occuring in $p_r : i$;
                           if $V$ is not a dndv then
                              total_switch := false;
                           for each edge $(U, p_\ell : k) \in G(r)$ do
                              if $(V, p_r : k) \notin G(r)$ then
                                 total_switch := false;
```

```
            end {for}
          end
        else if $U$ is a sndv then
          if $U$ appears in an EDB subgoal then
            begin
              for each edge $(U,p_r:k) \in G(r)$ do
                  if $p_r : k \neq U$ then
                    total_switch := false;
              end {for}
            end
          else /* $U$ is a sndv not appearing in an EDB subgoal */
            begin
              if $p_r : i \neq$ a sndv then
                  total_switch := false;
              else
                  begin
                    Let $p_r : i = V$;
                    if $V$ appears in EDB then
                      total_switch := false;
                    else
                      begin
                          for each edge $(U, p_r:k) \in G(r)$ do
                              if $p_r : k \neq V$ then
                                total_switch := false;
                          end {for}
                          for each edge $(V,p_r:k) \in G(r)$ do
                              if $p_r : k \neq U$ then
                                total_switch := false;
                          end {for}
                      end
            end
      (b) Similarly, Test if $p_f$ matches $p_r$'s matching ndv-patterns:
```

Step (5) **if** *normal, partial_switch, or total_switch* is **true** *then conclude* $T_{left} \subseteq T_1$ **else return** *("program is not 1-bounded").*

Step (6) *Perform the tests corresponding to the containment $T_{right} \subseteq T_1$: if both containments hold, conclude the program is 1-bounded.*

end.

As seen in Algorithm 5.4.1, we have adapted the conditions given in Theorem 5.3.1 into an elegant algorithm. The AV-graph representation of the sirup used in the algorithm makes a useful contribution to the actual efficiency of the computation. Since our characterization is based on identifying the syntactic properties of the sirup, the data structure provided by the AV-graph makes the traversal of node and arguments quite efficient.

NOTES: After initializing the 3 flags corresponding to Normal case, Partial case, and Total case in Step (0), we construct the set $sndv-set$ to identify the sndvs present in the sirup. Those ndvs which are not present in $sndv-set$ are dndvs. Remark that if FDs are given then the modified definition of related (as given in Definition 5.3.2) should be used. The purpose of the exercise to construct the $sndv-set$ is to identify which are the sndvs which contribute to relatedness to p_r in Step 1.2. Before, the syntactic conditions corresponding to the 3 cases can be applied directly, we need to identify the subgoals which are related to p_r. Step 1.2 performs this computation. Using the constructed set $sndv-set(p_r)$, it traverses the edges corresponding to each sndv present in $sndvs(p_r)$ and at the same time adds the subgoals which are visited during the traversal to $rel(p_r)$ to represent those subgoals which are related to p_r.

Step (2) tests for the condition for Normal case. For each dv X_i present in p_ℓ or in the subgoals related to p_r, it is checked whether p_ℓ saves that X_i or not. In this way the condition for dv-saving and subgoal preservation is verified.

Step(3) For each argument position i in p_r, if $p_\ell : i$ is a dv or a sndv, then the condition for partial switch case in the algorithm verifies that $p_r : i$ saves that argument. As given in Thereom 5.3.1, the check for dndvs is also performed.

Step(4) verifies various conditions with respect to Total Switch case.

Step (5) Checks the value of the flag whether one of the three containments holds or not i.e $T_{left} \subseteq T_1$. The algorithm is aborted in case the containment $T_{left} \subseteq T_1$ fails the tests.

Step (6) is performed when the containment for $T_{left} \subseteq T_1$ has tested positive from previous step. In Step (6) thes test for Normal, Partial and Total Switch case is done for the direction $T_{right} \subseteq T_1$.

Conclusion: The sirup is declared 1-bounded iff both containments hold.

We next have

Theorem 5.4.1 *Algorithm 5.4.1 correctly determines if a bilinear sirup is 1-bounded. It runs in time linear in the size of the sirup.*

Proof. The correctness is a direct consequence of Theorems 5.3.1 and 5.1.1. We shall show that the time complexity is proportional to the number of edges in the AV graph of the sirup. Since this latter number is linear in the sirup size, the result will follow. Step 0 is trivial. Step 1.1 takes time proportional to the sum of degrees of the ndv nodes of $G(r)$. This in turn is upper bounded by the number of edges in $G(r)$. Step 1.2 takes time proportional to the sum of degrees of the sndv nodes in $G(r)$. Step 2 takes no more time than the sum of degrees of the dv's of the sirup r. The time for Step 3 is the number of arguments of p_ℓ times the time spent for each argument. If the argument carries a dv or an sndv, the time spent is constant. If it is a dndv, the time spent is proportional to the degree of this node in $G(r)$. Thus, the overall time for Step 3 is upper bounded by the $max\{n, \Sigma_{V \text{ is a } dndv} deg(V)\}$, where n is the arity of p. Steps 4.1 and 4.2 take time no more than proportional to the sum of degrees of the dv nodes in $G(r)$. Step 4.3 is very similar to Step 2 and has an identical upper bound on the time. For Step 4.4, each of (a) and (b) take time at most proportional to the sum of degrees of nodes corresponding to the ndv's appearing in p_ℓ and p_r. The remaining steps are trivial. Step 6 corresponds to a similar test for the other direction of containment. Since no step of the algorithm takes more than $O(n)$ time, Algorithm 5.4.1 has a time complexity $O(n)$. □

Chapter 6

Stage Preserving Linearizability

The issue of linearization has been studied earlier in the literature. However, the focus has been to linearize programs without much concern with the evaluation efficiency of the resulting linear program. In this chapter, we focus our attention to the study of a linearization technique which is based on the idea that the resulting linear programs should not increase the amount of work done to evaluate the programs.

