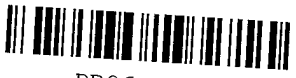
January 1991

Report No. STAN-CS-91-1356

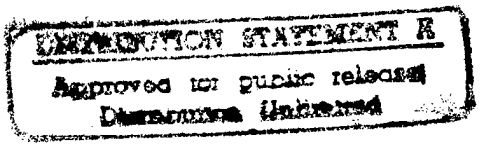
Thesis



SUBTREE-ELIMINATION ALGORITHMS IN DEDUCTIVE DATABASES

by

Yatin Saraiya



Department of Computer Science

Stanford University

Stanford, California 94305

SUBTREE-ELIMINATION ALGORITHMS IN DEDUCTIVE DATABASES

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Yatin Saraiya
January 1991

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Jeffrey D. Ullman
(Principal Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Vaughan Pratt

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Yehoshua Sagiv

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies

Abstract

A deductive database consists of a set of stored facts, and a set of logical rules (typically, recursive Horn clauses) that are used to manipulate these facts. A number of optimizations in such databases involve the transformation of sets of logical rules (*programs*) to simpler, more efficiently evaluable programs. We consider a class of optimizations in which the transformation is a simple syntactic restriction on the form of the original program, and in which the correctness of the transformation indicates the existence of a normal form for the proof trees generated by the program. For example, the existence of *basis-linearizability* in a nonlinear program indicates that the program is inherently linear, and permits the use of special-purpose query evaluators for linear recursions. The canonical example of a basis-linearizable program is the program that computes the transitive closure of a binary relation; the corresponding normal form for the proof trees is that of right-linearity. Similarly, if a program is *sequencable*, then it is conducive to a pipelined evaluation. In addition, the existence of *k-boundedness* in a program permits the elimination of recursion overhead in evaluating the program. We investigate the complexity of detecting such optimization opportunities, and provide correct (but not always complete) algorithms for this purpose.

Each of the problems that are mentioned above may be described in terms of the *subtree-elimination problem*, which we define and analyze. We relate the detection of basis-linearizability, sequencability and 1-boundedness to the complexity classes $\mathcal{NC}$, $\mathcal{P}$ and $\mathcal{NP}$, and show that the first two of these problems are, in general, undecidable. The techniques used in our analysis provide a complete description of the complexity of deciding the equivalence of *conjunctive queries* (single-rule, nonrecursive programs), and tight undecidability results for the detection of program equivalence.

Acknowledgements

First and foremost, I would like to thank Jeff Ullman, who had a major role in defining and shaping this thesis. His insight and interest contributed greatly to my understanding of the problem, and his kindness and generosity made it a pleasure to be a member of this Department.

I would also like to thank Vaughan Pratt for his patience, Shuky Sagiv for his numerous and invaluable comments at various stages in the development of this thesis, and Moshe Vardi for posing hard problems.

I'd like to express my appreciation to the following people: Tomás Feder and Jack Snoeyink, for letting me bounce my ideas off them; members of the NAIL! group, past and present, for the stimulating environment that they provided; Gene Klotz, for giving me my first shot at research; and Ed Skeath, for getting me started on this whole business. Ed will be sorely missed.

I am grateful to the following organizations for their support: NSF (IRI-88-12791), AFOSR (88-0266), and IBM (equipment grant).

On a personal note, I'd like to thank my mother Indu, my father Pratap and my brother Nakul for all the love and support they have shown me over the years. I'd also like to thank all the friends who have helped out at various stages; they know who they are, and they know I know.

Y. P. S.
Stanford, CA

Contents

List of Tables

List of Figures

Chapter 1

Subtree eliminations

1.1 Introduction

A *deductive database system* represents the use of predicate logic as a programming language for database systems. One may think of this programming language as the extension of *relational algebra* ([11]) through the use of recursion. This extension provides a strict increase in the expressive power of the database query language ([2]), but makes query evaluation potentially more expensive; that is, a general-purpose query evaluator is likely to be inefficient when applied to a "simple" program. Many of the optimization strategies that have been incorporated into the experimental deductive database systems currently under construction ([23, 22, 21], for example) are based on the recognition of programs on which limited yet efficient query evaluators may be used. In this dissertation, we provide an alternative optimization strategy: the replacement of programs by semantically equivalent but syntactically simpler programs, such that efficient algorithms may be used with respect to the transformed programs. The optimizations that we investigate are based on the detection of "normal forms" for the proof trees generated by the program in question. The problems that we address are *decision* problems; that is, given a program and a normal form, we ask whether the normal form applies to the given program. In this thesis, we present a uniform framework for the description of normal forms, a mechanism for the construction of conditions that are sufficient (but not always necessary) for the detection of each such normal form, and complexity results for the detection of three common normal forms. Our results have implications to the complexity of deciding equivalence among recursive and nonrecursive programs.

1.2 Deductive databases

For our purposes, a deductive database system[1] consists of a finite set of stored ground facts (the *extensional database* or *EDB*), and a finite set of rules (Horn clauses) that are used to manipulate the EDB. A set of rules is termed a *program*. The program comprises

[1] See [35, 36] for a comprehensive treatment.

the *intensional database* or *IDB*. Relations are defined in terms of predicates; that is, for any predicate p, the relation for p is the set of tuples $\vec{a}$ such that $p(\vec{a})$ is true. In this report, we will use the terms "predicate" and "relation" interchangeably. A predicate that corresponds to an EDB relation is termed an *extensional* or *EDB* predicate, and a predicate that is defined by a rule is termed *intensional* or *IDB*. We assume without loss of generality that no predicate is both intensional and extensional.

1.2.1 Syntax

Programs will be written using Prolog syntax. A rule is of the form

$$p(\vec{X}) :- b_1(\vec{Y_1}), \ldots, b_n(\vec{Y_n}).$$

The " :- " represents the "if" operator, and a comma represents the "and" operator. The atomic formula (*atom*) $p(\vec{X})$ is termed the *head* of the rule, and the conjunction on the right of the "if" symbol is termed the *body* of the rule. Each atomic formula in the body is termed a *subgoal*. The rule defines the predicate p, and p is hence intensional. The variables appearing in the head of the rule are termed *distinguished*, and all other variables are termed *nondistinguished*. Distinguished variables are universally quantified over the rule, and nondistinguished variables are implicitly existentially quantified in the body of the rule. That is, if $W_1, \ldots, W_k$ are the distinguished variables in the rule and $Z_1, \ldots, Z_m$ are the nondistinguished variables, then the rule represents the formula

$$\forall W_1, \ldots, W_k((\exists Z_1, \ldots, Z_m b_1(\vec{Y_1}) \wedge \ldots \wedge b_n(\vec{Y_n})) \supset p(\vec{X}))$$

Example 1.1 The following program $\mathcal{P}$ consists of the two rules r_1 and r_2, and defines the intensional predicate p. We assume that b is an extensional predicate.

$r_1 : p(X,Y) :- p(X,U), p(U,Y).$
$r_2 : p(X,Y) :- b(X,Y).$

$p(X,Y)$ is the head of each rule. The conjunction $p(X,U), p(U,Y)$ is the body of rule r_1, and $b(X,Y)$ is the body of rule r_2. The meaning of rule r_1 is: for all X and Y, $p(X,Y)$ is true if for some U, $p(X,U)$ and $p(U,Y)$ are both true; that is, r_1 represents the formula

$$\forall X, Y((\exists U p(X,U) \wedge p(U,Y)) \supset p(X,Y))$$

□

If a predicate p appears in the head of a rule and q appears in the body of the rule, then p is said to *depend* on q. A predicate p is termed *recursive* if p depends transitively upon itself, and a rule is termed recursive if a predicate q appearing in the body of the rule depends transitively upon the predicate p appearing in the head of the rule. All other predicates and rules are termed *nonrecursive*. A program is termed recursive if any rule is recursive, and nonrecursive otherwise. In the example above, the predicate p and rule r_1 are

recursive (so the program is recursive), and the predicate b and the rule r_2 are nonrecursive. A nonrecursive rule is also termed a *basis* or *initialisation* rule. A rule is said to be *linear* if at most one subgoal in the rule is intensional, *linear recursive* if exactly one subgoal in the body is recursive with the head of the rule, and *bilinear* if exactly two subgoals are recursive with the head. Rule r_1 in Example 1.1 is bilinear, and rule r_2 is linear. Rule r_1 in Example 1.2 below is also linear. A program is termed linear if every rule in the program is linear, and linear recursive if every rule is linear recursive or nonrecursive.

1.2.2 Semantics

The accepted semantics for Horn-clause programs consists of the *unique minimum Herbrand model* or *least fixed point* ([38]). The idea is that we may think of the "application" of a rule as the bottom-up (forward-chaining) use of the Horn clause represented by the rule. Then, the relation for each intensional predicate is the smallest relation that satisfies each of the rules in the program; that is, the smallest relation that is closed under the application of the rules in the program as described above. Alternatively, we may generate the intensional relations by initialising each intensional predicate to be empty, adding all facts generated by basis rules and then applying the rules in a bottom-up (forward-chaining) manner until no new facts are generated. The first of these views, that of the relation for a predicate being the smallest relation satisfying the initialisation rules and closed under the recursive rules, is integral to the approach taken by this report. Since programs may be viewed as generalised closures, we have focussed our attention on optimizations that may be performed on programs that compute such common relations as the symmetric and transitive closures of a binary relation.

Example 1.2 The program

$r_1: \ p(X,Y) :- \ p(Y,X).$
$r_2: \ p(X,Y) :- \ b(X,Y).$

computes the symmetric closure of the basis predicate b. The recursive rule r_1 states that p is symmetric, the basis rule insists that $b \subset p$ and minimality is imposed by the semantics of the program. Note that the program is linear (and linear recursive). □

Example 1.3 We repeat here the program of Example 1.1.

$r_1 : p(X,Y) :- \ p(X,U), p(U,Y).$
$r_2 : p(X,Y) :- \ b(X,Y).$

The first rule says that p is transitive and the second requires inclusion; thus, the program computes the transitive closure of b. If we think of b as the "parent" relation, then this program computes the "ancestor" relation. This program is important in that there is no nonrecursive program computing the transitive closure of b ([2]), justifying our earlier claim that the addition of recursion to relational algebra increases the expressive power of the language. □

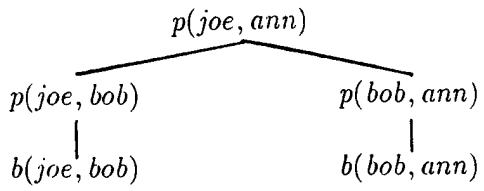


Figure 1.1: Proof tree.

Example 1.4 The program

$r_1 : p(X,Y) :- p(X,U), p(U,Y).$
$r_2 : p(X,Y) :- p(Y,X).$
$r_3 : p(X,Y) :- b(X,Y).$

computes the symmetric, transitive closure of b. □

A rule is said to be *safe* if for every rule, every distinguished variable appears in the rule body. A program is said to be *Datalog* if it is safe and function-free; the programs of Examples 1.1 – 1.4 are all Datalog programs. In this report, we will for the most part restrict our attention to Datalog programs. Such programs are commonly used because they are powerful enough for the description of many real-life problems, but always generate finite relations from a finite database (because the Herbrand universe is finite).

1.2.3 Proof trees

For the purposes of this report, it will be convenient to view the relation generated by a program in another way. We say that an atom is *ground* if no variable appears in it. A rule is termed *instantiated* if every atom in the rule is replaced with a ground atom with which it unifies, such that the implied unifications are consistent. A *proof tree* is a tree in which the vertices are ground atoms such that

1. The leaves of the tree are atoms appearing in the EDB; and
2. If an interior node a has the children $b_1, \ldots, b_n$, then there is an instantiated rule in the program whose head is a and whose body is $b_1, \ldots, b_n$.

If c is the root of a proof tree, then we say that the proof tree *establishes* the fact c.

Example 1.5 Assume that the relation for b in Example 1.3 is $\{b(joe, bob), b(bob, ann)\}$. The proof tree of Figure 1.1 establishes $p(joe, ann)$. □

Now, given any program and database, the relation produced by the program for some predicate p from this database is precisely the set of facts $p(\vec{a})$ that are established by proof trees.

1.2.4 Optimizations

In typical database applications, the extensional database is much larger than the size of a program that manipulates it. Hence, the preferred optimization techniques are *data-independent*. Two programs are said to be *equivalent* if they generate the same relations for every predicate from every extensional database; the optimizations that we consider are based on program equivalence, and are hence independent of any particular database. A variety of efficient query evaluation techniques have been proposed (see [7],[36] for overviews) and these techniques vary in application domain and efficiency. In this report, we investigate opportunities for transforming programs into equivalent programs for which efficient query evaluation techniques become available. The following examples illustrate optimizations that may be performed on the closure programs of the preceding examples; these optimizations will serve as canonical examples for three optimization problems that we will consider throughout this dissertation.

Example 1.6 Consider the symmetric closure program (say, $\mathcal{P}$) of Example 1.2. We may think of the basis relation b as the edge relation in a directed graph; that is, $b(u, v)$ is true precisely when there is an edge $u \rightarrow v$ in the graph. Then, the symmetric closure of the graph may be obtained by adding the edge $v \rightarrow u$ to the graph, where $u \rightarrow v$ is any edge in the original graph. That is, the program of Example 1.2 is equivalent to the following nonrecursive program $\mathcal{Q}$.

r_1' : $p(X, Y) :- b(Y, X)$.
r_2 : $p(X, Y) :- b(X, Y)$.

The program $\mathcal{Q}$ is obtained from the program $\mathcal{P}$ by replacing the recursive subgoal $p(Y, X)$ in the body of rule r_1 by the nonrecursive subgoal $b(Y, X)$. The gains of such a replacement stem from the elimination of recursion overhead in evaluating the program with respect to a database; that is, we may use a query evaluator that is specific to nonrecursive programs. □

Example 1.7 Consider the bilinear program $\mathcal{P}$ of Example 1.3, computing the transitive closure of the basis predicate b. It is a well-known fact that $\mathcal{P}$ is equivalent to the following linear recursive program $\mathcal{Q}$.

r_1' : $p(X, Y) :- b(X, U), p(U, Y)$.
r_2 : $p(X, Y) :- b(X, Y)$.

The program $\mathcal{Q}$ is obtained from $\mathcal{P}$ by replacing the first recursive atom in the body of r_1, $p(X, U)$, with the nonrecursive atom $b(X, U)$. An intuitive understanding of the equivalence is as follows. Assume that b stands for the "parent" relation, and that p is the "ancestor" relation, as we have previously discussed. Then, the recursive rule in $\mathcal{P}$ says that "Y is X's ancestor if X has some *ancestor* U whose ancestor is Y", and the recursive rule in $\mathcal{Q}$ says that "Y is X's ancestor if X has some *parent* U whose ancestor is Y." The gains of replacing

$\mathcal{P}$ by $\mathcal{Q}$ is that fast query evaluators that are specific to linear recursions become available for application to the linear program $\mathcal{Q}$. Specifically, assume that we wish to answer the query $p(joe, Y)$?; that is, "who are the ancestors of *joe*". Then, the "magic sets" technique ([36]) applied to $\mathcal{P}$ generates the entire p relation, and takes time that is quadratic in the size of b (in the worst case); however, "right-linear evaluation" ([36]) computes the answer in time that is linear in the size of b [2]. A proof of this claim is contained in [36]. □

Example 1.8 Finally, consider the program $\mathcal{P}$ of Example 1.4, computing the symmetric, transitive closure of b. Again, we may think of b as the edge relation in a directed graph; then, $p(u, v)$ is true iff there is a path from u to v in the graph, obtained by following edges in a forward or backward manner (in a mixed fashion). A little thought should suffice to convince the reader that p may be computed by

1. First, computing the symmetric closure of the graph.
2. Then, computing the transitive closure of the result.

That is, the program $\mathcal{P}$ is equivalent to the following program, where the new predicate q is the symmetric closure of b.

r_1 : $p(X, Y) :- p(X, U), p(U, Y)$.
s_1 : $p(X, Y) :- q(X, Y)$.
r_2' : $q(X, Y) :- q(Y, X)$.
r_3' : $q(X, Y) :- b(X, Y)$.

Now, since p is defined only by rules r_1 and s_1, and since q is defined by rules r_2' and r_3', we may apply the optimizations of the previous examples to create the equivalent program

r_1' : $p(X, Y) :- q(X, U), p(U, Y)$.
s_1 : $p(X, Y) :- q(X, Y)$.
r_2'' : $q(X, Y) :- b(Y, X)$.
r_3' : $q(X, Y) :- b(X, Y)$.

Finally, since q is now defined only by the nonrecursive rules r_2'' and r_3', we may substitute these rules for the predicate q to obtain the linear recursive program

r_1'' : $p(X, Y) :- b(X, U), p(U, Y)$.
r_1''' : $p(X, Y) :- b(U, X), p(U, Y)$.
s_1' : $p(X, Y) :- b(Y, X)$.
s_1'' : $p(X, Y) :- b(X, Y)$.

The gains of such a replacement are again obtained from the use of a query evaluator that is specific to linear recursions. That is, magic sets takes quadratic time (in the size of b) to evaluate $p(joe, Y)$? with respect to the original program, but right-linear evaluation takes linear time when applied to the linear recursive program above. □

[2] Right-linear evaluation may be extended to the nonlinear transitive-closure program, but does not extend to arbitrary nonlinear programs.

1.3 Top-down expansions

Recall that the facts generated by a program from a database are precisely those facts that are established by proof trees. A *top-down expansion* is obtained by lifting the arguments of a piece of a proof tree to variables; that is, the top-down expansion specifies the relation between the leaves and the root of a proof-tree piece. Alternatively, one may think of a top-down expansion as a state in a top-down query evaluation.

Definition 1.1 Consider a (not necessarily function-free) program $\mathcal{P}$ with rules $r_1, \ldots, r_n$, and let p be any intensional predicate defined by $\mathcal{P}$. Assume that p has arity k. A *top-down expansion of* p by $\mathcal{P}$ is defined inductively, as follows.

1. The tree with root $p(X_1, \ldots, X_k)$ and leaf $p(X_1, \ldots, X_k)$, where $X_1, \ldots, X_k$ are distinct variables, is a top-down expansion of p by $\mathcal{P}$. By convention, this top-down expansion is said to have depth 0.

2. Let r_1 be a rule

$$h :- B.$$

   in which the rule head h has principal functor p. The tree T with root h and leaves B (in which the order of subgoals in B is preserved) is a top-down expansion of p by $\mathcal{P}$. The depth of this top-down expansion is 1.

3. Consider any top-down expansion T of p. Assume that $q(\vec{Z})$ is the ith leaf in T. Let R be a top-down expansion of depth 1, in which all variables have been renamed to make them distinct from the variables in T, and let τ be the mgu of $q(\vec{Z})$ with the root of R. Let S be the expansion R, in which each variable Y appearing in the root of R is replaced by $\tau(Y)$ throughout R. Replace every variable Y in T that appears in the root of S by $\tau(Y)$, and replace the ith leaf in the result by the subtree S. The result is a top-down expansion of p by $\mathcal{P}$; the depth of the expansion is the depth of the resulting tree.

□

Recall that a variable appearing in a rule is distinguished if it appears in the head of the rule, and nondistinguished otherwise. The reason that nondistinguished variables are renamed at each stage of the expansion is that these variables are implicitly existentially quantified in the body of the rule. Distinguished variables are renamed only to make them distinct from variables (perhaps nondistinguished) in the parent tree.

Note that the definition of a top-down expansion requires that subgoals in a rule are written in a way that preserves the order of the subgoals in every rule. Throughout this report, we will assume that both proof trees and top-down expansions are written in a way that preserves the left-to-right order of the subgoals in every rule[3].

[3]This assumption refers to the writing down of proof trees, and has no bearing on the order in which subgoals are evaluated by a query evaluator.

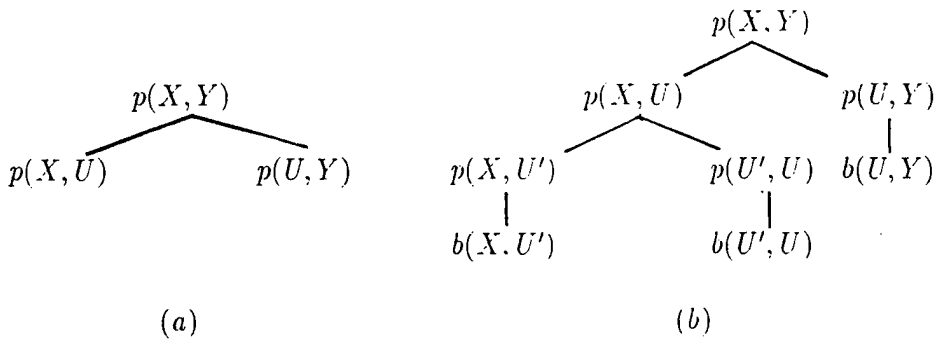


Figure 1.2: Top-down expansion.

Note also that the construction of a top-down expansion is polynomial in the size of the expansion (since unification is polynomial [36]). For Datalog programs, this procedure may easily be performed in LOGSPACE.

Example 1.9 Consider the program of Example 1.3. Recall that this program computes p as the transitive closure of the basis predicate b; that is, we may think of b as the *parent* relation and p as the *ancestor* relation. The trees of Figure 1.2 are top-down expansions of $p(X, Y)$ using the rules in this program.

Note that the variable U is renamed at depth 2 in the tree of Figure 1.2 (b). Intuitively, the top-down expansion states that Y is an ancestor of X if X has some parent U', U' has some parent U and U has the parent Y. That is, Y is X's great-grandparent in this case. □

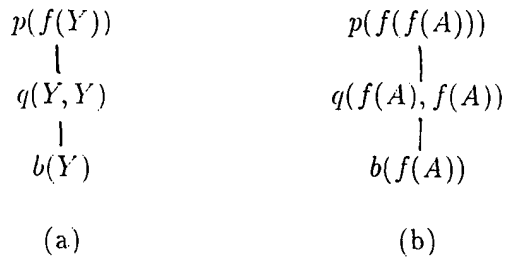
We will also speak of a top-down expansion of an atom $p(\vec{X})$; that is, we will specify an atom to be unified with the root of the tree. Let T be a top-down expansion and $p(\vec{X})$ an atom. Assume that this atom unifies with the root of T under the most general unifier τ. Then, the expression $\tau(T)$ represents the expansion obtained by replacing every variable Y in T that appears in the root by $\tau(Y)$. We say that $\tau(T)$ is a *top-down expansion of* $p(\vec{X})$ *by* $\mathcal{P}$.

Example 1.10 Consider the program

$r_1 : p(f(X)) :- q(X, U).$
$r_2 : q(X, X) :- b(X).$

Figure 1.3 (a) shows a top-down expansion of p with depth 2, and Figure 1.3 (b) exhibits a top-down expansion of $p(f(f(A)))$.

In Figure 1.3 (a), the variable X in rule r_2 has been renamed to the variable Y. □



(a) (b)

Figure 1.3: Top-down expansion with specified root.

1.3.1 Conjunctive queries

A single-rule, nonrecursive program is called a *conjunctive query* ([10]). That is, a conjunctive query is of the form

$$p(\vec{X}) :- B.$$

where B is a conjunction of atomic formulae all of whose principal functors are EDB predicate names. Each top-down expansion with EDB predicates at the leaves defines a conjunctive query in a natural way; that is, if T is such a top-down expansion, then the corresponding conjunctive query is

$$h :- B.$$

where h is the root of T, and where B is the conjunction of the fringe of T.

Example 1.11 The top-down expansion of Figure 1.2 (b) represents the conjunctive query

$$p(X, Y) :- b(X, U'), b(U', U), b(U, Y).^4$$

□

We will also speak of top-down expansions with IDB predicates at the leaves as conjunctive queries. The intention, in this case, is that the conjunctive query is applied nonrecursively.

Example 1.12 The top-down expansion of Figure 1.2 (a) represents the conjunctive query

$$p(X, Y) :- p(X, U), p(U, Y).$$

The idea is that this conjunctive query represents all ways in which two p-facts (presumably generated by the program) may be combined to produce a new p-fact using the top-down expansion. □

[4] This conjunctive query is often written $\{XY \mid b(X, U'), b(U', U), b(U, Y)\}$; however, we will use the notation illustrated above for the purposes of uniformity

We may now view a program as the (possibly infinite) union of the conjunctive queries that it generates, such that the bodies of the conjunctive queries are occupied by only extensional predicates. In fact, we will also speak of an arbitrary union of conjunctive queries as a "program".

1.3.2 Containment and equivalence

Let $\mathcal{P}$ and $\mathcal{Q}$ be programs defining the predicate p (perhaps among other IDB predicates). We say that $\mathcal{P}$ *is contained in* $\mathcal{Q}$ *with respect to* p (written $\mathcal{P} \subset_p \mathcal{Q}$, or just $\mathcal{P} \subset \mathcal{Q}$ if p is understood) iff for every database, the relation for p that is produced by $\mathcal{P}$ is a subset of that produced by $\mathcal{Q}$. The programs $\mathcal{P}$ and $\mathcal{Q}$ are said to be *equivalent with respect to* p (written $\mathcal{P} \equiv_p \mathcal{Q}$) iff $\mathcal{P} \subset_p \mathcal{Q}$ and $\mathcal{Q} \subset_p \mathcal{P}$; as before, we omit all references to the predicate p if this predicate is understood from the context.

Since a conjunctive query is a single-rule, nonrecursive program, these definitions apply equally to such queries. However, for the purpose of deciding containment among conjunctive queries, we need make no reference to the predicates being defined by the queries because each conjunctive query defines a unique predicate. That is, if the heads of two conjunctive queries are labelled by two different predicates, then there is no containment; otherwise, the containment is defined with respect to the common predicate defined by the queries.

Chandra and Merlin ([10]) have proposed a syntactic test for the containment of one conjunctive query in another. Consider the conjunctive queries

$C_1 : p(\vec{X}) :- B_1.$
$C_2 : q(\vec{Y}) :- B_2.$

where B_1 and B_2 are conjunctions. Let f be a function on the variables in C_2. We extend f to all symbols in C_2 by requiring f to be the identity on constants. Finally, we may extend f to terms (and atomic formulae) in the obvious way; that is, we define $f(q(d_1, \ldots, d_k))$ to be $f(q)(f(d_1), \ldots, f(d_k))$. We say that f is a *containment mapping from* C_2 *into* C_1 (written $f : C_2 \rightarrow C_1$) iff the following are true.

1. $f(q(\vec{Y})) = p(\vec{X})$.
2. For each atom t in B_2, the atom $f(t)$ appears in B_1.

For any atom in C_2, $f(t)$ is termed the *destination* of t under f.

The value of containment mappings is illustrated in the following theorem of Chandra and Merlin ([10]).

Theorem 1.1 *Containment mapping theorem (Chandra and Merlin).*
For any conjunctive queries C_1 and C_2, $C_1 \subset C_2$ iff there is a containment mapping $f : C_2 \rightarrow C_1$. □

In Chapter 2 of this dissertation, we will present a dual to the containment mapping, the *conjunct mapping*, which is also a necessary and sufficient condition for the detection

of conjunctive query containment. The value of the concept of conjunct mappings is that the concept permits of a complete description of the complexity of deciding containment among conjunctive queries; the description is also contained in Chapter 2.

Example 1.13 Consider the conjunctive queries C_1 and C_2, as defined below.

C_1 : $p(X)$:- $a(X,B), b(A,B), b(C,B), c(B,B), c(A,D)$.
C_2 : $p(X)$:- $a(X,V), b(U,V), c(U,W)$.

The function f defined by $f(X) = X, f(V) = B, f(U) = A, f(W) = D$ is a containment mapping from C_2 into C_1. The destination (under f) of $p(X)$ is $p(X)$, of $a(X,V)$ is $a(X,B)$, of $b(U,V)$ is $b(A,B)$ and of $c(U,W)$ is $c(A,D)$.

However, there is no containment mapping $g : C_1 \rightarrow C_2$. Assume such a g exists. Then, the destination of $c(B,B)$ under g would have to be $c(U,W)$ (the only c-atom in C_2). However, $g(c(B,B)) = c(U,W)$ requires that $g(B)$ be both U and W, contradicting the functionality of g. □

Now, we may define the containment of top-down expansions (with IDB predicates permitted at the leaves) as the containment of the corresponding conjunctive queries. Given the relations between rules, top-down expansions and conjunctive qeuries, we will sometimes also speak of the containment of a top-down expansion in a rule.

Finally, recall that we may think of a program as a (possibly infinite) union of conjunctive queries, where the bodies of these queries are atomic formulae whose principal functors are EDB predicates. Sagiv and Yannakakis ([29]) use this idea to reduce program containment to the containment of conjunctive queries. The theorem of Sagiv and Yannakakis says that, for a program $\mathcal{P}$ to be contained in a program $\mathcal{Q}$, each conjunctive query generated by $\mathcal{P}$ must be contained in some conjunctive query generated by $\mathcal{Q}$; that is, there can be no "mixing and matching".

Theorem 1.2 *(Sagiv and Yannakakis).*
Consider programs $\mathcal{P}$ and $\mathcal{Q}$, defining the IDB predicate p. Then, $\mathcal{P} \subset_p \mathcal{Q}$ iff for every conjunctive query $C_{\mathcal{P}}$ generated by $\mathcal{P}$ with EDB predicates at the leaves and p at the root, there is a conjunctive query $C_{\mathcal{Q}}$ generated by $\mathcal{Q}$ with EDB predicates at the leaves such that $C_{\mathcal{P}} \subset C_{\mathcal{Q}}$. □

1.3.3 Tree shapes

As we will show in the next section, the optimizations of the previous section can be described in terms of *normal-form* proof trees. That is, each optimization is made possible because all facts generated by the relevant program are generated by proof trees of a certain "shape".

Recall that proof trees (and top-down expansions) are written in a way that preserves the left-to-right order of the subgoals in any rule. The idea of a proof-tree shape is the intuitively obvious one, given this assumption. Following Helm ([13]), we may formalise the concept by defining a shape as a list over the rule names.

Definition 1.2 Let $\mathcal{P}$ be a program with rules $r_1, \ldots, r_n$. A *tree shape* (or just *shape*) is a list that is defined as follows.

1. The empty list [] is a shape. This list represents all top-down expansions of depth 0.

2. Let r_i be a rule defining the predicate p, with intensional subgoals (in order) $t_1, \ldots, t_k$ (that is, the principal functor of each t_i is an intensional predicate name). Then. the list
$$[r_i \overbrace{[]\ldots[]}^{k}]$$
is a shape. This shape represents the top-down expansion of p with depth 1, in which r_i is used to construct the expansion. The $(j+1)$th component of this list is said to *represent the leaf* t_j.

3. Let S be a nonempty shape, representing some top-down expansion T of some predicate p. Assume that the ith occurrence of the empty list in S represents the ith leaf $q(\vec{Z})$ in T. Assume further that the head of rule r_j unifies with this leaf under the mgu τ, and that r_j has the intensional subgoals (in order) $q_1(\vec{Z}_1), \ldots, q_k(\vec{Z}_k)$. Construct the top-down expansion T' by expanding the ith leaf $q(\vec{Z})$ in T through rule r_j. Let S' be the shape S, where the empty list representing the ith leaf $q(\vec{Z})$ in T is replaced by the list
$$[r_j \overbrace{[]\ldots[]}^{k}]$$
Then S' is a shape, representing the top-down expansion T'. Further, the lth occurrence of the empty list in the above list is said to represent the lth leaf $q_l(\tau(\vec{Z}_l))$ in the newly-added subtree.

□

It is clear that every top-down expansion has a well-defined shape. We will also speak of the shape of a top-down expansion of a specified atom, and of the shape of a proof tree.

Example 1.14 Figure 1.2 contains top-down expansions generated by the program of Example 1.3. The expansion of Figure 1.2 (a) has the shape $[r_1[][]]$, and expansion (b) in that figure has the shape $[r_1[r_1[r_2][r_2]][r_2]]$. Similarly, Figure 1.3 exhibits top-down expansions generated by the program of Example 1.10. Figure 1.3 (a) contains a top-down expansion with the shape $[r_1[r_2]]$, and part (b) of that figure contains a top-down expansion with the same shape, but with the specified root $p(f(f(A)))$. □

The structure of a shape can be simplified if the program in question is linear (recall that a program is linear if at most one atom in the body of any rule is intensional). In the linear case, each top-down expansion may be represented as a *string* over the rule names, and sets of top-down expansions may be represented by regular expressions over the rule names ([25]).

Example 1.15 Consider the linear logic program of Example 1.10. The top-down expansions of Figure 1.3 have the shape $r_1 r_2$; note that the order in which the rule names appear in the shape corresponds to the order in which rules are applied in a top-down fashion, starting with the root. The set of all top-down expansions generated by the program, with EDB predicates at the leaves, may be denoted by the regular expression $r_1^* r_2$. □

Extending functions

We formalise our earlier notions of the extensions of functions on the variables in an expansion, as follows. Let f be any function defined on some of the variables in a top-down expansion T. We extend f to all variables, and to constants, by requiring f to be the identity on all variables on which it is not defined, and on all constants. Similarly, for any term, we define $f(q(d_1, \ldots, d_k))$ to be $f(q)(f(d_1), \ldots, f(d_k))$. Finally, we define $f(T)$ to be the result of replacing every atom a in T by $f(a)$. The idea, as before, is that we merely replace every occurrence of every variable on which f is defined by its image under f. In the remainder of this chapter, we will assume that all functions have been extended as described above.

Labels

Consider any sequence $T_1, \ldots, T_n$ of top-down expansions, not necessarily of different shapes. We will assume that each node in each expansion is labelled to be distinct from all other nodes in the expansion, and from all nodes in every other expansion. Given a label l, the node to which l refers will be written $node(l)$. We assume that labels are preserved through the application of functions to a top-down expansion, as described above. Now, given any containment mapping $f : T_2 \rightarrow T_1$ where T_1 and T_2 are labelled top-down expansions, we assume the selection of a labelled destination for each labelled node in T_2 under f; that is, for each node $node(k)$ that is a leaf or the root in T_2, we select a unique label l such that $f(node(k)) = node(l)$.

Uniqueness of shapes

It is easily seen that every shape uniquely defines a top-down expansion, up to the renaming of variables. The idea is that the top-down application of rules "commute", and we may therefore construct a top-down expansion of a given shape using any order of rule applications. More formally, let T_1, T_2 and T_3 be top-down expansions in which all variables in each expansion are renamed to be distinct, and where T_2 and T_3 have depth 1 (i.e. each represents the application of a single rule). Let $node(l)$ and $node(k)$ be distinct leaves in T_1. Assume that we expand the leaf $node(l)$ in T_1 through T_2 under any unifier τ_1, and expand $node(k)$ in the result through T_3 under any unifier τ_2 to create some expansion S (see Figure 1.4).

Then, we may create S in the following way.

1. Expand $node(k)$ in T_1 through T_3 using the unifier $\sigma_2 = \tau_2(\tau_1)$ (recall that all functions are extended to be the identity on variables on which they are not defined).

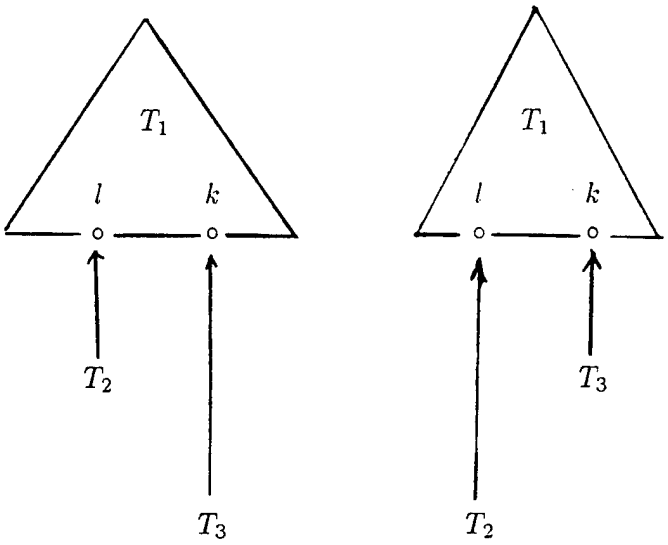


Figure 1.4: Changing the order of expansion.

2. Define the unification σ_1 to be the identity on variables in T_1 and T_3, and to be $\tau_2(\tau_1)$ on the variables in T_2. Expand $node(l)$ in the result of 1 above through T_2 using the unification σ_2.

That is, the unifications σ_1 and σ_2 permit us to construct the same expansion S by applying the expansions T_2 and T_3 to $node(l)$ and $node(k)$ in reverse order. Hence, if we use the mgu at each stage, then both orders of applications must yield equally general top-down expansions. An induction on n allows us to reach the same conclusion if the leaves $node(k_1), \ldots, node(k_n)$ are expanded through the depth-1 expansions $S_1, \ldots, S_n$ respectively. A straightforward induction on the size of a shape permits us to claim that every shape uniquely defines a top-down expansion, up to renaming of variables. Hence, we may abuse notation by speaking of the containment of a shape (or top-down expansion) in a shape.

Note that the uniqueness of shapes allows us to construct a top-down expansion from top-down expansions T_1, T_2 and T_3 of arbitrary depth, by renaming the variables of the expansions apart, expanding some leaf in T_1 through T_2 using the mgu of the leaf in T_1 with the root of T_2, and similarly expanding any leaf in the result through T_3.

1.3.4 Changing shapes

We present some results that will be of use in the following sections. The idea is that containment mappings between top-down expansions remain essentially unchanged under certain operations that are performed on the expansions in question, and expansions can therefore be robustly manipulated to change their shapes.

Replicating expansions

Let $T_1, \ldots, T_n$ be a sequence of expansions, not necessarily of different shapes. Assume that the label of each node in T_i is a list of the form $[i, l]$, where l is an integer unique to the relevant node in T_i. We will speak of *replicating* some tree T_i in the above sequence. To create k copies $T_{i1}, \ldots, T_{ik}$ of T_i, we construct k *rename* functions $ren_1, \ldots, ren_k$ that rename each variable in T_i to be distinct from any variables in $T_1, \ldots, T_n$ (or their copies, if any), such that the ranges of these rename functions are distinct. Then, for each j, T_{ij} is $ren_j(T_i)$. Further, the labels of each copy of T_i are propagated by setting the label of $node(m)$ in T_{ij} to $[j|m]$.

Theorem 1.3 *Expansion Theorem*
Let T_1 and T_3 be top-down expansions with distinct variables, and let T_2 and T_4 be expansions with distinct variables. Assume that $f : T_2 \rightarrow T_1$ and $g : T_4 \rightarrow T_3$ are containment mappings. Assume that R is obtained by expanding the leaf $node(l)$ in T_1 through T_2. Let $node(k_1), \ldots, node(k_n)$ be leaves in T_2 such that $f(node(k_i)) = node(l)$, and let S be the result of expanding each leaf $node(k_i)$ in T_2 through the ith replica T_{4i} of T_4. Then, there is a containment mapping $h : S \rightarrow R$ that preserves the destinations of f and g; that is, if $f(node(c)) = node(d)$ and $node(c)$ is a leaf in S, then $h(node(c)) = node(d)$; and, if $g(node(c)) = node(d)$, then $h(node([i|c]) = node(d)$ for all i. Figure 1.5 depicts this claim.

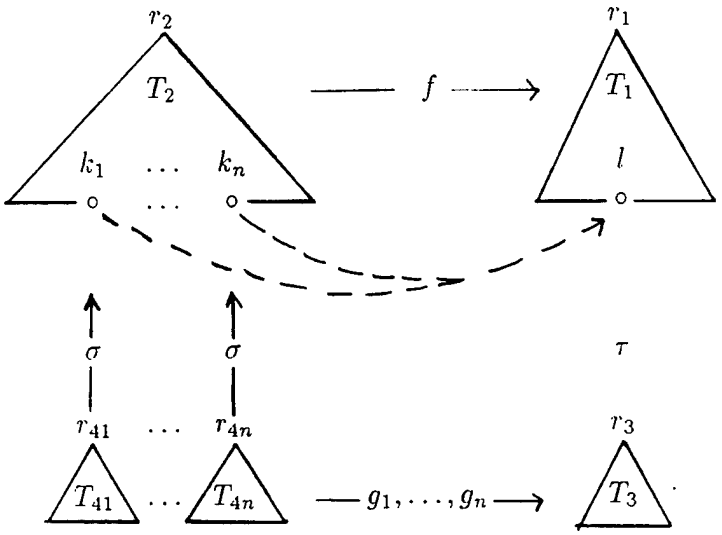


Figure 1.5: The Expansion Theorem.

Proof. Let the roots of T_1, T_2, T_3 and T_4 be r_1, r_2, r_3 and r_4 respectively. Similarly, let the root of T_{4j} be r_{4j} for all j. Let τ be the most general unifier of $node(l)$ with the root r_3 of T_3, and σ the most general unifier of the sequence $r_{41}, \ldots, r_{4n}$ with the sequence of leaves $node(k_1), \ldots, node(k_n)$ in T_2. Since g is a containment mapping from T_4 into T_3, the domain-disjoint functions $g_j = g(ren_j{}^{-1})$ are containment mappings from T_{4j} into T_3 that preserve the destinations of g. Define the function ρ as follows.

$$\rho(V) = \begin{cases} \tau(f(V)) & \text{for } V \text{ in } T_2 \\ \tau(g_j(V) & \text{for } V \text{ in } T_{4j} \end{cases}$$

Now, ρ is a unification under which the leaves $node(k_1), \ldots, node(k_n)$ may be expanded through the subtrees $T_{41}, \ldots, T_{4n}$ respectively (to create some expansion S', say). Consider any leaf $node(m)$ in T_2; assume that the destination of $node(m)$ under f is $node(i)$ in T_1. Then, $node(m)$ in S' is syntactically identical to $node(i)$ in R. Similarly, consider any leaf $node(m)$ in T_4; assume that the destination of $node(m)$ under g is $node(i)$ in T_3. Then for each j, $node([j|m])$ in S' is syntactically identical to $node(i)$ in R. That is, the identity I is a containment mapping from S' into R that preserves the destinations of f and g. If σ is the most general unifier of $node(k_1), \ldots, node(k_n)$ with the roots $r_{41}, \ldots, r_{4n}$ of $T_{41}, \ldots, T_{4n}$, then by the properties of the most general unifier there is some function h such that $\rho = h(\sigma)$; hence, h is a containment mapping from S into S' such that $h(node(i)) = node(i)$ for all nodes in S, and our result follows by composing h and I. □

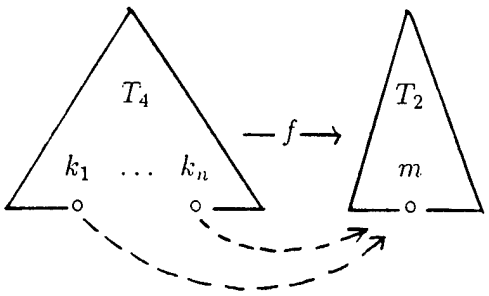


Figure 1.6: Assumption for the Splicing Theorem.

Theorem 1.4 *Splicing Theorem*
Let T_1, T_2, T_3 and T_4 be top-down expansions with distinct variables. Assume that $node(l)$ is a leaf in T_1 and $node(m)$ is a leaf in T_2, and that there is a containment mapping $f : T_4 \rightarrow T_2$. Let $node(k_1), \ldots, node(k_n)$ be the leaves in T_4 whose destination under f is $node(m)$ (see Figure 1.6). Construct the expansion R by expanding $node(l)$ in T_1 through T_2, and expanding $node(m)$ in the result through T_3. Construct S by expanding $node(l)$ in T_1 through T_4, and expanding each leaf $node(k_j)$ in the result through the jth replica T_{3_j} of T_3. Then, there is a containment mapping h from S into R such that the destination of each leaf $node(i)$ in S is as follows.

1. If $node(i)$ is a node in T_1, then $h(node(i)) = node(i)$.
2. If $node(i)$ is a node in T_4, then $h(node(i)) = f(node(i)$.
3. If i is of the form $[p|3]$, then $h(node(i)) = node(p)$.

Figure 1.7 depicts the idea.
Proof. The identity is a containment mapping from T_1 into T_1, and the function $g_j = ren_j^{-1}$ is a containment mapping from the jth replica of T_3 into T_3; our result follows by two applications of the Expansion Theorem. □

1.4 Subtree eliminations

1.4.1 Normal-form optimizations

In this section, we will present a variety of optimizations that are defined in terms of *normal-form conjunctive queries*. The domain on which we will define these optimizations is that of *single-IDB* programs; that is, programs in which there is only one intensional predicate, p. We do not assume that the programs are either safe or function-free. The following

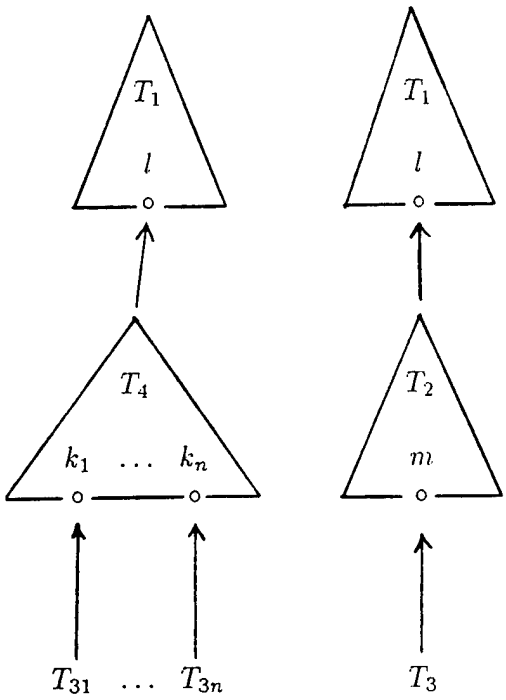


Figure 1.7: Illustrating the Splicing Theorem.

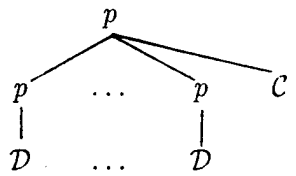


Figure 1.8: A one-bounded expansion.

program $\mathcal{P}$ will serve as the canonical example of a single-IDB program; the IDB predicate defined by the program is the predicate p.

Let $\mathcal{P}$ consist of the n recursive rules

$$r_1 : p(\vec{X}_{10}) :- p(\vec{X}_{11}), \ldots p(\vec{X}_{1k_1}), \mathcal{C}_1.$$
$$\vdots$$
$$r_i : p(\vec{X}_{i0}) :- p(\vec{X}_{i1}), \ldots p(\vec{X}_{ik_i}), \mathcal{C}_i.$$
$$\vdots$$
$$r_n : p(\vec{X}_{n0}) :- p(\vec{X}_{n1}), \ldots p(\vec{X}_{nk_n}), \mathcal{C}_n.$$

and the m nonrecursive rules

$$b_1 : p(\vec{Y}_{10}) :- \mathcal{D}_1.$$
$$\vdots$$
$$b_j : p(\vec{Y}_{j0}) :- \mathcal{D}_j.$$
$$\vdots$$
$$b_m : p(\vec{Y}_{m0}) :- \mathcal{D}_m.$$

where the $\mathcal{C}_i$ and $\mathcal{D}_j$ are arbitrary conjunctions of EDB predicates. Examples 1.1 – 1.4 exhibit such programs.

Let us define a top-down expansion to be *closed* if all its leaves are EDB predicates.

1.4.2 One-boundedness: definition and results

Define a top-down expansion to be *one-bounded* if at most the root is expanded through a recursive rule; that is, if the top-down expansion has depth at most 2 (see Figure 1.8). The program $\mathcal{P}$ is said to be *one-bounded* iff one-boundedness is a normal form for the proof trees generated by the program; that is, iff every closed conjunctive query generated by $\mathcal{P}$ is contained in some closed one-bounded conjunctive query.

If $\mathcal{P}$ is one-bounded, then $\mathcal{P}$ can be converted into an equivalent nonrecursive program $\mathcal{Q}$ by expanding each recursive atom in each rule through each nonrecursive rule. That is, the equivalent nonrecursive program $\mathcal{Q}$ is constructed as follows.

Let q be a new predicate symbol. Construct the program $\mathcal{Q}$, with $n + m + 1$ rules $\{r_i' \mid 1 \leq i \leq n\} \cup \{b_i' \mid 1 \leq i \leq m\} \cup \{c\}$, as follows. If r_i is a recursive rule, then replace r_i by the rule

$r_i' : \ p(\vec{X}_{i0}) :- \ q(\vec{X}_{i1}), \ldots, q(\vec{X}_{ik_i}), \mathcal{C}_i.$

That is, we replace all recursive occurrences of p in r_i with a corresponding occurrence of q.

Next, we introduce the rule c, which merely initialises p to q:

$c : \ p(V_1, \ldots, V_k) :- \ q(V_1, \ldots, V_k).$

The Vs are distinct variables. Finally, each nonrecursive rule b_i is replaced by the rule

$b_i' : \ q(\vec{Y}_{i0}) :- \ \mathcal{D}_i.$

which initialises q using the nonrecursive rules for p. Assume $\mathcal{P}$ is one-bounded; then, the closed top-down expansions of $\mathcal{P}$ are isomorphic to the closed top-down expansions of $\mathcal{Q}$; that is, each such top-down expansion generated by either program can also be generated by the other program. The equivalence of $\mathcal{P}$ and $\mathcal{Q}$ then follows by the theorem of Sagiv and Yannakakis (Theorem 1.2).

Example 1.16 Consider the program $\mathcal{P}$ of Example 1.2. The program $\mathcal{Q}$ is the program

$r_1' : \ p(X, Y) :- \ q(Y, X).$
$c : \ p(X, Y) :- \ q(X, Y).$
$b_1' : q(X, Y) :- \ b(X, Y).$

The intermediate predicate q can, in this case, be eliminated to obtain the linear program $\mathcal{Q}'$ of Example 1.6, repeated below.

$r_1' : \ p(X, Y) :- \ b(Y, X)$
$r_2 : p(X, Y) :- \ b(X, Y).$

The closed one-bounded expansions of $\mathcal{P}$ are easily seen to be isomorphic to the closed expansions of $\mathcal{Q}'$, as indicated by Figure 1.9. □

The gains of converting $\mathcal{P}$ into $\mathcal{Q}$ in this way are obtained from the elimination of recursion overhead in query processing.

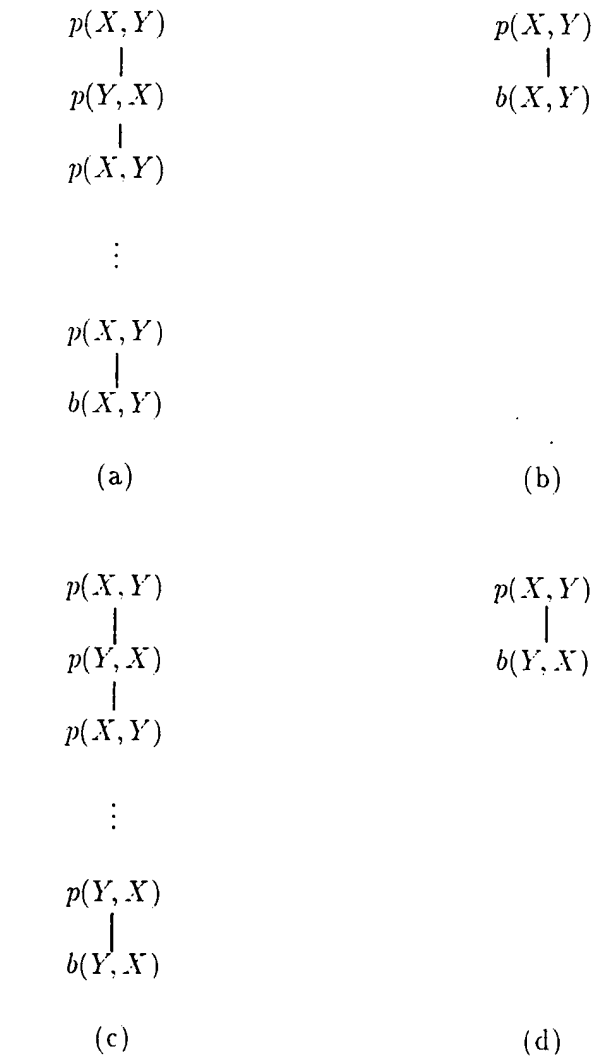


Figure 1.9: Illustrating Example 1.16.

Results

One-boundedness is easily seen to be decidable; we sketch a proof in Section 1.5. The complexity of one-boundedness has been investigated by Kanellakis ([20]) for a restricted class of programs. A *sirup* is a single-IDB , Datalog program with a single recursive rule, and a basis rule of the form

$$p(X_1, \ldots, X_k) :- b(X_1, \ldots, X_k).$$

where the Xs are distinct variables, and where the EDB predicate b appears nowhere else in the program. Kanellakis ([20]) has shown that one-boundedness in $\mathcal{NP}$-hard for linear sirups; that is, for programs that in addition to the basis rule above contain the recursive rule

$$p(\vec{Y}) :- p(\vec{Z}), e_1(\vec{W}_1), \ldots, e_n(\vec{W}_n).$$

Kanellakis' result, however, assumes an unbounded number of repetitions of EDB predicates in the body of the recursive rule. In Chapter 2, we will show that one-boundedness is $\mathcal{NP}$-hard even if there are no more than 4 repetitions of any EDB predicate in the body of the recursive rule. In the same chapter, we will show that one-boundedness is decidable in polynomial time for linear sirups in which no EDB predicate is repeated in the body of the recursive rule. Finally, in Section 1.5, we will present a polynomial-time algorithm that is sufficient (but not necessary) to detect one-boundedness in arbitrary single-IDB programs; the idea is a reduction to the decision procedure for linear sirups.

1.4.3 Base-case linearizability: definition and results

Recall our running assumption that top-down expansions are written in a way that preserves the left-to-right order of the subgoals in every rule. Define a top-down expansion generated by $\mathcal{P}$ to be *right-linear* if only the rightmost occurrence of p is ever recursively expanded in the expansion (see Figure 1.10). The program $\mathcal{P}$ is said to be *linearizable by basis* iff right-linearity is a normal form for the proof trees generated by the program; that is, iff every closed conjunctive query generated by $\mathcal{P}$ is contained in some closed right-linear conjunctive query.

If $\mathcal{P}$ is basis-linearizable, then $\mathcal{P}$ can be converted into an equivalent linear recursive program, as follows.

Let q be a new predicate symbol. Construct the program $\mathcal{Q}$, with $n + m + 1$ rules $\{r'_i \mid 1 \leq i \leq n\} \cup \{b'_i \mid 1 \leq i \leq m\} \cup \{c\}$, as follows. If r_i is a nonlinear rule ($k_i > 1$), then replace r_i by the linear rule

$$r'_i : \; p(\vec{X}_{i0}) :- q(\vec{X}_{i1}), \ldots, q(\vec{X}_{i_k-1}), p(\vec{X}_{ik_i}), \mathcal{C}_i.$$

That is, we replace all but the last recursive occurrence of p in r_i with a corresponding occurrence of q. If r_i is linear ($k_i = 1$), then r'_i is the same as r_i:

$$r'_i : \; p(\vec{X}_{i0}) :- p(\vec{X}_{i1}), \mathcal{C}_i.$$

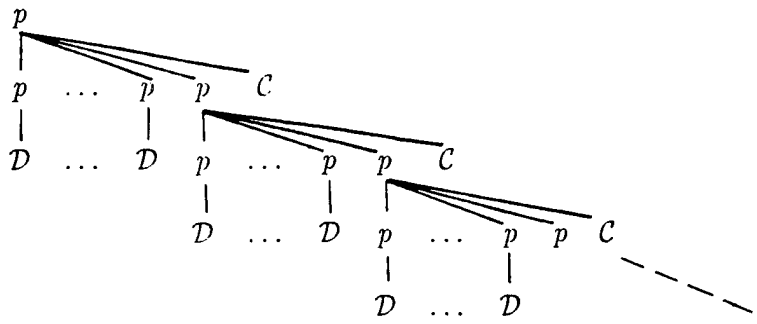


Figure 1.10: A right-linear expansion.

Next, we introduce the rule c, which merely initialises p to q:

$c: p(V_1, \ldots, V_k) :- q(V_1, \ldots, V_k).$

The Vs are distinct variables. Finally, each nonrecursive rule b_i is replaced by the rule

$b_i': q(\vec{Y}_{i0}) :- \mathcal{D}_i.$

which initialises q using the nonrecursive rules for p.

If $\mathcal{P}$ is basis-linearizable, then the closed right-linear top-down expansions of $\mathcal{P}$ are isomorphic to the closed top-down expansions of $\mathcal{Q}$; that is, each such top-down expansion generated by either program can also be generated by the other program. The equivalence of $\mathcal{P}$ and $\mathcal{Q}$ then follows by the theorem of Sagiv and Yannakakis (Theorem 1.2).

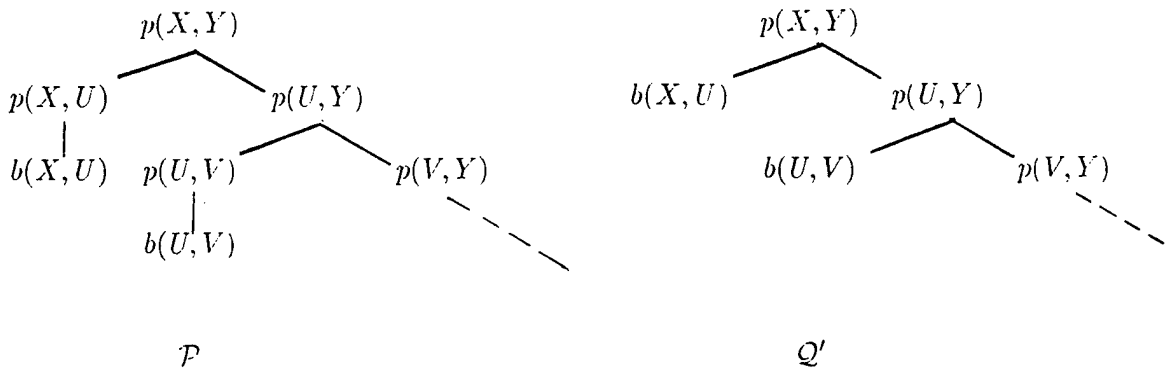
Example 1.17 Consider the program $\mathcal{P}$ of Example 1.3. The program $\mathcal{Q}$ is the program

$r_1': p(X,Y) :- q(X,U), p(U,Y).$
$c: p(X,Y) :- q(X,Y).$
$b_1': q(X,Y) :- b(X,Y).$

The intermediate predicate q can, in this case, be eliminated to obtain the linear program $\mathcal{Q}'$ of Example 1.7, repeated below.

$r_1': p(X,Y) :- b(X,U), p(U,Y).$
$r_2: p(X,Y) :- b(X,Y).$

The closed right-linear expansions of $\mathcal{P}$ are easily seen to be isomorphic to the expansions of $\mathcal{Q}'$, as indicated by Figure 1.11. □



$\mathcal{P}$ $\mathcal{Q}'$

Figure 1.11: Illustrating Example 1.17.

The gains of replacing $\mathcal{P}$ by $\mathcal{Q}$ are obtained through the use of special-purpose query evaluators for linear recursive programs. In fact, Ullman and van Gelder ([37]) have shown that the evaluation of programs with the *polynomial fringe property* (a superset of linear recursive programs) may be performed in $\mathcal{NC}$.

Results

Basis-linearizability was proposed by Zhang et al. ([40][5]), who studied this property in terms of a bilinear sirup with rectified rule heads (i.e., no variable is repeated in the head of any rule), such that there is at most one EDB subgoal in the body of the recursive rule. That is, they consider programs of the following form.

$$p(X_1, \ldots, X_k) :- p(\vec{Y}), p(\vec{Z}), e(\vec{W})).$$
$$p(X_1, \ldots, X_k) :- b(X_1, \ldots, X_k).$$

They claim a polynomial-time decision procedure for the detection of basis-linearizability for such programs, although their proof has a flaw in a key lemma; we will discuss the error in Chapter 3. In Chapter 3, we will show that basis-linearizability is decidable for bilinear sirups with an unbounded number of EDB subgoals in the recursive rule, as long as the EDB predicates are distinct; that is, we consider programs of the following form, in which the es are distinct.

$$p(X_1, \ldots, X_k) :- p(\vec{Y}), p(\vec{Z}), e_1(\vec{W}_1), \ldots, e_n(\vec{W}_n).$$
$$p(X_1, \ldots, X_k) :- b(X_1, \ldots, X_k).$$

The techniques of Chapter 2 can be used to show that our decision procedure is polynomial. We will show in Chapter 3 that the proof of [40] does not directly extend to such programs,

[5]This paper has recently been published in the ACM Transactions on Database Systems, but the proof has been omitted.

and in fact we also cover (as they do not) the case in which the program can be linearized in a different way; however, our proof has been motivated in large part by their treatment.

Ramakrishnan et al. ([25]) have shown that the decision procedure of Chapter 3 does not extend to the case in which EDB repetitions are permitted in the body of the recursive rule, and show that detecting basis-linearizability is $\mathcal{NP}$-hard in this case; however, their reduction involves an unbounded number of repetitions of EDB predicates in the recursive rule. In Chapter 2, we show that the detection of basis-linearizability is $\mathcal{NP}$-hard for bilinear sirups, even if no EDB predicate appears more than 4 times in the body of the recursive rule.

In Chapter 4, we show that the detection of basis-linearizability in head-rectified, single-IDB Datalog programs in undecidable even if the program has one bilinear rule, an unbounded number of linear rules and only 5 basis rules. Our treatment also shows that program containment is undecidable for a restricted class of linear Datalog programs.

Finally, in Section 1.5, we provide polynomial-space and polynomial-time algorithms that are sufficient (but not necessary) for the detection of basis-linearizability in arbitrary single-IDB Datalog programs.

1.4.4 Sequencability: definition and results

Define a top-down expansion generated by $\mathcal{P}$ to be *sequenced* if a recursive atom generated by the (top-down) application of a recursive rule r_i is never expanded through the rule r_j, if $i > j$ (see Figure 1.12). The program $\mathcal{P}$ is said to be *sequencable* iff sequencing is a normal form for the proof trees generated by the program; that is, iff every closed conjunctive query generated by $\mathcal{P}$ is contained in some closed, sequenced conjunctive query. If $\mathcal{P}$ is sequencable, then it may be replaced by the following program $\mathcal{Q}$.

Let $q_1 \ldots q_n$ be new and distinct predicate symbols, and construct the program $\mathcal{Q}$ from program $\mathcal{P}$, as follows. First, replace the rule r_1 by the two rules r_1' and s_1, where r_1' is the same as r_1:

r_1' : $p(\vec{X}_{10}) :- p(\vec{X}_{11}), \ldots p(\vec{X}_{1k_1}), \mathcal{C}_1$.
s_1 : $p(V_1, \ldots, V_k) :- q_1(V_1, \ldots, V_k)$.

The Vs are distinct variables. Next, replace each recursive rule r_i ($i > 1$) by the two rules

r_i' : $q_{i-1}(\vec{X}_{(i-1)0}) :- q_{i-1}(\vec{X}_{i1}), \ldots q_{i-1}(\vec{X}_{ik_i}), \mathcal{C}_i$.
s_i : $q_{i-1}(V_1, \ldots, V_k) :- q_i(V_1, \ldots, V_k)$.

As before, the Vs are distinct variables. Finally, replace each nonrecursive rule b_i by the rule

b_i' : $q_n(\vec{Y}_{i0}) :- \mathcal{D}_i$.

$\mathcal{Q}$ computes those facts which would be produced by a bottom-up evaluation of $\mathcal{P}$, in which we initialise p using the b_i, and then, in sequence, close under $r_n, r_{n-1}, \ldots, r_1$. It is easily

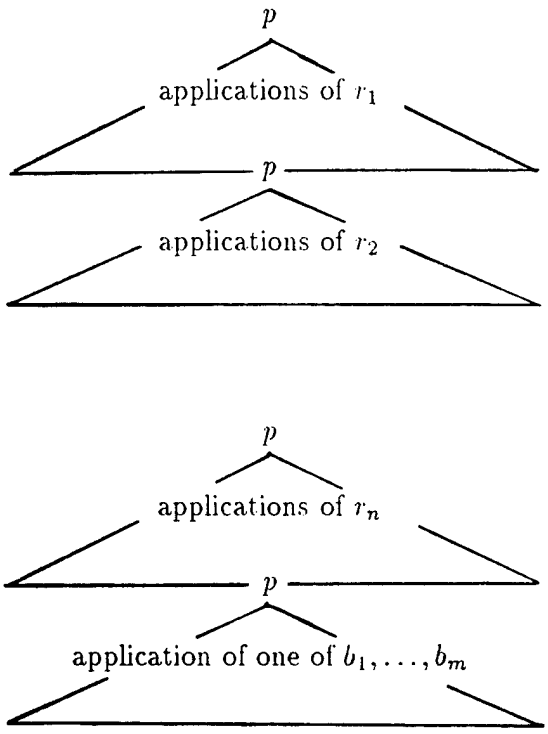


Figure 1.12: A sequenced expansion.

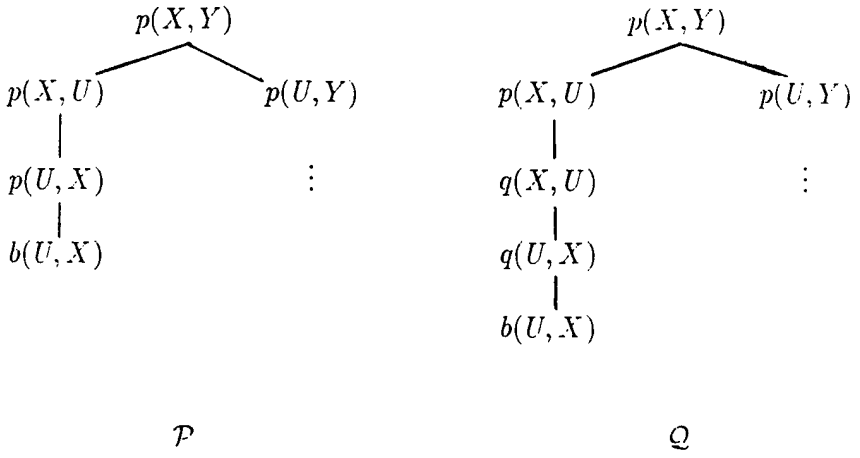


Figure 1.13: Illustrating Example 1.18.

seen that the normal-form closed conjunctive queries of $\mathcal{P}$ are isomorphic to the closed conjunctive queries of $\mathcal{Q}$. Therefore, if $\mathcal{P}$ is sequencable, then $\mathcal{P}$ and $\mathcal{Q}$ are equivalent.

Example 1.18 For the program of Example 1.4, the program $\mathcal{Q}$ is the program

r_1' : $p(X,Y) :- p(X,U), p(U,Y).$
s_1 : $p(X,Y) :- q(X,Y).$
r_2' : $q(X,Y) :- q(Y,X).$
s_2 : $q(X,Y) :- b(X,Y).$
b_1' : $q_2(X,Y) :- c(X,Y).$

Figure 1.13 sketches the idea behind the isomorphism between the closed sequenced expansions of $\mathcal{P}$ and the closed expansions of $\mathcal{Q}$. □

The gains of replacing $\mathcal{P}$ by $\mathcal{Q}$ are obtained in several ways. For example, the evaluation of the transformed program $\mathcal{Q}$ can be pipelined. Most importantly, however, the detection of sequencability can often set up further optimizations, as in Example 1.8. Sequencability is also essential to the detection of *separability* ([24]) in linear programs.

Results

Sequencability is not known to be decidable for any interesting classes of programs. In Chapter 4, we will show that the detection of sequencability is undecidable for head-rectified, single-IDB Datalog programs with only two recursive rules and 9 basis rules. Our treatment provides a tight undecidability result for program equivalence.

Sequencability has only been studied for linear Datalog programs with two recursive rules and a single basis rule, and the decidability of sequencability is open even on this restricted domain. A variety of sufficient conditions have been proposed ([25], [17]) in this case. In Chapter 2, we show that sequencability is $\mathcal{NP}$-hard for such programs if EDB predicates are allowed to appear up to 3 times in the body of each recursive rule, and in fact that all the proposed sufficient conditions are also $\mathcal{NP}$-hard in this case. We also show that a popular sufficient condition is polynomial if no EDB predicates are allowed to repeat in any recursive rule.

Finally, we provide in Section 1.5 a polynomial-space algorithm that is sufficient (but not necessary) for the detection of sequencability in arbitrary Datalog programs.

1.4.5 Subtree eliminations as a descriptive mechanism

The normal forms implied by these problems are susceptible to a uniform description, as the elimination of subtrees of a certain shape from the proof trees generated by a program. As a dual to the concept of a closed top-down expansion, let us define a top-down expansion to be *open* iff the only rules used to construct the expansion are recursive rules.

Definition 1.3 Let $\mathcal{P}$ be the canonical single-IDB program of the preceding subsection. A *subtree elimination instance* is the program $\mathcal{P}$ and a finite set S of finite shapes over the recursive rules $r_1 \ldots r_n$ (that is, the shapes define open expansions). Let $\mathcal{Q}$ be the set of closed top-down expansions generated by $\mathcal{P}$, such that no subtree of the expansion is described by a shape in S. Then the answer to the instance is "yes" iff $\mathcal{P} \equiv \mathcal{Q}$ (recall that we speak of any union of closed top-down expansions as a program). □

The set of such instances is called the subtree-elimination problem (SEP). Since $\mathcal{Q}$ is a subset of the conjunctive queries generated by $\mathcal{P}$, and by the theorem of Sagiv and Yannakakis (Theorem 1.2), we may conclude that the answer to an SEP instance is "yes" iff each conjunctive query generated by $\mathcal{P}$ is contained in some conjunctive query generated by $\mathcal{Q}$.

The transformations of the previous sections may all be described within the framework of SEP. The idea is that the violations of the required normal form are represented as subtree shapes.

One-boundedness

Let $\mathcal{P}$ be the canonical single-IDB program of this section. Let a *minimum-depth violation of one-boundedness* (or just a *violation*) be an open top-down expansion of depth 2, and let $S = \{s_1, \ldots, s_n\}$ be the set of shapes of the minimum-depth violations. It is easily seen that the set of one-bounded top-down expansions generated by $\mathcal{P}$ is precisely the set of expansions in which no subtree has the shape s_i for any i. Thus, $\mathcal{P}$ is one-bounded iff the answer to the SEP instance $< \mathcal{P}, S >$ is "yes".

Example 1.19 For the program of Example 1.2, the set S is the singleton $\{[r_1[r_1[]]]\}$, representing the single minimum-depth violation of Figure 1.14. □

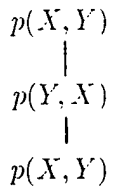


Figure 1.14: Minimum-depth violation of one-boundedness.

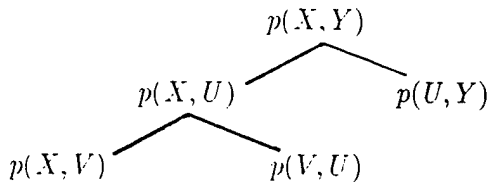


Figure 1.15: Minimum-depth violation of basis-linearizability.

Linearizability by basis

Let a *minimum-depth violation of right-linearity* (or just a *violation*) be an open top-down expansion of depth 2, in which the rightmost recursive subgoal that is a child of the root is not expanded, and in which at least one other child of the root is expanded. Let $S = \{s_1, \ldots, s_n\}$ be the set of shapes of the minimum-depth violations. It is easily seen that the set of right-linear top-down expansions generated by $\mathcal{P}$ is precisely the set of expansions in which no subtree has the shape s_i for any i. Thus, $\mathcal{P}$ is linearizable by basis iff the answer to the SEP instance $< \mathcal{P}, S >$ is "yes".

Example 1.20 For the program of Example 1.3, the set S is the singleton $\{[r_1[r_1[][]][]]\}$, representing the single minimum-depth violation of Figure 1.15. □

Sequencability

Let a *minimum-depth violation of sequencability* (or just a *violation*) be an open top-down expansion of depth 2, in which if r_i is used to expand the root, then at least one child of the root is expanded through some rule r_j such that $j < i$. Let $S = \{s_1, \ldots, s_n\}$ be the set of shapes of the minimum-depth violations. It is easily seen that the set of sequenced top-down expansions generated by $\mathcal{P}$ is precisely the set of expansions in which no subtree has the shape s_i for any i. Thus, $\mathcal{P}$ is sequencable iff the answer to the SEP instance $< \mathcal{P}, S >$ is "yes".

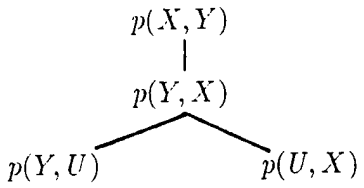


Figure 1.16: Minimum-depth violation of sequencability.

Example 1.21 For the program of Example 1.4, the set S is the singleton $\{[r_2[r_1[][]]]\}$. The corresponding minimum-depth violation is depicted in Figure 1.16. □

1.5 Subtree-elimination algorithms

We now present a uniform framework for the generation of sufficient conditions for the solution of an SEP instance. In each case, the program that we consider is the generic single-IDB program $\mathcal{P}$ of Section 1.4.1.

1.5.1 Basis-independence

Recall that an expansion is termed open if only recursive rules are used in the construction of the expansion, and closed if all the leaves are EDB predicates. Our sufficient conditions will focus on the rectification of open top-down expansions; that is, we will show that every open expansion generated by $\mathcal{P}$ is contained in a normal-form open expansion. This procedure is sufficient to prove that every closed top-down expansion is contained in a closed, normal-form expansion, as we show below; the only wrinkle is introduced if the containing expansion has depth 0. Recall that the top-down expansion of depth 0 (the expansion with shape []) is the top-down expansion generating the conjunctive query

$$p(X_1, \ldots, X_k) :- p(X_1, \ldots, X_k).$$

where the Xs are distinct variables. Now, assume that an expansion T_1 is contained in [], and that f is the containment mapping proving the containment. By the properties of the containment mapping, T_1 must have some leaf that is syntactically identical to its root.

Theorem 1.5 Assume that T_1 is an open top-down expansion that is contained in the expansion of depth 0. Then every closed top-down expansion obtained by applying basis rules to the leaves of T_1 is contained in an initialisation rule.

Proof. Since T_1 is contained in the depth-0 expansion, there must be some leaf (labelled l, say) that is syntactically identical to the root. Hence, if the basis rule b_i is used to expand $node(l)$ in T_1, then the result is contained in b_i by the Expansion Theorem. □

By our assumption that all violations in our SEP instance are open, we conclude that each basis rule (considered as a top-down expansion) is in the indicated normal form; that is, no such expansions are prohibited. Now, every closed top-down expansion is either a basis rule, or is obtained by expanding every p-leaf in an open expansion through some basis rule. Hence, by the Expansion Theorem and Theorem 1.5, the rectification of all open expansions is sufficient to prove that the answer to a given SEP instance is "yes"; that is, that every closed top-down expansion is contained in a closed, normal-form expansion. Note that this result holds independently of the set of initialisation rules in $\mathcal{P}$; that is, the transformations that we will consider are *basis-independent.*

The following theorem shows that if an expansion T is contained in the empty expansion, then the result of expanding any leaf of T through the expansion U is contained in either the empty expansion or in U.

Theorem 1.6 Assume that T_1 is an open top-down expansion that is contained in the expansion of depth 0 (the expansion with shape []). Let T_3 be the expansion obtained by expanding $node(l)$ in T_1 through the expansion T_2. Then, either $T_3 \subset []$, or there is a containment mapping from T_{21} (a replica of T_2) into T_3 such that for any leaf $node([1|j])$ in T_{21}, the destination of $node([1|j])$ is $node(j)$ in T_3.

Proof. Let f be a containment mapping from [] into T_1. As before, there is some leaf (say, $node(k)$) in T_1 that is syntactically identical to the root of T_1.. There are two cases. If $k \neq l$, then f is a containment mapping from [] into T_3. If $k = l$, then the subtree rooted at $node(l)$ in T_1 is $\tau(T_2)$, where τ is the mgu of the root of T_2 with $node(l)$ in T_1; hence, $\tau(ren_1^{-1})$ is a containment mapping from T_{21} into T_3, where ren_1 is the *rename* function that creates T_{21} from T_2. □

1.5.2 Generating sufficient conditions

Ramakrishnan, Sagiv, Ullman and Vardi ([25]) have proposed a framework for the construction of conditions that are sufficient (but not necessary) to prove that the proof trees of a program satisfy a normal form. Let $< \mathcal{P}, S >$ be an SEP instance, where $S = \{v_1, \ldots, v_k\}$ is a complete set of violations to the desired normal form. The process of [25] consists of two steps:

1. For each v_i, show that there is a containment mapping of a restricted form from some normal-form top-down expansion q_i into v_i.

2. Show that the results of (1) may be used to inductively rectify all top-down expansions of $\mathcal{P}$.

Such a technique is called a *proof-tree transformation technique.*

One-boundedness

By Theorem 1.5 and the Expansion Theorem, the containment of every open expansion of depth at least two in an open expansion of depth at most one suffices to prove one-boundedness in $\mathcal{P}$.

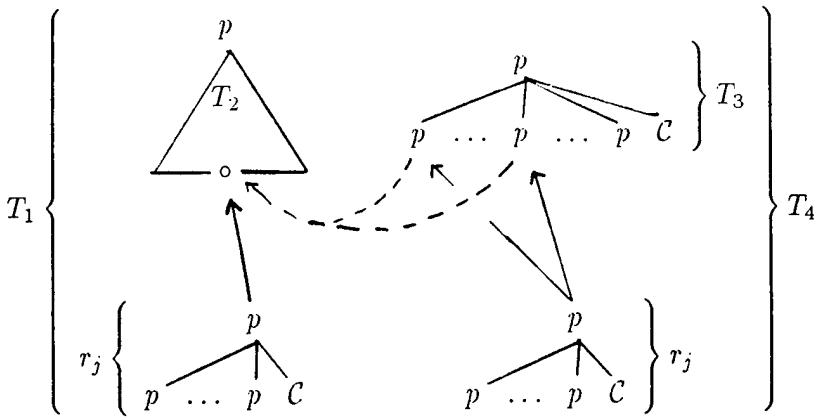


Figure 1.17: Illustrating Theorem 1.7.

Theorem 1.7 Assume that every minimum-depth violation of one-boundedness is contained in an open expansion of depth at most 1. Then $\mathcal{P}$ is one-bounded.

Proof. By induction on the number i of rule applications in the top-down expansion, we will show that every expansion of depth at least 2 is contained in an expansion of depth at most 1. The basis, $i = 2$, follows by assumption. For the induction, assume the truth of the hypothesis for $2 \leq l < i$.

Consider a top-down expansion T_1 obtained through i rule applications. T_1 is obtained by expanding some leaf $node(n)$ in some expansion T_2 (constructed using $i-1$ rule applications) through some rule r_j (see Figure 1.17). By our inductive hypothesis, T_2 is contained in a tree T_3 of depth at most one; assume that f is a containment mapping proving the containment. If T_3 has depth 0, then by Theorem 1.6, $T_3 \subset []$ or $T_3 \subset r_j$, and our result follows.

Assume that T_3 has depth 1. By the Expansion Theorem, we may expand each node in T_3 whose destination under f is $node(n)$ through r_j, to produce an expansion T_4 of depth 2 such that $T_1 \subset T_4$ (see Figure 1.17). However, by assumption, T_4 is contained in a top-down expansion of depth at most 1, and our result follows because the composition of containment mappings is a containment mapping. □

Example 1.22 Consider the program of Example 1.2. The minimum-depth violation of one-boundedness is contained in the top-down expansion of depth 1, under the identity mapping $f(X) = X, f(Y) = Y$ (see Figure 1.18). Hence, the program is one-bounded. □

A similar procedure on *closed* top-down expansions may be used to show that one-boundedness is decidable. That is, $\mathcal{P}$ is one-bounded iff every closed expansion of depth 3 is contained in a closed expansion of depth at most 2. Our treatment has, however, been motivated by the interests of exposition, and by the setting up of an efficient algorithm for the detection of one-boundedness; the latter is contained in the next section.

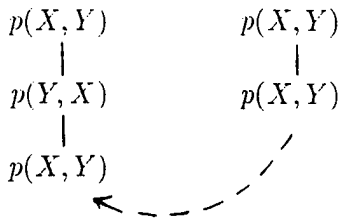


Figure 1.18: Proving one-boundedness.

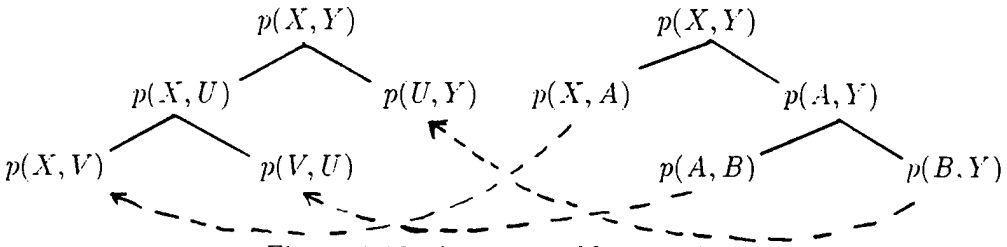


Figure 1.19: An acceptable mapping.

Linearizability by basis

By definition, the open expansion of depth 0 is right-linear. By Theorem 1.5 and the Expansion Theorem, if we show that every non-right-linear open expansion is contained in a right-linear open expansion, then we may conclude that $\mathcal{P}$ is basis-linearizable. The following treatment is an extension of a result of [25]; they consider a Datalog program with a single recursive rule, which is bilinear.

Let T_1 and T_2 be open expansions, let *node*(l) be the rightmost p-leaf in T_1 and let *node*(k) be the rightmost p-leaf in T_2. We say that a containment mapping $f : T_2 \rightarrow T_1$ is *acceptable for right-linearity* (or just *acceptable*) iff *node*(l) is the destination (under f) of no leaf, or is the destination of *node*(k) only.

Example 1.23 Consider the program of Example 1.3. Figure 1.19 shows that the minimum-depth violation of right-linearity is contained in a right-linear expansion under the acceptable mapping $f(X) = X, f(Y) = Y, f(A) = V, f(B) = U$. □

Theorem 1.8 Assume that every minimum-depth violation of right-linearity is contained in a right-linear expansion, and that the containment is provable by an acceptable containment mapping. Then $\mathcal{P}$ is basis-linearizable.

Proof. By induction on the number i of rule applications in the top-down expansion, we show that every non-right-linear expansion is contained in a right-linear expansion. The basis, $i = 0$ (representing the top-down expansion of depth 0), is trivial. For the induction, assume the truth of the hypothesis for $0 \leq l < i$, where $i > 0$.

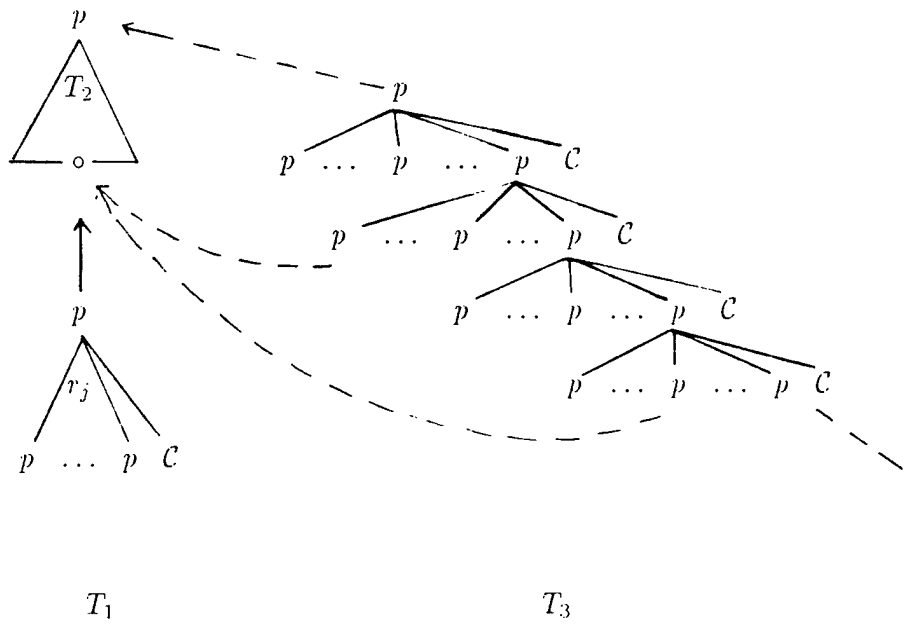


Figure 1.20: T_1 in Theorem 1.8.

Consider a top-down expansion T_1 obtained through i rule applications. If T_1 is right-linear, then the result follows; if T_1 is a minimum-depth violation, then the result follows by assumption. Otherwise, T_1 is obtained by expanding some leaf $node(n)$ in some expansion T_2 (constructed using $i-1$ rule applications) through some rule r_j (see Figure 1.20).

By our inductive hypothesis, T_2 is contained in a right-linear tree T_3. If T_3 has depth 0, then our result follows by Theorem 1.6.

Assume that T_3 has depth at least 1. By the Expansion Theorem, we may expand each node in T_3 whose destination under f is $node(n)$ through r_j, to produce an expansion T_4 such that $T_1 \subset T_4$. If T_4 is right-linear, the result follows. Otherwise, T_4 is a tree that is obtained from some right-linear tree by expanding some non-rightmost p-leaves in the right-linear tree through the rule r_j. We call such an expansion *almost right-linear*.

We may rectify T_4 in a bottom-up manner, using the Splicing Theorem and the fact that the composition of containment mappings is a containment mapping, as follows. At any stage in the process, we have a situation as shown in Figure 1.21; that is, we have an almost-right-linear tree T_5 whose rightmost p-node is expanded through a violation V, such that the rightmost p-node of V is a leaf or is expanded through some right-linear tree T_6.

By assumption, there is an acceptable mapping from some right-linear expansion R into V. Hence, by the Splicing Theorem, we may splice R in for V; the p-leaves of R remain

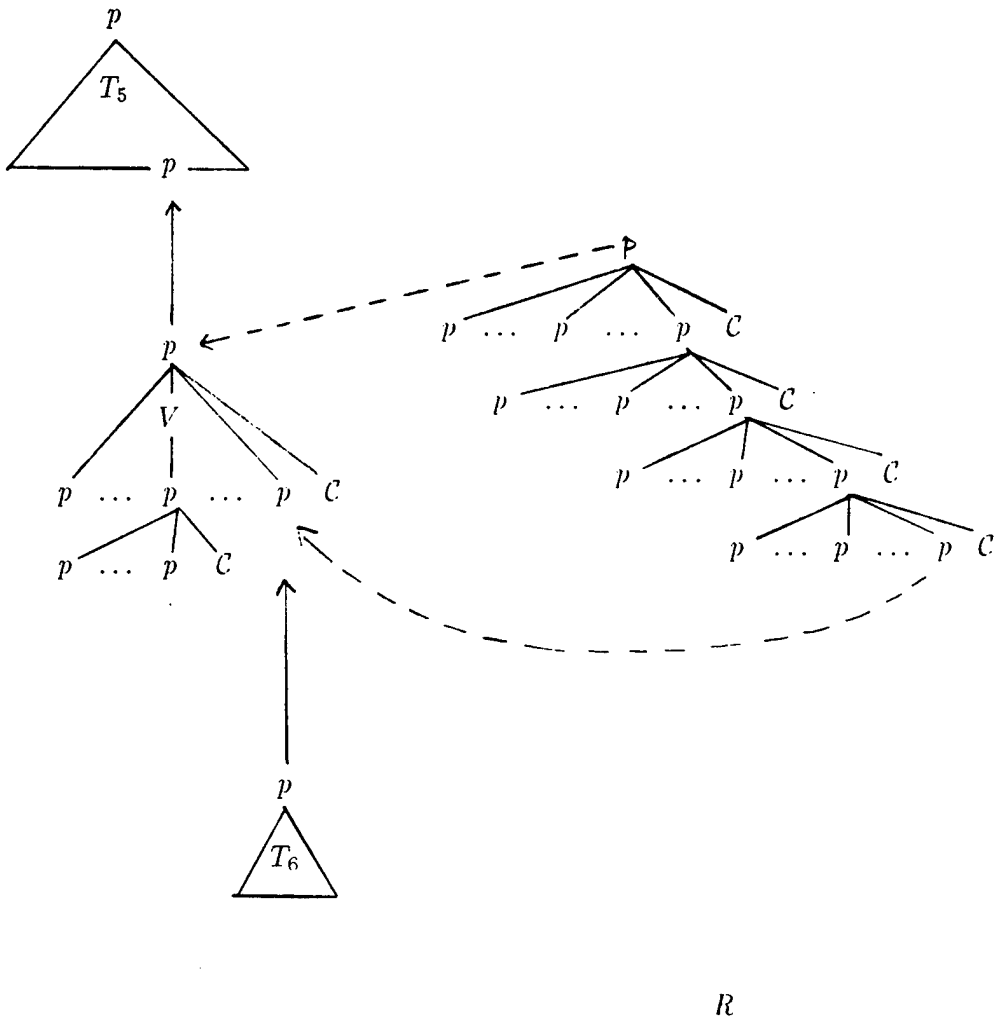


Figure 1.21: The rectification procedure of Theorem 1.8.

leaves, except for the rightmost p-leaf in R which may be expanded through the right-linear tree T_6. An inductive repetition suffices to complete the rectification. □

By the observations of Example 1.23, we may now claim that the transitive-closure program of Example 1.3 is basis-linearizable.

Complexity If $\mathcal{P}$ is a Datalog program, then the minimum-depth violations may be constructed in polynomial time, and the containments under acceptable mappings may be tested in polynomial space by the chase algorithm of [25]. In the non-Datalog case, the chase may not terminate; however, we may generate sufficient conditions by placing a bound on the depth of the containing right-linear expansion. A suitable heuristic may be one based on size preservation; that is, we may insist that the containing query be no bigger than the contained expansion.

Sequencability

By convention, the open expansion of depth 0 is sequenced. By Theorem 1.5 and the Expansion Theorem, if we show that every non-sequenced open expansion is contained in a sequenced open expansion, then we may conclude that $\mathcal{P}$ is sequencable. The following result is an extension of an algorithm proposed independently by [25] and [17]; they consider Datalog programs with only two recursive rules, both of which are linear.

For any $i < j$, define an *r_i-r_j violation of sequencability* (or just an r_i-r_j violation) to be an open expansion of depth 2, such that r_j is used to expand the root, one or more of the children are expanded through r_i, and all other children of the root are leaves.

Let us further define an *r_i-r_j-sequenced expansion* to be a sequenced expansion using only the rules r_i and r_j such that r_i is used at most once (that is, r_i can only be used to expand the root).

Example 1.24 Consider the program of Example 1.4. The minimum-depth violation of sequencability in this program is the expansion $[r_2[r_1[][]]]$ (see Figure 1.22); this expansion is an r_1-r_2 violation. The minimum-depth violation is contained in the r_1-r_2-sequenced expansion $[r_1[r_2[]][r_2[]]]$, as shown in the figure; the containment follows because both expansions have the same root and leaves. □

Let us define a *node violation of* r_i to be the root of an r_i-r_j violation for any j.

Theorem 1.9 Assume that for all i and j, each r_i-r_j violation is contained in some r_i-r_j-sequenced expansion. Then $\mathcal{P}$ is sequencable.

Proof. By induction on the number m of rule applications in a top-down expansion T generated by $\mathcal{P}$, we prove that T is contained in a sequenced expansion. The proof is similar to that of Theorem 1.8; we merely sketch the inductive step. Assume that the mth rule applied is r_i. By the inductive hypothesis and the Expansion Theorem, T is contained in some top-down expansion T', which is obtained by expanding some leaves in a sequenced expansion through r_i. We inductively rectify T' using the Splicing Theorem, by "bubbling

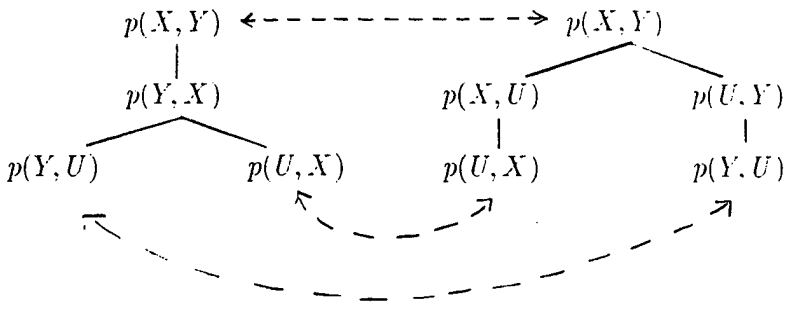


Figure 1.22: Proving sequencability.

up" the node violations of r_i as follows. Consider any node violation at maximum depth; assume it is the root of the r_i-r_j violation V, in which some leaves may be expanded through sequenced expansions in which every rule used is of the form r_k for $k \geq j$ (where $j > i$). By assumption, V is contained in an r_i-r_j-sequenced expansion S. By the Splicing Theorem, we may splice S in for V; this process either reduces the number of node violations of r_i by 1 or replaces the node violation by a node violation at a smaller depth. Further, the splicing-in of S for V does not create any r_k-r_l-violations for $k \neq i$. Hence, we may proceed in stages, in each stage removing all node violations of r_i at maximum depth, until the tree T' is sequenced. The situation is depicted in Figure 1.23. In the figure, $R_1 \ldots R_k$ are sequenced subtrees in which the rules $r_1, \ldots, r_i$ are not used. □

By the observations of Example 1.24, we may conclude that the program of Example 1.4 (computing the symmetric, transitive closure) is sequencable.

Complexity As before, the tests of Theorem 1.9 may be performed in polynomial space for Datalog programs through the chase algorithm of [25]. In the non-Datalog case, we may construct sufficient conditions by placing a bound on the depth of each containing r_i-r_j-sequenced expansion, perhaps by using the size-maintaining heuristic that was presented in the previous subsection.