We study the problem of Stage Preserving Linearization (sp-linearization) for a class of bilinear sirups with multiple (but distint) EDBs. The linearization technique we propose in this chapter is based on proof-tree transformations technique. We identify 6 different kinds of proof-tree transformations which lead to sp-linearized programs. Using these transformations we are able to identify several unique class of programs as linearizable. Such equivalent linear programs cannot be obtained using previously known techniques.

The organization of this chapter is as follows. In Section 6.1 we will formally introduce the notion of Stage Preserving Linearization. The associated proof-tree transformations will be presented in Section 6.2. In Section 6.3 we include an informal comparison between sp-linearization with ZYT-linearization. This section will also be supplemented with arguments supporting the conjecture that the work done per iteration is much less with sp-linearized programs, than those obtained via other linearization techniques. We will conclude with a short summary of this chapter.

<u>Remark</u>: With respect to the study of Stage preserving Linearization we have developed syntactic conditions (corresponding to the different kinds of proof transformations) and also

a polynomial time algorithm. These contributions are beyond the scope of this thesis. The complete results regarding the Stage preserving Linearizability appear in [17], [15].

6.1 Introduction

Before we can formally define Stage Preserving Linearization, we need to introduce the following terms. Recall the least fixpoint semantics introduced in Chapter 1.

Definition 6.1.1 *The stage of a tuple in a bottom-up fixpoint evaluation is the minimum number of iterations needed to produce that tuple. By the* closure ordinal *of a program* Π *w.r.t.* $\mathcal{D}$ *we mean the smallest* m *such that* $\Pi^m(\mathcal{D}) = \Pi^\omega(\mathcal{D})$.

<u>Remark</u>: In this paper, we will consider semi-naive evaluation as our basic model of evaluation for defining the stage.

Informally, *Stage Preserving Linearization* (sp-linearization) means that the nonlinear program can be replaced by an equivalent linear program without increase the stage of each derived tupl[illegible]. to a bottom-up fixpoint evaluation.

The formal definition of Stage Preserving Linearization follows.

Definition 6.1.2 *Let* Π *be any program and* Σ *be any equivalent linear program obtained from* Π *as a result of any proof-tree transformation. Then we say that this transformation* preserves stages *provided for every derived tuple* $p(t)$, *its stage w.r.t.* Σ *and* $\mathcal{D}$ *is no more than its stage w.r.t.* Π *and* $\mathcal{D}$. *In particular, when a linearization preserves stages, we say that it is* stage preserving *and that the original non-linear program is* stage preserving linearizable *(sp-linearizable).*

Notice that the definition of sp-linearizability does not impose any restrictions on the form of the equivalent program Σ. In particular it does not say anything about the number of rules in Σ. We will exploit this point in the next section to demonstrate, that the original nonlinear (sp-linearizable) could be equivalent to *one* of the *four* different kinds of sp-linear programs, depending on the particular type(s) of proof-tree transformation(s) involved w.r.t that nonlinear program.

Example 6.1.1 *Consider the following nonlinear sirup $\mathcal{R}$.*

r_0: $p(X_1, X_2, X_3)$:- $e(X_1, X_2, X_3)$.
r_1: $p(X_1, X_2, X_3)$:- $a(U, X_2), b(X_1, X_2), c(W, X_3), p(X_1, V, W), p(X_1, U, X_3)$.

This program is not equivalent to any linear program obtainable using the well-known ZYT-linearization technique, e.g., the programs $\{r_0, r_1$-left$\}$ and r_0, r_1-right$\}$ corresponding to program $\mathcal{R}$. However, it can be shown using the results of this paper that this program is sp-linearizable and is equivalent to the following linear program $\mathcal{S}$.

r_0: $p(X_1, X_2, X_3)$:- $e(X_1, X_2, X_3)$.
r_1-*left*: $p(X_1, X_2, X_3)$:- $a(U, X_2), b(X_1, X_2), c(W, X_3), e(X_1, V, W), p(X_1, U, X_3)$.
r_1-*right*: $p(X_1, X_2, X_3)$:- $a(U, X_2), b(X_1, X_2), c(W, X_3), p(X_1, V, W), e(X_1, U, X_3)$.