1.5.3 Fast algorithms

Finally, we present restricted algorithms for the detection of one-boundedness and basis-linearizability. In Chapter 2, we will show that for Datalog programs, these algorithms are polynomial in the size of $\mathcal{P}$; they perform polynomial-time reductions to restricted programs for which the detection of the appropriate normal form is known to be decidable in polynomial time. The idea behind the algorithms is to further restrict the destinations of an atom under a mapping that proves the containment of a violation in a normal-form

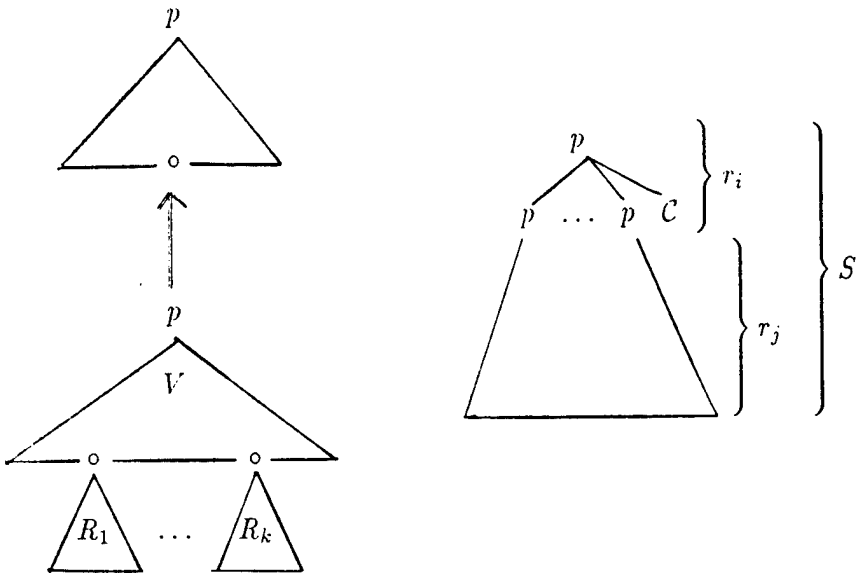


Figure 1.23: The rectification process of Theorem 1.9.

expansion. That is, we essentially rename different occurrences of the recursive atoms in each rule.

Let us assume that for any i and j, the jth subgoal in the rule i is given the superscript ij, and that this superscript is carried through all top-down expansions[6]. That is, if r_i is used to expand a p-atom and the jth subgoal has principal functor q, then the jth child of the expanded atom is referred to as a *q^{ij}-atom* (or just q^{ij}). The root of the expansion, and the body of the top-down expansion of depth 0, are merely p-atoms. Figure 1.24 illustrates this notation on the non-right-linear top-down expansion of Figure 1.19, representing the minimum-depth violation of basis-linearizability in the program of Example 1.3.

One-boundedness

For all i, define a *violation of degree i* to be a minimum-depth violation of one-boundedness in which exactly i children of the root are expanded. Now, for any open top-down expansions T_1 and T_2, we say that a containment mapping $f : T_1 \rightarrow T_2$ is *restricted* if the destination of any q^{ij}-atom in T_1 under f is a q^{ij}-atom in T_2. Note that the number and size of the violations of degree 1 are polynomial in the size of $\mathcal{P}$, and that each violation can be constructed in polynomial time.

[6]Note that this concept is distinct from the idea of labels in a top-down expansion

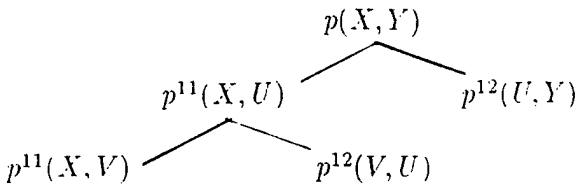


Figure 1.24: Notation.

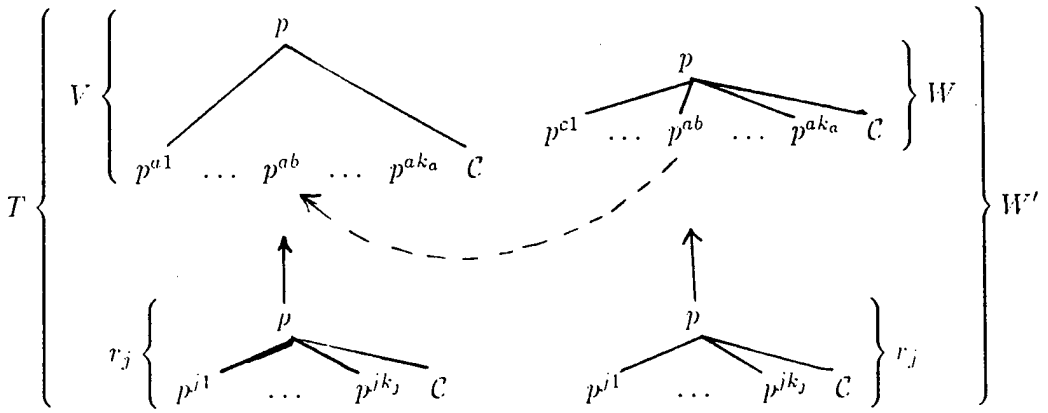


Figure 1.25: W' in Theorem 1.10.

Theorem 1.10 Assume that every violation of degree 1 is contained in a top-down expansion of depth no more than 1, and that the containments are proved by restricted mappings. Then, $\mathcal{P}$ is one-bounded.

Proof. We prove by induction on i that every violation of degree i is contained in a tree of depth at most one, and that the containment is proved by a restricted mapping; our result then follows by Theorem 1.7. The basis, $i = 1$, follows by assumption.

Now, assume the truth of the inductive hypothesis for $1 \leq l < i$. Consider any violation T of degree i. This violation must be obtained by expanding some p^{ab}-leaf at depth 1 in some violation V of degree $i - 1$ through some rule r_j. By our inductive hypothesis, V is contained in some expansion W of depth at most 1 under a restricted containment mapping.

If the depth of W is 0, then by Lemma 1.6 and the Expansion Lemma, T is contained in [] or r_j under a restricted mapping.

If the depth of W is 1, then let W' be the result of expanding the p^{ab}-leaf in W through the rule r_j (see Figure 1.25). By the Expansion Lemma, T is contained in W' under a restricted mapping. However, W' is a violation of degree 1, and is (by assumption) contained in a tree of depth at most 1 under a restricted mapping; our result follows because

the composition of restricted mappings is a restricted mapping. □

This theorem leads to the following algorithm that is sufficient (but not necessary) to prove one-boundedness.

Algorithm 1.1
INPUT: A single-IDB program $\mathcal{P}$.
OUTPUT: "yes" only if $\mathcal{P}$ is one-bounded.

(1) Construct all violations of degree 1, and all expansions of depth at most 1.

(2) If each violation of degree 1 is contained in an expansion of depth at most 1 under a restricted mapping, then answer "yes"; otherwise, answer "no".

□

There are a polynomial number of degree-1 violations, so Step (1) may be accomplished in polynomial time. Step (2) involves a polynomial number of containment tests, each involving the existence of a restricted mapping from an expansion of depth 0 or 1 into a violation. Testing for the existence of a mapping from a depth-0 expansion into any expansion is clearly polynomial. Consider tests of the second kind; that is, tests for the existence of a restricted mapping from a depth-1 expansion T into a violation V. Each atom in T has at most 2 possible destinations in V under a restricted mapping, and we will show in Chapter 2 that such tests may be accomplished in polynomial time. Hence, Algorithm 1.1 is in $\mathcal{P}$.

Basis-linearizability

A similar process may be applied to create sufficient conditions for the detection of basis-linearizability.

As before, we define a *violation of degree i* to be a minimum-depth violation of right-linearity in which exactly i children of the root are expanded. Now, for any violation T_1 and right-linear expansion T_2, we say that a containment mapping $f : T_1 \rightarrow T_2$ is *restricted* if for any q^{ij}-leaf in T_1 that is not the rightmost p-leaf in T_1, the destination of this q^{ij}-atom in T_1 under f is a q^{ij}-atom in T_2. Note that a restricted containment mapping is acceptable for right-linearity. Again, we observe that the number and size of the violations of degree 1 are polynomial in the size of $\mathcal{P}$, and that each violation can be constructed in polynomial time.

Theorem 1.11 Assume that every violation of degree 1 is contained in a right-linear top-down expansion, and that the containments are proved by restricted mappings. Then, $\mathcal{P}$ is basis-linearizable.

Proof. We prove by induction on i that every violation of degree i is contained in a right-linear expansion, and that the containment is proved by a restricted mapping; our result then follows by Theorem 1.8. The basis, $i = 1$, follows by assumption.

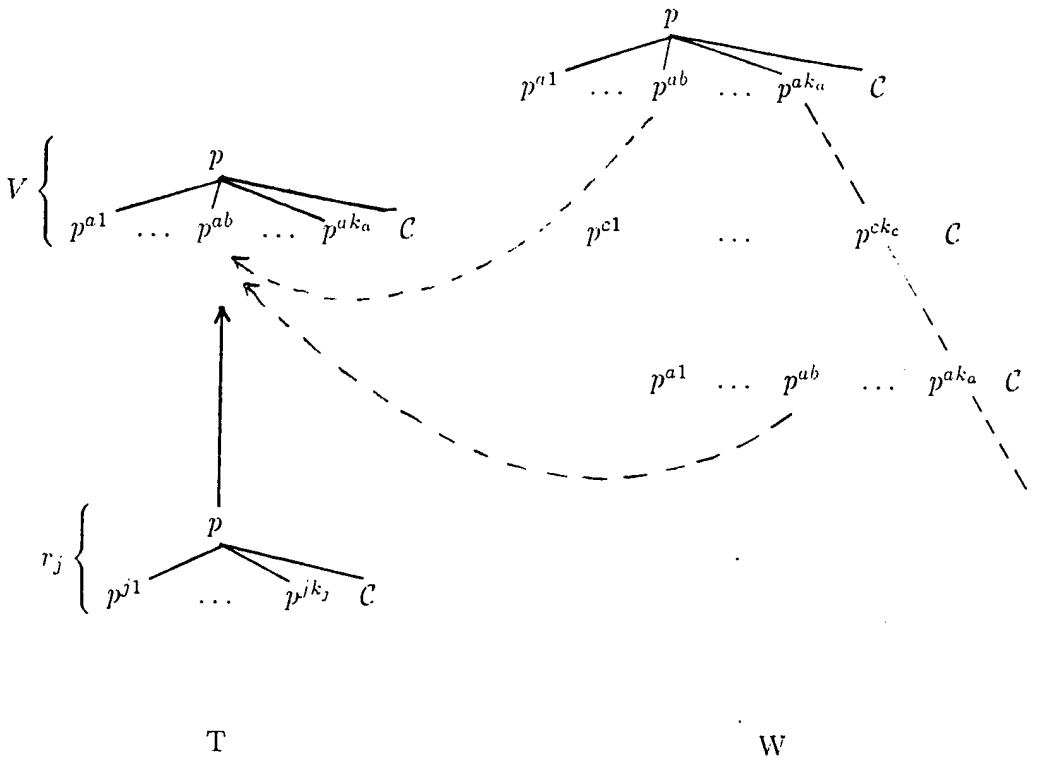


Figure 1.26: W in Theorem 1.11.

Now, assume the truth of the inductive hypothesis for $1 \leq l < i$. Consider any violation T of degree i. This violation must be obtained by expanding some p^{ab}-leaf at depth 1 in some violation V of degree $i-1$, such that this leaf is not the rightmost child of the root, through some rule r_j (see Figure 1.26). By our inductive hypothesis, V is contained in some right-linear expansion W under a restricted containment mapping.

If the depth of W is 0, then by Theorem 1.6, T is contained in r_j or [] under a restricted mapping.

If the depth of W is 1 or larger, then let W' be the result of expanding every p^{ab}-leaf in W whose destination is the newly-expanded leaf in V through the rule r_j. By the Expansion Theorem, T is contained in W' under a restricted mapping. Note that W' is obtained by expanding some p^{ab}-leaves in a right-linear expansion through r_j; that is, at any stage of the tree, at most two sibling p-atoms are expanded. By assumption, each violation of degree 1 is contained in a right-linear expansion under a restricted mapping, and a bottom-up rectification as in the proof of Theorem 1.8 serves to complete the proof. □

This theorem yields the following sufficient (but not necessary) condition for the detection of basis-linearizability. We will show in Chapter 2 that the algorithm is polynomial:

the key is to test whether every degree-1 violation is contained in a right-linear tree *of depth at most* 2 under a restricted mapping[7].

Algorithm 1.2
INPUT: A single-IDB program $\mathcal{P}$.
OUTPUT: "yes" only if $\mathcal{P}$ is basis-linearizable.

(1) Construct all violations of degree 1, and all right-linear expansions of depth at most 2.

(2) If each violation of degree 1 is contained in a right-linear expansion of depth at most 2 under a restricted mapping, then answer "yes"; otherwise, answer "no".

□

There are a polynomial number of degree-1 violations and a polynomial number of right-linear trees of depth at most 2, so Step (1) may be accomplished in polynomial time. Step (2) involves a polynomial number of containment tests, each involving the existence of a restricted mapping from a right-linear expansion of depth 0, 1 or 2 into a violation. As we mentioned in our treatment of one-boundedness in the previous subsection, testing for the existence of a mapping from a depth-0 expansion into any expansion is polynomial. Consider tests for the existence of a restricted mapping from a depth-1 or depth-2 right-linear expansion T into a violation V. Let N be the size of V: remember that N is polynomial in the size of the program $\mathcal{P}$. By the definition of a restricted mapping, one atom (say, a) in T has up to N possible destinations in V, and each other atom has at most 2 possible destinations. Hence, by a case analysis on the destinations of a, we may reduce the test to N tests for the existence of a mapping from T into V such that each atom in T has at most 2 allowed destinations in V. We will show in Chapter 2 that such tests may be accomplished in polynomial time. Hence, Algorithm 1.2 is in $\mathcal{P}$.

Sequencability

A similar algorithm may be devised for the detection of sequencability, but we omit the treatment in the interest of brevity.

1.6 Overview of Chapters 2, 3 and 4

In the remainder of this report, we will investigate the complexity of the subtree elimination problem, focussing on the detection of one-boundedness, basis-linearizability and sequencability.

In Chapter 2, we investigate the complexity of detecting containments among conjunctive queries. We extend the conjunctive query containment problem to the *k-containment* problem, and show that k-containment and $kSAT$ are essentially the same problem. This investigation results in a complete description of the complexity of conjunctive query containment. These results are then extended to show that one-boundedness, basis-linearizability

[7]The algorithm is polynomial for any given choice of depth for the right-linear trees.

	Polynomial time	$\mathcal{NP}$-hard	Undecidable
One-boundedness	linear sirup, ≤ 1 reps	linear sirup, ≤ 4 reps	never
Basis-linearizability	bilinear sirup, ≤ 1 reps	bilinear sirup, ≤ 4 reps	1 nonlinear rule, 5 basis rules.
Sequencability	???	2 recursive rules (both linear), ≤ 3 reps, 1 basis rule	2 recursive rules, 9 basis rules.

and sequencability are $\mathcal{NP}$-hard for restricted classes of programs. In this chapter, we also prove that the sufficient conditions of Section 1.5.3 (Algorithms 1.1 and 1.2) are in $\mathcal{P}$.

In Chapter 3, we provide a decision procedure for the detection of basis-linearizability in a class of recursive programs. The decision procedure can be seen to be polynomial using the techniques of Chapter 2.

Finally, in Chapter 4, we show that sequencability and basis-linearizability are undecidable for multi-rule, nonlinear programs. The techniques of this chapter also provide tight undecidability results for the detection of program equivalence.

1.6.1 Complexity results

The results of our investigation are presented in Table 1.1. The programs considered are all Datalog. The expression $\leq i$ *reps* means that each recursive rule in the program has at most i occurrences of an EDB predicate in the body of each recursive rule.

Chapter 2

The complexity of conjunctive query containment

2.1 Introduction

In this chapter, we will characterize the complexity of testing containments among pairs of conjunctive queries. Recall that a conjunctive query is a single-rule, nonrecursive program, and that the theorem of Sagiv and Yannakakis (Theorem 1.2) relates containments among recursive programs to containments among the conjunctive queries generated by these programs.

In Section 2.2, we characterize the complexity of the conjunctive query containment problem. Our results are obtained by defining and analysing a closely related problem, which we term the *k-containment problem*. We also show that a restricted version of the 2-containment problem is in $\mathcal{NC}$, and is hence efficiently computable in parallel.

In Section 2.3, we extend the results of Section 2.2 to provide $\mathcal{NP}$-hardness results for the one-boundedness, sequencability and basis-linearizability problems. We also justify the title of Section 1.5.3 by showing that the algorithms of that section are in $\mathcal{P}$, as we claimed.

2.2 The k-containment problem

Let us consider the complexity of the conjunctive query containment problem. In this section, we will define the k-containment problem as a parametrized version of the conjunctive query containment problem, and show that while the 2-containment problem is in $\mathcal{P}$, the 3-containment is $\mathcal{NP}$-complete. We also show that for Datalog (that is, function-free) conjunctive queries, the 2-containment problem is in NLOGSPACE (and hence in $\mathcal{NC}$).

2.2.1 Conjunctive query containment

Recall from Section 1.3.1 that a conjunctive query is a single-rule, nonrecursive program. That is, a conjunctive query is of the form

$$p(\vec{X}) :- \mathcal{B}.$$

where $\mathcal{B}$ is an arbitrary conjunction of atomic formulae. Strictly speaking, the predicate p must not occur within $\mathcal{B}$; however, we will ignore this restriction when it is clear from the context that such an expression is to be applied nonrecursively.

Consider the (not necessarily function-free) conjunctive queries

$C_1 : a_0(\vec{U}_0) :- a_1(\vec{U}_1), \ldots a_m(\vec{U}_m).$
$C_2 : b_0(\vec{W}_0) :- b_1(\vec{W}_1), \ldots b_l(\vec{W}_l).$

We assume without loss of generality that there are no repetitions of any atomic formula in the body of C_1, or in the body of C_2 (although predicates may appear repeatedly). That is, for no i and j, $i \neq j$, are $a_i(\vec{U}_i)$ and $a_j(\vec{U}_j)$ syntactically identical (or $b_i(\vec{W}_i)$ and $b_j(\vec{W}_j)$ identical). Subgoal repetitions can clearly be identified and eliminated in polynomial time. The conjunctive queries C_1 and C_2 will serve as generic conjunctive queries in the remainder of this chapter.

The theorem of Chandra and Merlin

The following treatment is a synopsis of Section 1.3.2, which presents a syntactic test for the containment of a conjunctive query in another (see Theorem 1.1). The basic tool used is the containment mapping, as described below in terms of the generic conjunctive queries C_1 and C_2 presented at the beginning of this section.

Let f be a function on the symbols in C_2 that leaves constants unchanged. We may extend f to terms (and atoms) in the obvious way; that is, we define $f(q(d_1, \ldots, d_k))$ to be $f(q)(f(d_1), \ldots, f(d_k))$, where the d_i are arbitrary terms. Such a function is said to be a *containment mapping* from C_2 into C_1 if the following are both true.

1. $f(b_0(\vec{W}_0)) = a_0(\vec{U}_0)$.
2. For $1 \leq j \leq l$ there is an i, $1 \leq i \leq m$, such that $f(b_j(\vec{W}_j)) = a_i(\vec{U}_i)$.

If there is a containment mapping $f : C_2 \to C_1$, then for every atom t in C_2, we say that $f(t)$ is the *destination* of t under f. Since f is a function and there are (by assumption) no subgoal repetitions in C_1, each atom in C_2 has a unique destination under f.

The theorem of Chandra and Merlin (Theorem 1.1) states that for any conjunctive queries C_1 and C_2, $C_1 \subset C_2$ if and only if there is a containment mapping $f : C_2 \to C_1$.

Example 2.1 Consider the conjunctive queries C_3 and C_4, as defined below.

$C_3 : p(X) :- a(X, B), b(A, B), b(C, B), b(D, D), c(B, B), c(C, B), c(A, D).$
$C_4 : p(X) :- a(X, V), b(U, V), c(U, W).$

The function f defined by $f(X) = X, f(V) = B, f(U) = A, f(W) = D$ is a containment mapping from C_4 into C_3. The destination (under f) of $p(X)$ is $p(X)$, of $a(X,V)$ is $a(X,B)$, of $b(U,V)$ is $b(A,B)$ and of $c(U,W)$ is $c(A,D)$.

However, there is no containment mapping $g : C_3 \rightarrow C_4$. If there were such a mapping g, then the destination of $c(B,B)$ under g would have to be $c(U,W)$ (the only c-atom in C_4); however, the requirement $g(c(B,B)) = c(U,W)$ would imply $g(B) = U$ and $g(B) = W$, contradicting the assumed functionality of g. □

2.2.2 Conjunct mappings

The theorem of Chandra and Merlin is based on the existence of a mapping on symbols. An alternative viewpoint, and one which lends itself to a description of the complexity of deciding conjunctive query containment, is based on the existence of a mapping on atoms.

Definition 2.1 Let s and t be two arbitrary atomic formulae. We say that there is a *partial mapping* from s to t (written $s \rightarrow t$) if there is a substitution for the variables in s under which s is made syntactically identical to t. □

Each partial mapping $s \rightarrow t$ (if it exists) implies a unique substitution for each variable in s; the set of such assignments is termed the *assignment set induced by the partial mapping.* Two assignment sets are said to be *consistent* if no variable is assigned a different value by the two assignment sets, and a pair of partial mappings is said to be consistent if the assignment sets induced by the mappings are consistent.

Example 2.2 In Example 2.1, the partial mapping $b(U,V) \rightarrow b(A,B)$ induces the assignment set $\{U := A, V := B\}$. Further, the partial mappings $a(X,V) \rightarrow a(X,B)$ and $b(U,V) \rightarrow b(A,D)$ are inconsistent, since the variable V is assigned a different value by these mappings. Finally, as indicated in Example 2.1, there is no partial mapping $c(B,B) \rightarrow c(U,W)$. □

The existence of a partial mapping may be tested and the induced assignment set generated through *term-matching* (Ullman ([36])), and may hence be accomplished in time that is polynomial in the total size of s and t. Testing for consistency is clearly polynomial in the size of the assignment sets. The following lemma shows that for function-free atoms, both these procedures may be accomplished in LOGSPACE. The central idea is the fact that in the function-free case, the existence of a partial mapping and the consistency of a pair of partial mappings may each be tested by testing the equality or inequality of arguments in the relevant atoms.

Example 2.3 Consider the queries of Example 2.1. There is a partial mapping $b(U,V) \rightarrow b(A,B)$, but no partial mapping $c(B,B) \rightarrow c(U,W)$. In the latter case, both arguments of $c(B,B)$ are equal, but the arguments of $c(U,W)$ are unequal. □

For any atom $p(\vec{X})$ and integer i, let $p(\vec{X})[i]$ denote the ith argument of $p(\vec{X})$.

Theorem 2.1 Let $p(\vec{X})$ and $q(\vec{Y})$ be atoms. Then, there is a partial mapping $p(\vec{X}) \rightarrow q(\vec{Y})$ iff the following conditions are true.

1. p and q are the same predicate, and have the same arity (say, n).
2. For $1 \leq i \leq n$, if $p(\vec{X})[i]$ is the constant c, then $q(\vec{Y})[i]$ is the same constant c.
3. For $1 \leq i < j \leq n$, if $p(\vec{X})[i] = p(\vec{X})[j]$, then $q(\vec{Y})[i] = q(\vec{Y})[j]$.

Proof. If conditions 1, 2 and 3 are met, then the set

$$\{p(\vec{X})[i] := q(\vec{Y})[i] \mid 1 \leq i \leq n,\ p(\vec{X})[i] \text{ is a variable}\}$$

is a substitution under which $p(\vec{X})$ is made syntactically identical to $q(\vec{Y})$. The converse follows by the definition of a partial mapping. □

Theorem 2.2 Let p be a predicate of arity n and q a predicate of arity m. Assume that the partial mappings $p(\vec{X}) \rightarrow p(\vec{Y})$ and $q(\vec{W}) \rightarrow q(\vec{Z})$ exist. These partial mappings are consistent iff for $1 \leq i \leq n, 1 \leq j \leq m$, if $p(\vec{X})[i] = q(\vec{W})[j]$, then $p(\vec{Y})[i] = q(\vec{Z})[j]$.
Proof. If the condition is met, then the set

$$\{p(\vec{X})[i] := p(\vec{Y})[i] \mid 1 \leq i \leq n,\ p(\vec{X})[i] \text{ is a variable}\}$$
$$\cup$$
$$\{q(\vec{W})[j] := q(\vec{Z})[j] \mid 1 \leq j \leq m,\ q(\vec{W})[j] \text{ is a variable}\}$$

is single-valued for each variable in $\vec{X} \cup \vec{W}$, and is a substitution under which $p(\vec{X})$ is made syntactically identical to $p(\vec{Y})$ and $q(\vec{W})$ made syntactically identical to $q(\vec{Z})$. The converse follows by the definition of consistency. □

The importance of partial mappings lies in the duality that such mappings enjoy with containment mappings. More precisely, if there is a function f and atoms s and t such that $f(s) = t$, then (by definition) there is a partial mapping $s \rightarrow t$; in addition, the existence of a partial mapping $s \rightarrow t$ uniquely defines a function f such that $f(s) = t$. Further, the assignment set induced by the partial mapping is merely an extensional definition of the function f.

Example 2.4 Consider the queries C_3 and C_4 of Example 2.1. The function f defined by $f(U) = A, f(V) = B$ maps $b(U, V)$ to $b(A, B)$ (that is, $f(b(U, V)) = f(A, B)$). The partial mapping $b(U, V) \rightarrow b(A, B)$ exists, and the assignment set induced by the partial mapping is $\{U := A, V := B\}$. □

These observations lead us to the following characterization for conjunctive query containment.

Definition 2.2 Let C_1 and C_2 be the generic conjunctive queries of the preceding subsection. We say that a *conjunct mapping* M from C_2 into C_1 (written $M : C_2 \rightarrow C_1$) is a sequence $< m_0, \ldots, m_l >$ of (not necessarily distinct) atoms in C_1 such that the following are all true.

1. m_0 is $a_0(\vec{U}_0)$.
2. For $1 \leq j \leq l$, m_j is $a_i(\vec{U}_i)$ for some $i \in [1,m]$.
3. For each j, $0 \leq j \leq l$, there is a partial mapping $b_j(\vec{W}_j) \rightarrow m_j$.
4. Each pair of partial mappings is consistent.

For each j, m_j is the *destination* of $b_j(\vec{W}_j)$ under the conjunct mapping. The partial mapping $b_0(\vec{W}_0) \rightarrow a_0(\vec{U}_0)$ is termed the *head mapping.* □

The value of conjunct mappings is illustrated by the following theorem.

Theorem 2.3 (*Conjunct mapping theorem*) For any conjunctive queries C_1 and C_2, there is a containment mapping from C_2 into C_1 iff there is a conjunct mapping from C_2 into C_1.
Corollary $C_1 \subset C_2$ iff there is a conjunct mapping $M : C_2 \rightarrow C_1$.
Proof. Let C_1 and C_2 be as above. If there is a containment mapping $f : C_2 \rightarrow C_1$, then the sequence $< f(b_0(\vec{W}_0)), \ldots f(b_l(\vec{W}_l)) >$ defines a conjunct mapping from C_2 into C_1. Conversely, assume that M is a conjunct mapping from C_2 into C_1. The function defined by the union of the assignment sets induced by the partial mappings $b_j(\vec{W}_j) \rightarrow m_j$, for $0 \leq j \leq l$, is a containment mapping from C_2 into C_1. The proof of the corollary follows by the containment mapping theorem. □

Example 2.5 Consider the conjunctive queries C_3 and C_4 of Example 2.1. There is a conjunct mapping $M : C_4 \Rightarrow C_3$ in which the destination of $p(X)$ is $p(X)$, of $a(X,V)$ is $a(X,B)$, of $b(U,V)$ is $b(A,B)$ and of $c(U,W)$ is $c(A,D)$. □

2.2.3 The k-containment problem

We extend the conjunctive query containment problem by permitting the placement of restrictions on the destinations that a conjunct mapping may include.

Definition 2.3 Let C_1 and C_2 be the generic conjunctive queries of Section 2.2.1, and let $\mathcal{D}$ be the sequence $< D_0, D_1, \ldots D_l >$, where

1. $D_0 = \{a_0(\vec{U}_0)\}$, and
2. For $1 \leq j \leq l$, $D_j \subset \{a_i(\vec{U}_i) \mid 1 \leq i \leq m\}$.

We say that $C_1 \subset C_2$ *under* $\mathcal{D}$ (written $C_1 \overset{\mathcal{D}}{\subset} C_2$) if there is a conjunct mapping $M : C_2 \Rightarrow C_1$ such that $m_j \in D_j$, for all j . Determining such a containment is an instance of the *distinguished-destination problem.* The D_j are termed *destination sets*, since they limit the possible destinations of each atom under a conjunct mapping. If any destination set D_j is empty, then $C_1 \overset{\mathcal{D}}{\not\subset} C_2$.

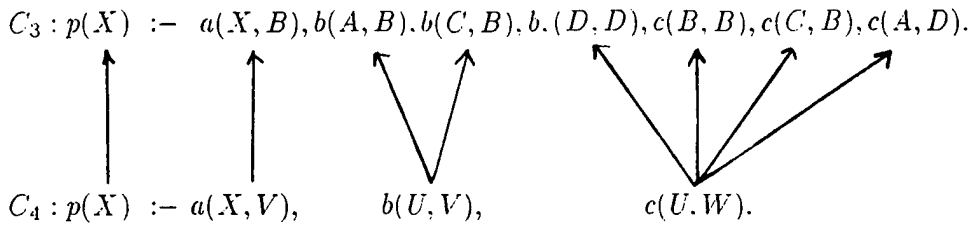


Figure 2.1: Distinguished-destination instance.

Similarly, if we are given conjunctive queries C_1 and C_2, and destination sets $\mathcal{D}_1$ and $\mathcal{D}_2$, then we may define the *equivalence* of C_1 and C_2 under $\mathcal{D}_1$ and $\mathcal{D}_2$ (written $C_1 \overset{\mathcal{D}_1,\mathcal{D}_2}{\equiv} C_2$) to be $C_1 \overset{\mathcal{D}_1}{\subset} C_2$ and $C_2 \overset{\mathcal{D}_2}{\subset} C_1$. □

Example 2.6 Consider the conjunctive queries C_3 and C_4 of Example 2.1, and define $\mathcal{D}$ by the arrows in Figure 2.1. $< C_1, C_2, \mathcal{D} >$ is a distinguished-destination instance. Note that the atom $b(D,D)$ is not an allowed destination for $b(U,V)$, but is an allowed destination for $c(U,W)$. □

We further parametrize the problem by the maximum cardinality of the destination sets D_j.

Definition 2.4 An instance of the *k-containment problem* is an instance of the distinguished-destination problem, in which $|D_j| \leq k$ for all j: that is, no destination set has more than k elements. □

Example 2.7 The problem of Example 2.6 is an instance of the 4-containment problem. □

2.2.4 Pruning

Given conjunctive queries C_1 and C_2 (the generic conjunctive queries of Section 2.2.1) and a set $\mathcal{D}$ of destination sets, we may *prune* the destination sets in $\mathcal{D}$ as follows. For each j, let D_j be the set $\{d_{jq} | 1 \leq q \leq n_j\}$, where n_j is the cardinality of D_j.

Definition 2.5 We say that the distinguished-destination problem $< C_1, C_2, \mathcal{D} >$ is *pruned* iff

1. For all p, s, if $d_{ps} \in D_p$ then $b_p(\bar{W}_p)$ -- d_{ps}: that is, there is a partial mapping from every atom in C_2 to each of its allowed destinations; and

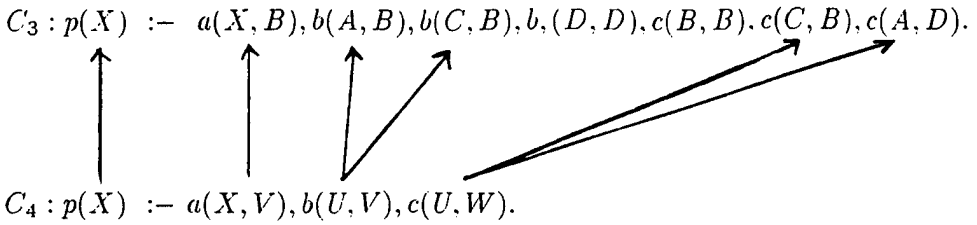


Figure 2.2: Pruning.

2. For all p, s, if $d_{ps} \in D_p$, then for all q there is a t such that $d_{qt} \in D_t$ and the partial mapping $b_p(\vec{W}_p) \rightarrow d_{ps}$ is consistent with the partial mapping $b_q(\vec{W}_q) \rightarrow d_{qt}$; that is, every allowed destination for some atom is consistent with at least one allowed destination for every other atom.

□

Define any $d_{ps} \in D_p$ to be a *violation of Class 1* if it violates condition (1), and a *violation of Class 2* if it violates condition (2). If d_{ps} is a violation of Class 1 or 2, then by the definition of a conjunct mapping, the destination of $b_p(\vec{W}_p)$ cannot be d_{ps} under any conjunct mapping. Hence, the removal of a Class 1 or Class 2 violation d_{ps} from the destination set D_p does not affect the existence of a containment. We prune the destination sets D_j by iteratively removing all violations to produce a set $\mathcal{D}'$ such that $C_1 \overset{\mathcal{D}}{\subset} C_2$ iff $C_1 \overset{\mathcal{D}'}{\subset} C_2$.

Example 2.8 Consider the distinguished-destination instance of Example 2.6. There is no partial mapping $c(U,W) \rightarrow b(D,D)$ (since b and c are different predicates), and the partial mapping $c(U,W) \rightarrow c(B,B)$ is inconsistent with every choice of destination for $b(U,V)$. The pruned distinguished-destination instance is shown in Figure 2.2. Note that the instance is now a 2-containment problem.

□

Algorithm 2.1
INPUT: $C_1, C_2, \mathcal{D}$ as above.
OUTPUT: $\mathcal{D}'$, containing no violations, such that $C_1 \overset{\mathcal{D}'}{\subset} C_2$ iff $C_1 \overset{\mathcal{D}}{\subset} C_2$.

(1) *change* $\leftarrow$ *true*
(2) **while** *change*
(3) *change* $\leftarrow$ *false*
(4) **for** $0 \leq j \leq l$
(5) **for** $s \in D_j$

```
                % If s is a Class 1 violation, remove it.
(6)             if b_j(W̄_j) ≠ s
(7)                 D_j ← D_j − {s}
(8)             else
                % If s is a Class 2 violation, remove it.
(9)             violation ← false
(10)                for 0 ≤ i ≤ l
(11)                    temp ← true
(12)                    for t ∈ D_i
(13)                        if b_j(W̄_j) → s is consistent with b_i(W̄_i) → t
(14)                            temp ← false
(15)                    violation ← violation ∨ temp
(16)        if violation
(17)            D_i ← D_i − {s}
(18)            change ← true
```

$\mathcal{D}'$ is the set of the resulting D_j. □

The correctness of the algorithm follows from the fact that the removal of a violation cannot affect the existence of a containment. Termination is guaranteed since each assignment of *true* to *change* (at line 1) accompanies the deletion of a destination from a destination set. The algorithm is easily seen to run in polynomial time, and in LOGSPACE for Datalog queries by Theorems 2.1 and 2.2.

In the remainder of this chapter, we assume that all distinguished-destination instances have been pruned.

2.2.5 Equivalence of the containment and distinguished-destination problems

It turns out that the conjunctive query containment and distinguished-destination problems are essentially the same problem, in the sense that these problems are polynomially equivalent.

Given conjunctive queries C_1 and C_2 and an integer k such that no predicate appears more than k times in the body of C_1, we may construct a k-containment instance by setting D_0 to be a singleton set containing the head of C_1, and setting each other D_j to be the set of the atoms in the body of C_1 that have the same principal functor as b_j. That is, each atom in the body of C_2 is allowed to map to every occurrence of the same predicate in the body of C_1.

The following algorithm performs the reduction in the opposite direction.

Algorithm 2.2
INPUT: a k-containment instance C_1, C_2 and $\mathcal{D}$.
OUTPUT: conjunctive queries C_1' and C_2' such that no predicate appears more than k times in the body of C_1', and such that $C_1' \subset C_2'$ iff $C_1 \overset{\mathcal{D}}{\subset} C_2$.

(1) The heads of C_1' and C_2' are those of C_1 and C_2, respectively.[1]

(2) Let the jth atom in the body of C_2 be $e(\vec{W}_j)$. Since $\mathcal{D}$ is pruned, each member of the destination set D_j must also have principal functor e. Let D_j be $\{e(\vec{U}_{j_q})|1 \leq q \leq n_j\}$, where (by assumption) $n_j \leq k$. Create a new predicate symbol g_j. Then, add the atom $g_j(\vec{W}_j)$ to the body of C_2', and an atom $g_j(\vec{U}_{j_q})$ to the body of C_1' for $1 \leq q \leq n_j$.

□

Example 2.9 Consider the pruned distinguished-destination instance of Example 2.8. The algorithm produces the following queries.

$C_3' : p(X) \ :- \ d(X,B), e(A,B), e(C,B), g(C,B), g(A,D).$
$C_4' : p(X) \ :- \ d(X,V), e(U,V), g(U,W).$

□

The algorithm is clearly polynomial. In step 2, each predicate g_j is made to appear at most $n_j \leq k$ times in the body of C_1', and thus no predicate appears more than k times in the body of C_1. Finally, to prove that $C_1 \overset{\mathcal{D}}{\subset} C_2$ iff $C_1' \subset C_2'$, we observe the following.

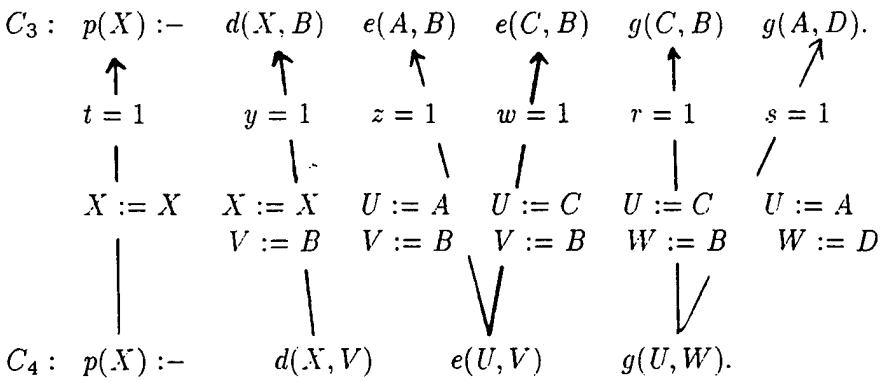
1. Both instances involve the same head mapping, yielding the same induced assignment set.

2. If the j-th atom in the body of C_2 is $e(\vec{W}_j)$ with destination set $D_j = \{e(\vec{U}_{j_q})|1 \leq q \leq n_j\}$, then for any q, there is a partial mapping $e(\vec{W}_j) - e(\vec{U}_{j_q})$ inducing the assignment set S iff there is a partial mapping $g_j(\vec{W}_j) - g_j(\vec{U}_{j_q})$ inducing the assignment set S.

2.2.6 Complexity of k-containment

The k-containment problem is clearly in $\mathcal{NP}$; merely guess a conjunct mapping and verify using the conjunct mapping theorem. It turns out that the k-containment problem and kSAT are essentially the same problem. That is, for $k \geq 2$, the k-containment problem is no harder than kSAT; since 2SAT is known to be polynomial, we may conclude that the 2-containment problem is also polynomial. In fact, the reduction may be performed in LOGSPACE for Datalog queries, and the 2-containment problem is therefore in NLOGSPACE (and hence $\mathcal{NC}$ [9]). Further, for $k \geq 3$, kSAT is no harder than the k-containment problem, and the 3-containment problem is therefore $\mathcal{NP}$-complete.

[1] To preserve the safety of the queries, we may create a new predicate f, and place the atom $f(\vec{U}_0)$ in the body of C_1' and the atom $f(\vec{W}_0)$ in the body of C_2'.



2SAT instance: Class 1 clauses: $\{t\}\{y\}\{z+w\}\{r+s\}$
Class 2 clauses $\{\bar{z}+\bar{r}\}\{\bar{w}+\bar{s}\}$

Minimal satisfying truth assignments: $t=y=z=s=1$
$t=y=w=r=1$

Figure 2.3: Testing the containment $C_3 \subset C_4$ in Example 2.9

Polynomial time

We will provide a polynomial-time reduction from any instance of the k-containment problem to an instance of kSAT, for $k \geq 2$. We assume that the conjunctive queries C_1 and C_2 are the generic queries described in Section 2.2.1.

The basic idea is that we carry a Boolean variable representing each choice of destination for each atom. The clauses produced are of two kinds.

1. Clauses that enforce the requirement that each atom in C_2 have a legal destination in C_1. We call such clauses *Class 1* clauses.

2. Consistency constraints that disallow inconsistent pairs of partial mappings. Such clauses are termed *Class 2* clauses.

Example 2.10 illustrates the construction.

Example 2.10 Consider the pruned 2-containment instance of Example 2.9. Let the following Boolean variables indicate the following choices of destinations (see Figure 2.3).

Boolean variable	atom	destination
t	$p(X)$	$p(X)$
y	$d(X,V)$	$d(X,B)$
z	$e(U,V)$	$e(A,B)$
w	$e(U,V)$	$e(C,B)$
r	$g(U,W)$	$g(C,B)$
s	$g(U,W)$	$g(A,D)$

Recall that the Class 1 clauses represent the statement "Every atom in C_4 has a legal destination". Now, the only allowed destination for $p(X)$ is $p(X)$, yielding the Class 1 clause $\{t\}$. Similarly, the allowed destinations for $b(U,V)$ are $b(A,B)$ and $b(C,B)$; the corresponding Class 1 clause is $\{z+w\}$. The set of Class 1 clauses is $\{t\}\{y\}\{z+w\}\{r+s\}$.

The Class 2 clauses enforce consistency of the corresponding partial mappings. The mapping $b(U,V) \rightarrow b(A,B)$ is inconsistent with $c(U,W) \rightarrow c(C,B)$ (yielding the clause $\{\bar{z}+\bar{r}\}$), and the mapping $b(U,V) \rightarrow b(C,B)$ is inconsistent with $c(U,W) \rightarrow c(A,D)$ (yielding the clause $\{\bar{w}+\bar{s}\}$).

The 2SAT instance created is $\{t\}\{y\}\{z+w\}\{r+s\}\{\bar{z}+\bar{r}\}\{\bar{w}+\bar{s}\}$, with the two satisfying truth-assignments $t=y=z=s=1$ and $t=y=w=r=1$. The first such assignment gives a conjunct mapping as indicated by the heavy arrows in Figure 2.3. Note that in the general case, the cardinality k of the Class 1 clauses is the parameter of the k-containment instance, and Class 2 clauses always have cardinality 2. □

In general, more that one member of a Class 1 clause may be true in a satisfying truth assignment (signifying more than one possible destination for some atom). In this case, any one choice of destination suffices.

The following is a formal statement and proof of the algorithm.

Algorithm 2.3
INPUT: a pruned k-containment instance $< C_1, C_2, \mathcal{D} >$, with $k \geq 2$.
OUTPUT: a k-SAT instance $\mathcal{I}$ that is satisfiable iff there is a k-containment.

(1) If any destination set D_j is empty, output the unsatisfiable instance $\{x\}\{\bar{x}\}$.

(2) Create $(l+1)k$ Boolean variables $\{x_{ji} | 0 \leq j \leq l, 1 \leq i \leq k\}$ and $(l+1)k$ set variables $\{A_{ji} | 0 \leq j \leq l, 1 \leq i \leq k\}$; the former will be used to construct the kSAT instance, and the latter to hold induced assignment sets. Each destination set D_j is $\{d_{ji} | 1 \leq i \leq n_j\}$, where by assumption $n_j \leq k$.

(3) **for** $0 \leq j \leq l$

(4) add the clause $\{x_{j1} + \ldots + x_{jn_j}\}$ to $\mathcal{I}$

(5) **for** $1 \leq q \leq n_j$

(6) $A_{jq} \leftarrow$ the assignment set induced by the partial mapping $b_j(\vec{W}_j) - d_{jq}$

(7) **for** $0 \leq j < i \leq l$

(8) **for** $1 \leq q \leq n_j, 1 \leq p \leq n_i$

(9) **if** A_{jq} is inconsistent with $A[ip]$

(10) add $\{\overline{x_{jq}} + \overline{x_{ip}}\}$ to $\mathcal{I}$

□

The algorithm is clearly polynomial. As before, we term the clauses that are added to $\mathcal{I}$ at Step 4 *Class 1* clauses, and those that are added at Step 10 *Class 2* clauses. Since Class 1 clauses have cardinality at most $n_j \leq k$ and Class 2 clauses are doubletons, $\mathcal{I}$ is a kSAT instance.

By the form of the Class 1 clauses, we may observe that in any satisfying truth-assignment for $\mathcal{I}$, some x_{jq} is *true* for each j. Define a satisfying truth-assignment to be *minimal* iff exactly one x_{jq} is *true* for each j.

Lemma 2.1 $\mathcal{I}$ is satisfiable iff it has a minimal truth-satisfying assignment.
Proof. If $\mathcal{I}$ has a minimal satisfying truth-assignment, then it is clearly satisfiable. For the converse, assume that S is a satisfying truth-assignment for $\mathcal{I}$. By previous discussion, at least one member of each Class 1 clause is *true* under S. Arbitrarily pick one such member of each Class 1 clause, and set every other member to be *false*. Such a procedure cannot make either a Class 1 clause or a Class 2 clause untrue if it was true under S; hence, the result is a minimal satisfying truth- assignment for $\mathcal{I}$. □

Lemma 2.2 If $C_1 \overset{\mathcal{D}}{\subset} C_2$, then $\mathcal{I}$ is satisfiable.
Proof. If $C_1 \overset{\mathcal{D}}{\subset} C_2$, no destination set D_j is empty, and the algorithm does not terminate at Step 1. Let $< d_{0s_0}, \ldots, d_{ls_l} >$ be a conjunct mapping from C_2 into C_1, where $d_{js_j} \in D_j$ for all j. Construct a truth-assignment for $\mathcal{I}$ by setting x_{js_j} to be *true*, and all other variables to be *false*. Such an assignment satisfies all Class 1 clauses. Further, by the properties of a conjunct mapping, each Class 2 clause is satisfied as well. Hence, this truth-assignment satisfies $\mathcal{I}$. □

Lemma 2.3 If $\mathcal{I}$ is satisfiable, then $C_1 \overset{\mathcal{D}}{\subset} C_2$.
Proof. Assume $\mathcal{I}$ is satisfiable, and let S be a minimal satisfying truth-assignment for I. Assume that the variables that are true under S are $x_{0s_0}, \ldots, x_{ls_l}$. Then $< d_{0s_0}, \ldots, d_{ls_l} >$ is a conjunct mapping from C_2 into C_1 that obeys $\mathcal{D}$, since the Class 1 clauses enforce the requirement that $d_{js_j} \in D_j$, and the Class 2 clauses enforce the requirement that no two choices of destination are inconsistent. □

Note that a similar reduction may be performed from a k-equivalence problem $< C_1, C_2, \mathcal{D}_1, \mathcal{D}_2 >$ to a kSAT instance $\mathcal{I}$. That is, we perform the reduction for $< C_1, C_2, \mathcal{D}_1 >$ and $< C_2, C_1, \mathcal{D}_2 >$ separately, to produce kSAT instances $\mathcal{I}_1$ and $\mathcal{I}_2$ respectively, where the variables in $\mathcal{I}_1$ and $\mathcal{I}_2$ are distinct; then, the conjunction of $\mathcal{I}_1$ and $\mathcal{I}_2$ yields a kSAT instance that is satisfiable iff $C_1 \overset{\mathcal{D}_1,\mathcal{D}_2}{\equiv} C_2$.

Theorem 2.4 Algorithm 2.3 is correct.

Corollary 1. The 2-containment (or 2-equivalence) problem is in $\mathcal{P}$.

Corollary 2. The 2-containment (or 2-equivalence) problem is in NLOGSPACE (and hence $\mathcal{NC}$) for Datalog queries.

Proof. By Lemmas 2.3 and 2.4. Corollary 1 follows because 2SAT is in $\mathcal{P}$. Corollary 2 follows because 2SAT is complete for NLOGSPACE, and NLOGSPACE is in $\mathcal{NC}$ ([9]). □

Related work Sagiv and Yannakakis ([29]) propose an algorithm to decide containment among function-free conjunctive queries C_1 and C_2 in which each atom in C_2 has only two destinations consistent with the head mapping; the primary differences between their algorithm and Algorithm 2.3 are that their algorithm is based on function-free queries, and performs pruning only in terms of atoms whose destinations are inconsistent with the head mapping. Related work on polynomial-time algorithms includes the minimization algorithms of Aho et al. ([3, 4]) and Johnson and Klug ([19]). The most general result among these three papers is that of Johnson and Klug, who consider "fanout-free" conjunctive queries; they test for equivalence between two queries by minimizing each query, and then determining whether the minimal queries are isomorphic. The conjunctive query

$$C_1 : p(X) \ :- \ a(X, C), b(B, C), b(C, C), c(B, D), c(C, D), c(C, C).$$

is not fanout-free. However, C_1 is equivalent to the conjunctive query

$$C_2 : p(X) \ :- \ a(X, U), b(U, U), c(U, U).$$

as may be verified through two uses of Algorithm 2.4.

$\mathcal{NP}$-completeness

We now show that the k-containment problem is $\mathcal{NP}$-complete for $k \geq 3$, for a restricted class of conjunctive queries. We will begin by defining the concept of a valid labelling for a kSAT instance, and showing that a kSAT instance is satisfiable iff it has a valid labelling. We will then reduce the problem of finding a valid labelling for a given kSAT instance to that of solving a k-containment instance.

3SAT instance:	$c_1 = \{x_1 + x_2 + \bar{x}_3\}$	$c_2 = \{\bar{x}_1 + \bar{x}_4 + x_5\}$
Satisfying truth assignment:	$x_1 = true, x_4 = false$, x_2, x_3, x_5 arbitrary.	
Valid labelling:	$<< T_{11}, D_{12}, D_{13} >$	$< D_{21}, T_{22}, D_{23} >>$

Figure 2.4: Valid labelling.

Valid labelling Let $\mathcal{I}$ be a kSAT instance consisting of the p clauses $c_1, \ldots, c_p$ over the q variables $x_1, \ldots, x_q$. Without loss of generality, we assume that $\mathcal{I}$ contains no tautological clauses (clauses that contain a pair of complementary literals); such clauses do not affect the satisfiability of $\mathcal{I}$, and may be removed in polynomial time. We also assume that no literal appears more than once in any clause; literal repetitions may similarly be removed in polynomial time [2]. For all i, let the ith clause have $n_i \leq k$ literals. We will use the notation l_{ij} to denote the jth literal in clause c_i.

Let $\{T_{ij} \mid 1 \leq i \leq p, 1 \leq j \leq k\} \cup \{D_{ij} \mid 1 \leq i \leq p, 1 \leq j \leq n_i\}$ be a set of $2km$ distinct constants. A *labelling* A of $\mathcal{I}$ is an p-vector of tuples $< A[i1], \ldots, A[in_i] >$, where $A[ij]$ is either T_{ij} or D_{ij} for each i and j. Intuitively, the assignment of T_{ij} to $A[ij]$ represents a satisfying truth assignment under which the literal l_{ij} is "true"; similarly, D_{ij} denotes a "don't care" for the value of l_{ij}.

A labelling of $\mathcal{I}$ is *valid* if

1. For each i, there is exactly one j such that $A[ij]$ is T_{ij}; that is, exactly one literal in each clause is "true"; and

2. For any $A[ij]$ and $A[ml]$, $i \neq m$, if l_{ij} and l_{ml} are complementary literals, then either $A[ij] = D_{ij}$ or $A[ml] = D_{ml}$ (or both); that is, no pair of complementary literals in different clauses are both true.

Example 2.11

Consider the 3SAT instance I consisting of the two clauses $c_1 = \{x_1 + x_2 + \bar{x}_3\}$ and $c_2 = \{\bar{x}_1 + \bar{x}_4 + x_5\}$. The sequence $<< T_{11}, D_{12}, D_{13} >< D_{21}, T_{22}, D_{23} >>$ is a valid labelling for I; that is, we set the first literal in clause c_1 and the second literal in clause c_2 to "true", and set all other literals to "don't care" (see Figure 2.4). □

[2] That is, $\mathcal{I}$ is an instance of at-most-kSAT

It turns out that the existence of a valid labelling is necessary and sufficient for the existence of a satisfying truth-assignment.

Theorem 2.5 A kSAT instance is satisfiable iff it has a valid labelling.
Proof. Consider a kSAT instance $\mathcal{I}$ as above. If $\mathcal{I}$ is satisfiable, then it has some satisfying truth-assignment S. Construct a labelling V as follows: if any literal l_{ij} it *true* under S, then $A[ij]$ is T_{ij}; otherwise, it is D_{ij}. Since S satisfies each clause in $\mathcal{I}$, at least one $A[ij]$ is T_{ij} for each i. Now, for each i, select any $A[ij]$ that is assigned T_{ij}, and assign D_{im} to every other $A[im]$. This procedure cannot create violations of requirement 2 in the definition of a valid labelling, and the result is a valid labelling for $\mathcal{I}$.

For the converse, assume that V is a valid labelling for $\mathcal{I}$. Construct a truth assignment for $\mathcal{I}$ as follows: for every literal l_{ij} such that $A[ij]$ is T_{ij}, if l_{ij} is the positive literal x_p then set x_p to *true* under S, and if it is the negative literal $\bar{x}_p$, then set x_p to *false*; set all unassigned variables to *false*. This procedure assigns a unique value to each variable in $\mathcal{I}$, since a valid labelling never assigns T-values to complementary literals. Since at least one literal in each clause is true under S, S is a satisfying truth-assignment for $\mathcal{I}$. □

The reduction The idea of a valid labelling permits a reduction from a kSAT instance I to a k-containment instance as illustrated below.

Example 2.12 Consider the kSAT instance I of Example 2.11, consisting of the two clauses $c_1 = \{x_1 + x_2 + \bar{x}_3\}$ and $c_2 = \{\bar{x}_1 + \bar{x}_4 + x_5\}$. We construct conjunctive queries C_1 and C_2 as follows.

Let us use the Datalog variable L_{ij} to represent the jth literal in clause c_i. For example, L_{11} represents the first literal in clause 1; that is. the (occurrence of) x_1 in c_1. Similarly, L_{23} represents the literal x_5 in clause c_2. Further, the Datalog variables T_{ij} and D_{ij} respectively represent a choice of "true" or "don't care" for the jth literal in clause c_i.

The head of each of C_1 and C_2 will be the O-ary predicate h (see Figure 2.5).

To represent the clause c_1, we construct the atom $c_1(L_{11}, L_{12}, L_{13})$ in the body of C_2, and the three possible destinations $c_1(T_{11}, D_{12}, D_{13})$, $c_1(D_{11}, T_{12}, D_{13})$, $c_1(T_{11}, D_{12}, D_{13})$. Note that any containment mapping from C_2 into C_1 enforces the fact that exactly one literal in clause c_1 is "true", as required by part (1) of the definition of a valid labelling (see Figure 2.5). We similarly construct an occurrence of a predicate c_2 in C_2 and three occurrences of this predicate in C_1 to represent the clause c_2.

Finally, we must impose requirement (2) in the definition of a valid labelling: that is, that no two complementary occurrences of any literal are both assigned "true". In our example, there is only one such violation: the first literal in c_1 and the first literal in c_2 are complementary. Hence, we construct a new "enforcer" predicate e_{1121}. and add the atom $e_{1121}(L_{11}, L_{21})$ to the body of C_2 and the three atoms $e_{1121}(T_{11}, D_{21})$, $e_{1121}(D_{11}, T_{21})$ and $e_{1121}(D_{11}, D_{21})$ to the body of C_1. The completed instance is shown in Figure 2.5.

Note that in general, the number of predicate repetitions added in the first stage is the maximum cardinality of any clause in the kSAT instance (i.e. k), and the number added in the second stage is always 3. □

3SAT instance: $c_1 = \{x_1 + x_2 + \bar{x}_3\}$ $c_2 = \{\bar{x}_1 + \bar{x}_4 + x_5\}$

Satisfying truth assignment: $x_1 = true, x_4 = false, x_2, x_3, x_5$ arbitrary.

Valid labelling: $<< T_{11}, D_{12}, D_{13} >$ $< D_{21}, T_{22}, D_{23} >>$

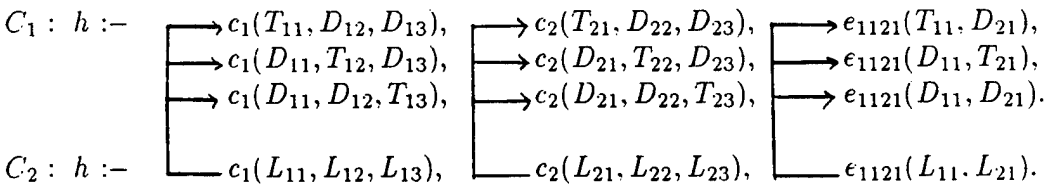


Figure 2.5: Illustrating the construction.

We formalise the procedure as follows.

Let $\mathcal{I}$ be a kSAT instance, for $k \geq 3$. We construct conjunctive queries C_1 and C_2 such that no predicate appears more than k times in either query, such that no variable is repeated in any conjunct, and such that $C_1 \subset C_2$ iff $\mathcal{I}$ has a valid labelling. Further, the construction will yield queries C_1 and C_2 such that $C_2 \subset C_1$; hence, $C_1 \equiv C_2$ iff $\mathcal{I}$ is satisfiable. The time expended in the reduction is polynomial in the size of C_1 and C_2.

Algorithm 2.4
INPUT: A kSAT instance $\mathcal{I}$
OUTPUT: Conjunctive queries C_1 and C_2 as above, such that $C_1 \subset C_2$ ($C_1 \equiv C_2$) iff $\mathcal{I}$ is satisfiable

(1) C_1 and C_2 each has the rule head h (with no arguments). That is, the relation for h is one of *true* and *false*. [3]

(2) **for** $1 \leq i \leq p$

(3) Consider the ith clause $c_i = \{l_{i1} + \ldots + l_{in_i}\}$

(4) Create new nondistinguished variables $L_{i1}, \ldots L_{in_i}$ and the predicate symbol c_i

(5) Add to the body of C_2 the atom $c_i(L_{i1}, \ldots, L_{in_i})$

(6) add n_i c_i-atoms to the body of C_1, where the jth argument of the jth such atom is T_{ij} and the rth argument is D_{ir} for all $r \neq j$

(7) **for** $1 \leq i < j \leq p$

(8) **for** $1 \leq l \leq n_i, 1 \leq m \leq n_j$

(9) **if** l_{il} and l_{jm} are complementary literals

(10) create a new predicate constant e_{iljm}

(11) add $e_{iljm}(L_{il}, L_{jm})$ to the body of C_2

(12) add $e_{iljm}(T_{il}, D_{jm})$, $e_{iljm}(D_{il}, T_{jm})$ and $e_{iljm}(D_{il}, D_{jm})$ to C_1

□

The algorithm is clearly polynomial. Define conjuncts that are added to the bodies of C_1 and C_2 in Steps 5 and 6 to be *Class 1* conjuncts, and those that are added at Steps 11 and 12 to be *Class 2* conjuncts. C_2 contains no repetitions of any predicate. C_1 contains at most k repetitions of Class 1 predicates, and three repetitions of Class 2 predicates. Note that all conjuncts are rectified (that is, have no repeated arguments), and have arity at most k.

[3] For a reader who is offended by a 0-ary predicate, identical results may be obtained by making $h(X)$ the head of each rule, and adding the atom $a(X)$ to each rule body.

Lemma 2.4 There is a containment mapping $f : C_2 \rightarrow C_1$ iff $\mathcal{I}$ has a valid labelling.
Proof. Assume f is a containment mapping from C_2 into C_1. Construct a labelling for $\mathcal{I}$ by setting $A[ij]$ to be $f(L_{ij})$ for all i and j. The possible destinations for Class 1 atoms require that condition 1 in the definition of a valid labelling is satisfied; similarly, the possible destinations for Class 2 atoms enforce condition 2, and the labelling thus constructed is valid.

Assume V is a valid labelling for $\mathcal{I}$. Construct a function f on the variables of C_2 by setting $f(L_{ij})$ to the value of $A[ij]$ for all i and j, and extend f to atoms (see Section 2.2.1). Since $f(h) = h$, the head mapping exists. By condition 1 in the definition of a valid labelling, and by construction, each Class 1 atom in C_2 has exactly one possible destination in C_1 under f. The possible destinations for Class 2 atoms, along with condition 2 in the definition of a valid labelling, ensure the functionality of f. □

Lemma 2.5 $C_2 \subset C_1$.
Proof. The function defined by $f(D_{ij}) = f(T_{ij}) = L_{ij}$ is a containment mapping from C_1 into C_2. □

Theorem 2.6 Algorithm 2.4 is correct.
Corollary. 3-containment (3-equivalence) is $\mathcal{NP}$-complete.
Proof. By Algorithm 2.3, Theorem 2.4 and Theorem 2.5, we conclude that 3-containment is polynomially equivalent to 3-containment, and is hence $\mathcal{NP}$-complete. □

Related results Chandra and Merlin ([10]) have shown that testing conjunctive query containment is $\mathcal{NP}$-complete even for function-free qeries, although their result assumes up to six repetitions of each predicate in the bodies of the queries. Sagiv and Yannakakis ([29]) have shown that the problem is $\mathcal{NP}$-complete even if no predicate appears more than three times in the body of either query; however, their reduction assumes the repetition of variables in the arguments of some conjuncts.

2.3 Applications

Let us turn to the complexity of the optimization problems that we discussed in Chapter 1. It turns out that the results of the previous section allow us to show that restricted versions of the one-boundedness, sequencability and basis-linearizability problems are $\mathcal{NP}$-hard, and that the algorithms of Section 1.5.3 are polynomial.

2.3.1 Approach and notation

The $\mathcal{NP}$-hardness results of this section are based on the following idea. Given any 3SAT instance, the techniques of the preceding section permit us to construct conjunctive queries

$$C_1 : \; h \; :- \; B_1.$$
$$C_2 : \; h \; :- \; B_2.$$

such that $C_1 \subset C_2$ iff the 3SAT instance is satisfiable. That is, B_1 and B_2 are the bodies of the conjunctive queries generated by Algorithm 2.5. Let us abuse notation by saying that $B_1 \subset B_2$ whenever there is a conjunct mapping $M : B_2 \Rightarrow B_1$; that is, a consistent choice of destinations (in B_1) for the atoms in B_2. Now, the conjunctive query C_1 is contained in the conjunctive query C_2 iff there is a conjunct mapping from B_2 into B_1, since h (the head of C_1 and C_2) is a 0-ary predicate. Hence, the original 3SAT instance is satisfiable iff $B_1 \subset B_2$. Note also that by construction, there is a conjunct mapping $B_1 \Rightarrow B_2$; that is, that $B_2 \subset B_1$.

Example 2.13 Consider the conjunctive queries C_1 and C_2 of Example 2.12. The conjunctions B_1 and B_2 are as follows.

$$\begin{aligned} B_1 :\ & c_1(T_{11}, D_{12}, D_{13}), c_1(D_{11}, T_{12}, D_{13}), c_1(D_{11}, D_{12}, T_{13}), \\ & c_2(T_{21}, D_{22}, D_{23}), c_2(D_{21}, T_{22}, D_{23}), c_2(D_{21}, D_{22}, T_{23}), \\ & e_{1121}(T_{11}, D_{21}), e_{1121}(D_{11}, T_{21}), e_{1121}(D_{11}, D_{21}). \\ B_2 :\ & c_1(L_{11}, L_{12}, L_{13}), c_2(L_{21}, L_{22}, L_{23}), e_{1121}(L_{11}, L_{21}). \quad \square \end{aligned}$$

Now, the $\mathcal{NP}$-hardness results of the following subsections are obtained by embedding the conjunctions B_1 and B_2 (perhaps with added arguments) into the bodies of recursive rules, such that the resulting rules have desired properties iff $B_1 \subset B_2$. Let us adopt the convention that for any variable X, $X|B_1$ denotes the conjunction B_1 in which each conjunct is given the additional first argument X.

Example 2.14 Consider the conjunction B_2 of Example 2.13. The rule

$$p(X) :- p(U), X|U|B_2.$$

stands for the rule

$$p(X) :- p(U), c_1(X, U, L_{11}, L_{12}, L_{13}), c_2(X, U, L_{21}, L_{22}, L_{23}), e_{1121}(X, U, L_{11}, L_{21}). \quad \square$$

Finally, let us adopt the convention that in "unwinding" a recursion, every nondistinguished variable in a rule is renamed by "priming" at each stage of the expansion: that is, a nondistinguished variable U in the body of a rule is renamed

$$U^{\overbrace{\prime \ldots \prime}^{i}}$$

(written $U^{(i)}$) for some i. For a linear recursion, the version of a nondistinguished variable U at depth $i+1$ is the variable $U^{(i)}$.

Example 2.15 Consider the linear rule of Example 2.14. Figure 2.6 exhibits our notation on expansions of $p(X)$ using this rule. $\square$

Let us return to the conjunctions B_1 and B_2 as described at the beginning of this section. Let $B_j^{(i)}$ denote the conjunction B_j (that is, one of B_1 and B_2) in which every variable is primed i times. It is clear that if $B_1^{(i)} \subset B_2^{(k)}$ for any i and k, then $B_1 \subset B_2$.

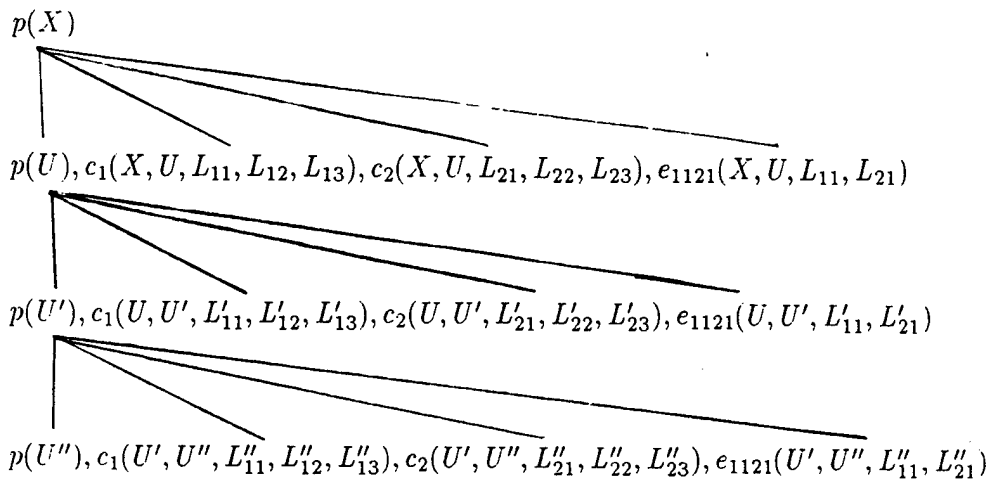


Figure 2.6: Priming.

2.3.2 One-boundedness

Consider the safe, (not necessarily function-free) recursive rule

$r_1 : \ p(\vec{X}) :- \ p(\vec{U_1}), \ldots p(\vec{U_n}), a_1(\vec{W_1}), \ldots a_k(\vec{W_k}).$

where $\vec{X}$ is a vector of distinct variables. Recall that the sirup r_1 is said to be *1-bounded* if every top-down expansion of $p(\vec{X})$ using r_1 is contained in a top-down expansion of depth at most 1. Recall also our convention that the top-down expansion representing the rule

$\epsilon : p(\vec{X}) :- \ p(\vec{X}).$

has depth 0. If r_1 is 1-bounded, then the program consisting of r_1 and any basis rule of the form

$r_2 : \ p(\vec{X}) :- \ b(\vec{X}).$

may be reduced to a nonrecursive program, in which r_1 is replaced by the rule

$r'_1 : \ p(\vec{X}) :- \ b(\vec{U_1}), \ldots b(\vec{U_n}), a_1(\vec{W_1}), \ldots a_k(\vec{W_k}).$

Kanellakis ([20]) has shown that deciding 1-boundedness is $\mathcal{NP}$-hard for linear sirups defining a predicate of arity four; however, the reduction involves an unbounded number of repetitions of EDB predicates in the body of the sirup. We present the following result.

Theorem 2.7 Let r_1 be a linear, Datalog, head-rectified sirup defining a binary predicate, such that no EDB predicate appears more than four times in its body, or has arity greater than five. Testing 1-boundedness is $\mathcal{NP}$-hard for such rules.

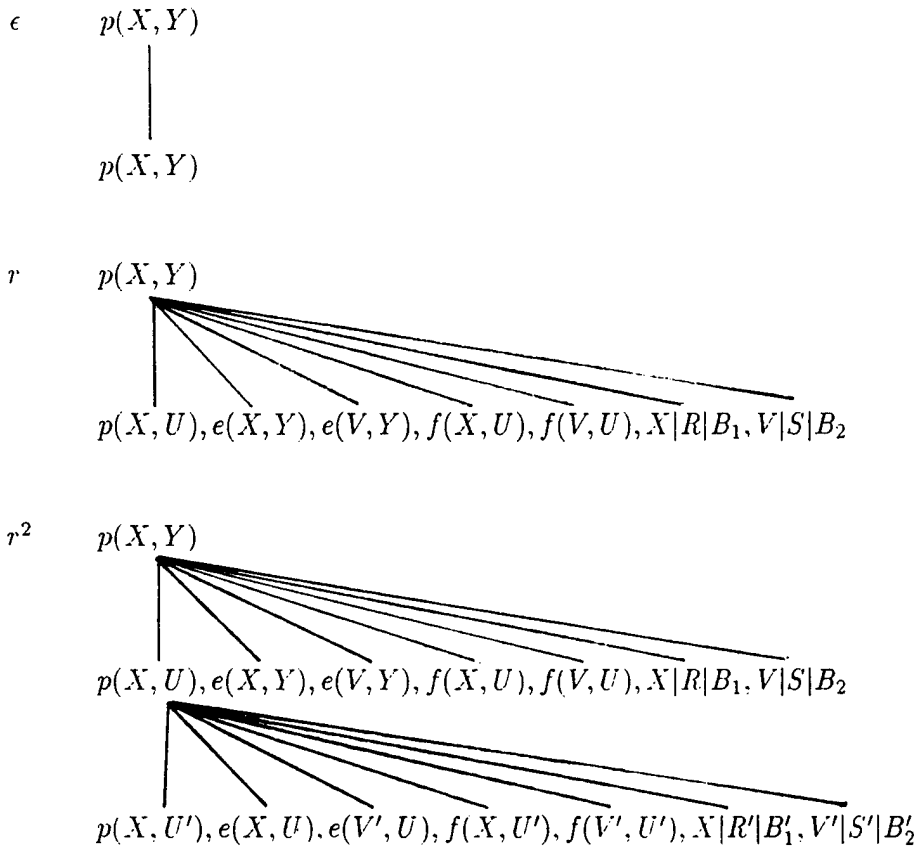


Figure 2.7: The expansions of Theorem 2.7

Proof. Given a 3SAT instance $\mathcal{I}$, construct the conjunctive queries $C_1 : h :- B_1.$ and $C_2 : h :- B_2.$ as in Algorithm 2.4 . We construct a program $\mathcal{P}$ that is 1-bounded iff there is a conjunct mapping $M : B_2 \Rightarrow B_1$. Recall that by construction, $B_1 \Rightarrow B_2$. $\mathcal{P}$ is the sirup

$$r : p(X,Y) :- p(X,U), e(X,Y), e(V,Y), f(X,U), f(V,U), X|R|B_1, V|S|B_2.$$

where X, Y, U, V, R and S are distinct variables not appearing in B_1 or B_2, and where p, e and f are distinct predicate symbols not appearing in B_1 or B_2.

Consider the top-down expansions r^2, ϵ and r (see Figure 2.5).

$\mathcal{P}$ is 1-bounded iff $r^2 \subset \epsilon$ or $r^2 \subset r$. In either case, the head mapping induces the assignment set $\{X := X, Y := Y\}$. The nondistinguished variable U' appears in the p-atom generated by r^2, so that the only choice of destination for the body of ϵ violates the head mapping, and thus $r^2 \not\subset \epsilon$.

Thus, if $\mathcal{P}$ is 1-bounded, then $r^2 \subset r$. The destination for the atom $p(X, U)$ in r requires that $U := U'$. Then, the possible destinations for the atom $f(V, U)$ require that $V := X$ or $V := V'$, and the possible destinations for $e(V, Y)$ require that $V := X$ or $V := V$. Thus, the conjunct mapping $M : r \Rightarrow r^2$ must assign $V := X$, and we may conclude that there is a conjunct mapping $V|S|B_2 \Rightarrow X|R|B_1$ or $V|S|B_2 \Rightarrow X|R'|B_1'$, as desired.

For the converse, assume $C_1 \subset C_2$; then $B_2 \Rightarrow B_1$. We may select the destination of $X|R|B_1$ to be $X|R|B_1$, and select the destination of every other atom in r_1 as in the previous paragraph, to obtain a conjunct mapping from r into r^2. By Theorem 1.7, $\mathcal{P}$ is one-bounded. □

On the other hand, we may use Algorithm 2.3 to decide 1-boundedness in polynomial time, for linear sirups in which no predicate is repeated in the body of the recursive rule (i.e., $n = 1$ and the a_i are distinct in r_1). The reason is that testing 1-boundedness reduces to two 2-containment tests.

Algorithm 2.5
INPUT: A sirup r_1 as described above, with $n = 1$ and with no predicate repetitions among the a_i.
OUTPUT: "yes" if the sirup is 1-bounded, "no" otherwise.

1. Construct the empty rule $\epsilon : p(\vec{X}) :- p(\vec{X})$., and the top-down expansion $r_1 r_1$.

2. Use Algorithm 2.3 to determine whether $r_1 r_1 \subset \epsilon$ or $r_1 r_1 \subset r_1$. If either containment holds, return "yes"; otherwise, return "no".

□

Theorem 2.8 Algorithm 2.5 is correct.
Proof. Necessity follows by the definition of 1-boundedness, and sufficiency by Theorem 1.7. □

Theorem 2.9 Algorithm 2.5 runs in polynomial time for arbitrary sirups, and in NLOGSPACE (and hence in $\mathcal{NC}$) for Datalog queries.
Proof. Step (1) may be performed in polynomial time for arbitrary sirups, and in LOGSPACE for Datalog queries. The proof follows by Theorem 2.4 and the fact that NLOGSPACE is in $\mathcal{NC}$ ([9]). □

Finally, we show that Algorithm 1.1 (see Section 1.5.3) runs in polynomial time.

Theorem 2.10 Algorithm 1.1 runs in polynomial time.
Proof. Each containment in the algorithm is an instance of the 2-containment problem. □

2.3.3 Rule sequencability

Consider the safe, Datalog, linear rules

$r_1 : p(\vec{X}) :- p(\vec{U}_0), a_1(\vec{U}_1), \ldots a_k(\vec{U}_k).$
$r_2 : p(\vec{X}) :- p(\vec{W}_0), b_1(\vec{W}_1), \ldots b_l(\vec{W}_l).$

where $\vec{X}$ is a vector of distinct variables. Define ϵ to be the following rule.

$\epsilon : p(\vec{X}) :- p(\vec{X}).$

Recall that r_1 is *sequencable under* r_2 if $(r_1 + r_2)^* \subset r_2^* r_1^*$. Rule sequencability is not known to be decidable. However, the problem is at least as hard as any problem in $\mathcal{NP}$, as the following theorem shows.

Theorem 2.11 Let r_1 and r_2 be rules as above, with the additional restrictions that no predicate appears more than three times in the body of r_1 or more than once in the body of r_2, and that all predicates have arity at most four. Detecting rule sequencability is $\mathcal{NP}$-hard for rules of this form.

Proof. Consider an arbitrary 3SAT instance $\mathcal{I}$, and apply Algorithm 2.5 to obtain conjunctive queries $C_1 : h :- B_1$ and $C_2 : h :- B_2$. We construct rules r_1 and r_2 such that $C_1 \subset C_2$ iff r_1 is sequenceable under r_2. By the discussion of the beginning of this section, we know that $B_1 \Rightarrow B_2$, and that $C_1 \subset C_2$ iff $B_2 \Rightarrow B_1$. The rules r_1 and r_2 are

$r_1 : p(X, Y, Z, W) :- p(Y, X, Z, W), X|B_1.$
$r_2 : p(X, Y, Z, W) :- p(X, Y, W, Z), X|B_2.$

where X, Y, Z and W are new, distinct variables and g is a new predicate symbol.

Note that the p-atom in the expansions r_1^2 and r_2^2 is $p(X, Y, W, Z)$, so r_1^2 and r_2^2 are contained in ϵ and each rule is 1-bounded.

Assume r_1 is sequencable under r_2. Then $r_1 r_2$ is contained in some expansion in $r_2^* r_1^*$. By the 1-boundedness of r_1 and r_2, $r_1 r_2$ must be contained in one of ϵ, r_1, r_2 and $r_2 r_1$ (see Figure 2.8). In each case, the head mapping induces the assignment set $\{X := X, Y := Y, Z := Z, W := W\}$. However, in the first three containments, the destination for the p-atom generated by the expansion is inconsistent with the head mapping, and we may conclude that $r_1 r_2 \subset r_2 r_1$.

Consider any conjunct mapping $M : r_2 r_1 \Rightarrow r_1 r_2$ (see Figure 2.8). As we observed, the head mapping yields the assignment $\{X := X\}$. Now, by definition (see the introduction to this section), the first argument of every atom in the conjunctions $X|B_2$ and $X|B_1'$ is X, and therefore both these conjunctions must map to the conjunction $X|B_1$ in $r_1 r_2$. Hence, the conjunct mapping M must map $X|B_2$ and $X|B_1'$ into $X|B_1$, so $B_2 \Rightarrow B_1$, as required.

For the converse, assume that $B_2 \Rightarrow B_1$. Then the mapping M indicated in Figure 2.8 proves that $r_1 r_2 \subset r_2 r_1$, which in turn suffices to prove sequencability by Theorem 1.9. □

In view of the lack of a known algorithm to detect sequencability, a variety of conditions have been proposed that are sufficient (but not necessary) to detect sequencability in pairs

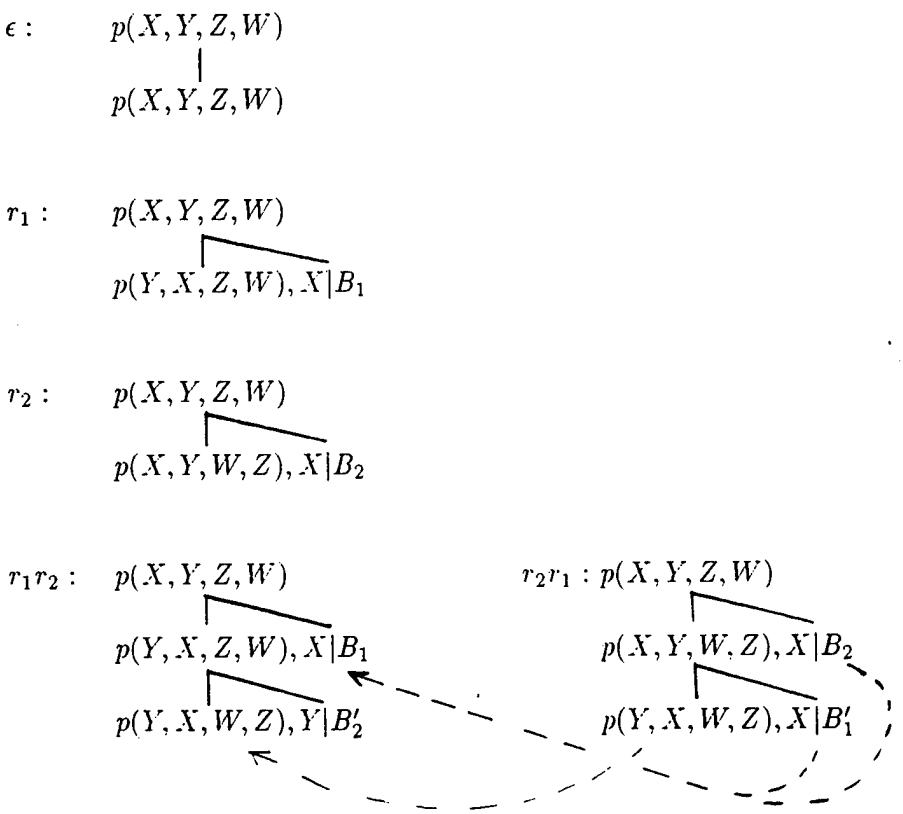


Figure 2.8: The construction of Theorem 2.11.

of linear rules. The most popular condition is that of *commutativity*, $r_1 r_2 \subset r_2 r_1$. The more general condition $r_1 r_2^* \subset r_2 r_1^*$ was proposed independently by Ramakrishnan et al. ([25]) and Ioannidis ([17]); the former shows that the condition is verifiable in polynomial space. The following theorem considers the complexity of such conditions.

Theorem 2.12 Let r_1 and r_2 be rules obeying the conditions of Theorem 2.11. Then testing each of the following conditions, each sufficient to prove the sequencability of r_1 under r_2, is $\mathcal{NP}$-hard.
(a) $r_1 \subset r_2$
(b) $r_1 = r_2$
(c) $r_1 \subset r_2^*$
(d) $r_1 r_2 \subset r_2 r_1$
(e) $r_1 r_2 = r_2 r_1$
(f) $r_1 r_2 \subset r_2 r_1^*$.
Proof. Let $\mathcal{I}$ be a 3SAT instance, and let B_1 and B_2 be the conjunctions resulting from the application of Algorithm 2.5. We provide reductions such that conditions (a) - (f) are satisfied iff there is a conjunct mapping $M : B_2 \Rightarrow B_1$.

To prove (a), (b) and (c), we construct the rules

$r_1 : p(X, Y) :- p(Y, X), B_1.$
$r_2 : p(X, Y) :- p(Y, X), B_2.$

The observation that each rule is 1-bounded, and that $B_1 \Rightarrow B_2$ by construction, suffices to complete the proof.

The construction of Theorem 2.11 yields $\mathcal{NP}$-hardness reductions for conditions (d) - (f). □

Conditions (a), (b), (d) and (e) are each in $\mathcal{NP}$; conditions (c) and (f) may be tested in polynomial space by the chase algorithm of Ramakrishnan et al. ([25]).

Theorem 2.10 implies that the conditions (a) through (f) can probably not be tested in polynomial time. However, conditions (a), (b), (d) and (e) reduce to the testing of containments among pairs of conjunctive queries, and Theorem 2.4 may apply to these conditions over various classes of rules that arise in practice. For example, if r_1 and r_2 each have no repetitions of predicates in their respective bodies, then this algorithm provides a polynomial-time test of conditions (a),(b),(d) and (e). Ioannidis ([16]) has also proposed an algorithm to test commutativity (condition (e)) in this restricted case.

2.3.4 Basis-linearizability

Consider the safe, simple recursive rule

$r_1 : p(\vec{X}) :- p(\vec{U}_1), \ldots, p(\vec{U}_n), e_1(\vec{W}_1), \ldots, e_k(\vec{W}_k).$

where $n > 1$ and $\vec{X}$ is a vector of distinct variables. Recall that r_1 is *basis-linearizable* if, for every basis rule

$$r_2 : p(\vec{X}) :- b(\vec{X}).$$

we may replace r_1 with the rule

$$r_1' : p(\vec{X}) :- b(\vec{U}_1), \ldots, b(\vec{U}_{n-1}), p(\vec{U}_n), e_1(\vec{W}_1), \ldots, e_k(\vec{W}_k).$$

to obtain an equivalent program. The linear rule r_1' is obtained from the nonlinear rule r_1 by replacing all but the last occurrence of p with a corresponding occurrence of the basis predicate b. Recall that basis-linearizability indicates that right-linearity is the normal form for the conjunctive queries generates by the program. That is, the sirup r_1 is basis-linearizable iff every top-down expansion of $p(\vec{X})$ using r_1 is contained in a right-linear expansion.

Linearizability of this sort was investigated by Zhang et al. ([40] [4]), who claim a polynomial-time decision procedure for bilinear, function-free rules with one nonrecursive subgoal (i.e., $n = 2$ and $k = 1$), although (as we will show in the next chapter) their proof is flawed. In Chapter 3, we will extend their result to include all bilinear recursions, as long as no nonrecursive predicate appears more than once in the body of the rule. We will also show that the algorithm runs in polynomial time.

Basis-linearizability is not known to be decidable in the case in which predicates are allowed to appear repetitively among the subgoals. Ramakrishnan et al. ([25]) show that detecting basis-linearizability in bilinear recursions is $\mathcal{NP}$-hard; their reduction involves a recursive predicate of arity 6, and places no bound on the number of predicate repetitions in the body of the rule. We tighten that result in the following theorem.

Theorem 2.13 Let r_1 be a bilinear rule as above, with the restrictions that p has arity 2, all other predicates have arity at most 5, and no predicate appears more than four times in the body of r_1. Then, deciding basis-linearizability is $\mathcal{NP}$-hard for rules of this form.

Proof. Let $\mathcal{I}$ be a 3SAT instance to which Algorithm 2.5 has been applied to produce conjunctions B_1 and B_2. We construct a rule r_1 that is basis-linearizable iff there is a conjunct mapping $M : B_2 \Rightarrow B_1$. Note that $B_1 \Rightarrow B_2$ by construction. The rule is

$$r_1 : p(X, Y) :- p(X, U), p(T, Y), e(X, Y), e(V, Y), f(X, U), f(V, U), X|R|B_1, V|S|B_2.$$

where X, Y, U, T, V, R and S are new and distinct variables.

If the rule r_1 is basis-linearizable, then the tree T_1 in Figure 2.9 is contained in a right-linear tree (a top-down expansion in which only $p(T, Y)$ is expanded through r_1). By convention, the empty rule

$$\epsilon : p(X, Y) :- p(X, Y).$$

is a right-linear tree of depth 0.

[4] This paper has recently been published in TODS ([41]), but the proof has been omitted.

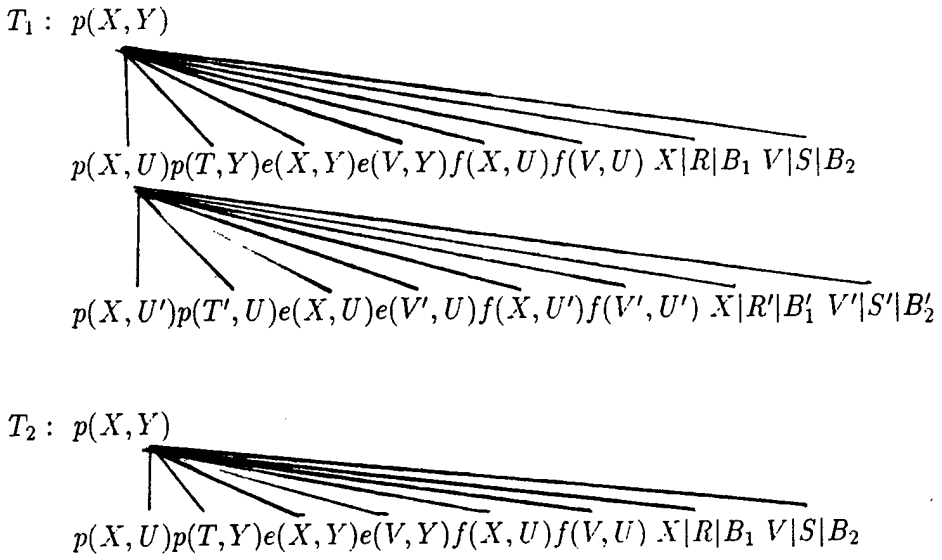


Figure 2.9: The construction for Theorem 2.13

Since $p(X,Y)$ is not a leaf in T_1, every destination for $p(X,Y)$ among the leaves of T_1 is inconsistent with the head mapping, and we may conclude that $T_1 \not\subseteq \epsilon$; that is, T_1 is contained in a right-linear tree T_2 in which the head is expanded using rule r_1.

Now, the head mapping from T_2 into T_1 induces the assignments $\{X := X, Y := Y\}$. Hence, the atom $p(X,U)$ must have destination $p(X,U')$, and every conjunct mapping from T_2 into T_1 must induce the assignment $U := U'$. The only destinations for $e(V,Y)$ consistent with these assignments force $V := X$ or $V := V$, and the only such destinations for the atom $f(V,U)$ force the assignments $V := V'$ or $V := X$. Hence, every conjunct mapping $M : T_2 \Rightarrow T_1$ must induce the assignment $V := X$, forcing the mapping $V|S|B_2 \Rightarrow X|R|B_1$ or $V|S|B_2 \Rightarrow X|R'|B_1'$; in either case, we may conclude $B_2 \Rightarrow B_1$.

For sufficiency, assume that $B_2 \Rightarrow B_1$. Then, the conjunct mappings in the previous paragraph, along with the partial mappings $p(T,Y) \rightarrow p(T,Y)$ and $X|R|B_1 \rightarrow X|R|B_1$, suffice to prove basis-linearizability by Theorem 1.18. □

Finally, we justify our claim that Algorithm 1.2 (in Section 1.5.3) runs in polynomial time.

Theorem 2.14 Algorithm 1.2 runs in polynomial time.

Proof. Each containment test in the algorithm is an instance of the 2-containment problem. □

Chapter 3

A decision procedure for basis-linearizability

3.1 Introduction

Consider the safe and function-free ("Datalog") logic program $\mathcal{P}$ with a single doubly-recursive ("bilinear") rule of the form

$$r_1: \quad p(X_1, \ldots, X_m) \quad :- \quad p_{(1)}(\vec{Y}), p_{(2)}(\vec{Z}), e_1(\vec{U}_1), \ldots, e_N(\vec{U}_N).$$

and a single basis rule of the form

$$r_2: \quad p(X_1, \ldots, X_m) \quad :- \quad b(X_1, \ldots, X_m).$$

where we have subscripted the recursive occurrences of p for ease of reference. We will refer to atoms in the body of the recursive rule by their principal functors, using the subscripts to disambiguate the recursive atoms; that is, the term "$p_{(1)}$" will be used to refer to $p_{(1)}(\vec{Y})$ (the first recursive p-atom in the body of r_1), and the term "e_i" to refer to the atom $e_i(\vec{U}_i)$.

We assume that the rules r_1 and r_2 satisfy the following requirements.

1. The variables appearing in the head are distinct. Recall that these variables variables are termed *distinguished*, and that all other variables are termed *nondistinguished*.

2. The rules are *range-restricted* or *safe*; that is, every distinguished variable appears in the rule body.

3. The base predicate b and the subgoals e_i are EDB predicates, and these predicates are distinct. Recall from Chapter 1 that EDB relations are stored by extension in the database, and that the predicate p is termed *intensional* or *IDB*.

Recall that $\mathcal{P}$ is termed *linearizable by basis* iff right-linearity is a normal form for the proof trees (or conjunctive queries) generated by the program. That is, $\mathcal{P}$ is basis-linearizable iff $\mathcal{P}$ is equivalent to the following linear program $\mathcal{Q}$.

r_1 : $p(X_1,\ldots,X_m)$:- $b(\vec{Y}), p_{(2)}(\vec{Z}), e_1(\vec{U}_1),\ldots,e_N(\vec{U}_N)$.
r_2 : $p(X_1,\ldots,X_m)$:- $b(X_1,\ldots,X_m)$.

$\mathcal{Q}$ is obtained from $\mathcal{P}$ by replacing the first recursive occurrence of p in the body of r_1 (that is, the atom $p_{(1)}(\vec{Y})$) with the base atom $b(\vec{Y})$.

Example 3.1 We repeat here the program of Example 1.1, computing the transitive closure of b.

r_1 : $p(X,Y)$:- $p(X,U), p(U,Y)$.
r_2 : $p(X,Y)$:- $b(X,Y)$.

$p(X,U)$, the first recursive atom in r_1, is referred to as $p_{(1)}$, and $p(U,Y)$ as $p_{(2)}$. Recall that we proved this program to be basis-linearizable in Section 1.5.2; that is, the program is equivalent to the following linear logic program.

r_1' : $p(X,Y)$:- $b(X,U), p(U,Y)$.
r_2 : $p(X,Y)$:- $b(X,Y)$.

The gains of performing the transformation are obtained from the use of query evaluators specific to linear recursions. □

In this chapter, we present a decision procedure for the recognition of basis-linearizability in bilinear recursions of the form of $\mathcal{P}$. The running time of the algorithm is polynomial [1] in the size of the program $\mathcal{P}$.

3.1.1 Related results

Consider the program $\mathcal{P}$ of the preceding section. As we showed in Chapter 2, if repetitions are allowed in the EDB subgoals of the recursive rule in this program, then the detection of basis-linearizability is $\mathcal{NP}$-hard; in fact, the decidability of basis-linearizability for such programs is open. In this chapter, we show that if no such repetitions are allowed, then basis-linearizability may be detected in polynomial time; hence, in some sense, our result represents a boundary between tractability and intractability. Finally, as we will show in the next chapter, if we consider programs with a single bilinear rule, an unbounded number of linear rules and 5 basis rules, then the detection of basis-linearizability becomes undecidable.

This chapter is an extension of the work of Zhang, Yu and Troy ([40] [2] who proposed the problem of basis-linearizability in the restricted case $N \leq 1$; that is, in the case in which the recursive rule has at most one EDB subgoal. They claim a polynomial time algorithm for this case; however, the proof of correctness of their algorithm is flawed, and we will touch upon this flaw in Section 3.5.3. Their algorithm also ignores so-called *deletion-linearizable* recursions, on the grounds that such programs may be linearized in a different way (see Section 3.4).

[1] In fact, it can be performed in linear time.
[2] This result has recently been published in TODS ([41]), but the proof has been omitted.