Four different kinds of Sp-linear Programs

Let us consider the four types of linear programs which can be derived from the original program $\mathcal{P}$. Recall that sp-linearizability of $\mathcal{P}$ involves being (stage-)equivalent to *not* one fixed type of linear program but to one of a number of possible linear programs, depending on the conditions satisfied by $\mathcal{P}$.

Consider a bilinear sirup $\mathcal{P}$, given in schematic form:

$\mathcal{P} = \{r0: \quad p \quad :- \quad e;$
$r_1: \quad p \quad :- \quad pl, pr, a1, \ldots, ak\}.$

We first identify these linear programs and state when $\mathcal{P}$ can be expected to be equivalent to each linear program.
(1) $Q_{right} = \{r0: \quad p \quad :- \quad e;$
$r1_{right}: \quad p \quad :- \quad e_l, pr, a1, \ldots, ak\}.$

(2) $Q_{left} = \{r0: \quad p \quad :- \quad e;$
$r1_{left}: \quad p \quad :- \quad pl, e_r, a1, \ldots, ak\}.$

(3) $Q_{mixed} = \{r0: \quad p \quad :- \quad e;$
$r1_{right}: \quad p \quad :- \quad e_l, pr, a1, \ldots, ak;$
$r1_{left}: \quad p \quad :- \quad pl, e_r, a1, \ldots, ak\}.$

(4) $Q_{squash} = \{r0: \ p :- \ e;$
$r1_{right}: \ p \ :- \ e_l, p_r, a1, \dots, ak;$
$r1_{left}: \ p \ :- \ p_l, e_r, a1, \dots, ak;$
$r1_{squash}: \ p :- \ p*, a1, \dots, ak\}$, where $p*$ is obtained from p_l and p_r.

It has been shown in [17], [15] that a nonlinear program P is sp-linearizable iff it is equivalent to one of the four types of linear programs listed above.

We say that P is *right-linearizable* if it is equivalant to the linear program Q_{right}. Similarly we say P is *left-linearizable* if it is linearizable using the program Q_{left}. Certain kinds of sp-linear programs contain both the left-linear as well as right-linear rules. Such types of linear programs are called mixed-linear. The program Q_{mixed} is an example of such a linear program. In this case, we say that P is *mixed-linearizable*. Finally, if P is equivalent to Q_{squash}, then we say P is *squash-linearizable*.

An example of Squash-linearizable program is given next.

Example 6.1.2 *(Squash-linearizable Program)*

r0: c_r(P, C, D) :- com(P, C, D).
r1: c_r(P, C, D) :- c_r(P, E, F), pre(E, C), c_r(P, G, H), pre(H, D).

Here the (exit) predicate com(P, C, D) says that P is an immediate prerequisite to both C and D; the predicate pre(E, C) states that E is an immediate prerequisite to C; finally, the recursive predicate c_r(P, C, D) asserts that P is a common prerequisite to the courses C and D. This program is a non-linear (bilinear) strap. It can be shown using the technique developed in this paper that this program is equivalent to the following linear program Q.

r0: c_r(P, C, D) :- com(P, C, D).
$r1_{right}$: c_r(P, C, D) :- com(P, E, F), pre(E, C), c_r(P, G, H), pre(H, D).
$r1_{left}$: c_r(P, C, D) :- c_r(P, E, F), pre(E, C), com(P, G, H), pre(H, D).
$r1_{squash}$: c_r(P, C, D) :- c_r(P, E, H), pre(E, C), pre(H, D). □

Notice in particular the rule $r1_{squash}$ and the recursive predicate *c_r(P, E, H)*. We say that *c_r (P, E, H)* is obtained by "squashing" together *c_r(P, E, F)* and *c_r(P, G, H)* in the body of the original rule r_1.