The algorithm of [40] does not extend in an obvious way to the programs that we consider (in which N is unbounded). That is, the representation of $e_1, \ldots, e_N$ as their "join" (a single atom), followed by an application of the algorithm of [40], is insufficient to detect basis-linearizability. The following example illustrates this point.

Example 3.2 Consider the program

$r_1 : p(X,Y) :- p(X,U), p(U,Y), c(U), d(Y).$
$r_2 : p(X,Y) :- b(X,Y).$

where b, c and d are distinct EDB predicates. This program is basis-linearizable, as the algorithm of Section 3.2 shows. Assume that we represent the EDB predicates c and d by their Cartesian product, in some new EDB predicate e. Then, we obtain the program

$r_1 : p(X,Y) :- p(X,U), p(U,Y), e(U,Y).$
$r_2 : p(X,Y) :- b(X,Y).$

which is not basis-linearizable, as the algorithm of Section 3.2 also shows. □

The proof of [40] also does not extend directly to the proof of correctness of our algorithm. Both their proof and ours rely heavily on the idea of safety; that is, on the requirement that every distinguished variable appears in the rule body. In the case $N = 1$, every distinguished variable X_i must appear among the arguments of $p_{(1)}$, $p_{(2)}$ or the single EDB subgoal e_1 by the assumption of safety; however, if N is unbounded, then X_i may appear as the only argument to some "new" EDB predicate e_j.

3.2 The algorithm

In this section, we present an algorithm that decides whether the program $\mathcal{P}$ is basis-linearizable, and show that the algorithm is polynomial in the size of $\mathcal{P}$.

Let us say that a nondistinguished variable that appears only in the arguments of the atom t in r_1 is said to be *local* to t; all other arguments are termed *nonlocal*.

Recall that we will refer to atoms in the body of the rules in $\mathcal{P}$ by their principal functors, using subscripts to disambiguate the recursive p-atoms.

Definition 3.1 We say that $p_{(1)}$ is an *adjunct* to $p_{(2)}$ in $\mathcal{P}$ if there is a partial mapping $p_{(1)} \rightarrow p_{(2)}$ that induces an assignment set that is the identity on nonlocal variables in $p_{(1)}$. That is, $p_{(1)}$ is obtainable from $p_{(2)}$ by replacing 0 or more occurrences of each variable in $p_{(2)}$ with a new, local variable. If $p_{(1)}$ is an adjunct, then $\mathcal{P}$ is said to be *deletion-linearizable*. □

The intuitive importance of adjuncts is that they may be deleted from the recursive rule to produce an equivalent, linear program, as we will show in Section 3.4.

Definition 3.2 Define $r_1 - p_{(1)}$ to be the rule r_1 in which $p_{(1)}$ has been deleted, and let $\mathcal{P} - p_{(1)}$ be the resulting program. If $p_{(1)}$ is an adjunct to $p_{(2)}$, then every distinguished

variable appearing among the arguments of $p_{(1)}$ also appears in $p_{(2)}$, so we conclude that this deletion preserves safety. □

For any atom t in the rule r_1, or in any top-down expansion, let us use the notation $t[i]$ to refer to the ith argument of t. We will always reserve the keyword X_i to refer to the ith distinguished variable in the head of the recursive rule.

Definition 3.3 The *home* position of any distinguished variable X_i is the ith position in any p-atom. □

Definition 3.4 $p_{(1)}$ is said to be a *trivial* adjunct to $p_{(2)}$ if $p_{(1)}$ is an adjunct, $p_{(1)}$ contains no nonlocal nondistinguished variables and, whenever the distinguished variable X_i appears in $p_{(1)}$, then $p_{(2)}[i]$ is X_i (that is, X_i appears in its home position in $p_{(2)}$). □

Example 3.3 Consider the program $\mathcal{P}$ defined as below.

r_1 : $p(X, Y, W, Z) :- p(U, X, A, B), p(X, X, A, W), e(Y, W, Z)$.
r_2 : $p(X, Y, W, Z) :- b(X, Y, W, Z)$.

The nondistinguished variables U and A are local to $p_{(1)}$ (that is, to the atom $p(U, X, A, B)$). The atom $p_{(1)}$ is an adjunct to the atom $p_{(2)}$ (that is, the atom $p(X, X, A, W)$); in fact, $p_{(1)}$ is a trivial adjunct since X appears in its home position in $p_{(2)}$. The rule $r_1 - p_{(1)}$ is presented below.

$r_1 - p_{(1)}$: $p(X, Y, W, Z) :- p(X, X, A, W), e(Y, Z)$.

Note that this rule is safe. □

Recall from Chapter 1 that top-down expansions generated by $\mathcal{P}$ are written in a way that preserves the left-to-right order of the subgoals in every rule, and that an expansion or proof tree is termed *right-linear* if only the rightmost p-atom in the recursive rule is ever recursively expanded. Finally, recall that a top-down expansion generated by $\mathcal{P}$ is termed *open* if only the recursive rule is used in constructing the expansion.

Example 3.4 The expansions T_1, T_2, T_3 and T_4 of Figure 3.1 are all open. T_1 is the minimal violation of right-linearity in $\mathcal{P}$, and the expansions T_2, T_3 and T_4 are the right-linear expansions of depth 0, 1 and 2 respectively. □

We will use the terms *tree* and *expansion* interchangeably. As we mentioned in Chapter 1, these expansions may all be constructed in time that is polynomial in the size of $\mathcal{P}$.

Now, let f be a containment mapping (equivalently, M a conjunct mapping) from T_2, T_3 or T_4 into T_1. Recall from Section 1.5.2 that the mapping f (or M) is termed *acceptable* if, under the mapping, the $p_{(2)}$-child of the root of T_1 is the destination of no $p_{(1)}$- leaf.

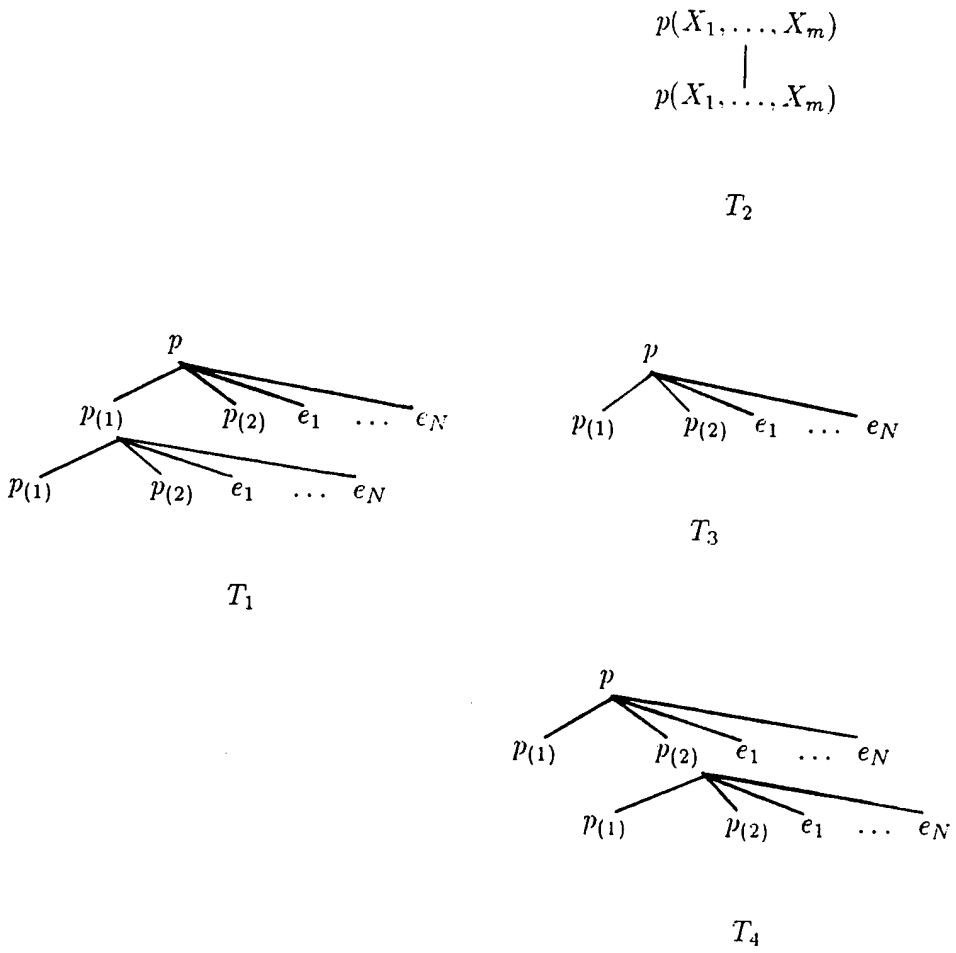


Figure 3.1: Expansions used in the algorithm.

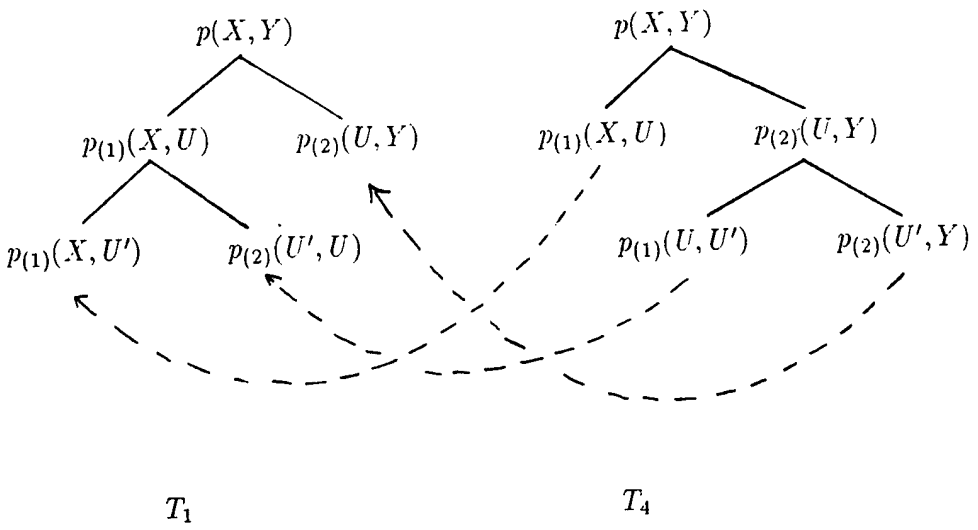


Figure 3.2: Acceptable mapping

Example 3.5 Consider the transitive-closure program of Example 3.1. The containment mapping f defined by $f(X) = X, f(Y) = Y, f(U) = U', f(U') = U$ is an acceptable mapping from T_4 into T_1, as indicated in Figure 3.2. □

We now state the main results of this chapter.

Theorem 3.1 $\mathcal{P}$ is basis-linearizable iff at least one of the following is true.

(1) $p_{(1)}$ is a trivial adjunct to $p_{(2)}$ in $\mathcal{P}$;

(2) $p_{(1)}$ is an adjunct, and $\mathcal{P} - p_{(1)}$ is 1-bounded; or

(3) There is an acceptable mapping from one of T_2, T_3 and T_4 into T_1.

□

The proof of this theorem is an extensive combinatorial analysis, and will occupy the remainder of this chapter. Before we begin this proof, let us observe that the algorithm implied by Theorem 3.1 is polynomial in the size of $\mathcal{P}$.

Theorem 3.2 The procedure of Theorem 1 is in NLOGSPACE.
Corollary 1. The procedure of Theorem 1 is in $\mathcal{NC}$.
Corollary 2. The procedure of Theorem 1 is polynomial.
Proof. Determining whether $p_{(1)}$ is an adjunct, or a trivial adjunct, is easily determined in LOGSPACE, so Condition (1) is in LOGSPACE. $\mathcal{P} - p_{(1)}$ is a linear sirup with no repeated

EDB subgoals; by Theorem 2.9, Condition (2) is in NLOGSPACE. By a case analysis on the destinations of the $p_{(1)}$-atom in T_3 and T_4, Condition (3) may be tested through nine 2-containment tests, each of which may be accomplished in NLOGSPACE by Theorem 2.4. Hence, the algorithm is in NLOGSPACE. Corollary 2 follows by [9], and Corollary 3 immediately follows. □

3.3 Proof Outline

In Section 3.4, we will investigate the behaviour of programs in which $p_{(1)}$ is an adjunct to $p_{(2)}$, and show that in this case, the recursive atom $p_{(1)}$ is redundant in the recursive rule r_1. Section 3.5 contains the proof of Theorem 3.1.

3.4 Adjuncts

Let us assume that $p_{(1)}$ is an adjunct to $p_{(2)}$ in $\mathcal{P}$. It turns out that, in this case, the atom $p_{(1)}$ is redundant in the recursive rule r_1; hence, we call such programs *deletion-linearizable*. We will prove this fact in the remainder of this subsection. As before, by the form of the basis rule, we will restrict our attention to proof trees (ground or otherwise) in which the basis rule is never used. That is, we adopt the convention that p itself is both an EDB and an IDB relation. In this subsection, we will consider *ground* or *instantiated* expansions: that is, trees in which all variables have been replaced by constants.

An important property of adjuncts is treated in the following lemma.

Lemma 3.1 If $p_{(1)}$ is an adjunct to $p_{(2)}$, then any p-fact that unifies with $p_{(2)}$ also unifies with $p_{(1)}$, and the unifications agree on the values of all nonlocal variables in $p_{(1)}$.
Proof. Let f represent the assignment set induced by the partial mapping $p_{(1)}(\vec{Y}) - p_{(2)}(\vec{Z})$, and assume that $p(a_1, \ldots, a_m)$ unifies with $p_{(2)}(\vec{Z})$ under the substitution τ. The function $\tau(f)$ is then a substitution under which $p_{(1)}$ unifies with $p(a_1, \ldots, a_m)$. Since f is the identity on nonlocal variables, τ and $\tau(f)$ agree on these variables. □

Definition 3.5 Let T be a proof tree (or expansion) generated by $\mathcal{P}$. The *right strut* of T is the proof tree (or expansion) obtained from T by discarding all $p_{(1)}$- atoms. □

Lemma 3.2 If $p_{(1)}$ is an adjunct, then for any tree T generated by $\mathcal{P}$ from any database D, the right strut of T is a proof tree generated by $\mathcal{P} - p_{(1)}$ from D and yielding the same fact as T.
Corollary. $\mathcal{P} \subset \mathcal{P} - p_{(1)}$.
Proof. Straightforward induction on the depth of the right strut of T. □

Example 3.6 Consider the program $\mathcal{P}$ of Example 3.3, in which $p_{(1)}$ is an adjunct to $p_{(2)}$. The right strut of the expansion of Figure 3.3 (a) is shown in Figure 3.3 (b). □

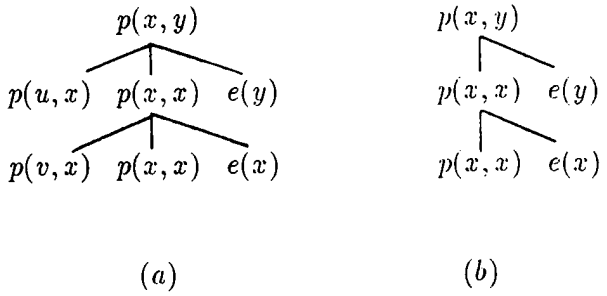
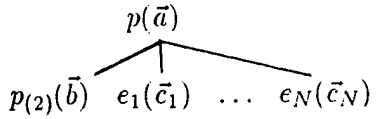
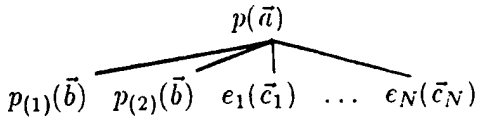


Figure 3.3: Right strut

Lemma 3.3 Consider any database D, and assume that $p_{(1)}$ is an adjunct. For $n \geq 1$, if S is a proof tree of depth n generated by $\mathcal{P} - p_{(1)}$ from D, then there is a complete tree R of depth n that is generated by $\mathcal{P}$ from D such that S is the right strut of R.
Corollary 1. $\mathcal{P} - p_{(1)} \subset \mathcal{P}$.
Corollary 2. $\mathcal{P} \equiv \mathcal{P} - p_{(1)}$.
Proof. The proof is a straightforward induction on n, using Lemma 3.1. For the basis ($n = 1$), let S be the proof tree



By Lemma 3.1, $\mathcal{P}$ generates the proof tree R defined as



For the induction, assume the truth of the hypothesis for $1 \leq i < n$. Let S be a tree of depth n as in Figure 3.4, in which the top level of the expansion is the ground query

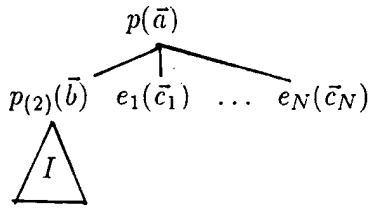
$$p(\vec{a}) :- p_{(2)}(\vec{b}), e_1(\vec{c}_1), \ldots, e_N(\vec{c}_N).$$

and where $p_{(2)}(\vec{b})$ is generated from $\mathcal{P} - p_{(1)}$ by a tree I of depth $n-1$. By hypothesis, $\mathcal{P}$ generates a complete tree J of depth $n-1$ from the leaves of I, establishing the fact $p(\vec{b})$, for which I is a right strut. By Lemma 3.1, $\mathcal{P}$ generates a complete proof tree of depth n establishing the fact $p(\vec{a})$, such that the first level of the proof tree is the query

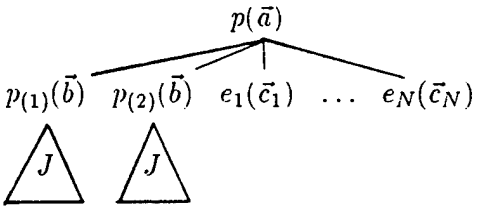
$$p(\vec{a}) :- p_{(1)}(\vec{b}), p_{(2)}(\vec{b}), e_1(\vec{c}_1), \ldots, e_N(\vec{c}_N).$$

and where the subtrees establishing $p_{(1)}$ and $p_{(2)}$ are both J (see Figure 3.4).

The proof of Corollary 1 is immediate. Corollary 2 is proved using Lemma 3.2. □



(a)



(b)

Figure 3.4: The induction

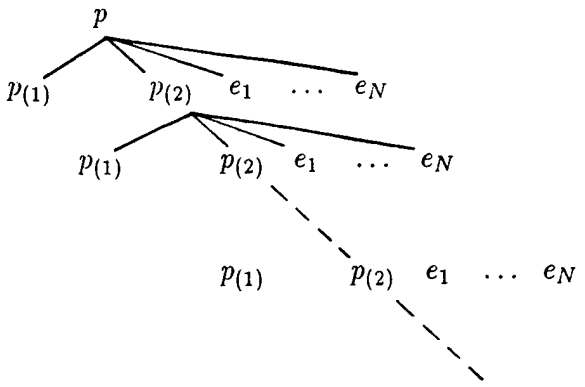


Figure 3.5: Right-linear expansion

3.5 Proof

This section will be devoted to the proof of Theorem 3.1.

Recall that an expansion is termed *open* if the basis rule is never used in constructing the expansion, and *closed* iff there are no p-leaves in it; that is, the basis rule is used to "close off" all intensional atoms in the latter case. Recall also that an expansion or proof tree is termed *right-linear* if only the $p_{(2)}$-atom in the recursive rule is ever recursively expanded (see Figure 3.6.)

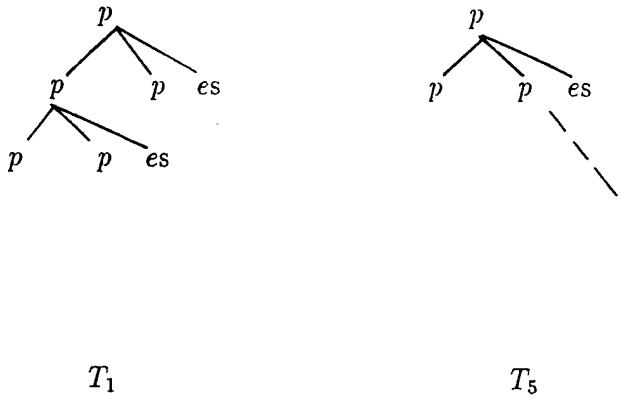
By definition, $\mathcal{P}$ is basis-linearizable iff every non-right-linear closed expansion is contained in some closed right-linear expansion (see Section 1.4.3). However, the form of the basis rule permits us to deal exclusively with open expansions; that is, the application of the basis rule to an open expansion amounts to the "renaming" of every p-leaf in the open expansion to the EDB predicate b, and every closed expansion is obtainable in this manner. Thus, $\mathcal{P}$ is basis-linearizable iff every open expansion is contained in a right-linear, open expansion. We may now think of the basis-linearizability of the sirup r_1 as the right-linearity of the proof trees generated by r_1 from every database, for *every* set of initialisation rules. The proof of correctness of Theorem 3.1 will be based exclusively on open expansions.

The outline of the proof is as follows.

3.5.1 Proof outline

1. In Section 3.5.2, we will show that the conditions of Theorem 3.1 are sufficient to show that each non-right-linear expansion is contained in a right-linear expansion; that is, that these conditions suffice to prove basis-linearizability in $\mathcal{P}$.

2. Section 3.5.3 is the heart of this chapter. In this section, we prove that the conditions



T_1 T_5

Figure 3.6: Necessity: $T_1 \subset T_5$

of Theorem 3.1 are necessary for $\mathcal{P}$ to be basis-linearizable. The proof of necessity proceeds as follows, under the assumption of basis-linearizability in $\mathcal{P}$. Note that, in this case, the minimal violation of right-linearity (expansion T_1 in Figure 3.6) must be contained in some right-linear expansion (T_5 in Figure 3.6).

(a) If $p_{(1)}$ is an adjunct to $p_{(2)}$, then $p_{(1)}$ is trivial or $\mathcal{P} - p_{(1)}$ is one-bounded.

(b) Assume that the minimal violation of right-linearity (expansion T_1 in Figure 3.7) is contained in a right-linear expansion of depth at most 1 (one of T_2 and T_3 in Figure 3.7); then, the containment is provable by an acceptable containment mapping.

(c) Assume that T_1 is contained in a right-linear expansion of depth at least 2 (see Figure 3.8). Then one of the following must hold.

- $p_{(1)}$ is an adjunct to $p_{(2)}$.
- T_1 is contained in a right-linear expansion of depth at most 1.
- T_1 is contained in the right-linear of depth 2, and the containment is provable by an acceptable mapping. In fact, the mapping must have the form shown in Figure 3.9.

3.5.2 Sufficiency

In this section, we prove that the conditions of Theorem 3.1 are each sufficient to prove basis-linearizability in $\mathcal{P}$. The sufficiency of Condition 3 follows by Theorem 1.8. Consider Conditions 1 and 2.

Recall our convention that $t[i]$ refers to the ith argument of the atom t, and that $p_{(1)}$ is

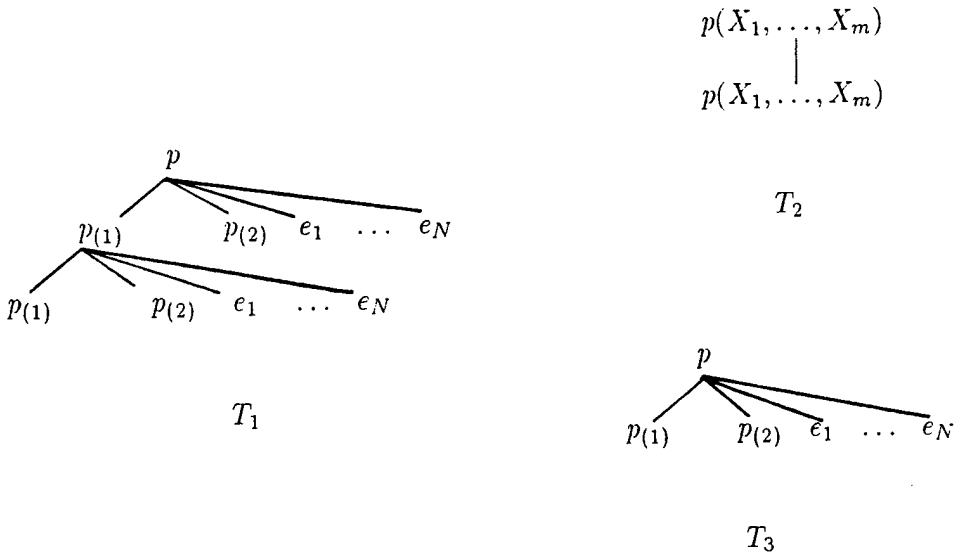


Figure 3.7: Assume $T_1 \subset T_2$ or $T_1 \subset T_3$

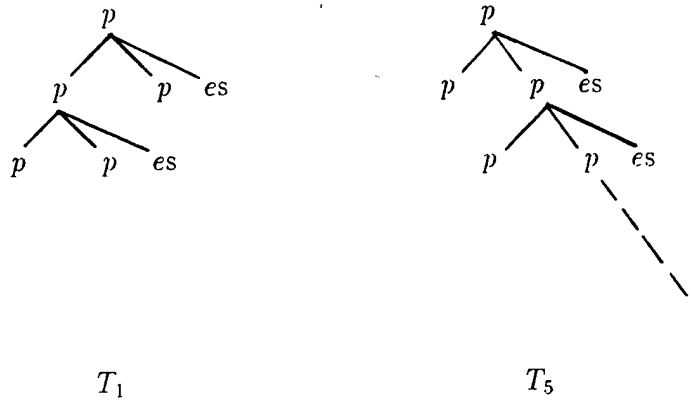


Figure 3.8: Assume $T_1 \subset T_5$.

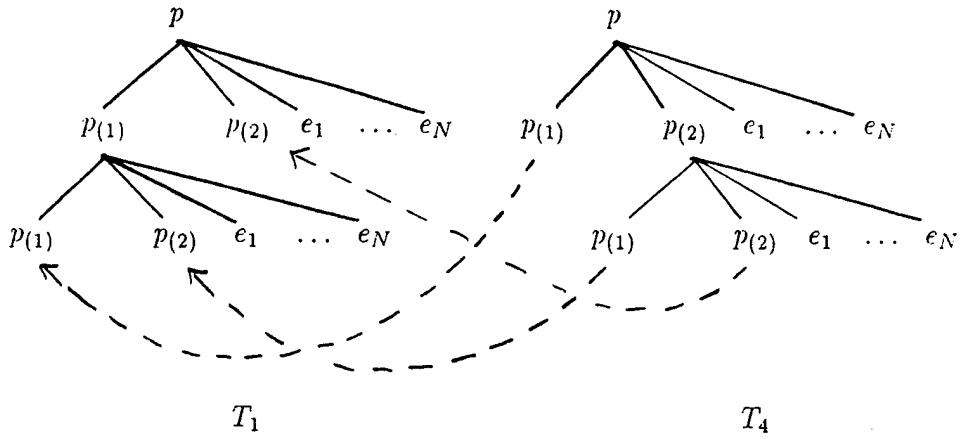


Figure 3.9: Show $T_1 \subset T_4$.

said to be a *trivial* adjunct to $p_{(2)}$ if $p_{(1)}$ is an adjunct, $p_{(1)}$ contains no nonlocal nondistinguished variables and, whenever the distinguished variable X_i appears in $p_{(1)}$, then $p_{(2)}[i]$ is X_i (that is, X_i appears in its "home" position in $p_{(2)}$). Throughout this chapter, X_i will always be used to refer to the ith distinguished variable in the head of the recursive rule.

Consider any tree T. We assume that every nondistinguished variable U in r_1 is renamed by 'priming' at each successive level of the expansion; that is, the version of U in sibling atoms in T is renamed to the new variable obtained by superscripting U with a "$\prime$" i times for some i (written $U^{(i)}$). If T is a linear expansion (that is, at most one p-atom is recursively expanded at any level), then the version of U at depth i in the expansion is $U^{(i-1)}$. Figure 3.6 illustrates this convention.

Let $q = q^1$ represent the version of the atom q at depth 1 in a tree T. If s is an atom in T, then $p_{(j)}s$ represents the atom s in the subtree T of some tree S that is rooted at the atom $p^1_{(j)}$. By convention, $p^i_{(j)}$ is the $p_{(j)}$-atom obtained by expanding $p^{(i-1)}_{(j)}$ through the recursive rule r_1.

Example 3.7 Consider the transitive-closure program of Example 3.1. The right-linear expansion of depth 3 has leaves and arguments as shown in Figure 3.10. □

Trivial adjuncts satisfy the following property.

Lemma 3.4 Assume that $p_{(1)}$ is a trivial adjunct to $p_{(2)}$ in $\mathcal{P}$. For all right-linear expansions T of depth n, for all integers $i, j < n$, and for all l:

1. If $p_{(1)}[l]$ is the distinguished variable X_k, then $p_{(2)}{}^i p_{(1)}[l]$ is also X_k; and
2. Any p-atom that unifies with $p_{(2)}{}^i p_{(1)}$ unifies with $p_{(2)}{}^j p_{(1)}$, and the unifications yield the same substitutions for all variables that are nonlocal to either of these two atoms in the conjunctive query represented by T.

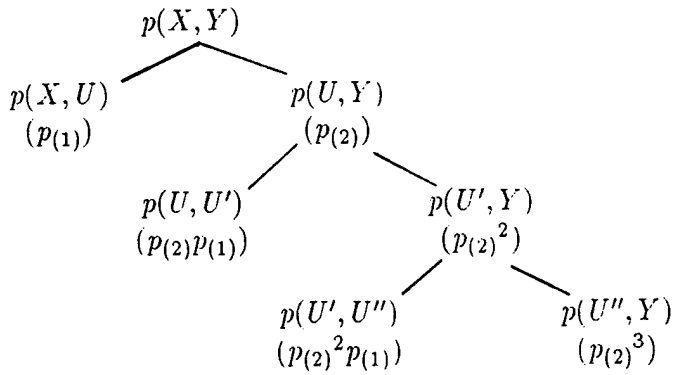


Figure 3.10: Naming and renaming conventions

Proof. Assume $p_{(1)}[l]$ is X_k. Since $p_{(1)}$ is a trivial adjunct, $p_{(2)}[k]$ is X_k. A straightforward induction on $r \geq 1$ shows that ${p_{(2)}}^r[k]$ is X_k, which immediately implies 1.

To prove 2, we note that the only nonlocal variables that appear in $p_{(1)}$ are distinguished variables, which (by 1) appear in the same positions in ${p_{(2)}}^i p_{(1)}$ and ${p_{(2)}}^j p_{(1)}$. All other variables in $p_{(1)}$ are local in r_1, and their primed versions therefore appear in only one atom in T (that is, $U^{(i)}$ appears only in the atom ${p_{(2)}}^i p_{(1)}$). □

Example 3.8 Consider the rule r_1 in Example 3.3. Figure 3.11 depicts an expansion using this rule. Note that X persists in its home position. □

Lemma 3.5 Assume that $p_{(1)}$ is a trivial adjunct to $p_{(2)}$, and let S be a proof tree of depth k generated by $\mathcal{P} - p_{(1)}$ from a database D. Then $\mathcal{P}$ generates a right-linear proof tree R of depth k from D such that S is its right strut (see Figure 3.12).

Corollary. If $p_{(1)}$ is a trivial adjunct to $p_{(2)}$, then $\mathcal{P}$ is basis-linearizable.

Proof. S has only one p-leaf, say ${p_{(2)}}^k(\vec{a})$. We may eliminate all other p-facts from D without altering the fact produced by S. By Lemma 3.3, $\mathcal{P}$ generates a complete tree I with depth k from D; hence, in I, the atom ${p_{(2)}}^{k-1}p_{(1)}$ unifies with the single p-fact in D. By Lemma 3.4, all the atoms ${p_{(2)}}^i p_{(1)}$ may be consistently unified with this p-fact to construct a right-linear proof tree R of depth k that is generated by $\mathcal{P}$ from D (see Figure 3.12).

The corollary is proved by observing that, by Lemma 3.2, every proof tree generated by $\mathcal{P}$ from a database D is also generated by $\mathcal{P} - p_{(1)}$ from D. □

Lemma 3.6 Assume that $p_{(1)}$ is an adjunct, and that $\mathcal{P} - p_{(1)}$ is 1-bounded. Then $\mathcal{P}$ is basis-linearizable.

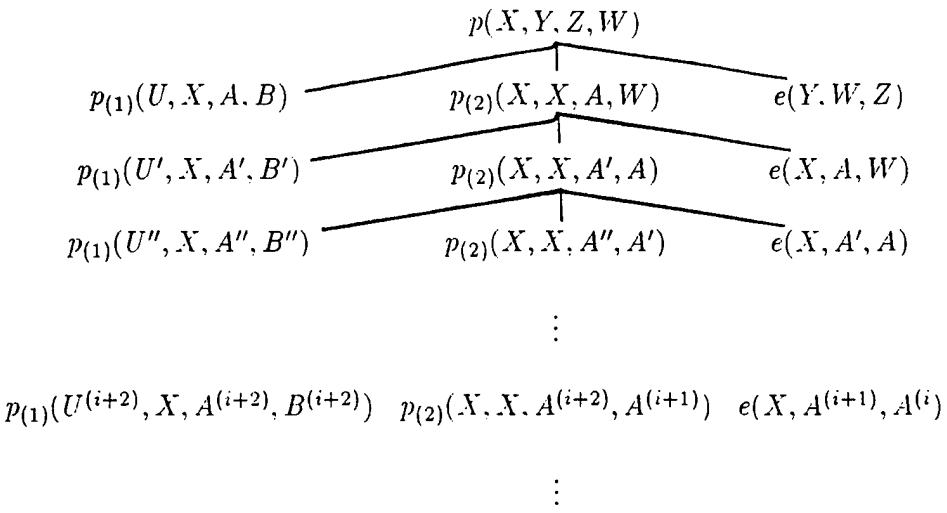


Figure 3.11: Persistence of X.

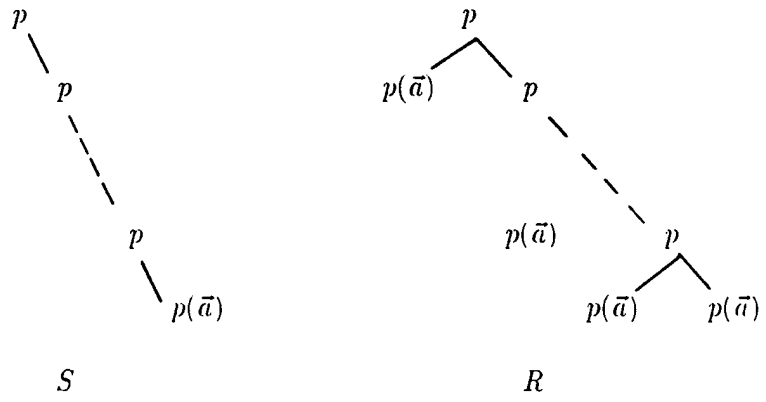
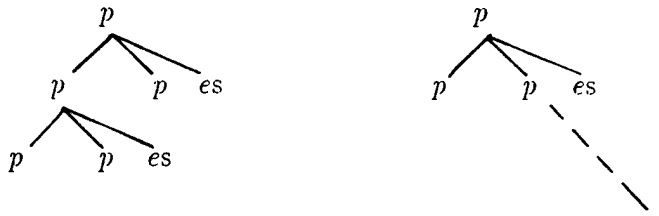


Figure 3.12: Trees S and R



T_1 T_5

Figure 3.13: Assume $T_1 \subset T_5$

Proof. Consider any database D, and any proof-tree S of depth k generated by $\mathcal{P}$ that establishes a fact $p(\vec{a})$. By Lemma 3.2, there is a proof tree I of depth k generated by $\mathcal{P} - p_{(1)}$ that also establishes this p-fact. If $\mathcal{P}$ is 1-bounded, then either $p(\vec{a}) \in D$ or $p(\vec{a})$ is generated by $\mathcal{P} - p_{(1)}$ using a tree J of depth 1. Applying Lemma 3.3 completes the proof. □

3.5.3 Necessity

This section will be devoted to the proof of necessity of the conditions in Theorem 3.1. If $\mathcal{P}$ is basis-linearizable, then every non-right-linear open expansion generated by $\mathcal{P}$ is contained in a right-linear expansion (i.e., that $T_1 \subset T_5$ in Figure 3.13). We will focus our attention on containment mappings that must exist from right-linear expansions into the minimal violation of right-linearity (T_1 in Figure 3.13).

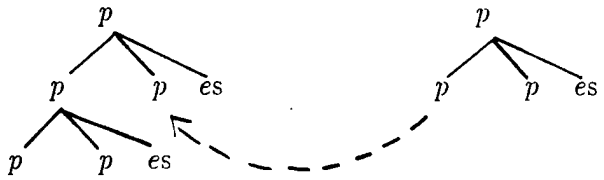
First, we will prove that if $\mathcal{P}$ is basis-linearizable and $p_{(1)}$ is an adjunct, then $p_{(1)}$ is a trivial adjunct or $\mathcal{P} - p_{(1)}$ is one-bounded.

Next, we will show that if T_1 is contained in a right-linear expansion of depth at most 1 under an unacceptable containment mapping (see Figure 3.14), then $p_{(1)}$ is an adjunct to $p_{(2)}$.

The remaining subsections concerns mappings that cannot exist from long right-linear trees into T_1. Specifically, we will show that if T_1 is contained in a right-linear tree T_5 of depth at least 2 (see Figure 3.15), then one of the following holds.

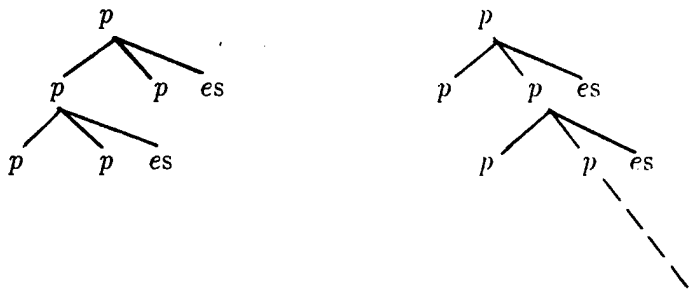
1. $p_{(1)}$ is an adjunct to $p_{(2)}$.

2. T_1 is contained in an expansion of depth at most 1.



T_1 T_3

Figure 3.14: Unacceptable mapping from T_3 into T_1.



T_1 T_5

Figure 3.15: Assume $T_1 \subset T_5$

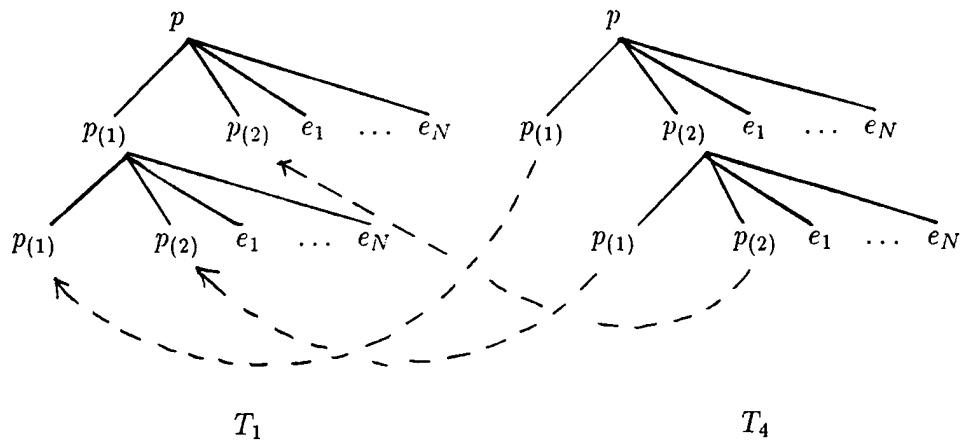


Figure 3.16: Conclude $T_1 \subset T_4$

3. T_5 has depth 2, and the containment mapping is acceptable. In fact, the mapping is of the form shown in Figure 3.16.

This procedure completes our proof. While reading these subsections, keep in mind the fact that if f is a containment mapping from some tree S into some tree T, the notation $f(q) = w$ implies that q is a leaf in S and w is a leaf in T.

Notation We will use the following notation to describe the arguments of p-atoms in the rule body. The expression

$$p_{(2)}(\ \overset{i}{A}\quad \overset{j}{X_i}\quad \overset{k}{X_j}\ )$$

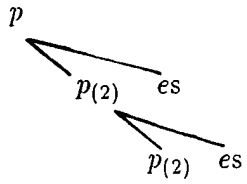
denotes the situation that some variable A appears in position i in $p_{(2)}$, the distinguished variable X_i appears in position j and the distinguished variable X_j appears in position k. Unless otherwise stated, A is an arbitrary variable (perhaps X_i), and any or all of $\{i, j, k\}$ may be equal.

Necessity of conditions 1 and 2

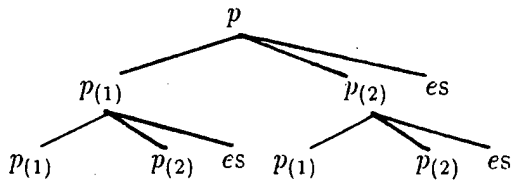
Assume that $\mathcal{P}$ is basis-linearizable. We will prove that, if $p_{(1)}$ is an adjunct to $p_{(2)}$ in $\mathcal{P}$, then either $\mathcal{P} - p_{(1)}$ is one-bounded or $p_{(1)}$ is trivial. Recall that nondistinguished variables in r_1 will be consistently renamed in each tree by *priming*.

Lemma 3.7 If f is a containment mapping from any tree S into any tree T, then $f(X_i) = X_i$ for every distinguished variable X_i.
Proof. Both trees have the root $p(X_1, \ldots, X_m)$. □

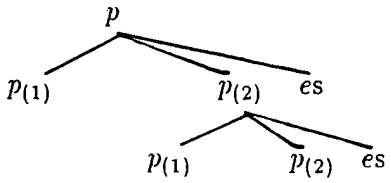
Assume that $p_{(1)}$ is an adjunct to $p_{(2)}$. Consider the expansion T_6 of Figure 3.17(b); since $\mathcal{P}$ is assumed to be basis-linearizable, the top-down expansion described by T_6 in



(a) T_8



(b) T_6



(c) T_9

Figure 3.17: $T_8 \subset T_6 \subset T_9$

Figure 3.17 is contained in some right-linear expansion T_9. Assume, now, that $\mathcal{P} - p_{(1)}$ is not one-bounded; we will show that $p_{(1)}$ is a trivial adjunct to $p_{(2)}$. By Lemma 3.3, the right strut T_8 of T_6 is contained in T_6, and we may conclude that T_8 is contained in T_9 (see Figure 3.17). Thus, $T_8 \subset T_9$, and T_9 has height at least two since $\mathcal{P} - p_{(1)}$ is assumed not to be 1-bounded.

Let f be a containment mapping from T_9 into T_8. Note that the only p-leaf in T_8 is $p_{(2)}p_{(2)}$, and this leaf must be the destination of every p-leaf in T_9.

Lemma 3.8 If $p_{(1)}[j]$ is X_i, then $p_{(2)}[i]$ is X_i.
Proof. By the definition of an adjunct, $p_{(2)}[j]$ is X_i. The picture is

$$p_{(1)}(\ \overset{j}{X_i}\ ) \qquad p_{(2)}(\ \overset{i}{C}\quad \overset{j}{X_i}\ )$$

. The mapping $f(p_{(1)}) = p_{(2)}p_{(2)}$ forces $f(X_i) = C$. Since f preserves distinguished variables, $C = X_i$. □

Lemma 3.9 No nonlocal nondistinguished variable appears in $p_{(1)}$.
Proof. Assume that the nonlocal nondistinguished variable A appears in position i in $p_{(1)}$; by the definition of an adjunct, $p_{(2)}[i] = A$. The picture is

$$p_{(1)}(\ \overset{i}{A}\ ) \qquad p_{(2)}(\ \overset{i}{A}\ )$$

Examine the lowest two levels of the tree T_9, which we assume has depth $n+1$ for some $n > 0$. The destination for ${p_{(2)}}^{(n-1)}p_{(1)}$ forces $f(A^{(n-1)}) = A'$.

By safety, X_i appears in the rule body. By Lemma 3.8, X_i cannot appear in $p_{(1)}$. Assume that it appears in position k in some e_q. Then, the kth argument of ${p_{(2)}}^{(n-1)}e_q$ is $A^{(n-1)}$. However, the kth argument of e_q is X_i by assumption, and the kth argument of $p_{(2)}e_q$ is therefore A (see Figure 3.11 for an illustration). Further, these two atoms are the only possible destinations for ${p_{(2)}}^{(n-1)}e_q$ because of our assumption of no repeated EDB predicates in the recursive rule. Hence, the possible destinations for ${p_{(2)}}^{(n-1)}e_q$ forces $f(A^{(n-1)})$ to be a nonprimed variable, which is a contradiction. Hence, X_i must appear in $p_{(2)}$ only, say in position $j \neq i$. The picture is

$$p_{(1)}(\ \overset{i}{A}\ ) \qquad p_{(2)}(\ \overset{i}{A}\quad \overset{j}{X_i}\ )$$

But then, the jth argument of ${p_{(2)}}^{(n+1)}$ is $A^{(n-1)}$, and the j-th argument of $p_{(2)}p_{(2)}$ is $A \neq A'$, a contradiction.

□

By Lemmas 3.8 and 3.9, $p_{(1)}$ must be a trivial adjunct to $p_{(2)}$.

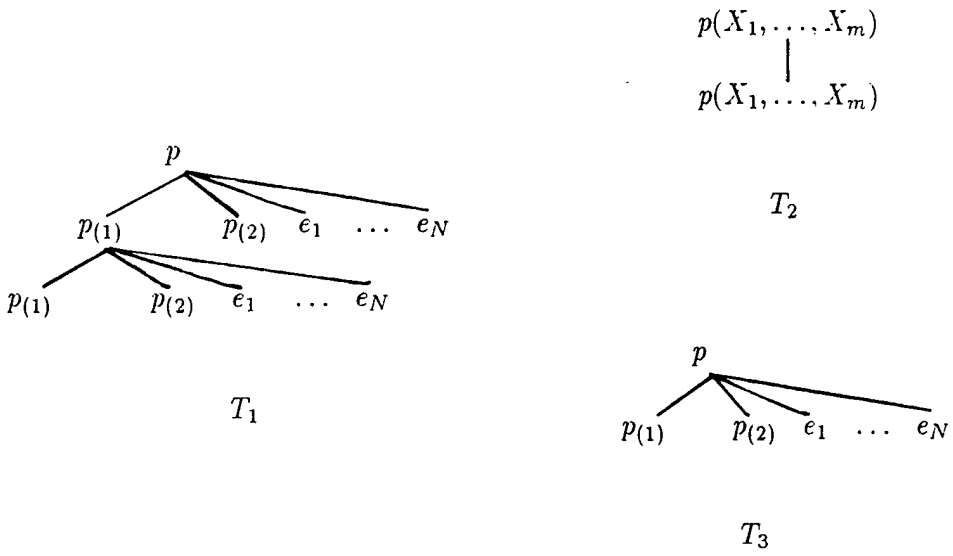


Figure 3.18: Minimal program: $T_1 \subset T_2$ or $T_1 \subset T_3$

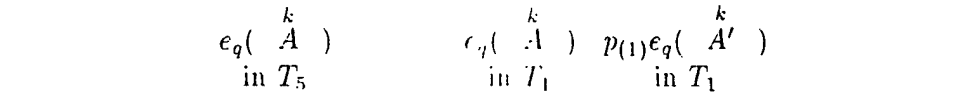
Minimal programs

Let us say that $\mathcal{P}$ is *minimal* iff the minimal violation of right-linearity (T_1 in Figure 3.18) is contained in a right-linear expansion of depth at most 1 (that is, in one of the expansions T_2 and T_3 in the same figure).

Assume that $\mathcal{P}$ is basis-linearizable and minimal. We will show that if $p_{(1)}$ is not an adjunct to $p_{(2)}$, then there is an acceptable containment mapping from T_2 or T_3 into T_1. Since any mapping from T_2 into T_1 is acceptable, we will concern ourselves only with an unacceptable containment mapping f from T_3 into T_1; that is, a mapping in which $f(p_{(1)}) = p_{(2)}$ (see Figure 3.19).

Lemma 3.10 Assume that a nondistinguished variable A appears in an EDB subgoal e_q and that f is a containment mapping from a right-linear tree T_5 of depth at least 1 into T_1. Then $f(A) = A$ or $f(A) = A'$.

Proof. Assume that A appears in the kth position in e_q. Since T_5 has depth at least 1, the atom e_q appears as a leaf. Because there are no repetitions of any EDB predicate in the subgoals of the recursive rule, the only e_q-atoms in T_1 are e_q and $p_{(1)}e_q$; the kth argument of the former is A and the kth argument of the latter is A'.



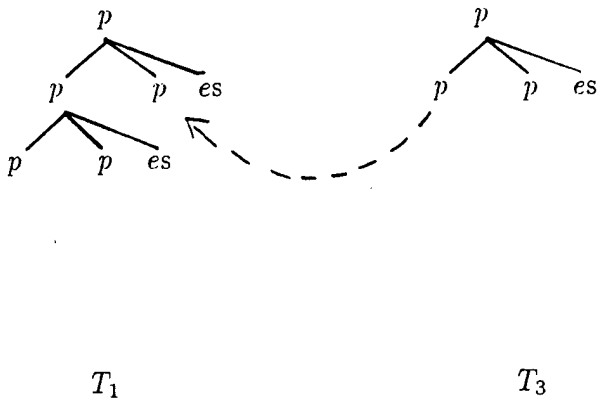


Figure 3.19: Unacceptable containment mapping

Considering all possible destinations in T_1 for the e_q-leaf in T_5 suffices to complete the proof. □

Recall our convention that $t[i]$ refers to the ith argument of the atom t.

Lemma 3.11 Assume that $\mathcal{P}$ is minimal, and that f is a containment mapping from T_3 into T_1 under which the destination of $p_{(1)}$ is $p_{(2)}$ (see Figure 3.19). Then $p_{(1)}$ is an adjunct to $p_{(2)}$.

Proof. Assume that a distinguished variable X_i appears in some position j in $p_{(1)}$. By Lemma 3.7 and the assumed mapping $f(p_{(1)}) = p_{(2)}$, $p_{(2)}[j]$ is X_i. To complete the proof, we show that if any nondistinguished nonlocal variable A appears in any position i in $p_{(1)}$, then A also occupies position i in $p_{(2)}$. Assume the existence of a nondistinguished variable A in the ith argument position in $p_{(1)}$, such that A appears in $p_{(2)}$, or in some e_q.

If A appears as an argument to some e_q, then by Lemma 3.10, $f(A)$ is A or A'. Since no primed variable appears in $p_{(2)}$, the mapping $f(p_{(1)}) = p_{(2)}$ forces $p_{(2)}[i] = A$.

Assume now that A appears in the jth position in the arguments of $p_{(2)}$. The picture is

$$p_{(1)}(\ \overset{i}{A}\quad \overset{j}{B}\ ) \qquad p_{(2)}(\ \overset{i}{C}\quad \overset{j}{A}\ )$$

Now, B cannot be distinguished, otherwise by previous discussion, A would also have to be distinguished. By the assumed mapping on $p_{(1)}$, we have $f(A) = C$. Since the jth arguments of $p_{(1)}p_{(1)}$ and $p_{(1)}p_{(2)}$ are both primed variables, we conclude that $f(p_{(2)}) = p_{(2)}$. Then $f(A) = A$ and $C = A$. □

Connectivity

It turns out that if $\mathcal{P}$ is basis-linearizable and not minimal, then the p-atoms $p_{(1)}$ and $p_{(2)}$ in the body of the recursive rule must be connected. The primary tools used in the remainder of the proof are connectivity and safety.

Two atoms in the body of the recursive rule r_1 are said to be *directly connected* if they share a nondistinguished variable; *connectivity* is defined as the transitive closure of the direct connectivity relation. Similarly, two nondistinguished variables are connected if they appear in a pair of connected atoms. The special case of connectivity between the recursive atoms $p_{(1)}$ and $p_{(2)}$ is formally defined below; the formality is necessary to the results of the remainder of this chapter.

Definition 3.6 Assume that the nondistinguished variable A appears in the arguments of $p_{(1)}$, and that the nondistinguished variable B appears in the arguments of $p_{(2)}$. We say that A and B are *directly connected* if $A = B$. The atoms $p_{(1)}$ and $p_{(2)}$ are said to be directly connected if they share a nondistinguished variable A. We say that A and B are *indirectly connected* if $A \neq B$, and if for $n > 0$, there are nondistinguished variables $U_1, \ldots, U_{n+1}$ and distinct EDB subgoals $e_{k_1}, \ldots, e_{k_n}$ such that

1. $A = U_1$.
2. $B = U_{n+1}$.
3. For $1 \leq i \leq n$, the variables U_i and U_{i+1} appear in the arguments of e_{k_i}.

The sequence $< e_{k_1}, \ldots, e_{k_n} >$ is termed a *connection sequence between A and B*, and the sequence $< U_1, \ldots, U_{n+1} >$ is termed the *corresponding variable sequence.*

□

As we mentioned earlier, nondistinguished variables in r_1 will be consistently renamed in each tree by *priming.* That is, if A is a nondistinguished variable, then its occurrences in any tree will be $A^{(i)}$ where i is an integer indicating superscripts of i "$'$"s. Further, occurrences of the nondistinguished variables A and B in sibling atoms will bear the same superscript. If the tree is linear, then $A^{(i)}$ is the occurrence of A at depth $i-1$ in the tree.

Let f be a containment mapping from a right-linear expansion R of depth at least 1 into an expansion S. Recall that if f is a mapping from some tree S into some tree T, the notation $f(q) = w$ implies that q is a leaf in S and w is a leaf in T.

Lemma 3.12 Assume that the nondistinguished variables V and W appear in the arguments of some EDB predicate e_q. For any i, assume that f is a containment mapping from a right-linear tree R of depth at least $(i+1)$ into an expansion S, and let $V^{(i)}$ and $W^{(i)}$ be the respective occurrences of V and W at depth $(i+1)$ in R. Then, there is some j such that $f(V^{(i)}) = V^{(j)}$ and $f(W^{(i)}) = W^{(j)}$.

Proof. Assume that V and W appear in the positions k and l in e_q, respectively. The kth and lth arguments of the e_q-atom at depth $(i+1)$ in R are $V^{(i)}$ and $W^{(i)}$ respectively.

Since there are no EDB repetitions in the recursive rule, every e_q-atom in S has kth and lth arguments $V^{(j)}$ and $W^{(j)}$ respectively, for some j. □

Lemma 3.13 Assume that R is a right-linear tree of depth m, S is any expansion and f if a containment mapping from R into S. Assume that the nondistinguished variable A in $p_{(1)}$ is connected to the variable B in $p_{(2)}$. Then for $0 \leq i < m$, exactly one of the following is true.

1. $A = B$ and $f(A^{(i)}) = f(B^{(i)})$.
2. $A \neq B$, and there is some j such that $f(A^{(i)}) = A^{(j)}$ and $f(B^{(i)}) = B^{(j)}$.

Proof. If A and B are directly connected, then $A = B$ and (1) above is true. If A and B are indirectly connected, then $A \neq B$.

Let

$$< e_{k_1}, \ldots, e_{k_n} >$$

be a connection sequence between A and B, and let

$$< U_1, \ldots, U_{n+1} >$$

be the corresponding variable sequence. Recall that $A = U_1$ and $B = U_{n+1}$. We show that (2) above is true.

Consider the possible destinations for the atom ${p_{(2)}}^i e_{k_1}$, which contains the variables $A^{(i)} = U_1^{(i)}$ and $U_2^{(i)}$. By Lemma 3.12, there is some j such that $f(A^{(i)}) = A^{(j)}$ and $f(U_2^{(i)}) = U_2^{(j)}$. If $U_2 = B$, then the result follows. Otherwise, the variables $U_2^{(i)}$ and $U_3^{(i)}$ appear in the leaf ${p_{(2)}}^i e_{k_2}$; the use of Lemma 3.12 and the single-valuedness of f shows that $f(U_3^{(i)}) = U_3^{(j)}$. An inductive repetition suffices to complete the proof. □

Note that the assumption that there are no EDB repetitions in the body of the recursive rule is essential to the proof of Lemmas 3.12 and 3.13.

In the remainder of this subsection, we will show that if $\mathcal{P}$ is basis-linearizable, but $p_{(1)}$ and $p_{(2)}$ are not connected, then $\mathcal{P}$ is minimal. Recall that that $\mathcal{P}$ is termed minimal if the small non-right-linear tree T_1 is contained in a right-linear tree of height at most one (that is, in one of T_2 and T_3).

Lemma 3.14 If $\mathcal{P}$ is basis-linearizable and not connected, then $\mathcal{P}$ is minimal.

Proof. Assume that f is a containment mapping from some right-linear tree T_5 of height $n > 1$ into T_3. We construct a containment mapping g from T_3 into T_1. We define the destinations of g as follows: g preserves the destination of $p_{(1)}$, and of any e_q connected to $p_{(1)}$; all other atoms in r_1 are mapped to "themselves" under g. More formally, we partition nondistinguished variables into two classes. A nondistinguished variable is termed *tied* if it appears in $p_{(1)}$ or in any predicate that is connected to $p_{(1)}$, and *free* otherwise. We define g as under.

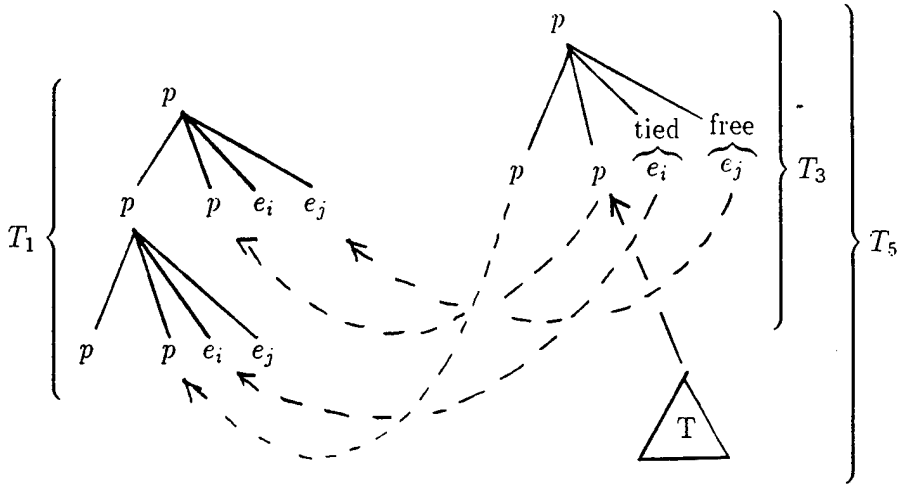


Figure 3.20: $g : T_3 - T_1$

$$g(V) = \begin{cases} V & \text{if } V \text{ is distinguished} \\ f(V) & \text{if } V \text{ is tied} \\ V & \text{if } V \text{ is free} \end{cases}$$

It is easily seen that g is a containment mapping from the depth-1 expansion T_3 into T_1 (see Figure 3.20). □

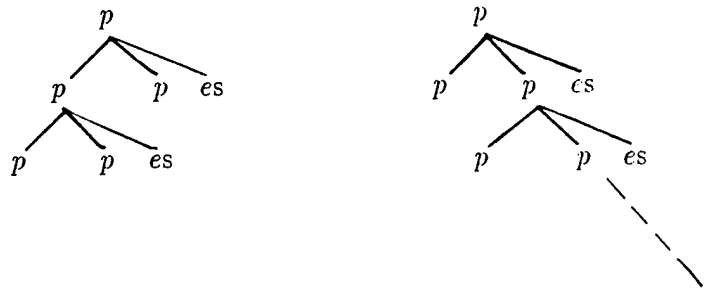
Non-minimal programs

Finally, let us turn our attention to the case in which the minimal violation of right-linearity (T_1 in Figure 3.21) is contained in a right-linear expansion T_5 of depth at least 2 (see Figure 3.21).

Assume that $\mathcal{P}$ is basis-linearizable, and that f is a containment mapping from T_5 into T_1. By a case analysis on the possible destinations for the leaf $p_{(1)}$ (at depth 1) in the right-linear tree T_5, we will show the following.

1. If $f(p_{(1)}) = p_{(1)}p_{(1)}$ and $\mathcal{P}$ is not minimal, then T_5 has depth 2, $f(p_{(2)}p_{(1)}) = p_{(1)}p_{(2)}$ and $f(p_{(2)}p_{(2)}) = p_{(2)}$; i.e., f is an acceptable containment mapping. (Recall that an expression of the form "$f(t) = s$" means that t is a leaf in the containing tree, and s is a leaf in the contained tree). Figure 3.22 describes this claim.

2. If $f(p_{(1)}) = p_{(1)}p_{(2)}$ or $f(p_{(1)}) = p_{(2)}$ (see Figure 3.23), then $p_{(1)}$ is an adjunct to $p_{(2)}$ or $\mathcal{P}$ is minimal.



T_1 T_5

Figure 3.21: Assume $T_1 \subset T_5$

Let us end this section with some observations on the nature of the containment mapping f.

Lemma 3.15 If the nondistinguished variable A appears in some e_q, then $f(A) = A$ or $f(A) = A'$, and $f(A') = A$ or $f(A') = A'$.
Proof. Assume the presence of the atom

$$e_q(\overset{i}{A})$$

in r_1. Since T_5 is of depth at least 2, the two leaves

$$e_q(\overset{i}{A}) \qquad p_{(2)}e_q(\overset{i}{A'})$$

must appear in it. Since there are no subgoal repetitions in r_1, the only e_q-atoms in T_1 are of the form

$$e_q(\overset{i}{A}) \qquad p_{(1)}e_q(\overset{i}{A'})$$

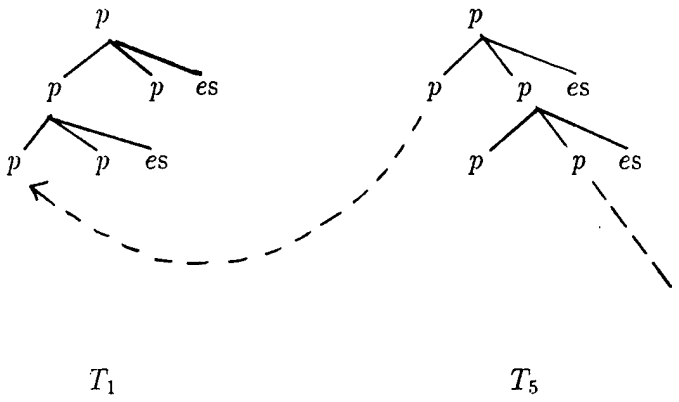
□

Lemma 3.16 Assume that the following variables appear in the indicated positions in r_1:

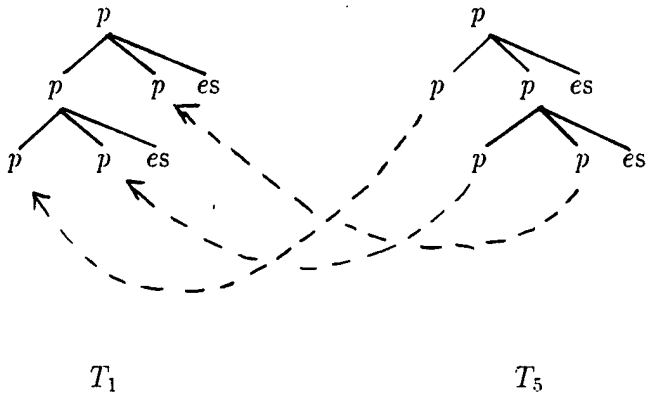
$$p_{(1)}(\overset{i}{C}) \qquad p_{(2)}(\overset{i}{A}) \qquad e_q(\overset{j}{X_i})$$

Then $f(A)$ is X_i or C.

Proof. Since T_5 is of height at least 2, the atom $p_{(2)}e_q$ exists in T_5, with form $e_q(\overset{j}{A})$. The only e_q-atoms in T_1 are of the form $e_q(\overset{j}{C})$ and $e_q(\overset{j}{A})$. □



(a) Assumption.



(b) Conclusion.

Figure 3.22: The case $f(p_{(1)}) = p_{(1)}p_{(1)}$.

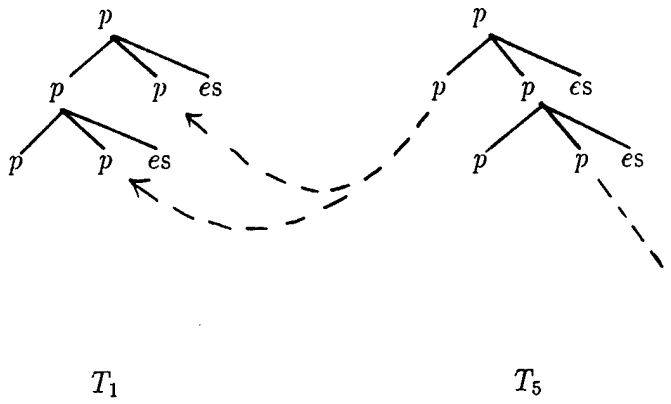


Figure 3.23: The cases $f(p_{(1)}) = p_{(1)}p_{(2)}$ and $f(p_{(1)}) = p_{(2)}$.

The case $f(p_{(1)}) = p_{(1)}p_{(1)}$ Assume that $\mathcal{P}$ is basis-linearizable. Then the minimal violation T_1 of right linearity (see Figure 3.24) is contained in a right-linear expansion T_5. Assume that T_5 has depth at least 2, and that f is a containment mapping from T_5 into the minimal violation T_1, such such that $f(p_{(1)}) = p_{(1)}p_{(1)}$ (see Figure 3.24(a)). We will show that $\mathcal{P}$ is minimal, or that $f(p_{(2)}p_{(1)}) = p_{(2)}$, T_5 has height 2 and $f(p_{(2)}p_{(2)}) = p_{(2)}$ (see Figure 3.24(b)).

If $p_{(1)}$ and $p_{(2)}$are not connected, then $\mathcal{P}$ is minimal by Lemma 3.14. Assume that $p_{(1)}$ and $p_{(2)}$ are connected. The mapping $f(p_{(1)}) = p_{(1)}p_{(1)}$ yields the following results.

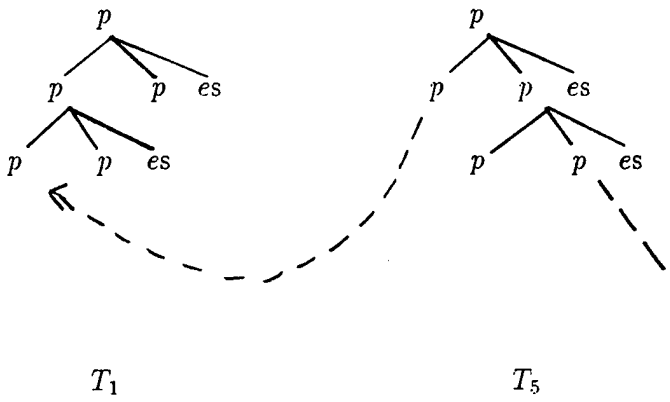
Definition 3.7 We say that $p_{(1)}$ is *invariant* if whenever any distingusihed variable X_i appears in it, then $p_{(1)}[i]$ is X_i. That is, X_i (also) appears in its "home" position in $p_{(1)}$. □

Lemma 3.17 Assume that $p_{(1)}$ is invariant. If X_i appears in any position j in $p_{(1)}$, then $p_{(1)}p_{(1)}[j]$ is X_i.
Proof. By invariance, $p_{(1)}[i] = X_i$, and our result follows immediately. □

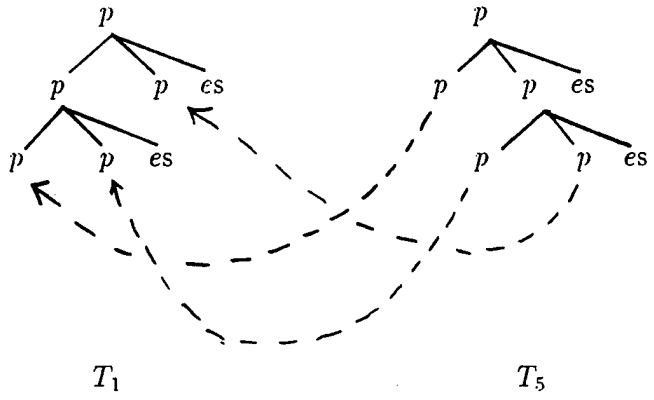
Lemma 3.18 If $f(p_{(1)}) = p_{(1)}p_{(1)}$, then $p_{(1)}$ is invariant, and $f(A) = A'$ for all nondistinguished variables A in $p_{(1)}$. Further, any variable B in $p_{(2)}$ that is connected to a variable A in $p_{(1)}$also satisfies $f(B) = B'$.
Proof. Assume that X_i appears in position $j \neq i$ in $p_{(1)}$. The picture is

$$p_{(1)}(\ \overset{i}{C}\quad \overset{j}{X_i}\ )$$

The jth argument of $p_{(1)}p_{(1)}$ is C. By the assumed mapping $f(p_{(1)}) = p_{(1)}p_{(1)}$, $f(X_i) = C$; since $f(X_i) = X_i$, our result follows. Otherwise, if X_i appears in $p_{(1)}$, then it occupies position i. Hence, $p_{(1)}$ is invariant.



(a) Assumption.



(b) Conclusion.

Figure 3.24: The case $f(p_{(1)}) = p_{(1)}p_{(1)}$.

If the nondistinguished variable A appears in the ith position in $p_{(1)}$, then A' appears in the ith position in $p_{(1)}p_{(1)}$; hence, $f(A) = A'$. If a nondistinguished variable B appearing in $p_{(2)}$ is connected to A, then $f(B) = B'$ by Lemma 3.13. □

Proving that $f(p_{(2)}p_{(1)}) = p_{(1)}p_{(2)}$. We now prove that $f(p_{(2)}p_{(1)}) = p_{(1)}p_{(2)}$. Recall that f is a containment mapping from a right-linear expansion T_5 of depth at least 2 into the minimal violation T_1, such that $f(p_{(1)}) = p_{(1)}p_{(1)}$.

Definition 3.8 Define a *Class* 1 variable to be a distinguished variable X_j that appears only among the arguments of $p_{(2)}$ (that is, X_j does not appear in the arguments of $p_{(1)}$ or any e_q). Define a *Class* 2 variable to be a Class 1 variable X_j that appears in some position $k \neq j$ in the arguments of $p_{(2)}$, but does not appear in position j (its "home" position). Define a *Class* 3 variable to be a Class 2 variable X_k, such that the kth argument of $p_{(2)}$ is some Class 2 variable X_j. That is, the arguments of $p_{(2)}$ are as follows, where $C \neq X_j$ and $X_j \neq X_k$.

$$p_{(2)}(\ \overset{j}{C}\quad \overset{k}{X_j}\quad \overset{l}{X_k}\ )$$

For $1 \leq i \leq 3$, an argument position l is termed *Class* i iff $p_{(2)}[l]$ is a Class i variable. □

Lemma 3.19 Let l be a Class 3 position. Then X_l is a Class 3 variable.
Proof. Assume that X_k is a Class 3 variable appearing as the lth argument of $p_{(2)}$. By definition, $k \neq l$. The picture is as follows, where X_j and X_k appear only in $p_{(2)}$, $j \neq k, k \neq l$ and $C \neq X_j$.

$$p_{(2)}(\ \overset{j}{C}\quad \overset{k}{X_j}\quad \overset{l}{X_k}\ )$$

By safety, X_l appears in $p_{(1)}$, $p_{(2)}$ or some e_q. Since X_k does not appear in $p_{(1)}$ and $X_k \neq X_l$, X_l cannot appear in any e_q by Lemma 3.16. Assume, now that X_l appears in $p_{(1)}$; by invariance, it must occupy position l. Since T_5 has depth at least 2, one of the atoms $p_{(2)}p_{(2)}$ or $p_{(2)}p_{(2)}p_{(1)}$ must appear as a leaf, and the lth argument of each of these atoms is X_j. Since X_j does not appear in $p_{(1)}$, it cannot appear in either of the leaves $p_{(1)}p_{(1)}$ or $p_{(1)}p_{(2)}$ in T_1; further, the lth argument of $p_{(2)}$ is X_k, which is distinct from X_j by assumption. Hence, neither of the atoms $p_{(2)}p_{(2)}$ and $p_{(2)}p_{(2)}p_{(1)}$has a legal destination in T_1. Hence, by safety, X_l must appear only in $p_{(2)}$ and our result follows. □

Lemma 3.20 For all i, i is a Class 3 position iff X_i is a Class 3 variable.
Proof. The "only if" follows from Lemma 3.19. For the converse, note that by definition, if there are k Class 3 variables then there are at least k Class 3 positions; our result follows by Lemma 3.19 and by pigeonholing, recalling the fact that all distinguished variables are distinct. □

Lemma 3.21 The arguments of $p_{(1)}$ and $p_{(2)}$ cannot be of the form

$$p_{(1)}(\ \overset{i}{A}\ ) \qquad p_{(2)}(\ \overset{j}{B}\quad \overset{k}{X_i}\ )$$

where A and B are connected nondistinguished variables and X_j appears only in $p_{(2)}$.
Proof. Assume the converse. Since B is nondistinguished, $j \neq k$; that is, X_j is a Class 2 variable. By Lemma 3.18, $f(B) = B'$., and $p_{(1)}$ is invariant. By safety, X_k must appear in $p_{(1)}$, $p_{(2)}$ or some e_q.

Since X_j does not appear in $p_{(1)}$, Lemma 3.16 prohibits X_k from appearing in any e_q. If X_k appears only in $p_{(2)}$, then it is a Class 3 variable; by Lemma 3.20, k must be a Class 3 position, and X_j a Class 3 variable; but $p_{(1)}[j]$ is nondistinguished, a contradiction. Hence, X_k must appear in the arguments of $p_{(1)}$, and must occupy position k by invariance. The picture is

$$p_{(1)}(\ \overset{i}{A}\quad \overset{k}{X_k}\ ) \qquad p_{(2)}(\ \overset{j}{B}\quad \overset{k}{X_j}\ )$$

Now, one of $p_{(2)}p_{(2)}$ and $p_{(2)}p_{(2)}p_{(1)}$ must appear as a leaf in T_5, and B appears as the kth argument of each of these leaves. However, the kth argument of each leaf in T_1 is a nonprimed variable, contradicting the fact that $f(B) = B'$. □

Lemma 3.22 Assume that the arguments of $p_{(1)}$ and $p_{(2)}$ are as follows

$$p_{(1)}(\ \overset{i}{A}\ ) \qquad p_{(2)}(\ \overset{j}{B}\ )$$

where A and B are connected nondistinguished variables. Then $p_{(1)}[j]$ is X_j.
Proof. By Lemma 3.18 and connectivity, $f(B) = B'$. By safety, X_j must appear in the body. However, by Lemma 3.16, it cannot appear in e_q. By Lemma 3.21, X_j cannot appear only in $p_{(2)}$. Hence, X_j must appear in $p_{(1)}$, and the result follows by invariance. □

Lemma 3.23 Let f be a containment mapping from a right-linear expansion T_5 of depth at least 2 into the minimal violation T_1, and assume that $\mathcal{P}$ is not minimal. Then $f(p_{(2)}p_{(1)}) = p_{(1)}p_{(2)}$.
Proof. Since $\mathcal{P}$ is not minimal, $p_{(1)}$ and $p_{(2)}$ must be connected. By Lemma 3.22, we must have the following situation, where A and B are connected nondistinguished variables.

$$p_{(1)}(\ \overset{i}{A}\quad \overset{j}{X_j}\ ) \qquad p_{(2)}(\ \overset{j}{B}\ )$$

By Lemma 3.18, $f(A) = A'$ and $f(B) = B'$. Now, the jth argument of $p_{(2)}p_{(1)}$ is B, but the jth arguments of the leaves $p_{(1)}p_{(1)}$ and $p_{(2)}$ in T_1 are not B'; that is, the p-leaves in T_1 are as below.

$$p_{(1)}p_{(1)}(\ \overset{j}{X_j}\ ) \qquad p_{(1)}p_{(2)}(\ \overset{j}{B'}\ ) \qquad p_{(2)}(\ \overset{j}{B}\ )$$

Hence, $f(p_{(2)}p_{(1)}) = p_{(1)}p_{(2)}$. □

T_5 has height 2 Recall that a Class 1 variable is a distinguished variable that appears only in the arguments of $p_{(2)}$, and that any position in $p_{(2)}$ that is occupied by a Class 1 variable is a Class 1 position. It turns out that for all i, X_i is a Class 1 variable iff i is a Class 1 position.

Lemma 3.24 For all i, if i is a Class 1 position, then X_i is a Class 1 variable.
Corollary. i is a Class 1 position iff X_i is a Class 1 variable.
Proof. Assume that i is a Class 1 position; that is, $p_{(2)}[i]$ is a variable X_k that appears only in $p_{(2)}$. If $k = i$ then the result follows. Assume $k \neq i$. By safety, X_i appears in $p_{(1)}$, some e_q or $p_{(2)}$. By Lemma 3.16 and since X_k (by assumption) does not appear in $p_{(1)}$, X_i cannot appear in any e_q. If X_i appears in $p_{(1)}$, then by invariance it occupies position i. But then, the ith argument of $p_{(2)}p_{(1)}$ is X_k, which by assumption does not appear in $p_{(1)}$ (and therefore does not appear in $p_{(1)}p_{(2)}$); hence, $f(p_{(2)}p_{(1)}) \neq p_{(1)}p_{(2)}$, contradicting Lemma 3.23. Therefore, X_i must appear only in $p_{(2)}$.

To prove the corollary, we proceed as follows. By definition, if there are k Class 1 variables, then there are at least k Class 1 positions. By the preceding result, the number of Class 1 variables is no smaller than the number of Class 1 positions; thus, the number of Class 1 variables is the same as the number of Class 1 positions. The corollary now follows by the preceding result and pigeonholing. □

Now, we show that if A and B are connected nondistinguished variables, then $f(A') = A$ and $f(B') = B$. The idea is that we know that $p_{(2)}p_{(1)}[i] = A'$, and that $f(p_{(2)}p_{(1)}) = p_{(1)}p_{(2)}$; hence, we need only discover the value of $p_{(1)}p_{(2)}[i]$.

Lemma 3.25 Let f be a containment mapping as before, and assume the following picture

$$p_{(1)}(\ \overset{i}{A}\ ) \qquad p_{(2)}(\ \overset{j}{B}\ )$$

where A and B are connected nondistinguished variables. Then, either $A = B$ and $p_{(1)}p_{(2)}[i]$ is X_j, or $p_{(1)}p_{(2)}[i]$ is A.
Proof. By Lemma 3.23, we have the picture

$$p_{(1)}(\ \overset{i}{A}\ \ \overset{j}{X_j}\ ) \qquad p_{(2)}(\ \overset{j}{B}\ )$$

Now, the jth argument of the atoms $p_{(2)}p_{(2)}$ and $p_{(2)}p_{(2)}p_{(1)}$ is B', and one of these atoms appears as a leaf in T_5. The jth arguments of the leaves in T_1 are X_j, B' and B, so $f(B')$ is one of X_j, B' and B. By connectivity (Lemma 3.13), if $f(B') = X_j$, then $A = B$ and $f(A')$ is X_j; if $f(B') = B$ then $f(A') = A$; and if $f(B') = B'$ then $f(A') = A'$.

Consider the case $f(B') = B'$; then, by connectivity, we must have $f(A') = A'$. Now, we know by Lemma 3.23 that $f(p_{(2)}p_{(1)}) = p_{(1)}p_{(2)}$, and that A' appears as the ith argument of the former. If $p_{(1)}p_{(2)}[i]$ is A', then the variable A appears in the ith position in $p_{(1)}$ and $p_{(2)}$. The picture is

$$p_{(1)}(\ \overset{i}{A}\ ) \qquad p_{(2)}(\ \overset{i}{A}\ )$$

By safety, X_i must appear in the rule body. By Lemma 3.16, if X_i appears in any e_q, then $f(A)$ is one of X_i and A, contradicting Lemma 3.18. By invariance (since A is nondistinguished), X_i does not appear in $p_{(1)}$. However, X_i cannot appear only in $p_{(2)}$, since it would be a Class 1 variable and $p_{(2)}[i]$ is not Class 1. Hence, $f(B') \neq B'$ and our result follows. □

Lemma 3.26 Assume the following arguments for $p_{(1)}$ and $p_{(2)}$

$$p_{(1)}(\overset{i}{A}) \qquad p_{(2)}(\overset{j}{A})$$

where A is a nondistinguished variable. Then $p_{(1)}p_{(2)}[i] \neq X_j$.
Proof. Assume the converse. Then by assumption and invariance, the picture is

$$p_{(1)}(\overset{i}{A} \quad \overset{j}{X_j} \quad \overset{k}{X_j}) \qquad p_{(2)}(\overset{i}{X_k} \quad \overset{j}{A})$$

where $i \neq j, i \neq k$ since A is nondistinguished.

By Lemma 3.18, $f(A) = A'$, and Lemma 3.23 requires that $f(p_{(2)}p_{(1)}) = p_{(1)}p_{(2)}$; since the jth argument of $p_{(2)}p_{(1)}$is A, we conclude that $p_{(1)}p_{(2)}[k] = A'$ and $p_{(2)}[k] = A$.

Now, by invariance, X_i cannot appear in $p_{(1)}$. If X_i appears in any e_q, then by Lemma 3.16, we must have $f(X_k) = A$ or $f(X_k) = X_i$, a contradiction since $i \neq k$. Hence, X_i appears only in $p_{(2)}$, say in position l distinct from i, j and k. We now have the picture

$$p_{(1)}(\overset{i}{A} \quad \overset{j}{X_j} \quad \overset{k}{X_j}) \qquad p_{(2)}(\overset{i}{X_k} \quad \overset{j}{A} \quad \overset{k}{A} \quad \overset{l}{X_i})$$

where $i \neq j, i \neq k, i \neq l, j \neq l$. However, by the corollary to Lemma 3.24 and since X_i is Class 1, $p_{(2)}[i]$ (X_k) must also be Class 1; but $p_{(2)}[k]$ is nondistinguished (and hence not Class 1), contradicting Lemma 3.24. □

Lemma 3.27 Assume that the p-atoms in $\mathcal{P}$ are of the form

$$p_{(1)}(\overset{i}{A}) \qquad p_{(2)}(\overset{j}{B})$$

where A and B are connected nondistinguished variables. Then $f(A') = A$ and $f(B') = B$.
Proof. By Lemmas 3.23, 3.25 and 3.26, and by connectivity. □

Lemma 3.28 T_5 has height 2.
Proof. Assume that A and B are connected nondistinguished variables appearing in $p_{(1)}$ and $p_{(2)}$ respectively. The picture is

$$p_{(1)}(\overset{i}{A}) \qquad p_{(2)}(\overset{j}{B})$$

By Lemma 3.22, $p_{(1)}[j] = X_j$. Further, by Lemmas 3.23 and 3.27, $p_{(2)}[i] = X_l$ for some l such that $p_{(1)}[l] = A$. Thus, we have the picture

$$p_{(1)}(\ \overset{i}{A}\quad \overset{j}{X_j}\quad \overset{l}{A}\ ) \qquad p_{(2)}(\ \overset{i}{X_l}\quad \overset{j}{B}\ )$$

where A and B are connected, and where $X_l \neq X_j$. We assume T_5 has height at least 3, and force a contradiction.

Now, $p_{(2)}p_{(2)}p_{(1)}[i] = A''$ and $p_{(2)}p_{(2)}p_{(1)}[j] = B'$. The mapping $f(B') = B$ forces $f(p_{(2)}p_{(2)}p_{(1)}) = p_{(2)}$, yielding $f(A'') = X_l$. By connectivity, $A = B$. But then the jth argument of each of $p_{(2)}p_{(2)}p_{(2)}$ and $p_{(2)}p_{(2)}p_{(2)}p_{(1)}$ is A'', and X_l does not appear in position j in any leaf in T_1; that is, T_5 cannot have depth 3 or depth greater than 3, a contradiction. □

Hence, $p_{(2)}p_{(2)}$ is a leaf in T_5. Let A and B be as in the proof of the preceding theorem. By Lemma 3.27, $f(B') = B$. Now, B' appears in the jth position in $p_{(2)}p_{(2)}$ (in T_5) and $p_{(2)}$ (in T_1). Howeever, the jth arguments of $p_{(1)}p_{(1)}$ and $p_{(1)}p_{(2)}$ are X_j and B' respectively. Hence, we may conclude that $f(p_{(2)}p_{(2)}) = p_{(2)}$.

Cyclic programs Before we proceed to the cases $f(p_{(1)}) = p_{(1)}p_{(2)}$ and $f(p_{(1)}) = p_{(2)}$, we will investigate the behaviour of *cyclic* programs. Recall that a Class 2 variable is a distinguished variable X_j that appears only among the arguments of $p_{(2)}$, such that $p_{(2)}[j] \neq X_j$; that is, X_j appears "out of position." Recall also that a position i is termed Class 2 iff $p_{(2)}[i]$ is a Class 2 variable. We say that the program $\mathcal{P}$ is *cyclic* if for all j, if X_j is a Class 2 variable, then j is a Class 2 position.

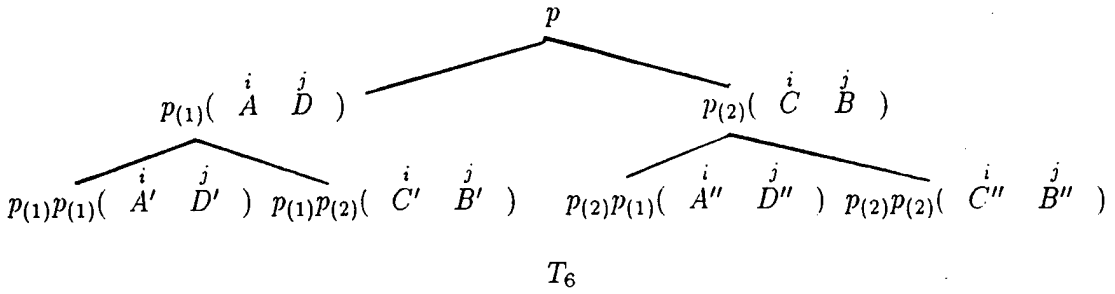
Assume that the arguments of the p-atoms in $\mathcal{P}$ are as follows, where A and B are connected nondistinguished variables (not necessarily distinct), C and D are arbitrary nondistinguished variables and i need not be distinct from j.

$$p_{(1)}(\ \overset{i}{A}\quad \overset{j}{D}\ ) \qquad p_{(2)}(\ \overset{i}{C}\quad \overset{j}{B}\ )$$

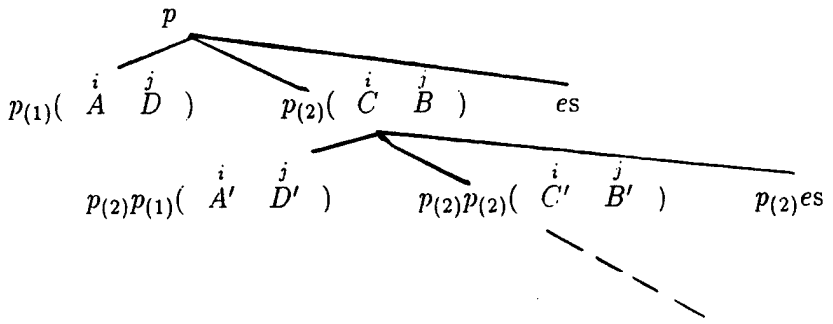
We will show that if $\mathcal{P}$ is cyclic and basis-linearizable, and has such arguments, then $\mathcal{P}$ is minimal. For this purpose, we will investigate the containment of the complete expansion of depth 2 (expansion T_6 in Figure 3.25) in a right-linear expansion; that is, for the purpose of this subsection, we will not be considering the minimal violation of right-linearity.

Consider the complete expansion T_6 of depth 2, illustrated in Figure 3.25. We will assume that each nondistinguished variable V is renamed to V' in the children of $p_{(1)}$, and to V'' in the children of $p_{(2)}$, as indicated in the figure.

Assume that $\mathcal{P}$ is basis-linearizable. Then, T_6 is contained in some right-linear expansion T_5 (see Figure 3.25): assume that f is a containment mapping proving the containment. Since a nondistinguished variable appears in at least one position in each p-leaf in T_5, T_6 is clearly not contained in the depth-0 expansion



T_6



T_5

Figure 3.25: Trees T_6 and T_5

$$p(\vec{X})$$
$$|$$
$$p(\vec{X})$$

The following lemma shows that, if $\mathcal{P}$ is cyclic, then T_6 is not contained in any right-linear expansion of depth greater than 1.

Lemma 3.29 Assume that the p-atoms in $\mathcal{P}$ are as described above, and that $\mathcal{P}$ is basis-linearizable and cyclic. Then the expansion T_6 is not contained in a right-linear expansion of depth greater than 1.

Proof. Assume that f is a containment mapping from some right-linear expansion T_5 of depth at least 2 into T_6; we force a contradiction.

The ith argument of the leaf $p_{(1)}$ in T_5 is A, and the ith arguments of the p-leaves in T_6 are A', C', A'' and C''; hence, $f(A)$ is one of these four variables. By connectivity, $f(B)$ is one of A', C', A'', C'', B' and B''.

By safety, the distinguished variable X_j must appear in the body of the recursive rule. Assume X_j appears in some e_q, in position k say. Since T_5 has depth at least 2, the leaf $p_{(2)}e_q$ appears in it, with kth argument B. The only possible destinations for this leaf in T_6 are the atoms $p_{(1)}e_q, p_{(2)}e_q$ and e_q; however, the kth argument of each of these atoms is a nonprimed variable, a contradiction.

Assume that X_j appears in the arguments of $p_{(1)}$, in position k say. The kth arguments of the leaves $p_{(1)}p_{(1)}$ and $p_{(2)}p_{(1)}$ are D and B respectively, both of which are nondistinguished; hence, we must have $f(p_{(1)}) = p_{(1)}p_{(2)}$ or $f(p_{(1)}) = p_{(2)}p_{(2)}$, and the kth argument of $p_{(1)}p_{(2)}$ or $p_{(2)}p_{(2)}$ must be the distinguished variable X_j. Hence, $p_{(2)}[k]$ is some distinguished variable X_l. The picture is

$$p_{(1)}(\ \overset{i}{A}\ \ \overset{j}{D}\ \ \overset{k}{X_j}\ ) \qquad p_{(2)}(\ \overset{i}{C}\ \ \overset{j}{B}\ \ \overset{k}{X_l}\ )$$

Now, since T_5 has depth at least 2, the leaf $p_{(2)}p_{(1)}$ must appear in it. The kth argument of this leaf is B; however, the kth argument of each leaf in T_6 is a nonprimed variable, and $p_{(2)}p_{(1)}$ has no legal destination.

Thus, X_j must appear only in $p_{(2)}$, in position k say. Since $p_{(2)}[j]$ is nondistinguished, $k \neq j$ and X_j is Class 2, which violates the assumed cyclicity of $\mathcal{P}$ since $p_{(2)}[j]$ is nondistinguished and hence is not Class 2.

□

Now, consider the case in which T_6 is contained in the right-linear expansion T_3 of depth 1. Let f be a containment mapping from T_3 into T_6. We say that f is *normalised* iff the following conditions all hold.

1. The destinations of the leaves $p_{(1)}$ and $p_{(2)}$ in T_3 have the same parent in T_6. That is, both p-leaves are mapped to the same "side" of the complete tree T_6.

2. The destination of each e_q-leaf in T_3 is either the e_q-leaf at depth 1 in T_6, or the e_q-leaf in T_6 with the same parent as the destinations of the p-leaves in T_3.

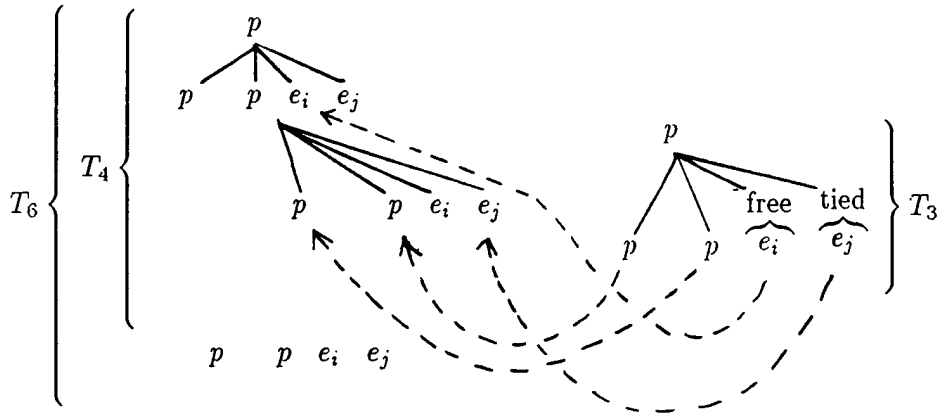


Figure 3.26: Normalised mapping

The following is the analog of Lemma 3.14 in Section 3.6.3.

Lemma 3.30 Assume that the arguments of the p-atoms in $\mathcal{P}$ are as described above, and that $T_6 \subset T_3$. Then, the containment is provable by a normalised mapping.

Proof. Let f be any mapping proving the containment; we construct a normalised mapping g from f. If $f(p_{(1)})$ is one of $p_{(1)}p_{(1)}$ and $p_{(1)}p_{(2)}$, then $f(A)$ is one of A' and C'; by connectivity, $f(B)$ must be A', C' or B'. However, the jth argument of each of $p_{(2)}p_{(1)}$ and $p_{(2)}p_{(2)}$ is a double-primed variable; hence, we must have $f(p_{(2)}) = p_{(1)}p_{(1)}$ or $f(p_{(2)}) = p_{(1)}p_{(2)}$. Similarly, if $f(p_{(1)})$ is one of $p_{(2)}p_{(1)}$ or $p_{(2)}p_{(2)}$, then $f(p_{(2)})$ must also be one of these atoms.