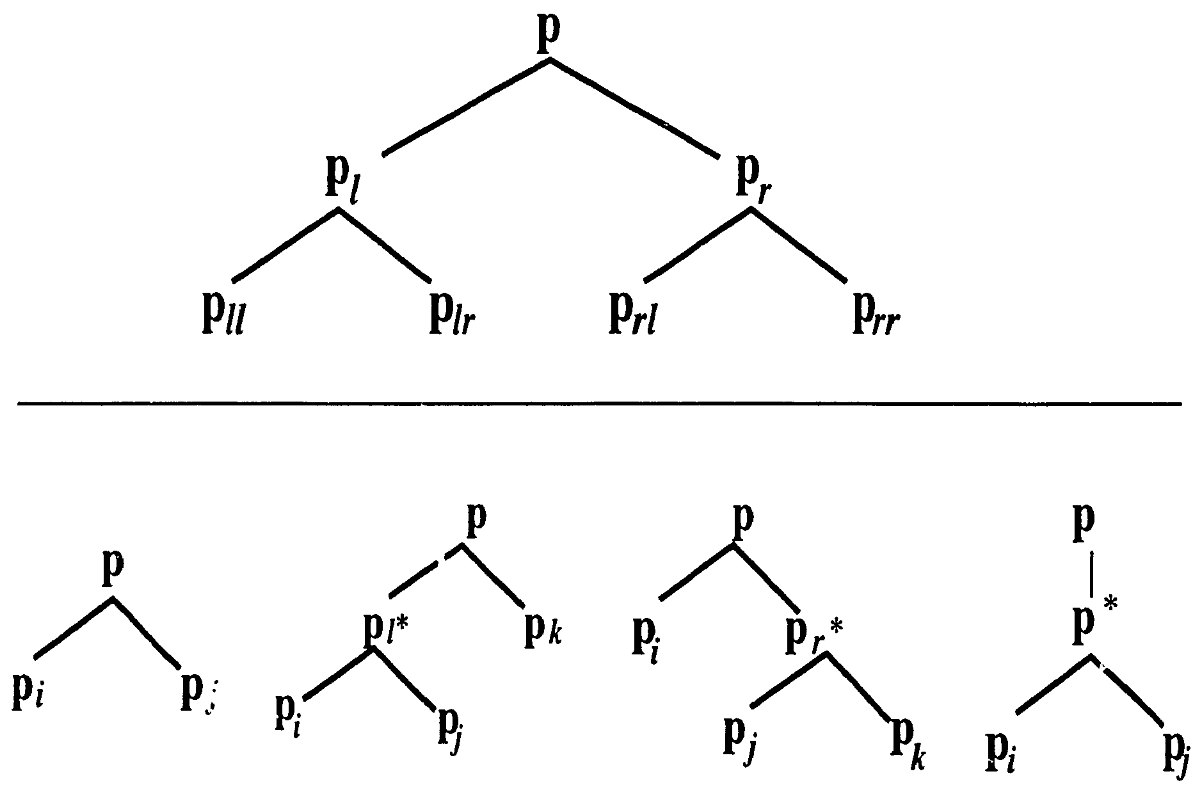


Figure 9: (a) The bad tree T_2 (b) Shows good trees

The nonlinear program given in Example 6.1.2 is squash-linearizable whereas the program given in Example 6.1.1 is mixed-linearizable.

6.2 Proof Tree Transformations associated with Sp-linearizability

In this section we will identify the different types of proof-tree transformations which lead to sp-linearzable programs. Proof tree transformations play a central role in Stage Preserving Linearization. In order to linearize an arbitrary nonlinear tree T_{non}, a transformation rule T—T' must satisfy the following two conditions, as shown in [21]: (i) $T \subseteq T'$ (ii) it should be possible to transform an arbitrary tree T_{non} into a linear tree by finitely many applications of the rule T—T'. Condition (i) makes sure that there is a valid mapping from T' to T, with the result that the set of tuples which could be derived using T could equivalently be derived using T'. Condition (ii) checks the finiteness of the tree transformation procedure i.e it imposes a condition to discourage transformation rules from reproducing previously obtained trees again.

Recall from Chapter 3 (Section 3.5) various proof-tree transformations and the notion of good and bad trees. In this section, we will identify the set of good trees which lead to a stage preserving transformation. Fig. 9 shows the good and the bad trees w.r.t to Sp linearization. Notice that we consider the tree T_2 as a bad tree. It represents non linearity since both the recursive predicates have been expanded further using the recursive rule. The criteria to find "good tree" are quite simple. "Good tree" represent the set of all those trees which (i) are linear (ii) are "stage preserving" with respect to the original nonlinear tree T_2.

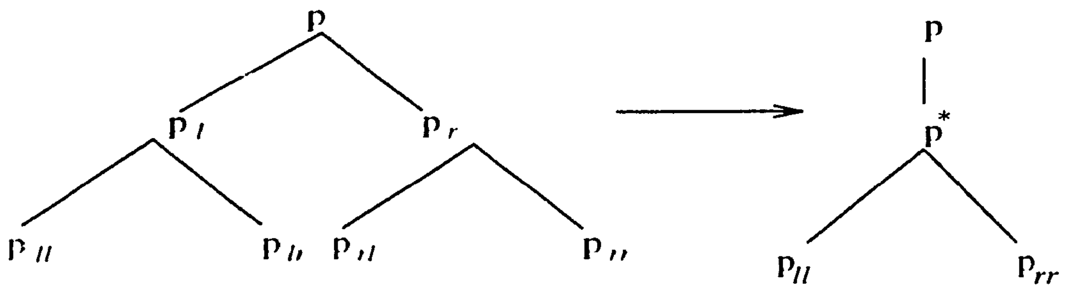
The set of linear trees which could be obtained from T_2 could be either a one level tree or it may have 2-levels. Within the one-level trees the only possibility is a tree of the form T_1. A tree of the form T_1 trivially satisfies the above 2 conditions. With respect to linear two-level trees a "good tree" will represent a tree with maximum one recursive predicate expanded. The different types of proof-trees which falls under this category are : (1) The right linear tree T_{right}. (2) The left linear tree T_{left} (3) Squash-linear tree. It can be verified that each of these trees is linear in nature; and they are "stage preserving" since the height of these trees is equal or less than that of T_2.