Partition the nondistinguished variables in T_3 into two classes. *Tied* variables are variables that appear in $p_{(1)}$ or $p_{(2)}$, or in any e_q that is connected to $p_{(1)}$ or $p_{(2)}$; *free* variables consist of the remainder. Define the function g as follows.

$$g(V) = \begin{cases} V & \text{if } V \text{ is distinguished} \\ f(V) & \text{if } V \text{ is tied} \\ V & \text{if } V \text{ is free} \end{cases}$$

That is, g preserves the destinations of f for all atoms that are connected to $p_{(1)}$ or $p_{(2)}$, and maps all other e_q to "themselves" in T_6 (see Figure 3.26). g is a normalised containment mapping from T_3 into T_6. □

Lemma 3.31 Assume that the arguments of $\mathcal{P}$ are as described above, and that f is a containment mapping from T_3 into T_6 such that the destination of the leaf $p_{(1)}$ in T_1 is a

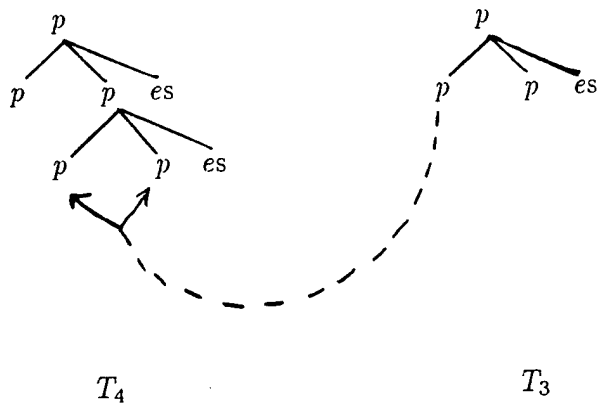


Figure 3.27: One-boundedness

child of the atom $p_{(1)}$ in T_6; that is, $f(p_{(1)}) = p_{(1)}p_{(1)}$ or $f(p_{(1)}) = p_{(1)}p_{(2)}$. Then $\mathcal{P}$ is minimal.

Proof. Construct a normalised containment mapping g from T_3 into T_6; g is a containment mapping from T_3 into the minimal violation T_1 of right-linearity. □

Lemma 3.32 Assume that the arguments of $\mathcal{P}$ are as described above, and that f is a containment mapping from T_3 into T_6 such that the destination of the leaf $p_{(1)}$ in T_1 is a child of the atom $p_{(2)}$ in T_6; that is, $f(p_{(1)}) = p_{(2)}p_{(1)}$ or $f(p_{(1)}) = p_{(2)}p_{(2)}$. If $\mathcal{P}$ is basis-linearizable, then $\mathcal{P}$ is minimal.

Proof. Construct a normalised containment mapping g from T_3 into T_6; g is a containment mapping from T_3 into the right-linear tree T_4 of depth 2 (see Figure 3.27). Let $\mathcal{Q}$ be the set of right-linear open expansions generated by $\mathcal{P}$; as we mentioned in Chapter 1, $\mathcal{Q}$ may be considered a program. Then, T_4 is the minimal violation of one-boundedness in $\mathcal{Q}$; by (the discussion following) Theorem 1.7, we may conclude that the set of right-linear expansions of $\mathcal{P}$ is one-bounded.

Now, consider the minimal violation T_1 of right-linearity in $\mathcal{P}$. Since we assumed that $\mathcal{P}$ is basis-linearizable, T_1 is contained in some right-linear expansion. However, since the set of right-linear expansions is one-bounded, we conclude that T_1 is contained in a right-linear expansion of depth at most 1, and the result follows. □

The following lemma details the property of cyclic programs that will be used in the following two sections. This lemma is key in the proof that if the minimal violation T_1 of right-linearity is contained in a long right-linear expansion under an unacceptable containment mapping, then T_1 is minimal (that is, T_1 is contained in the right-linear expansion of depth 1).

Lemma 3.33 Assume that $\mathcal{P}$ is basis-linearizable, and that the p-atoms in the recursive rule have the form

$$p_{(1)}(\ \overset{i}{A}\quad \overset{j}{D}\ ) \qquad p_{(2)}(\ \overset{i}{X}\quad \overset{j}{Y}\ )$$

where A and B are connected nondistinguished variables, and where X and Y are distinguished. Then $\mathcal{P}$ is minimal.

Proof. Consider the complete expansion T_6 of depth 2; since $\mathcal{P}$ is assumed to be basis-linearizable, T_6 is contained in some right-linear expansion. By discussion, and by Lemma 3.29, T_6 must be contained in the right-linear expansion T_3 of depth 1. By Lemmas 3.31 and 3.32, $\mathcal{P}$ is minimal. □

The result of Zhang, Yu and Troy Zhang et al. ([40]) claim a decision procedure for basis-linearizability in the restricted case in which the recursive rule has at most 1 EDB subgoal. As we mentioned in Section 3.1.1, the proof [3] of [40] is flawed. The flaw is, essentially, the fact that they neglect the case covered by Lemma 3.32. They claim the following result.

Define the program $\mathcal{P}$ to satisfy "Property 0" if

1. There is a partial mapping from $p_{(1)}$ into $p_{(2)}$ that preserves the distinguished variables in $p_{(1)}$.
2. There are two distinct nondistinguished variables A and C that appear only in $p_{(1)}$ and $p_{(2)}$, in the following positions:

$$p_{(1)}(\ \overset{i}{A}\quad \overset{j}{C}\ ) \qquad p_{(2)}(\ \overset{i}{C}\quad \overset{j}{A}\ )$$

3. $\mathcal{P}$ is not minimal.

The result of [40] is the claim that the complete tree T_6 of depth 2 is not contained in any right-linear expansion. The following program satisfies "Property 0," and yet T_6 is contained in the depth-1, right-linear expansion T_3.

r_1 : $p(X,Y,W,Z) :- p(U,X,A,C), p(X,X,C,A), e(Y,W,Z)$.
r_2 : $p(X,Y,W,Z) :- b(X,Y,W,Z)$.

The weaker result of Lemma 3.33 suffices for our purposes.

The case $f(p_{(1)}) = p_{(1)}p_{(2)}$ Let us return to the containment of the minimal violation T_1 of right-linearity (see Figure 3.28) in a right-linear expansion T_5 of depth at least 2. Let f be a containment mapping from T_5 into T_1 such that $f(p_{(1)}) = p_{(1)}p_{(2)}$ (see Figure 3.28). We will show that if $\mathcal{P}$ is basis-linearizable, then $\mathcal{P}$ is minimal. As before, if $p_{(1)}$ and $p_{(2)}$ are not connected, then $\mathcal{P}$ is minimal by Lemma 3.1.1. Assume that $p_{(1)}$ and $p_{(2)}$ are connected.

[3] The paper has been published in TODS ([41]) without a proof.

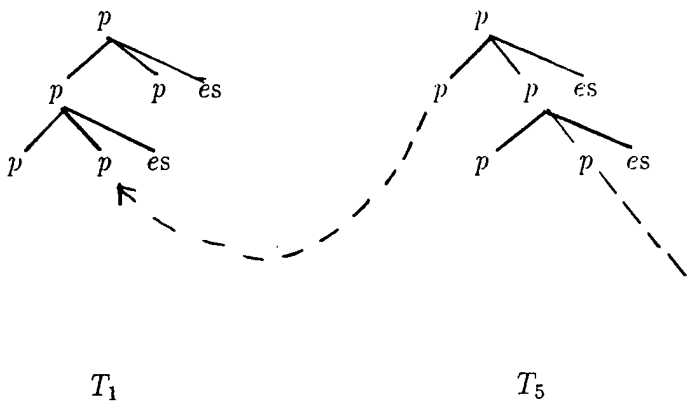


Figure 3.28: The case $f(p_{(1)}) = p_{(1)}p_{(2)}$.

Lemma 3.34 For all j, if $p_{(2)}[j]$ is nondistinguished, then so is $p_{(1)}[j]$.
Corollary. If $p_{(1)}[j]$ is a distinguished variable, then $p_{(2)}[j]$ is distinguished.
Proof. The proof follows from the fact that f preserves distinguished variables. If $p_{(2)}[j]$ is nondistinguished, then $p_{(1)}p_{(2)}[j]$ would also be nondistinguished; hence, by Lemma 3.7 and the assumed mapping $f(p_{(1)}) = p_{(1)}p_{(2)}$, $p_{(1)}[j]$ must be nondistinguished. □

Lemma 3.35 Assume that the arguments of the p-atoms in $\mathcal{P}$ are of the following form.

$$p_{(1)}(\ \overset{j}{D}\quad \overset{k}{X_j}\ ) \qquad p_{(2)}(\ \overset{j}{B}\quad \overset{k}{C}\ )$$

Then $f(B)$ is X_i, D or C.
Proof. The mapping $f(p_{(1)}) = p_{(1)}p_{(2)}$ yields the fact that $p_{(1)}p_{(2)}[k]$ is X_j. The kth arguments of $p_{(1)}p_{(1)}$ and $p_{(2)}$ are D and C respectively. The kth argument of $p_{(2)}p_{(1)}$ is B; our result follows by an examination of all possible destinations for this atom. □

Recall that a Class 2 variable is a distinguished variable X_j that appears only in $p_{(2)}$, such that $p_{(2)}[j] \neq X_j$. Recall also that a Class 2 position is an argument position in $p_{(2)}$ that is occupied by a Class 2 variable.

Lemma 3.36 If k is a Class 2 position, then X_k is a Class 2 variable.
Proof. The picture is as follows, where $C \neq X_j$, and where X_j appears only in $p_{(2)}$.

$$p_{(2)}(\ \overset{j}{C}\quad \overset{k}{X_j}\ )$$

By safety, X_k appears in the rule body. By Lemma 3.16, since X_j does not appear in $p_{(1)}$, X_k does not appear in any ϵ_q. Assume that X_k appears in some position l in $p_{(1)}$. Then,

X_j appears in position l in $p_{(2)}p_{(1)}$, and this atom is a leaf in T_5. Consider the possible destinations of this leaf. Now, X_j appears nowhere in $p_{(1)}$, and hence appears nowhere in $p_{(1)}p_{(1)}$ or $p_{(1)}p_{(2)}$; hence, $f(p_{(2)}p_{(1)}) = p_{(2)}$, and $p_{(2)}[l] = X_j$. The picture is

$$p_{(1)}(\ \overset{l}{X_k}\ ) \qquad p_{(2)}(\ \overset{j}{C}\ \ \overset{k}{X_j}\ \ \overset{l}{X_j}\ )$$

The assumed mapping $f(p_{(1)}) = p_{(1)}p_{(2)}$ requires that the lth position of $p_{(1)}p_{(2)}$ is X_k; that is, $p_{(1)}[j]$ is X_k. However, in this case, $p_{(2)}p_{(1)}[j]$ is X_j, and the mapping $f(p_{(2)}p_{(1)}) = p_{(2)}$ forces $C = X_j$, contradicting our assumption that X_j is Class 2.

□

Lemma 3.37 i is a Class 2 position iff X_i is a Class 2 variable.
Proof. By Lemma 3.36 and pigeonholing, as in the proof of Lemma 3.24. □

Lemma 3.38 $\mathcal{P}$ is cyclic.
Proof. By Lemmas 3.36 and 3.37. □

Lemma 3.39 Assume the picture

$$p_{(1)}(\ \overset{i}{A}\ \ \overset{k}{C}\ ) \qquad p_{(2)}(\ \overset{i}{X_k}\ \ \overset{k}{A}\ )$$

where A and C are nondistinguished and $X_k \neq X_i$. Then f cannot exist.
Proof. By safety, X_i must appear in the rule body. By Lemma 3.16, it cannot appear in any e_q. Assume it appears in $p_{(1)}$, in position l say. The mapping $f(p_{(1)}) = p_{(1)}p_{(2)}$ requires that $p_{(1)}p_{(2)}[l]$ is X_i; hence, $p_{(2)}[i]$ is some distinguished variable $X_m \neq X_k$ such that $p_{(2)}[m]$ is X_i. Now, the lth argument of the leaf $p_{(2)}p_{(1)}$ in T_5 is X_k, but X_k does not appear in popsition l in any leaf in T_1; that is, the lth arguments of the leaves in T_1 are as below.

$$p_{(1)}p_{(1)}(\ A\ ) \qquad p_{(1)}p_{(2)}(\ \overset{l}{X_i}\ ) \qquad p_{(2)}(\ \overset{l}{X_m}\ )$$

Hence, X_i appears only in $p_{(2)}$, and is thus Class 2. By Lemma 3.37, k must be a Class 2 position, a contradiction since A is nondistinguished. □

Lemma 3.40 Assume the picture

$$p_{(1)}(\ \overset{i}{A}\ \ \overset{j}{C}\ ) \qquad p_{(2)}(\ \overset{j}{B}\ )$$

where A and B are connected nondistinguished variables. If $\mathcal{P}$ is not minimal, then $C \neq B$. $A = B$ and $f(A)$ is X_j or C.
Proof. By Lemma 3.34, C is nondistinguished. If $C = B$ then $\mathcal{P}$ is minimal by Lemma 3.33. Assume that $C \neq B$. By safety, X_j appears in $p_{(1)}$, $p_{(2)}$or some e_q.

If X_j appears in some e_q, then our result follows by Lemma 3.16 and connectivity (Lemma 3.13). Assume that X_j appears in no e_q. By Lemma 3.37, X_j cannot appear only in $p_{(2)}$; hence, X_j appears in $p_{(1)}$, in position k say.

The assumption $f(p_{(1)}) = p_{(1)}p_{(2)}$ requires that $p_{(2)}[k]$ be some distinguished variable X_l different from X_j such that $p_{(1)}[l]$ is X_j. Since A and C are nondistinguished, $k \neq i$ and $k \neq j$. The situation is

$$p_{(1)}(\ \overset{i}{A}\ \ \overset{j}{C}\ \ \overset{k}{X_j}\ \ \overset{l}{X_j}\ )\qquad p_{(2)}(\ \overset{j}{B}\ \ \overset{k}{X_l}\ )$$

By Lemma 3.35, $f(B)$ is one of X_j, C and X_l. Since we assumed $C \neq A$, by connectivity we must have $A = B$. We will show that $f(A) \neq X_l$ to complete the proof.

For the assumed mapping on $p_{(1)}$, $p_{(2)}[i]$ must be X_m for some m ($m \neq l, m \neq j$) such that $p_{(1)}[m]$ is X_l. The new picture is

$$p_{(1)}(\ \overset{i}{A}\ \ \overset{k}{X_j}\ \ \overset{l}{X_j}\ \ \overset{m}{X_l}\ )\qquad p_{(2)}(\ \overset{i}{X_m}\ \ \overset{j}{A}\ \ \overset{k}{X_l}\ )$$

An examination of destinations for $p_{(2)}p_{(1)}$ suffices to show that we must have $f(p_{(2)}p_{(1)}) = p_{(2)}$. This mapping yields $f(A') = A$; however, in this case, neither $p_{(2)}p_{(2)}$ nor $p_{(2)}p_{(2)}p_{(1)}$ has a destination among the leaves of T_1. □

Lemma 3.41 Assume the picture

$$p_{(1)}(\ \overset{i}{A}\ \ \overset{j}{C}\ )\qquad p_{(2)}(\ \overset{j}{A}\ )$$

If $f(A) = C$, then $\mathcal{P}$ is minimal.

Proof. Assume $f(A) = C$. If $C = A$, then $\mathcal{P}$ is minimal by Lemma 3.33. Assume $C \neq A$: we will show that f cannot exist. The mapping $f(p_{(1)}) = p_{(2)}$, along with Lemma 3.34, requires that C is nondistinguished. The same mapping and the assumption $f(A) = C$ requires that $p_{(2)}[i]$ be some distinguished variable X_k such that $p_{(1)}[k]$ is C. Since $A \neq C$, we conclude that $X_k \neq X_i$. The new picture is

$$p_{(1)}(\ \overset{i}{A}\ \ \overset{j}{C}\ \ \overset{k}{C}\ )\qquad p_{(2)}(\ \overset{i}{X_k}\ \ \overset{j}{A}\ )$$

Now, the mapping $f(p_{(1)}) = p_{(1)}p_{(2)}$ yields $f(C) = A'$; hence, $p_{(2)}[k] = A$. By Lemma 3.39, f cannot exist. □

Lemma 3.42 Assume that the arguments of $p_{(1)}$ and $p_{(2)}$ are as follows, where A and C are nondistinguished (hence $i \neq j$).

$$p_{(1)}(\ \overset{i}{A}\ \ \overset{j}{C}\ \ \overset{k}{X_j}\ )\qquad p_{(2)}(\ \overset{i}{X_k}\ \ \overset{j}{A}\ )$$

Let T_5 be a right-linear tree of depth $r > 1$. Then, the kth argument of the $p_{(2)}$-leaf in T_5 (i.e. of $p_{(2)}{}^r$) is some $X_l \neq X_j$.
Proof. Consider the set $S = \{k_1, \ldots, k_n\}$ of all positions in $p_{(1)}$ that are occupied by X_j; by assumption, this set has cardinality at least 1. Since $p_{(1)}[i]$ and $p_{(1)}[j]$ are nondistinguished, $k_q \neq i$ and $k_q \neq j$ for all q. The requirement $f(p_{(1)}) = p_{(1)}p_{(2)}$, along with the fact that f is the identity on distinguished variables, shows that for each $l \in S$, $p_{(2)}[l]$ is X_m for some $m \in S$. A straightforward induction on $1 \leq q \leq r$ shows that for any $l \in S$, there is an $m \in S$ such that the lth argument of $p_{(2)}{}^q$ is X_m. Our result follows. □

Lemma 3.43 Assume that the arguments of $p_{(1)}$ and $p_{(2)}$ are as follows, where A and C are nondistinguished and $C \neq A$. Then $f(A) \neq X_j$.

$$p_{(1)}(\overset{i}{A}) \qquad p_{(2)}(\overset{j}{B})$$

Proof. If $f(p_{(1)}) = p_{(1)}p_{(2)}$ and $f(A) = X_j$, then we have the following picture where $k \neq i, k \neq j$.

$$p_{(1)}(\overset{i}{A}\ \overset{j}{C}\ \overset{k}{X_j}) \qquad p_{(2)}(\overset{i}{X_k}\ \overset{j}{A})$$

The kth argument of the leaf $p_{(1)}p_{(1)}$ in T_1 is the nondistinguished variable C. Since we assume $f(p_{(1)}) = p_{(1)}p_{(2)}$, the kth argument of $p_{(1)}p_{(2)}$ is X_j. The ith and jth arguments of the p-leaves in T_1 are as follows.

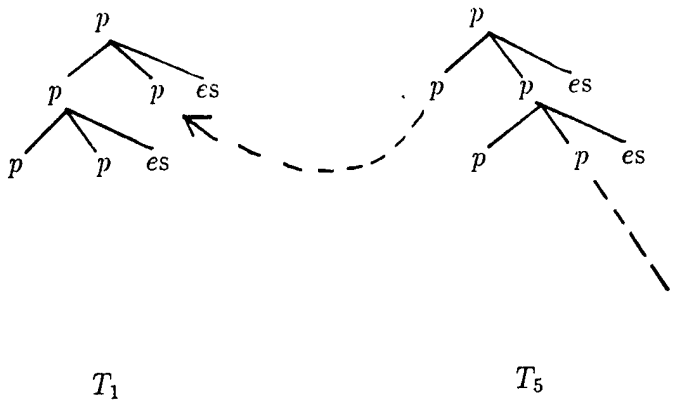
$$p_{(1)}p_{(1)}(\overset{i}{A'}\ \overset{j}{C'})\ p_{(1)}p_{(2)}(\overset{i}{X_j}\ \overset{j}{A'})\ p_{(2)}(\overset{i}{X_k}\ \overset{j}{A})$$

Let T_5 have depth $n + 1$, for some $n \geq 1$. Consider the lowest p-leaves in T_5; that is, the atoms $p_{(2)}{}^n p_{(1)}$ and $p_{(2)}{}^{n+1}$. By Lemma 3.42, the kth argument of $p_{(2)}{}^{n+1}$ is some distinguished variable distinct from X_j, and hence we must have $f(p_{(2)}{}^{n+1}) = p_{(2)}$. Now, the ith argument of $p_{(2)}{}^n p_{(1)}$ and the jth argument of $p_{(2)}{}^{n+1}$ are both A^n (recall that this is the variable A, primed n times). The mapping $f(p_{(2)}{}^{n+1}) = p_{(2)}$ forces $f(A^{(n)}) = A$; however, A does not appear in the ith position of any p-leaf in T_1, and hence $p_{(2)}{}^n p_{(1)}$ has no destination in T_1. □

Lemma 3.44 $\mathcal{P}$ is minimal.
Proof. By Lemmas 3.40, 3.41 and 3.43. □

The case $f(p_{(1)}) = p_{(2)}$ Assume that $\mathcal{P}$ is basis-linearizable, and that the minimal violation T_1 of Figure 3.29 is contained in a right-linear tree T_5 of depth at least 2. Assume further that the containment mapping f from T_5 into T_1 satisfies $f(p_{(1)}) = p_{(2)}$ (see Figure 3.29). We will show that $\mathcal{P}$ is minimal or $p_{(1)}$ is an adjunct to $p_{(2)}$, to complete the proof.

Lemma 3.45 For all i and j, if $p_{(1)}[j]$ is the distinguished variable X_i, then $p_{(2)}[j]$ is X_i.



T_1 T_5

Figure 3.29: The case $f(p_{(1)}) = p_{(2)}$.

Proof. Since f is a containment mapping from T_5 into T_1, and since both expansions have the root $p(X_1, \ldots, X_m)$, we conclude that $f(X_i) = X_i$ for all i. Our result follows because $f(p_{(1)}) = p_{(2)}$ by assumption. □

Recall that a Class 2 variable is a distinguished variable X_j that appears only in $p_{(2)}$, such that $p_{(2)}[j] \neq X_j$. Recall also that a Class 2 position is a position in $p_{(2)}$ that is occupied by a Class 2 variable.

Lemma 3.46 If k is a Class 2 position, then X_k is a Class 2 variable.
Proof. Assume the following scenario, where $C \neq X_j$ (so $j \neq k$), and where X_j appears only in $p_{(2)}$.

$$p_{(2)}(\ \overset{j}{C}\quad \overset{k}{X_j}\ )$$

By safety, X_k appears in the rule body. By Lemma 3.16, it cannot appear in any ϵ_q. Assume it appears in $p_{(1)}$, in position l; the mapping $f(p_{(1)}) = p_{(2)}$ forces $p_{(2)}[l] = X_k$ (so $l \neq j, l \neq k$). The picture is

$$p_{(1)}(\ \overset{l}{X_k}\ ) \qquad p_{(2)}(\ \overset{j}{C}\quad \overset{k}{X_j}\quad \overset{l}{X_k}\ )$$

Now, X_j appears in the lth position of the leaf $p_{(2)}p_{(1)}$ in T_5, but appears nowhere in the leaves $p_{(1)}p_{(1)}$ or $p_{(1)}p_{(2)}$ in T_1 (since X_j does not appear in $p_{(1)}$); also, the lth argument of $p_{(2)}$ is $X_k \neq X_j$, so $p_{(2)}p_{(1)}$ has no legal destination in T_1. Thus, X_k appears only in $p_{(2)}$.
□

Lemma 3.47 For all i, i is a Class 2 position iff X_i is a Class 2 variable.
Corollary. $\mathcal{P}$ is cyclic.

Proof. The proof follows by Lemma 3.46 and pigeonholing. □

Lemma 3.48 Assume that A is a nondistinguished variable appearing in $p_{(1)}$. Then $f(A) \neq X_j$.

Proof. Assume the converse. By the assumed mapping $f(p_{(1)}) = p_{(2)}$, the picture is the following.

$$p_{(1)}(\ \overset{i}{A}\quad \overset{j}{C}\ ) \qquad p_{(2)}(\ \overset{i}{X_j}\quad \overset{j}{A}\ )$$

We show that T_5 cannot have depth 2 or greater, a contradiction since T_5 is assumed to have depth at least 2.

Since A is nondistinguished, $i \neq j$. By Lemma 3.45, C must be nondistinguished. By safety, X_i must appear in the rule body.

If X_i appears in any e_q, then by Lemma 3.16, $f(X_j)$ is A or X_i, a contradiction. Assume that X_i appears in $p_{(1)}$, in position k say. Then, by Lemma 3.45, $p_{(2)}[k]$ is X_i. The picture is

$$p_{(1)}(\ \overset{i}{A}\quad \overset{j}{C}\quad \overset{k}{X_i}\ ) \qquad p_{(2)}(\ \overset{i}{X_j}\quad \overset{j}{A}\quad \overset{k}{X_i}\ )$$

Now, since T_5 has depth at least 2, the leaf $p_{(2)}p_{(1)}$ appears in it. The kth argument of this leaf is X_j. However, X_j does not appear in the kth position in any leaf in T_1, a contradiction.

Hence, X_i appears only in $p_{(2)}$. However, then X_i is a Class 2 variable, and we conclude by Lemma 3.47 that $p_{(2)}[i]$ (that is, X_j) is a Class 2 variable. However, this result contradicts Lemma 3.47 because $p_{(2)}[j]$ is a nondistinguished variable, and is hence not Class 2. □

Lemma 3.49 $p_{(1)}$ is an adjunct to $p_{(2)}$ or $\mathcal{P}$ is minimal.

Proof. We will show the following.

1. If a distinguished variable X_j appears in the ith position in $p_{(1)}$, then $p_{(2)}[i]$ is X_j.
2. If a nondistinguished variable A appears in $p_{(1)}$ (in the ith position, say) and in some e_q, then $p_{(2)}[i]$ is A.
3. If a nondistinguished variable appears in $p_{(1)}$ and $p_{(2)}$, then $\mathcal{P}$ is minimal.

Hence, if $\mathcal{P}$ is not minimal, then every nonlocal variable appearing in the arguments of $p_{(1)}$ appears in the same position in $p_{(2)}$; that is, $p_{(1)}$ is an adjunct to $p_{(2)}$.

To prove (1), we observe that by Lemma 3.45, every distinguished variable in $p_{(1)}$ appears in the same position in $p_{(2)}$.

To prove (2), we proceed as follows. Assume that $p_{(1)}$ shares a nondistinguished variable A with an EDB subgoal e_q, and that A is the ith argument of $p_{(1)}$. By Lemma 3.15, $f(A)$ is A or A'. However, no primed variable appears in $p_{(1)}$, so the mapping $f(p_{(1)}) = p_{(2)}$ forces $p_{(2)}[i] = A$.

Finally, we prove (3). Assume, now, that a nondistinguished variable A appears in position i in $p_{(1)}$, and in position j in $p_{(2)}$. We show that $\mathcal{P}$ is minimal. By Lemma 3.33, if

$i = j$ then $\mathcal{P}$ is minimal. Assume $i \neq j$, and let the jth argument of $p_{(1)}$ be C. The picture is

$$p_{(1)}(\ \overset{i}{A}\ \ \overset{j}{C}\ ) \qquad p_{(2)}(\ \overset{j}{A}\ )$$

By Lemma 3.45, since $p_{(2)}[j]$ is nondistinguished, C must be nondistinguished. Further, by Lemma 3.33, if $p_{(2)}[i]$ is nondistinguished then $\mathcal{P}$ is minimal. Assume that $p_{(2)}[i]$ is some distinguished variable X_k; the mapping $f(p_{(1)}) = p_{(2)}$ forces $f(A) = X_k$. The picture is

$$p_{(1)}(\ \overset{i}{A}\ \ \overset{j}{C}\ ) \qquad p_{(2)}(\ \overset{i}{X_k}\ \ \overset{j}{A}\ )$$

By safety, X_j appears in the rule body. Assume that X_j appears in some e_q; by Lemma 3.17, we must have $f(A) = X_j$ or $f(A) = C$. Since C is nondistinguished, the former must hold, so that $X_k = X_j$, contradicting Lemma 3.48. By Lemma 3.47, X_j cannot appear only in $p_{(2)}$, since in this case X_j would be Class 2 but $p_{(2)}[j] = A$ is nondistinguished. Thus, X_j must appear in $p_{(1)}$, in position m say. Then, the mapping $f(p_{(1)}) = p_{(2)}$ forces $p_{(2)}[m] = X_j$. The picture is

$$p_{(1)}(\ \overset{i}{A}\ \ \overset{j}{C}\ \ \overset{m}{X_j}\ ) \qquad p_{(2)}(\ \overset{i}{X_k}\ \ \overset{j}{A}\ \ \overset{m}{X_j}\ )$$

Now, the leaf $p_{(2)}p_{(1)}$ must appear in T_5, and the mth argument of this leaf is A. The mth arguments of the leaves $p_{(1)}p_{(1)}$ and $p_{(1)}p_{(2)}$in T_1 are each $C \neq X_k$, so we must have $f(p_{(2)}p_{(1)}) = p_{(2)}$. Thus, $p_{(2)}[m]$ must be X_k, so $X_k = X_j$, contradicting Lemma 3.48. □

The proof is now complete.

Chapter 4

Undecidability of the general problems

4.1 Introduction

Finally, let us turn to the decidability of basis-linearizability and sequencability. In this chapter, we will show that both these problems are undecidable for a restricted class of Datalog programs.

4.1.1 Definitions

Consider the following single-IDB, safe, Datalog program $\mathcal{P}$, in which the head of every rule is rectified (i.e., contains no repetitions of any variable). Recall that a program is single-IDB iff there is only one intensional predicate in $\mathcal{P}$.

Let $\mathcal{P}$ consist of the n recursive rules

$$r_1 : \ p(\vec{X}_0) :- p(\vec{X}_{11}), \ldots p(\vec{X}_{1k_1}), \mathcal{C}_1.$$
$$\vdots$$
$$r_i : \ p(\vec{X}_0) :- p(\vec{X}_{i1}), \ldots p(\vec{X}_{ik_i}), \mathcal{C}_i.$$
$$\vdots$$
$$r_n : \ p(\vec{X}_0) :- p(\vec{X}_{n1}), \ldots p(\vec{X}_{nk_n}), \mathcal{C}_n.$$

and the m nonrecursive rules

$$b_1 : \ p(\vec{X}_0) :- \mathcal{D}_1.$$
$$\vdots$$
$$b_j : \ p(\vec{X}_0) :- \mathcal{D}_j.$$
$$\vdots$$
$$b_m : \ p(\vec{X}_0) :- \mathcal{D}_m.$$

where the $\mathcal{C}_i$s and $\mathcal{D}_j$s are arbitrary conjunctions of EDB predicates.

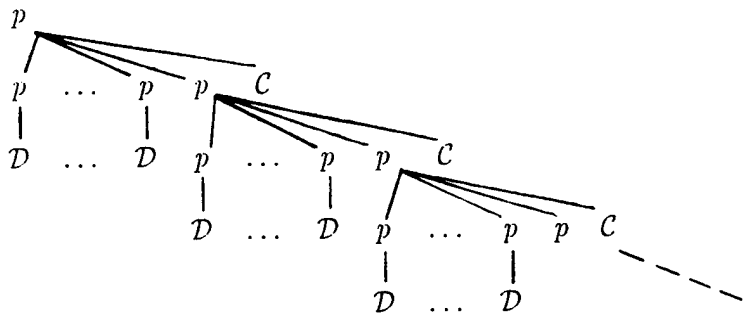


Figure 4.1: Right-linear query.

Base-case linearizability

Recall from Section 1.4.3 that a conjunctive query generated by $\mathcal{P}$ is *right-linear* if only the rightmost occurrence of p is ever recursively expanded (see Figure 4.1). Recall also that $\mathcal{P}$ is *basis-linearizable* iff every conjunctive query generated by $\mathcal{P}$ is contained in a right-linear conjunctive query.

Sequencability

Recall from Section 1.4.4 that a conjunctive query generated by $\mathcal{P}$ is termed *sequenced* if r_i is never used to expand a p-atom introduced by r_j in a top-down expansion generating this conjunctive query, if $i < j$ (see Figure 4.2). Recall also that $\mathcal{P}$ is termed *sequencable* iff every conjunctive query generated by $\mathcal{P}$ is contained in a sequenced query generated by $\mathcal{P}$.

4.2 Results

We state below the main results of this chapter. In the following statements, $\mathcal{P}$ and $\mathcal{Q}$ are safe, Datalog programs defining a single intensional predicate, using head-rectified rules.

Result 4.1 $\mathcal{P} \subset \mathcal{Q}$ ($\mathcal{P} \equiv \mathcal{Q}$) is undecidable, even if

(a) $\mathcal{P}$ and $\mathcal{Q}$ are linear, and have no more than five basis rules; or

(b) Each of $\mathcal{P}$ and $\mathcal{Q}$ contains only one recursive rule and nine nonrecursive rules.

□

Result 4.2 The base-case linearizability of $\mathcal{P}$ is undecidable, even if $\mathcal{P}$ contains only one nonlinear rule and five basis rules. □

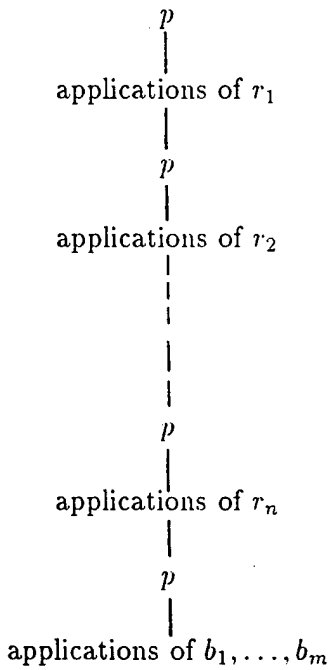


Figure 4.2: Sequenced query.

Result 4.3 The sequencability of $\mathcal{P}$ is undecidable, even if $\mathcal{P}$ contains only two recursive rules and nine basis rules. □

4.2.1 Related results

Shmueli ([33]) and Abiteboul ([1]) present general results that also yield undecidability results for program equivalence. Shmueli considers programs with a single recursive predicate; however, these programs have several IDB predicates, and include rules whose heads are not rectified. The assumption of head-rectification is integral to the proof of Theorem 3.1. Abiteboul's result concerns single-IDB programs with a single recursive rule; however, those programs have rules that are not head-rectified, and contain an unbounded number of initialisation rules.

4.3 Outline

Our undecidability results involve reductions from undecidable problems for context-free grammars. In Section 4.4, we will define certain normal forms for context-free grammars., and show that the containment of such normal-form grammars is undecidable. In this section, we will also show how unsafe Datalog programs may be made safe, allowing the reductions of the subsequent sections to involve unsafe programs.

In Section 4.5, we will prove Results 4.1(a) and 4.2. Section 4.6 contains the proof of Results 4.1(b) and 4.3.

4.4 Preliminaries

In this section, we present results that will be of use in the proofs of the next two sections.

4.4.1 Context-free grammars

The results of the following sections will be based upon reductions from undecidable problems in language theory. In this subsection, we establish some preliminary results. Our treatment is concise, and assumes concepts that are explained in [15].

Definition 4.1 For any grammar G, a production $A \rightarrow \beta$ is termed *intensional* if at least one nonterminal appears in β, and *extensional* otherwise. □

Definition 4.2 A grammar G over the alphabet $\Sigma = \{a, b\}$ is termed *bounded-basis* if G is ϵ-free and if there are only two extensional productions in G, which are of the form $N_a \rightarrow a$ and $N_b \rightarrow b$ where $N_a \neq N_b$, and where neither N_a nor N_b appears as the head of any other production. Hence, we may partition the nonterminals into *intensional* and *extensional* nonterminals, depending on whether they appear on the left-hand side of a intensional or extensional production. N_a is termed the *extensional nonterminal representing* a, and N_b is termed the *extensional nonterminal representing* b. □

Lemma 4.1 Given any linear grammar H over $\Sigma = \{a, b\}$, we may effectively construct a linear, bounded-basis grammar G such that $L(G) = L(H) - \{\epsilon\}$.
Proof. Construct a linear, ϵ-free grammar I for $L(H) - \{\epsilon\}$ by determining nullable nonterminals and following the procedure of Theorem 4.3 in Section 4.4 of [15]. Assume that S is the start symbol of I. We construct G from I, as follows. Introduce new nonterminals N_a and N_b, and the productions $N_a \rightarrow a$ and $N_b \rightarrow b$. Finally, consider any extensional production $A \rightarrow \alpha$ [1]. If α is a, replace the production by the unit production $A - N_a$; if α is b, replace the production by $A \rightarrow N_b$. Otherwise, α is of the form βa or βb. In the first case, replace the production by the production $A \rightarrow \beta N_a$, and in the second, replace the production by $A \rightarrow \beta N_b$. □

Definition 4.3 A *Modified Chomsky Normal Form (MCNF)* grammar G over the terminal alphabet $\Sigma = \{a, b\}$ is a grammar with the following properties.

1. G is bounded-basis.
2. Each intensional nonterminal N appears at the head of at most two productions. Further, if N does appear as the head of two productions, then both productions are unit productions. That is, intensional productions are of three types.
   (a) If $N \rightarrow M$ and $N \rightarrow K$ are productions, then these productions are *or-productions*.
   (b) $N \rightarrow MK$ is an *and-production*.
   (c) If N appears only on the left-hand side of the production $N - M$, then this production is a *copy production*.

□

Lemma 4.2 For every grammar H over $\Sigma = \{a, b\}$, there is an MCNF grammar G generating $L(H) - \{\epsilon\}$.
Proof. Construct a Chomsky Normal Form grammar for $L(H) - \epsilon$, with start symbol S. Introduce the nonterminals N_a and N_b, and the productions $N_a - a$ and $N_b - b$. Then, replace every other production of the form $N - c$, where c is a terminal, by the production $N \rightarrow N_c$. All productions other than $N_a - a$ and $N_b - b$ are now intensional. Replace every and-production $N \rightarrow MK$ by the two productions $N - L$ and $L - MK$, where L is a new nonterminal. At this point, the only violations of MCNF are the presence of nonterminals N such that $N - R_1, \ldots, N \rightarrow R_{k+1}$, for $k > 1$, are the productions with N on the left-hand side. Introduce new nonterminals $M_1, \ldots, M_k$; then, add the productions $N - M_1$ and $M_k - R_{k+1}$; finally, for $2 \leq i \leq k$, replace the production $N - R_i$ with the two productions $M_{i-1} \rightarrow R_i$ and $M_{i-1} - M_i$. □

Lemma 4.3 Assume $\Sigma = \{a, b\}$. It is undecidable, for an arbitrary context-free grammar (CFG) G over the alphabet Σ, whether $\Sigma^* \subset L(G)$. This result is true even if G is linear.

[1] α is a string of terminals.

Proof. Let $\Sigma = \{a_1, \ldots, a_m\}$; the corresponding problems over this alphabet are known to undecidable ([15]). The size of the alphabet can be reduced to 2 by padding and encoding. □

Lemma 4.4 Let G_1 and G_2 be bounded-basis grammars over the alphabet $\Sigma = \{a, b\}$. Then $G_1 \subset G_2$ is undecidable. This result is true even if G_1 and G_2 are required to be linear, or if both grammars are required to be in Modified Chomsky Normal Form.
Proof. Let G_1 be the obvious bounded-basis, linear (or MCNF) grammar generating Σ^+. Testing whether $\epsilon \in L(G_2)$ is decidable, and our result follows by Lemmas 4.1 and 4.3 (4.2 and 4.3 if G_2 is MCNF). □

Lemma 4.5 Let G be a linear, bounded-basis grammar over $\Sigma = \{a, b\}$. Then, $\Sigma^+ L(G) \subset \Sigma L(G)$ is undecidable.
Proof. $\Sigma^+ \subset L(G)$ iff

1. 1. $\Sigma \subset L(G)$ and
2. 2. $\Sigma^+ L(G) \subset \Sigma L(G)$.

Since 1 is decidable, our result then follows by Lemma 4.4.
□

4.4.2 Datalog programs

The following lemma will be of use in the following sections.

Lemma 4.6 Let C and D be the conjunctive queries

$C: \ p(\vec{X}) :- \mathcal{C}.$
$D: \ p(\vec{X}) :- \mathcal{D}, f(Z).$

where $\mathcal{C}$ and $\mathcal{D}$ are conjunctions of EDB predicates, Z is a distinguished variable (i.e. Z appears in $\vec{X}$) and $f(Z)$ does not appear in $\mathcal{C}$. Then $C' \not\subseteq D$.
Proof. Every containment mapping $g : D \to C$ must satisfy $g(Z) = Z$, since $g(p(\vec{X}))$ must be $p(\vec{X})$. However, then $g(f(Z))$ does not appear in the body of C. □

Safety

Recall that a rule is termed *safe* if every variable appearing in the rule head (a *distinguished* variable) appears in the rule body, and a program is termed safe if every rule in the program is safe. The constructions of the following sections will deal with programs that are unsafe. However, these programs may be made safe without altering the results of these sections, as follows.

Let

$$r: \ p(\vec{X}) :- \mathcal{C}.$$

be a (not necessarily safe) conjunctive query (or rule), and let e be a predicate not appearing in r. Then the notation $safe(r, e)$ represents the query obtained from r by adding conjuncts $e(A)$ to the body of r for every variable A that appears in r. Similarly, the notation $safe(\mathcal{P}, e)$ represents the replacement of every rule r in the program $\mathcal{P}$ with the rule $safe(r, e)$ for some predicate e that appears nowhere in $\mathcal{P}$.

Example 4.1 If r is the rule

$r: \ p(X, Y) :- b(X, U).$

then $safe(r, e)$ is the rule

$p(X, Y) :- b(X, U), e(X), e(Y), e(U).$ □

Lemma 4.7 For any conjunctive queries r and s and any predicate e that appears nowhere in r or s, there is a containment mapping $f : s \rightarrow r$ iff there is a containment mapping $g : safe(s, e) \rightarrow safe(r, e)$.
Proof. The containment mapping $g : safe(s, e) \rightarrow safe(r, e)$ is a containment mapping from s into r. For the converse, assume that f is a containment mapping from s into r. Consider any atom $e(A)$ in $safe(s, e)$: by construction, A appears in s, and therefore $f(A)$ appears in r. Also by construction, $e(f(A))$ appears in $safe(r, e)$, and f is therefore a containment mapping from $safe(s, e)$ into $safe(r, e)$. □

Lemma 4.8 Let $\mathcal{P}$ be a program, and let e be a predicate not appearing in $\mathcal{P}$. Then, $\mathcal{P}$ generates the top-down expansion r iff $safe(\mathcal{P}, e)$ generates the top-down expansion $safe(r, e)$.
Corollary. Let $\mathcal{P}$ and $\mathcal{Q}$ be programs, and assume that e does not appear in $\mathcal{P}$ or $\mathcal{Q}$. Then, $\mathcal{P} \subset \mathcal{Q}$ iff $safe(\mathcal{P}, e) \subset safe(\mathcal{Q}, e)$.
Proof. Straightforward induction on the number of rule applications in $\mathcal{P}$. The corollary follows by Lemma 4.7 and the theorem of Sagiv and Yannakakis (Theorem 1.2). □

For the remainder of this chapter, we will consider unsafe programs with the understanding that these programs will be made safe as in the above lemma[2].

4.5 Linear Logic Programs

In this section, we prove Result 4.1(a) and Result 4.2. The basic idea is the simulation of bounded-basis grammars using single-IDB programs with head-rectified rules and a bounded number of basis rules.

[2]The predicate e essentially represents the DOM relation ([36])

4.5.1 The construction

Let G be an ϵ-free grammar over $\Sigma = \{a_1, \ldots, a_k\}$ with nonterminals $\{N_1, \ldots, N_m\}$. We will construct a program $\mathcal{P}$, defining the IDB predicate p, to simulate G. The EDB for $\mathcal{P}$ will consist of binary predicates $\{a_1, \ldots, a_k\}$ and a unary predicate f.

Let X, Y, W and Z be new variable names; the head of each rule in $\mathcal{P}$ is

$$p(X, Y, W, Z, N_1, \ldots, N_m)$$

Note that each rule head is rectified (i.e. contains no repeated variables). Let $<< N_i >>$ denote an m-vector in which the ith component is W, and all other components are Z. Further, let us use the notation

$$p(< A, B >< N_i >>)$$

to represent the p-atom

$$p(A, B, W, Z, << N_i >>)$$

Example 4.2 Let G be a linear grammar over $\Sigma = \{a, b\}$ with start symbol S and the productions

$S \rightarrow aS \qquad S \rightarrow bB \qquad B \rightarrow b.$

G clearly generates a^*bb. The head of each rule in $\mathcal{P}$ is the atom $p(X, Y, W, Z, S, B)$, and $p(< U, Y >< B >>)$ denotes the atom $p(U, Y, W, Z, Z, W)$. □

Definition 4.4 We define a transformation from symbols in G (terminal or nonterminal) to atomic formulae, as follows. Let U and V be distinct variables. For any terminal a_i, the *$\{U, V\}$-atom corresponding to a_i* is the atom $a_i(U, V)$. The $\{U, V\}$-atom corresponding to a nonterminal N_i is the atom $p(< U, V >< N_i >>)$. This transformation may be extended to strings, as follows. Let $s = s_1 \ldots s_n$ be a nonempty string of terminals and nonterminals in G. Assume that $U_1, \ldots, U_{n+1}$ are distinct variables. We define the *$\{U_1, \ldots, U_{n+1}\}$-chain corresponding to s* to be the conjunction $c = c_1, \ldots, c_n$ where for all i, c_i is the $\{U_i, U_{i+1}\}$-atom corresponding to s_i. □

Example 4.3 Let G be the grammar of Example 4.2. The $\{X, U\}$-atom corresponding to the terminal b is $b(X, U)$, and the $\{X, U, Y\}$-chain corresponding to the string bB is the conjunction $b(X, U), p(< U, Y >< B >>)$, or $b(X, U), p(U, Y, W, Z, Z, W)$. □

Definition 4.5 Let $s = s_1 \ldots s_n$ be a (possibly empty) string of terminals and nonterminals, let $\mathcal{C}$ and Γ be conjunctions of atomic formulae, and let C be the conjunctive query (or rule)

$C: \; p(\vec{H}) :- \Gamma, \mathcal{C}.$

Let D and E be variables appearing in C. Γ is termed a *chain from D to E, embedded in C* and *representing s* iff one of the following is true.

1. $s = \epsilon$, Γ is the empty conjunction *true* and $D = E$.
2. $s = s_1$ (a terminal or nonterminal), D and E are distinct and Γ is the $\{D, E\}$ atom corresponding to s_1.
3. $s = s_1 \ldots s_n$ with $n > 1$, D and E are distinct and Γ is the $\{D, U_1, \ldots, U_{n-1}, E\}$-chain corresponding to s for some distinct variables $U_1, \ldots, U_{n-1}$ distinct from D and E, and not appearing in $\vec{H}$ or among the arguments of the conjuncts in $\mathcal{C}$.

If s is a nonempty string of terminals and D and E appear in $\vec{H}$, then Γ is a *binary chain* ([33]). □

Example 4.4 Consider the conjunctive query

$$C_1 : \ p(X, Y, W, Z) :- \ a(X, U_1), b(U_1, U_2), a(U_2, Y), f(W), f(Z).$$

The conjunction $a(X, U_1), b(U_1, U_2), a(U_2, Y)$ is a binary chain from X to Y representing the terminal string aba and embedded in C_1. □

Binary chains may be used to simulate strings in a language, as follows.