Fig. 10 is intended to serve as representative example of proof-tree transformation w.r.t sp-linearization.

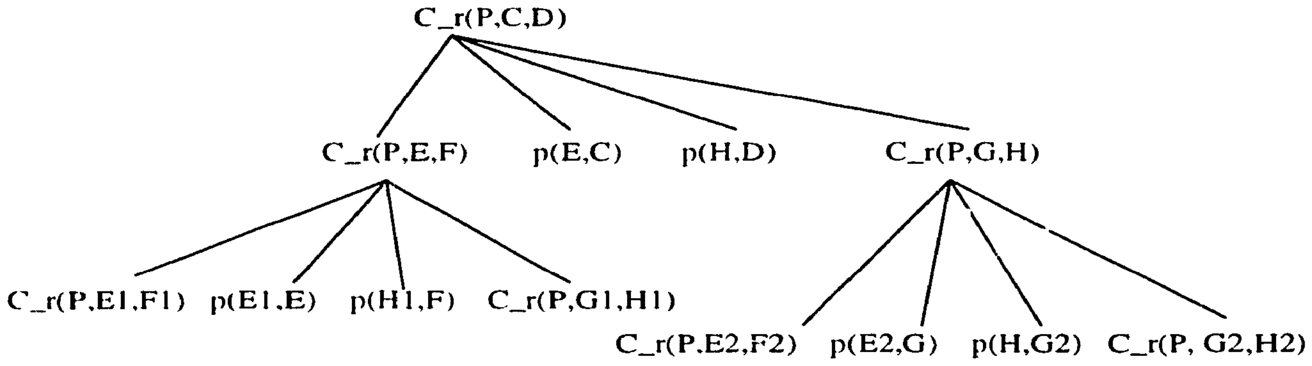
6.3 Comparison

In this section we shall do an informal comparison between Stage preserving Linearization and the well known linearization technique, ZYT linearization.

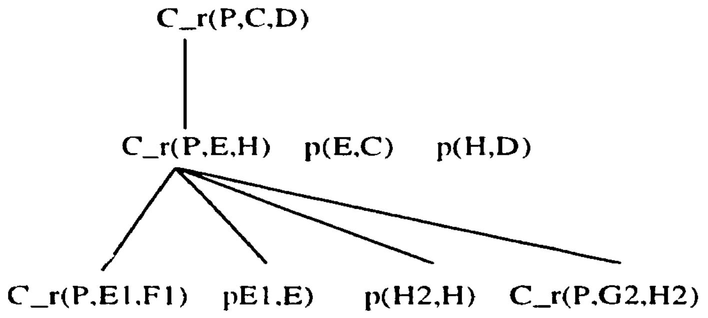
Recall from Chapter 1, Section 1, that ZYT linearization is a specific way of linearizing nonlinear recursive programs, by replacing exactly one of the recursive subgoals of the sirup with an exit predicate. The class of nonlinear programs for which the ZYT technique is applicable consists of bilinear sirups with single EDB, although this was subsequently extended to multiple EDB subgoals. However, in our case we consider a larger class of programs, consisting of bilinear sirups with multiple (but distinct) EDBs. ZYT-linearization technique uses *only* three different kinds of proof-tree transformations to linearize programs. In comparison, the scope of our result is much broader in the sense that we consider 4 different types



(a)



(b)



(c)

Figure 10: (a) Squash Transformation. (b),(c) T_2 and T_{squash} w.r.t to Example 6.1.2

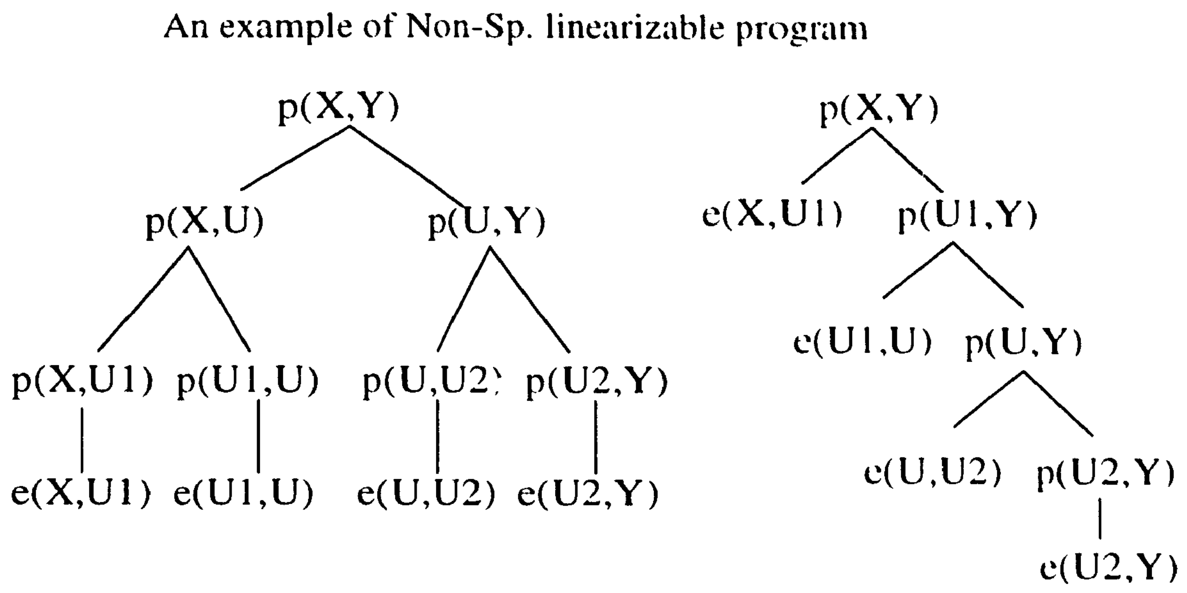


Figure 11: Showing the tree transformation w.r.t Transitive closure program

of proof-tree transformations. Corresponding to these tranformations 4 different kinds of linear programs can be obtained.

The study of stage preserving linearizability has been a great challenge and has led to the discovery of programs which cannot be identified using previously known techniques of linearization (*e.g.*mixed-linear and squash linear program..).

Finally, for every nonlinear sirup Π for which an equivalent sp-linear program Σ is obtained, it can be shown that for an arbitrary input database (edb) D, the *stage* of each derived tuple when the program Π_1 is evaluated on $\mathcal{D}$ is no more than its stage when Π is evaluated on $\mathcal{D}$. We refer to this aspect by saying that the stage is preserved by this linearization technique. In contrast, a technique such as ZYT-linearization does not preserve the stage in general. *E.g.*, when the bilinear transitive closure program $\{r_0 : t(X_1, X_2) :- e(X_1, X_2)$; $r_1 : t(X_1, X_2) :- t(X_1, Z), t(Z, X_2)\}$ is transformed into its linear equivalent $\{r_0 : t(X_1, X_2)$ $:- e(X_1, X_2)$; r_1-left: $t(X_1, X_2) :- e(X_1, Z), t(Z, X_2)\}$, this transformation does not preserve the stage of every derived tuple for t. Indeed, the stage of a derived tuple based on (semi naive evaluation of) the linear program for transitive closure can be as long as the length of the longest path whereas for the non-linear version, the fixpoint will be reached in a logarithm of this length. Thus, measured from the stage of a tuple derived using the non linear program, there can be an exponential increase in the stage of that tuple when it is derived using the linear program for transitive closure.

Example 6.3.1 *The well-known transitive closure program falls under the category of ZYT-linearized programs. For a tree of height 2, we illustrate using Fig. 11 that the resulting transformed tree has a greater height.* □

We shall now give an informal argument to show that each iteration based on the sp-linearized program Π_{splin} involves less work than a similar iteration based on a original nonlinear program Π. More specifically, we will show that the number of iterations needed to reach a fixpoint using Π_{splin} is less than or equal to the number of iterations needed using Π (See footnote [1]) For clarity, we shall present the argument using the example programs $\mathcal{P}$ and $\mathcal{Q}$. The ideas can be generalized to an arbitrary non-linear sirup which is linearized using our technique. Using appropriate abbreviations and using a generic notation for joins and projections, we can express the computation inside the loop of a bottom-up iteration based on $\mathcal{P}$ and $\mathcal{Q}$ as follows. We shall assume a semi-naive evaluation model for both, for convenience. (Recall the processing involved in semi-naive evaluation: Starting with a relation obtained using the exit rule; the recursive rule is fired repeatedly, new tuples generated are added to the result. This process continues until no new tuples can be obtained.)

In the expresion below, Step(1) represents the processing involved to derive new tuples, whereas Step(2) keeps the updated result of all the tuples generated so far.

Bottom-up iteration based on $\mathcal{P}$:

(1) $\Delta C\_R := \pi[(C\_R \bowtie P \bowtie \Delta C\_R \bowtie P) \cup (\Delta C\_R \bowtie P \bowtie C\_R \bowtie P)] - C\_R;$
(2) $C\_R := C\_R \cup \Delta C\_R;$

Here, we understand the arguments of projection and join implicitly (as suggested by the rules in the corresponding program) since they are inessential to our analysis. We can rewrite the above computation into the following equivalent one:

$\Delta C R := \pi[(C \bowtie P \bowtie \Delta C\_R \bowtie P) \cup ((C\_R - C) \bowtie P \bowtie \Delta C\_R \bowtie P) \cup$
$(\Delta C\_R \bowtie P \bowtie C \bowtie P) \cup (\Delta C\_R \bowtie P \bowtie (C\_R - C) \bowtie P)] - C\_R;$
$C\_R := C\_R \cup \Delta C\_R;$

We will refer to the four components in the expression for ΔC_R above as $C_1, \ldots, C_4$ respectively. Now, the computation in the bottom-up iteration loop based on $\mathcal{Q}$ can be

[1]Formally, this means that the closure ordinal of $\Pi_{splin} \leq$ the closure ordinal of Π, w.r.t. every input database $\mathcal{D}$

expressed as follows:

$\Delta CR := \pi[(C \bowtie P \bowtie \Delta C\_R \bowtie PR) \cup (\Delta C\_R \bowtie PR \bowtie C \bowtie PR)] \cup (\Delta C\_R \bowtie PR \bowtie PR)] - C\_R;\ C\_R := C\_R \cup \Delta C\_R;$

We will refer to the three components in the expression for ΔC_R above as D_1, D_2, D_3 respectively. Now, D_1, D_2 are identical to C_1, C_3 respectively. Thus it remains to compare the complexity of D_3 with that of of $C_2 \cup C_4$. Under typical assumptions, it can be shown that the complexity D_3 is much less than that of $C_2 \cup C_4$.

Finally, we will see that every non-linear sirup $\{r_0, r_1\}$ that is linearized by our technique is replaced by a subset of the rules $\{r_0, r_1\text{-left}, r_1\text{-right}, r_1\text{-squash}\}$. Since stage is preserved, the total number of iterations needed by the transformed program to evaluate any query is no more than needed by the original non-linear program. By the argument above, the amount of work done in each iteration based on the linear program is less than the per iteration work done by the non-linear program. Thus, linearization while preserving the stage achieves significant savings in the cost of query evaluation.

Conclusion: It was shown that a linearization that preserves stages achieves a substantial improvement in query processing efficiency without the aid of additional techniques during evaluation. Finally, several techniques developed for mixed linear recursions [22, 13] can be taken advantage of when evaluating specific queries against the transformed linear program.

In conclusion, we remark that ZYT-linearization and Sp-linearization offer complementary (and overlapping) ways of linearing nonlinear sirups, and implementations can take advantage of both.

6.4 Summary

In this section we introduced the fundamental notions regarding Stage Preserving Linearization. The main idea behind the stage preservation was briefed. Relevant proof tree transformations were identified. Using various examples we showed the different kinds of sp-linear programs that could result using our technique. We showed that stage preserving linearization results in significant improvement in the evaluation efficiency. An informal comparision was also done between sp-linearization and ZYT linearization.

Chapter 7

Concluding Remarks and Future Research

Important forms of query optimization in deductive databases are based on recognizing whether the recursion in a query program is essential. If it is not, and if this can be recognized, then the query program can be replaced by an equivalent non-recursive program, which can be subsequently optimized using the powerful optimization techniques developed for relational databases. In this thesis we focused on one such optimization technique, 1-boundedness. In addition, a brief introduction to a novel technique of linearization was also addressed. Our contribution included a syntactic characterization and a linear time algorithm for the recognizing 1-bounded bilinear sirups (with or without functional dependencies) We also identified the relevant proof-tree transformation - for both 1-boundedness and sp-linearizability. We also established the advantages of sp-linearization compared with previously studied linearization techniques such as ZYT-linearization.

7.1 Future Research

There are several open directions to pursue in order to extend the optimization problems we have studied. It would be interesting to investigate the following problems relevant to boundedness in future. (1) FDs constitute only one source of semantic knowledge. How can we take advantage of other integrity constraints and extend the results into a general scheme for semantic query optimization? (2) In the literature, positive results on (k-)boundedness have mostly concentrated on single recursive rule programs. An important question therefore is how this theory can be extended to programs with many recursive rules. (3) What can we say about 2-boundedness (more generally k-boundedness) of bilinear sirups? An interesting start to look into the solution of this problem would to use the insights and knowledge gained from the study of 1-boundedness, to characterize k boundedness for k > 1.

In the direction of extending the work on linearization, further research areas includes the possibility of linearization of higher order recursions, and the linearization of programs with multiple recursive rules. Several interesting problems remain open which are specific to sp-linearization: (1) Can we characterize bilinear recursions which can be linearized using mixed linear rules (without necessarily preserving stages)? (3) Can we account for the effect of semantic knowledge in the form of integrity constraints (*e.g.* functional dependencies) in recognizing sp-linearizability?

It is our hope that the framework of proof tree transformations studied in this thesis will offer a useful tool with which such optimization problems can be attacked.

Bibliography

[1] Abiteboul, Serge and Hull, Richard. Data functions, datalog and negation. *SIGMOD. International Conference on Management of Data*, 17(13):143–153, September 1988.

[2] Aho, A.V., Hopcroft, J.E., and Ullman, J.D. *The design and analysis of computer algorithms*. Reading, Mass.; Don Mills, Ont.: Addison-Wesley Pub. Co., 1974.

[3] Bancilhon, F. and Ramakrishnan, R. An amateur's introduction to recursive query-processing strategies. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 16–52, 1986.

[4] Ceri, S., Gottlob, G., and Tanca, L. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1), March 1989.

[5] Ceri, S., Gottlob, G., and Tanca, L. *Logic programming and Databases*. Berlin, New York: Springer-Verlag, 1990.

[6] Chandra, A.K. and Merlin, P.M. Optimal implementation of conjunctive queries in relational databases. In *Proc. 9th Annual ACM Symp. on the Theory of Computing*, pages 77–90, 1977.

[7] Gaifman, H., Mairson, H., Sagiv, Y., and Vardi, M.Y. Undecidable optimization problems for database logic programs. Technical report, IBM Research Report RJ 5583 (56702), Yorktown Heights, New York., April 3 1987.

[8] Hernandez, Hector, J., Gonzalez, Augustin, and Lakshmanan, Laks V.S. Testing implications of functional dependencies in linear sirups. Technical report, Concordia University, Montreal, Canada, December 1994.

[9] Hillebrand, G.G., Mairson, H.G., and Vardi, M.Y. Tools for datalog boundedness. In *Proc. ACM symp. PODS*, pages 1–12, 1991.

[10] Ioannidis, Y.E. A time bound on the materialization of some recursively defined views. In *Proc. 11th Int. Conf. of Very Large Data Bases*, pages 219 226, 1985.

[11] Kanellakis, P. Logic programming and parallel complexity. In *Foundations of Deductive Databases and Logic Programming*, pages 517 586, 1988. J. Minker ed., Morgan Kaufmann.

[12] Kanellakis, P. and Abiteboul, S. Database theory column: Deciding bounded recursion in database logic programs. *SIGACT News*, 20(1):17 23, 1989.

[13] Kemp, D.B., Ramamohanarao, K., and Somogyi, Z. Right-, left- and multi-linear rule transformations that maintain context information. In *Proc. 16th VLDB Conf.*, pages 235-242, 1989.

[14] Lakshmanan, Laks V.S. and Hernandez, H. Structural query optimization: A uniform framework for semantic query optimization in deductive databases. In *Proc. ACM Symp. PODS*, pages 102 114, Denver, CO, 1991.

[15] Lakshmanan, L.V.S., Ashraf, K., and Han, J. Homomorphic trees embeddings and their applications to recursive program optimization, July 1993. Submitted to a technical journal.

[16] Lakshmanan, L.V.S and Ashraf, Karima. Detecting 1-boundedness of non linear database logic programs in linear time, January 1994. Submitted to a technical journal. Revised in July 1995.

[17] Lakshmanan, L.V.S., Ashraf, Karima, and Han, Jiawei. Homomorphic tree embeddings and their applications to recursive program optimization. In *8th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 344 353, 1993.

[18] Lloyd, J. W. *Foundations of Logic Programming*. Springer Verlag, second edition, 1987.

[19] Maier, D. *The Theory of Relational Databases*. Computer Science Press, Rockville, Maryland, 1983.

[20] Naughton, J.F. Data independent recursion in deductive databases. In *Proc 5th ACM Symp. PODS*, pages 267 279, 1986.

[21] Naughton, J.F. Redundancy in function free recursive inference rules. In *Proc. of the IEEE Symposium on Logic Programming*, 1986.

[22] Naughton, J.F., Ramakrishnan, R., Sagiv, Y., and Ullman, J.D. Efficient evaluation of right-, left-, and multi-linear rules. In *Proc. ACM SIGMOD '89 International Conference on Management of Data*, pages 380–391, 1990.

[23] Premchand Sukumaran, Nair. *Optimization of Logic Queries in Knowledge Base Systems*. PhD thesis, Department of Computer Science, Concordia University, Montreal, Quebec, Canada, 1989.

[24] Ramakrishnan, R., Sagiv, Y., Ullman, J.D., and Vardi, M. Proof-tree transformation theorems and their applications. In *Proc. 8th ACM Symp. PODS*, pages 172–181, 1989.

[25] Sagiv, Y. On optimizing datalog programs. In *Proc. 6th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, 1987.

[26] Saraiya, Y. Linearizing nonlinear recursions in polynomial-time. In *Proc. 8th ACM SIGACT-SIGMOD-SIGACT Symp. of Principles of Database Systems*, pages 182–189, 1989.

[27] Saraiya, Y. Polynomial-time program transformations in deductive databases. In *Proc. 9th ACM SIGACT-SIGMOD-SIGACT Symp. on Principles of Database Systems*, pages 132–144, 1990.

[28] Ullman, J.D. *Principles of Database and Knowledge-Base Systems*, volume I. Computer Science Press, Maryland, 1989.

[29] Ullman, J.D. *Principles of Database and Knowledge-Base Systems*, volume II. Computer Science Press, Maryland, 1989.

[30] Vardi, M.Y. Decidability and undecidability results for boundedness of linear recursive queries. In *Proc. 7th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 341–351, 1988.

[31] Wood, P.T. Syntactic characterizations of 1-bounded datalog programs. In *International Conference on Deductive and Object-Oriented Databases*, 1991.

[32] Zhang, W., Yu, C.T., and Troy, D. Necessary and sufficient conditions to linearize doubly recursive programs in logic databases. *ACM Transactions on Database Systems*, pages 271–293, 1990.