Lemma 4.9 Consider the conjunctive queries

$$C_1 : \ p(\vec{H}) :- \ \Gamma_1, \mathcal{C}_1.$$
$$C_2 : \ p(\vec{H}) :- \ \Gamma_2, \mathcal{C}_2.$$

where $\Gamma_1, \Gamma_2, \mathcal{C}_1$ and $\mathcal{C}_2$ are conjunctions of EDB predicate occurrences. Let X and Y be distinguished variables (i.e. they appear in $\vec{H}$). Let Γ_1 be a binary chain from X to Y, embedded in C_1 and representing the terminal string $t = t_1 \ldots t_l$, and let Γ_2 be a binary chain from X to Y, embedded in C_2 and representing the terminal string $s = s_1 \ldots s_n$. Assume that there is a containment mapping g from

$$p(\vec{H}) :- \ \mathcal{C}_2.$$

into

$$p(\vec{H}) :- \ \mathcal{C}_1.$$

Then, there is a containment mapping $h : C_2 \rightarrow C_1$ iff $n = l$ and the strings $t_1 \ldots t_l$ and $s_1 \ldots s_n$ are identical.

Proof. Assume that Γ_1 and Γ_2 are as below.

$$\Gamma_1 : \ t_1(X, U_1), \ldots, t_l(U_{l-1}, Y).$$
$$\Gamma_2 : \ s_1(X, V_1), \ldots, s_l(V_{n-1}, Y).$$

Assume that the strings are identical. Let h be the identity mapping on distinguished variables (the variables in $\vec{H}$), and define $h(V_i) = U_i$ for all i. The function f defined by

$$f(A) = \begin{cases} h(A) & \text{if } h \text{ is defined on } A \\ g(A) & \text{if } g \text{ is defined on } A \end{cases}$$

is a containment mapping from C_2 into C_1.

For the converse, assume that h is a containment mapping from C_2 into C_1. Then $h(X) = X$ and $h(Y) = Y$, since X and Y are distinguished.

If $n = 0$, then $\Gamma_2 = s_1(X, Y)$. Since h is the identity of distinguished variables, and since s_1 does not appear in C_1 by assumption, we may conclude that $\Gamma_2 = h(s_1(X, Y)) = s_1(X, Y)$.

Consider any $n > 0$. If $l = 0$ (that is, $\Gamma_1 = t_1(X, Y)$), then $h(s_1(X, V_1))$ must be $t_1(X, Y)$, and $h(V_1) = Y$, a contradiction since Y does not appear as the first argument of any atom in Γ_1 (so $s_2(V_1, V_2)$ has no destination in Γ_1). If $l > 0$, then the only atom in Γ_1 whose first argument is X is $t_1(X, U_1)$. Thus, $h(s_1(X, V_1)) = t_1(X, U_1)$, yielding $s_1 = t_1$ and $h(V_1) = U_1$. An inductive repetition on l may be used to show that $l = n$, and $s_1 \ldots s_n = t_1 \ldots t_n$. □

Example 4.5 Let the conjunctive queries C_1 and C_2 be as follows.

C_1 : $p(X, Y, W, Z) :- a(X, U_1), b(U_1, U_2), a(U_2, Y), f(W), f(Z)$.
C_2 : $p(X, Y, W, Z) :- a(X, V_1), b(V_1, V_2), a(V_2, Y), f(W), f(Z)$.

Then, the mapping h defined by $h(X) = X, h(Y) = Y, h(W) = W, h(Z) = Z, h(V_1) = U_1, h(V_2) = U_2$ is a containment mapping from C_2 into C_1. However, there is no containment mapping from C_3 or C_4 into C_1, where C_3 and C_4 are defined as follows.

C_3 : $p(X, Y, W, Z) :- b(X, V_1), b(V_1, V_2), a(V_2, Y), f(W), f(Z)$.
C_4 : $p(X, Y, W, Z) :- a(X, V_1), a(V_1, Y), f(W), f(Z)$. □

The transformation from G to $\mathcal{P}$ is effected by the following algorithm.

Algorithm 4.1
INPUT: an ϵ-free grammar G with nonterminals $N_1, \ldots, N_m$.
OUTPUT: a single-IDB program $\mathcal{P}$ to simulate G[3].

1. The head of every rule is $p(X, Y, W, Z, N_1, \ldots, N_m)$.

2. Consider every production

   d_i : $N_j \to \beta$

   in G, where β is of length n. Let $\{U_1, \ldots, U_{n-1}\}$ be a set of new and distinct variables, and let γ be the $\{X, U_1, \ldots, U_{n-1}, Y\}$-chain representing β. We construct the rule

   r_i : $p(X, Y, W, Z, N_1, \ldots, N_m) :- \gamma, f(N_j), f(W)$.

[3]The manner of the simulation will be discussed later.

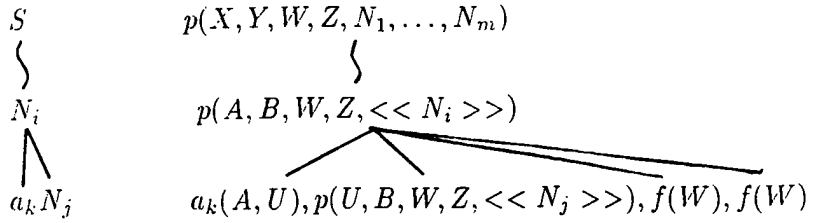


Figure 4.3: Simulating a derivation.

3. Add a single basis rule

$$b: \ p(X, Y, W, Z, N_1, \ldots, N_m) :- \ f(Z).$$

□

If G is linear, then $\mathcal{P}$ is linear. Further, if G has k extensional productions, then $\mathcal{P}$ has $k + 1$ initialisation rules.

Example 4.6 Consider the linear grammar G of Example 4.2. Our construction produces the following rules.

$p(X, Y, W, Z, S, B) :- a(X, U), p(U, Y, W, Z, W, Z), f(S), f(W).$
$p(X, Y, W, Z, S, B) :- b(X, V), p(V, Y, W, Z, Z, W), f(S), f(W).$
$p(X, Y, W, Z, S, B) :- b(X, Y), f(B), f(W).$ □

The importance of the variables W and Z is that they are *persistent*; that is, they appear in their "home" positions in every p-atom resulting from a top-down expansion in $\mathcal{P}$.

Lemma 4.10 Let $\mathcal{P}$ be a single-IDB program defining the predicate p, and let X_i be the ith variable in the head of every rule. Assume that the ith argument of every p-atom in the body of every rule is X_i. Then the ith argument of every p-leaf in any top-down expansion of p using the rules in $\mathcal{P}$ is X_i.
Proof. By induction on the number n of rule applications in the expansion. □

The intention of our construction is that the rules of $\mathcal{P}$ mimic derivations in G, to produce binary chains to represent every string in $L(G)$. The idea is illustrated in Figure 4.3. In the figure, the variable A may be a new nondistinguished variable, or the distinguished variable X. Similarly, B may be a nondistinguished variable, or the distinguished variable Y. U is a new nondistinguished variable.

Example 4.7 Figure 4.4 shows how a binary chain representing the string bb is generated by the program of Example 4.4. □

However, these rules also may be used to mimic "illegal" derivations in the grammar. That is, the production $N_i \rightarrow a_k N_j$ cannot be used to expand the nonterminal N_l, if

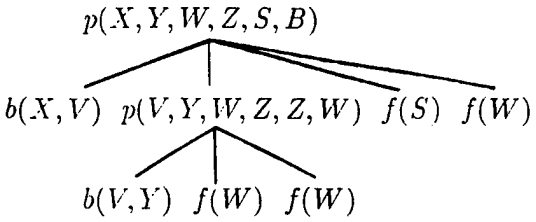


Figure 4.4: Generating a string.

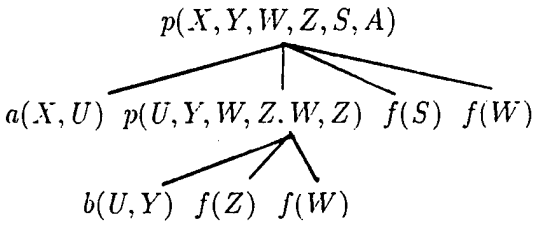


Figure 4.5: Illegal expansion.

$N_l \neq N_i$. However, the rule in $\mathcal{P}$ that corresponds to the production $N_i \rightarrow a_k N_j$ can, in fact, be used to expand a p-atom resulting from the application of the rule for $N_m \rightarrow a_o N_l$. We detect such illegal top-down expansions through the use of the conjuncts $f(N_i)$ in the rules of $\mathcal{P}$. A conjunctive query resulting from an illegal expansion as described above will contain the atom $f(Z)$, and is hence contained in the basis rule b. Figure 4.5 illustrates the attempt of $\mathcal{P}$ in Example 4.6 to expand the rule representing the production $S \rightarrow aS$ through the rule representing the production $B \rightarrow b$.

More formally, we say that a conjunctive query (or top-down expansion) generated by $\mathcal{P}$ is *illegal* if its body contains the conjunct $f(Z)$. Further, let us define *legal($\mathcal{P}$)* to be the union of all the conjunctive queries generated by $\mathcal{P}$ that are not illegal.

Lemma 4.11 Let C be an illegal conjunctive query generated by $\mathcal{P}$. Then $C \subset b$, where b is the nonrecursive rule added in step 3 of the algorithm.

Proof. Both conjunctive queries have the root $p(X, Y, W, Z, N_1, \ldots, N_m)$, and the identity mapping on variables in the root is a containment mapping from b into C. □

The simulation of G by $\mathcal{P}$ is formalised in the next three lemmas.

Lemma 4.12 Let S be the legal top-down expansion

S : $p(X, Y, W, Z, N_1, \ldots, N_m)$:- A, $p(< U_a, V_1 >< \ldots >)$, Γ, $f(N_i)$, $f(W)$.

where U_a and V_1 are distinct, A is a chain from X to U_a representing some string α in S, Γ is a chain from V_1 to Y representing some string γ in S, and the conjunction

$$(A, p(< U_a, V_1 >< N_j >>), \Gamma)$$

is a chain from X to Y representing $\alpha N_j \gamma$ in S.

Construct the top-down expansion T by expanding the indicated p-atom through some rule r_k constructed by Algorithm 4.1 from the production $d_k : N_m \rightarrow \beta$. Then

1. If $N_m \neq N_j$, T is illegal; and
2. If $N_m = N_j$, then T is of the form

$$T: \ p(X, Y, W, Z, N_1, \ldots, N_m) :- \Delta, f(N_i), f(W).$$

where Δ is a chain from X to Y, representing $\alpha\beta\gamma$ and embedded in T.

Proof. By the construction of the rule r_k by Algorithm 4.1, r_k is of the form

$$r_k: \ p(X, Y, W, Z, N_1, \ldots, N_m) :- B, f(N_m), f(W).$$

where B is a $\{X, R_1, \ldots, R_n, Y\}$-chain corresponding to β and the R_l are distinct variables not appearing in the rule head.

If $N_m \neq N_j$, then (since the N_m-argument of $p(< U_a, V_1 >< N_j >>)$ is Z), the body of T contains the conjunct $f(Z)$, and T is therefore illegal. However, if $N_m = N_j$, then by the persistence of W, the only f-atoms added are copies of $f(W)$. Assume that the R_l are renamed to new variables D_l in the expansion; then the children of the indicated p-atom in S forms a chain from U_a to V_1, representing β in T. Setting $\Delta = (A, B, \Gamma)$ completes the proof. □

Lemma 4.13 If β is a sentential form derived from the nonterminal N_i by G, then

$$T: \ p(X, Y, W, Z, N_1, \ldots, N_m) :- B, f(N_i), f(W).$$

is a top-down expansion in *legal*($\mathcal{P}$), where B is a chain from X to Y representing β and embedded in T.

Corollary. Assume $a_{i_1} a_{i_2} \ldots a_{i_k} \in$ *yield*(N_i). Then *legal*($\mathcal{P}$) generates the conjunctive query

$$C: \ p(X, Y, W, Z, N_1, \ldots, N_m) :- B, f(N_i), f(W).$$

where B is a binary chain representing $a_{i_1} \ldots a_{i_k}$.

Proof. Assume $N_i \stackrel{n}{\Rightarrow} \beta$. We prove our result by induction on n. If $n = 1$, then $N_i \rightarrow \beta$ is a production and our result follows by construction. Assume the truth of the hypothesis for $i < n$, where $n > 1$. Assume that $N_i \stackrel{n}{\Rightarrow} \delta$; then δ is of the form $\alpha\beta\gamma$, where $N_i \stackrel{n-1}{\Rightarrow} \alpha N_j \gamma$, and where $d_k : N_j \rightarrow \beta$ is a production. By our inductive hypothesis, $\mathcal{P}$ generates a legal top-down expansion

$S:\ p(X,Y,W,Z,N_1,\ldots,N_m) :- A,\ p(< U_a, V_1 >< N_j >>),\ \Gamma, f(N_i), f(W).$

where $(A, p(U_a, V_1) << N_j >>), \Gamma)$ is a chain from X to Y, embedded in S and representing $\alpha N_j \gamma$, A is a chain from X to U_a representing α in S, and Γ is a chain from V_1 to Y representing γ in S. We construct T by expanding the indicated p-atom through the rule r_k corresponding to the production d_k, and our result follows by Lemma 4.12. □

Lemma 4.14 Let T be a top-down expansion in *legal($\mathcal{P}$)*. Then T is of the form

$T:\ p(X,Y,W,Z,N_1,\ldots,N_m) :- B, f(N_i), f(W).$

where B is a chain from X to Y, embedded in T and representing a sentential form $\beta \in$ *yield*(N_i) for some N_i.
Corollary. Let C be a conjunctive query in *legal($\mathcal{P}$)*. Then C is of the form

$p(X,Y,W,Z,N_1,\ldots,N_m) :- B, f(N_i), f(W).$

where B is a binary chain representing some string $a_{i_1} \ldots a_{i_k} \in$ *yield*(N_i).
Proof. Let T be a top-down expansion in which n rules are applied. We prove our result by induction on n.

If $n = 1$, then the rule applied is of the form

$r_k:\ p(X,Y,W,Z,N_1,\ldots,N_m) :- B, f(N_i), f(W).$

where B is a chain representing some string β and

$d_k:\ N_i \rightarrow \beta$

is a production in G.

Assume the truth of the hypothesis for $i < n$, where $n > 1$. Let the following be a legal top-down expansion generated by n rule applications.

$T:\ p(X,Y,W,Z,N_1,\ldots,N_m) :- \Delta.$

T must be generated by the expansion of some p-atom in some legal top-down expansion S (generated by $n-1$ rule applications) through some rule r_k. By our inductive hypothesis, S is of the form

$S:\ p(X,Y,W,Z,N_1,\ldots,N_m) :- A,\ p(< U_a, V_1 >< N_j >>),\ \Gamma, f(N_i), f(W).$

where $N_i \Rightarrow \alpha N_j \gamma$, and where the conjunction

$$(A, p(U_a, V_1) << N_j >>), \Gamma)$$

is a chain from X to Y, embedded in S and representing $\alpha N_j \gamma$ (so that A is a chain from X to U_a representing α in S and Γ is a chain from V_1 to Y representing γ in S). Consider

the rule r_k that is used to expand the indicated p-atom in S to obtain T. By construction, the rule r_k is of the form

$r_k : p(X, Y, W, Z, N_1, \ldots, N_m) :- B, f(N_m), f(W)..$

such that
$d_k : N_m \rightarrow \beta$
is a production in G. By Lemma 4.12 and the assumed legality of T, we must have $N_m = N_j$; hence $N_i \Rightarrow \alpha\beta\gamma$ and our result follows by Lemma 4.12. □

4.5.2 Using the construction

Let G_1 be an ϵ-free grammar over $\Sigma = \{a_1, \ldots, a_k\}$, with start symbol N_1 and intensional nonterminals $N_1, \ldots, N_m$. Let G_2 be an ϵ-free grammar over Σ, with start symbol M_1 and nonterminals $M_1, \ldots, M_l$. Without loss of generality, we assume that the nonterminals of G_1 and G_2 are distinct.

Let S be a new nonterminal. Construct an ϵ-free grammar G_3 (with start symbol S) as the union of all the productions in G_1 and G_2, and add the two productions s_1 and s_2 described below.

$s_1 : S \rightarrow N_1$
$s_2 : S \rightarrow M_1$

$L(G_3)$ is clearly $L(G_1) \cup L(G_2)$.

Let G_4 be the ϵ-free grammar obtained from G_3 by deleting the production s_1. $L(G_4)$ is clearly the same as $L(G_2)$; we have merely introduced a unit production to change the start symbol. Note that for any N_i or M_i, the yield of this nonterminal is the same for G_3 and G_4; that is, the only nonterminal for which the yields may differ in G_3 and G_4 is the start symbol S.

Apply Algorithm 4.1 to G_3 and G_4 to create programs $\mathcal{P}$ and $\mathcal{Q}$ respectively, ensuring that the rule head in each case is $p(X, Y, W, Z, S, N_1, \ldots, N_k, M_1, \ldots, M_l)$; that is, the nonterminals of G_3 and G_4 appear in the same positions in the heads of all rules in $\mathcal{P}$ and $\mathcal{Q}$. Hence, every rule in $\mathcal{Q}$ also appears in $\mathcal{P}$, and we may conclude that $\mathcal{Q} \subset \mathcal{P}$.

Lemma 4.15 $\mathcal{P} \subset \mathcal{Q}$ iff $L(G_1) \subset L(G_2)$.
Corollary. $\mathcal{P} \equiv \mathcal{Q}$ iff $L(G_1) \subset L(G_2)$.
Proof.

By the construction of G_3 and G_4, the strings generated by any nonterminal $N \neq S$ are the same for G_3 and G_4.

Assume that $L(G_1) \subset L(G_2)$. Then $L(G_3) \subset L(G_4)$ by the construction of G_3 and G_4, and any string generated by any nonterminal in G_3 is also generated by the same nonterminal in G_4. Consider any conjunctive query C generated by $\mathcal{P}$; if C is illegal, then C is contained in the basis rule b of $\mathcal{Q}$. If C is legal, then by Lemma 4.14, assumption, Lemma 4.13 and Lemma 4.9, C is contained in some legal conjunctive query $\mathcal{D}$ generated by $\mathcal{Q}$.

For the converse, assume that $\mathcal{P} \subset \mathcal{Q}$. Then every legal conjunctive query $C_{\mathcal{P}}$ generated by $\mathcal{P}$ is contained in some legal conjunctive query $C_{\mathcal{Q}}$ generated by $\mathcal{Q}$, and Lemmas 4.13, 4.14 and 4.9 suffice to complete the proof. □

Theorem 4.1 Let $\mathcal{P}$ and $\mathcal{Q}$ be safe, linear, single-IDB programs with head-rectified rules and five basis rules. Then $\mathcal{P} \subset \mathcal{Q}$ ($\mathcal{P} \equiv \mathcal{Q}$) is undecidable.
Proof. Let G_1 and G_2 be linear bounded-basis grammars over the alphabet $\Sigma = \{a_1, a_2\}$ and apply the construction of the preceding lemma; our result follows by Lemmas 4.15 and 4.4. Safety is imposed as in Section 4.4. Note that, if G_1 and G_2 are bounded-basis, then Lemma 4.15 holds even if G_1 and G_2 have the same extensional nonterminals, and the number of basis rules can therefore be reduced to three. □

Now, let G_1 be the following linear grammar over $\Sigma = \{a_1, a_2\}$.

$d_1 : N_1 \rightarrow a_1 N_1$
$d_2 : N_1 \rightarrow a_2 N_1$
$d_3 : N_1 \rightarrow a_1$
$d_4 : N_1 \rightarrow a_2$

G_1 generates $\{a_1 + a_2\}^+$. Let G_2 be a linear, bounded-basis grammar over Σ, with nonterminals $M_1, \ldots, M_l$ and start symbol M_1. We assume without loss of generality that N_1 is distinct from each M_i. Construct the grammar G_3 with new start symbol S by taking the union of the productions in G_1 and G_2, and adding the production

$s : S \rightarrow N_1 M_1$.

G_3 generates $\Sigma^+ L(G_2)$. Apply Algorithm 4.1 to G_3 to produce the program $\mathcal{P}$. Note that $\mathcal{P}$ has five basis rules and only one nonlinear rule (corresponding to the production s).

Consider the legal conjunctive queries in $\mathcal{P}$. Since S appears on the left-hand side of only the production s and does not appear on the right-hand side of any production, the nonlinear rule r_s representing the production s is used only in the simulation of strings in $yield(S)$, and the rule can only be used at the root of a top-down expansion involved in such a simulation.

Theorem 4.2 Let $\mathcal{P}$ be a safe, head-rectified, single-IDB program with five basis rules and one nonlinear rule. The base-case linearizability of $\mathcal{P}$ is undecidable.
Proof. Let G_3 be the grammar of the preceding discussion, and let $\mathcal{P}$ be the result of applying Algorithm 4.1 to G_3. Any illegal conjunctive query generated by $\mathcal{P}$ is contained in the basis rule b. Every legal conjunctive query representing a string in $yield(N_1)$ or $yield(M_i)$ is linear, and hence right-linear. The legal conjunctive queries representing strings in $yield(S)$ simulate $\Sigma^+ L(G_2)$, and the right-linear subset of these queries simulates $\Sigma L(G_2)$; our result follows by Lemma 4.5. Safety is imposed as in Section 4.4. □

4.6 Single-recursive-rule programs

In this section, we present a construction whereby an arbitrary Modified Chomsky Normal Form (MCNF) grammar may be simulated using a head-rectified, single-IDB program with a single recursive rule and a bounded number of basis rules. The construction may be used to show that sequencability is undecidable, even for programs with only two recursive rules. In addition, the construction can be used to prove the undecidability of equivalence (or containment) of programs with a single recursive rule.

4.6.1 The construction

Let H be an MCNF grammar over $\Sigma = \{a_1, a_2\}$, with nonterminals $N_1, \ldots, N_m$, start symbol N_3 and extensional productions $N_1 \rightarrow a_1$ and $N_2 \rightarrow a_2$. We construct a program $\mathcal{P}$, with one nonrecursive rule, to simulate H.

The program $\mathcal{P}$ defines the IDB predicate p, and the head of each rule is

$$p(R, X, Y, W, Z, G, A, B, N_1, \ldots, N_m)$$

As before, $<< N_i >>$ denotes an m-vector in which the ith component is W, and in which all other components are Z. The expression

$$p(< K, L >< M >< R, T ><< N_i >>)$$

will be used to represent the p-atom

$$p(Z, K, L, W, Z, M, R, T, << N_i >>)$$

. That is, the first argument is always Z, and the 4th and 5th are always W and Z respectively (so that W and Z appear in their "home" positions). The EDB predicates in $\mathcal{P}$ are a_1, a_2, f, g and h.

The variables in the head of each rule have the following purposes:

1. R is a switch that is relevant only to the proof that sequencability is undecidable. The R-position in the arguments of each p-atom in the body of the recursive rule will be occupied by the variable Z, described below.

2. X and Y are the end-points of binary chains representing strings in $L(H)$, as in the preceding section.

3. W and Z are guard variables, as in the preceding section. They are used to weed out illegal conjunctive queries generated by the programs (i.e., queries representing impossible derivations in the grammar).

4. G is a guard position. Intuitively, a p-atom may be legally expanded through the recursive rule if its G-th argument is W, but not if its G-th argument is Z.

5. A and B are used to allow a choice in expanding one of two or-productions, in a manner to be described.

6. $N_1, ..., N_m$ represent the corresponding nonterminals in the grammar.

The rules of $\mathcal{P}$ are constructed as follows.

Algorithm 4.2
INPUT: an MCNF grammar G with nonterminals $N_1, \ldots, N_m$, where N_1 is the extensional nonterminal representing a_1 and N_2 is the extensional nonterminal representing a_2.
OUTPUT: a head-rectified, single-IDB program $\mathcal{P}$ with one recursive rule r and 9 basis rules $i_1 \ldots i_9$.

1. The head of each rule is $p(R, X, Y, W, Z, G, A, B, N_1, \ldots, N_m)$.

2. $\mathcal{P}$ has the following 9 basis rules:

$p(R, X, Y, W, Z, G, A, B, N_1, \ldots, N_m)$:-
i_1 : $a(X, Y), f(N_1), f(G)$.
i_2 : $b(X, Y), f(N_2), f(G)$.
i_3 : $f(Z)$.
i_4 : $g(G)$.
i_5 : $g(W)$.
i_6 : $f(U), g(U)$.
i_7 : $h(A, B), f(G)$.
i_8 : $h(U, V), h(V, W)$.
i_9 : $h(U, U)$.

3. The body of the recursive rule r for $\mathcal{P}$ has the atoms $f(W)$,$f(G)$, $g(Z)$, $g(N_1)$, $g(N_2)$ and $h(E, F)$, where E and F are nondistinguished variables that appear nowhere else in the program). The body of r also has p-atoms for each intensional production in the grammar, as follows:

   (a) Let $J \rightarrow K$ be a copy production, and let U_J be a new nondistinguished variable. Then, the recursive rule contains the atom

$$p(< X, Y >< J >< U_J, U_J >\!<< K >\!>)$$

   This atom (and any version of it in a top-down expansion) is called a *copy-atom representing* K in the production.

   (b) Let $J \rightarrow K$ and $J \rightarrow L$ be a pair of or-productions, and let T_J, U_J and V_J be new nondistinguished variables. Then, the body of $r_{\mathcal{P}}$ contains the atoms

$$p(< X, Y >< J >< T_J, U_J >\!<< K >\!>)$$
$$p(< X, Y >< J >< U_J, V_J >\!<< L >\!>)$$

The first of these is the *or-atom representing* K in the first production, and the second is the or-atom representing L in the second production. Versions of these two atoms in a top-down expansion such that both atoms have the same parent are called *sibling or-atoms*. The idea is that we need never recursively expand both these p-atoms, since either one (but not both) may be expanded using initialisation rule i_7.

(c) Let $J \rightarrow KL$ be an and-production, and let T_J, U_J and V_J be new nondistinguished variables. $r_{\mathcal{P}}$ has the two p-atoms

$$p(< X, T_J >< J >< U_J, U_J >\!<< K >\!>)$$
$$p(< T_J, Y >< J >< V_J, V_J >\!<< L >\!>)$$

The first of these atoms is called the *and-atom representing* K in the production, and the second is called the and-atom representing L in the production. Versions of these atoms that have the same parent are called *sibling and-atoms*. The idea is that both atoms are (recursively) expanded, to create chains from X to T_J, and from T_J to Y.

□

Example 4.8 The following MCNF grammar G over $\Sigma = \{c, d\}$ generates Σ^+.

$$S \rightarrow TT \quad T \rightarrow S \quad T \rightarrow H \quad H \rightarrow C \quad H \rightarrow D$$
$$C \rightarrow c \quad D \rightarrow d$$

The corresponding program is

r : $p(R, X, Y, W, Z, G, A, B, C, D, S, T, H)$:–
$p(Z, X, V, W, Z, S, I_1, I_1, Z, Z, Z, W, Z)$,
$p(Z, V, Y, W, Z, S, I_2, I_2, Z, Z, Z, W, Z)$,
$p(Z, X, Y, W, Z, T, I_3, I_4, Z, Z, W, Z, Z)$,
$p(Z, X, Y, W, Z, T, I_4, I_5, Z, Z, Z, Z, W)$,
$p(Z, X, Y, W, Z, H, I_6, I_7, W, Z, Z, Z, Z)$,
$p(Z, X, Y, W, Z, H, I_7, I_8, Z, W, Z, Z, Z)$.
$f(W), f(G), g(Z), g(C), g(D), h(E, F)$.

$p(Z, X, Y, W, Z, G, A, B, S, T, C, D)$:–
i_1 : $c(X, Y), f(C), f(G)$.
i_2 : $d(X, Y), f(D), f(G)$.
i_3 : $f(Z)$.
i_4 : $g(G)$.
i_5 : $g(W)$.
i_6 : $f(U), g(U)$.
i_7 : $h(A, B), f(G)$.
i_8 : $h(U, V), h(V, W)$.

$i_9 : h(U, U).$

In the recursive rule, the first two p-atoms represent the and-production $S \rightarrow TT$: hence, these atoms are and-atoms. The next two p-atoms represent the or-productions $T \rightarrow S$ and $T \rightarrow H$, and are therefore or-atoms. Similarly, the 5th and 6th p-atoms are or-atoms representing the or-productions $H \rightarrow C$ and $H \rightarrow D$. The initialisation rules i_1 and i_2 represent the extensional productions $C \rightarrow c$ and $D \rightarrow d$ respectively. □

Intuition

The intention is that the program $\mathcal{P}$ simulate the strings in the grammar G, by generating binary chains to represent the strings in $L(G)$. The basic ideas are as follows.

Illegal expansions As in the previous section, we detect illegal top-down expansions through the existence of the atom $f(Z)$ in the expansion. The idea is as follows. Consider any top-down expansion using only the recursive rule; as in the previous section, the variables W and Z are persistent in the p-atoms of the expansion. At each stage of the expansion, p-atoms in the body of the recursive rule are "activated" by placing W in the guard (6th) position of the atom, or "deactivated" by placing Z in this position. If the guard position of a p-atom is Z, then the application of the recursive rule will introduce the atom $f(Z)$, and the expansion is illegal; hence, such atoms must be terminated by a basis rule in all legal expansions. The activation of atoms in the body of the recursive rule enforces the fact that legal expansions correspond to derivations in the grammar.

Example 4.9 Consider the grammar and program of Example 4.8. Assume that at some stage, the atom corresponding to the production $T \rightarrow H$ has been activated: that is, the atom is of the form

$$p(Z, A, B, W, Z, W, I_4, I_5, Z, Z, Z, Z, W)$$

Applying the recursive rule to this atom yields atoms of the form

$$\begin{array}{l} p(Z, A, V, W, Z, Z, I_1, I_1, Z, Z, Z, Z, Z), \\ p(Z, V, B, W, Z, Z, I_2, I_2, Z, Z, Z, Z, Z), \\ p(Z, A, B, W, Z, Z, I_3, I_4, Z, Z, Z, Z, Z), \\ p(Z, A, B, W, Z, Z, I_4, I_5, Z, Z, Z, Z, Z), \\ p(Z, A, B, W, Z, W, I_6, I_7, W, Z, Z, Z, Z), \\ p(Z, A, B, W, Z, W, I_7, I_8, Z, W, Z, Z, Z), \\ f(W), f(W), g(Z), g(C), g(D), h(E, F). \end{array}$$

Note that the first four atoms are deactivated (with Z in position 6), and the last two atoms are activated (with W in position 6). Note also that the atoms that are activated are precisely the atoms representing productions whose head is H, as required by the fact that the atoms corresponding to the production $T \rightarrow H$ has been expanded. Finally, note that the 5th atom (the first to be activated) represents the production $H \rightarrow C$, and the arguments representing the nonterminals in this atom is $<< C >>$. Similarly, note that the

6th atom (the second to be activated) represents the production $H \rightarrow D$, and the arguments representing the nonterminals in this atom is $<< D >>$. □

Or-productions Another wrinkle is the fact that if an or-production is used in a derivation in the grammar, then there are two activated atoms in the simulating expansion. The idea, here, is that either one of these atoms may be terminated by the initialisation i_7, but not both (otherwise the result is contained in the initialisation rule i_8).

Example 4.10 Consider the expansion of Example 4.9, in which the or-atoms corresponding to the or-productions $H \rightarrow C$ and $H \rightarrow D$ have been activated. Either one of these atoms may be terminated by rule i_7. However, if both are terminated in this way, then the result contains the atoms $h(I_6, I_7), h(I_7, I_8),$, and the result is contained in i_8. □

The simulation

We say that a conjunctive query (or top-down expansion) generated by the program is *illegal* if it is contained in one of the initialisation rules $i_1 \ldots i_9$, and *legal* otherwise. No interesting top-down expansions are contained in i_1 or i_2, as the following lemma shows.

Lemma 4.16 Let T be a top-down expansion in which the recursive rule r is used. Then $T \not\subseteq i_1$ and $T \not\subseteq i_2$.
Proof. The root of T is expanded using r, since r is the only recursive rule in the program. T, i_1 and i_2 have the same root (the rule head that is common to all rules), and any containment mapping c from i_1 or i_2 into T must satisfy $c(N_1) = N_1$ and $c(N_2) = N_2$. Now, $f(N_1)$ appears in the body of i_1 and $f(N_2)$ appears in the body of i_2. However, neither N_1 nor N_2 appears in the body of r, and hence neither $f(N_1)$ nor $f(N_2)$ appears in T. Thus, there is no containment mapping c from i_1 or i_2 into T. □

The following lemmas show us that most of the initialisation rules may never be used in any interesting top-down expansion.

Lemma 4.17 Let T be a top-down expansion in which one of the basis rules i_3, i_5, i_6, i_8 and i_9 is used. Then T is illegal.
Corollary. Only the rules r, i_1, i_2 and i_7 are used in any legal top-down expansion.
Proof. The head of T is the rule head that is common to all rules. By Lemma 4.10, W and Z appear in their "home" positions in every p-atom in T. Hence, if i_j is used in T, then T is contained in i_j. □

Lemma 4.18 Assume that T is a legal top-down expansion that includes a p-atom in which the 6th argument (the "home" position for C) is W. Then one of r, i_1, i_2 and i_7 must be used to expand such a p-atom.
Proof. By Lemma 4.17, one of r, i_1, i_2, i_4 and i_7 must be used to expand this p-atom. However, if i_4 is used, then $g(W)$ appears in the body of T, and T is contained in i_5. □

Lemma 4.19 Assume that T is a legal top-down expansion that includes a p-atom in which the 6th argument (the "home" position for G) is Z. Then only i_4 is used to expand any such atoms.
Proof. By Lemma 4.17, rules i_3, i_5, i_6, i_8 and i_9 may not be used in the expansion. If one of r, i_1, i_2 and i_7 is used, then $f(Z)$ appears in the body of T, and $T \subset i_3$. □

Lemma 4.20 In any legal top-down expansion T, no copy-atom or and-atom is expanded using initialisation rule i_7.
Proof. Assume the converse; then the body of T contains the atom $h(G, G)$ for some variable G, and T therefore is contained in initialisation rule i_9. □

Lemma 4.21 In any legal top-down expansion T, no two sibling or-atoms are expanded using rule i_7.
Proof. Otherwise, T is contained in initialisation rule i_8. □

Lemma 4.22 Let T be a legal top-down expansion in which s and t are two sibling or-atoms (or sibling and-atoms). If s is expanded through the initialisation rule i_4, then so is t.
Proof. Since s and t are siblings, both have the same 6th argument (the argument in the home position for G), say S. If s is expanded through i_4, then $g(S)$ appears in the body of T. Since T is assumed to be legal, Lemma 4.17 requires that one of the rules r, i_1, i_2, i_4 and i_7 is used to expand t; however, if any rule but i_4 is used, then $f(S)$ is a conjunct in the body of C and T is contained in rule i_6 (and is therefore illegal). □

Lemma 4.23 In any top-down expansion T, i_7 may only be used to expand one of two sibling or-atoms, and the sibling or-atom is expanded using one of r, i_1 and i_2.
Proof. By Lemmas 4.21 and 4.22. □

Lemma 4.24 Let T be a legal top-down expansion in which some atom

$$p(< I_1, I_2 >< I_3 >< I_4, I_5 >\!<< N_k >\!>)$$

(with arbitrary I_j) is expanded through the recursive rule r. Then $N_k \neq N_1$ and $N_k \neq N_2$.
Proof. Assume the converse; then, the body of the recursive rule has the atom $g(W)$, and T is contained in i_5. □

Lemma 4.25 Let T be a legal top-down expansion in which some atom

$$p(< I_1, I_2 >< I_3 >< I_4, I_5 >\!<< N_k >\!>)$$

(with arbitrary I_j) is expanded through the initialisation rule i_j, where $j \in \{1, 2\}$. Then $N_k = N_j$.
Proof. Assume the converse. Then $f(Z)$ appears in the body of T, and $T \subset i_3$. □

Lemma 4.26 Let T be a conjunctive query generated by $\mathcal{P}$ in which the recursive rule r is used at least twice. Then every p-atom at depth $n > 1$ in T has as 6th argument (the argument in the "home" position of G) either W or Z.
Proof. By Lemma 4.10, W and Z appear in their home positions in every p-atom in T. By construction, every variable in the home position for any N_i in any p-atom in T is W or Z, and the "guard" position is therefore occupied by W or Z in any child of such an atom. □

Let us say that a conjunctive query is *restricted* if whenever one of two sibling or-atoms is expanded through r, i_1 or i_2, then its sibling atom is expanded through i_7.

Lemma 4.27 Every legal conjunctive query C is contained in a conjunctive query D that is both legal and restricted.
Proof. Let us assume that one of the sibling or-atoms

$$p(< U, V >< S >< I_1, I_2 >\!<< N_i >\!>)$$

and

$$p(< U, V >< S >< I_2, I_3 >\!<< N_j >\!>)$$

is expanded through r, i_1 or i_2 in the top-down expansion T establishing the conjunctive query C. Since C is legal, the root of T is expanded using rule r, and C therefore contains an atom of the form $h(E, F)$. By Lemma 4.26 and Lemma 4.19 , S is W or G, and by construction $f(S)$ appears in the body of C. By Lemmas 4.18 and 4.22 , the sibling atom is expanded through one of r, i_1, i_2 and i_7. If i_7 is not used, then since the distinguished variables A and B appear nowhere in the bodies of r, i_1 or i_2, the variables I_1, I_2 and I_3 appear nowhere in the fringe of T (i.e., in the body of C). Thus, we may construct a new conjunctive query C' from C, by expanding the first or-atom through rule i_7 to produce the atoms $f(S)$ and $h(I_1, I_2)$ for variables I_1 and I_2 that appear nowhere else in the new conjunctive query C'. C is contained in C' because $f(S)$ and an atom of the form $h(E, F)$ appears in C. An inductive repetition suffices to remove all violations of restrictiveness in C, while preserving the legality of C. □

For any distinct variables U and V, define a $\{U, V\}$-atom corresponding to an intensional nonterminal N_i to be any p-atom

$$p(< U, V >< W >< H, I >\!<< N_i >\!>)$$

for any variables H and I, and let chains be defined as in the preceding section. Note that, depending on the variables H and I in any p-atom, a string of terminals and nonterminals has many corresponding chains.

Definition 4.6 Define a top-down expansion T to be *closed* if T is legal and restricted, and if the following hold.

1. Any p-atom in which the 6th argument (the argument representing the guard G) is the distinguished variable Z is expanded through i_4; that is, no such p-atom is a leaf.

2. Consider any two sibling or-atoms neither of which is expanded using i_4. Then one of these atoms is expanded using rule i_7.

3. No p-atoms at depth 1 in the tree are leaves; that is, every p-child of the root is expanded through some rule.

By Lemmas 4.19, 4.18, 4.22 and 4.26, every top-down expansion establishing a legal, restricted conjunctive query is closed. □

The following Lemmas formalise the simulation of the grammar G by the program $\mathcal{P}$.

Lemma 4.28 Let

$$p(< U, V >< S >< I_1, I_2 >< < N_i > >)$$

be an atom in a closed top-down expansion R, where S is one of W and G, where $U \neq V$, and where I_1 and I_2 are arbitrary. If this p-atom is expanded using one application of one of r, i_1 and i_2 to produce a closed top-down expansion T, then the subtree rooted at this p-atom contains the following.

1. The atom $f(S)$, and 0 or more occurrences of the atoms $f(W)$ and $g(Z)$.

2. 0 or more occurrences of atoms $h(P, Q)$ for distinct variables P and Q.

3. A chain from U to V representing γ, where $N_i \rightarrow \gamma$ is a production.

Proof. By Lemma 4.25, the rule i_k ($k \in \{1, 2\}$) may be used only if $N_i = N_k$; the subtree then contains the atoms $f(S)$, $f(W)$ and $a_k(U, V)$. Recall that the extensional productions are of the form $N_1 \rightarrow a_1$ and $N_2 \rightarrow a_2$.

Assume that r is used. By Lemma 4.24, $N_i \neq N_1$ and $N_i \neq N_2$; that is, N_i is an intensional nonterminal. Application of r yields the atoms $f(S)$ and $f(W)$, two copies of $g(Z)$ (since $N_i \neq N_1$ and $N_i \neq N_2$), and an atom of the sort $h(E', F')$ where E' and F' are distinct, new variables obtained by renaming E and F in the top-down expansion. Consider the p-atoms that are generated by this application of r. By construction, the 6th position of any atom representing a production $N_j \rightarrow \beta$ is Z, if $N_j \neq N_i$; hence, by Lemma 4.19, every such atom must be expanded using rule i_4 (yielding the atom $g(Z)$). Also by construction, the 6th argument of the atom(s) representing any production $N_i \rightarrow \gamma$ is W. There are three cases.

1. $N_i \rightarrow N_k$ is a copy-production. By construction, the corresponding atom in the indicated application of r is

$$p(< U, V >< W >< J_1, J_1 >< < N_k > >)$$

and our result follows.

2. $N_i \rightarrow N_k$ and $N_i \rightarrow N_l$ are or-productions. By construction, the corresponding atoms in the indicated application of r are

$$p(< U, V >< W >< J_1, J_2 >\!<< N_k >\!>)$$
$$p(< U, V >< W >< J_2, J_3 >\!<< N_l >\!>)$$

for some nondistinguished variables J_1, J_2 and J_3 that appear nowhere else in T (since these variables are renamed in the expansion). By Lemma 4.18, and since T is assumed closed, one of these atoms must be expanded through i_7 to add the atoms $f(W)$ and $h(I_1, I_2)$ (or $h(I_2, I_3)$) to T. Our result follows.

3. $N_i \rightarrow N_k N_l$ is an and-production, and the corresponding atoms in the indicated application of r are

$$p(< U, I >< W >< J_1, J_1 >\!<< N_k >\!>)$$
$$p(< I, V >< W >< J_2, J_2 >\!<< N_l >\!>)$$

where I is a renaming of the variable T_{N_i} to a new variable that appears nowhere else in the expansion. Our result follows.

□

Lemma 4.29 Let

$$p(< U, V >< S >< I_1, I_2 >\!<< N_i >\!>)$$

be an atom in a closed top-down expansion R, where S is one of W and G, where $U \neq V$, and where I_1 and I_2 are arbitrary. If $N_i \rightarrow \gamma$ is a production in the grammar G, then there is a closed top-down expansion T obtained by expanding this p-atom using exactly one application of r, i_1 or i_2, but with an arbitrary number of uses of other rules, such that the subtree rooted at this p-atom contains the following atoms.

1. The atom $f(S)$, and 0 or more occurrences of the atoms $f(W)$ and $g(Z)$.
2. 0 or more occurrences of atoms $h(P, Q)$ for distinct variables P and Q.
3. A chain from U to V representing γ.

Proof. If $i = 1$ or $i = 2$, then we may apply i_1 or i_2 to the indicated p-atom to obtain the result.

Assume that N_i is an intensional nonterminal. Expand the indicated p-atom through the recursive rule r. Then the "non-p" children of the indicated p-atom are $f(S)$, $f(W)$, $g(Z)$, and $h(E', F')$ where E' and F' are distinct, new variables obtained by renaming E and F in the top-down expansion. Consider the p-atoms that are generated by this application of r. By construction, the 6th position of any atom representing a production $N_j \rightarrow \beta$ is Z, if $N_j \neq N_i$; hence, every such atom may be expanded using rule i_4 (yielding the atom $g(Z)$, which appears as a child of the root.) Also by construction, the 6th argument of the atom(s) representing any production $N_i \rightarrow \gamma$ is W. There are three cases.

1. $N_i \rightarrow N_k$ is a copy-production. By construction, the corresponding atom in the indicated application of r is

$$p(< U, V >< W >< J_1, J_1 >\!<< N_k >\!>)$$

and our result follows.

2. $N_i \rightarrow N_k$ and $N_i \rightarrow N_l$ are or-productions. By construction, the corresponding atoms in the indicated application of r are

$$p(< U, V >< W >< J_1, J_2 >\!<< N_k >\!>)$$
$$p(< U, V >< W >< J_2, J_3 >\!<< N_l >\!>)$$

for some nondistinguished variables J_1, J_2 and J_3 that appear nowhere else in T (since these variables are renamed in the expansion). Without loss of generality, assume that γ is N_k. Then, the latter atom may be expanded through i_7 to add the atoms $f(W)$ and $h(I_2, I_3)$ to T. Our result follows.

3. $N_i \rightarrow N_k N_l$ is an and-production. Then, the corresponding atoms in the indicated application of r are

$$p(< U, I >< W >< J_1, J_1 >\!<< N_k >\!>)$$
$$p(< I, V >< W >< J_2, J_2 >\!<< N_l >\!>)$$

where I is a renaming of the variable T_{N_i} to a new variable that appears nowhere else in the expansion. Our result follows.

□

For any nonterminal J in G, recall that *yield*(J) represent all the strings generated by J (i.e., the strings that would be generated if J were the start symbol of G).

Lemma 4.30 Let T be a closed top-down expansion generated by $\mathcal{P}$. Then the body of T consists of the atoms $f(W), f(G), g(Z), g(N_1)$ and $g(N_2)$; atoms of the form $h(U, V)$, where U and V do not appear in the root; and for $3 \leq i \leq m$, either $g(N_i)$, or $f(N_i)$ and a chain representing some string δ of terminals and nonterminals such that $N_i \Rightarrow \delta$.

Corollary. Let C be a conjunctive query generated by $\mathcal{P}$, established by a closed top-down expansion T. Then the body of C consists of the atoms $f(W), f(G), g(Z), g(N_1)$ and $g(N_2)$: atoms of the form $h(U, V)$, where U and V do not appear in the root; and for $3 \leq i \leq m$, either $g(N_i)$, or $f(N_i)$ and a chain representing some terminal string δ such that $N_i \Rightarrow \delta$.

Proof. Let T be a closed top-down expansion in G. Since T is assumed closed, the root is expanded through the recursive rule r. For any intensional nonterminal N_i, consider all the atoms representing productions of the form $N_i \rightarrow \alpha$. The proof proceeds by induction on n, the number of applications of r, i_1 or i_2 to atoms representing productions of which N_i is the head.

If $n = 1$, then by Lemma 4.22, rule i_4 is used to expand all such atoms, and the atom $g(G)$ is introduced.

If $n = 2$, then rule r is used and the result follows by Lemma 4.28; note that the 6th argument (the "home" position for G) in any p-atom at depth 1 is G by construction. The induction also follows by Lemma 4.28.

To prove the corollary, we observe that in any conjunctive query (with EDB's at the leaves), if $h(U, V)$ is obtained by the expansion of an or-atom through i_7, then the occurrence of U (or V) in the sibling or-atom disappears when r, i_1 or i_2 is used to expand the latter, since the distinguished variables A and B do not appear in the body of these rules. □

The converse is also true.

Lemma 4.31 Assume that $N_i \Rightarrow \delta$, where $i > 2$. Then, $\mathcal{P}$ generates a closed top-down expansion T, as follows. The body of T consists of the atoms $f(W), f(G), g(Z), g(N_1)$ and $g(N_2)$; atoms of the form $h(U, V)$, where U and V do not appear in the root; and either $g(N_i)$, or $f(N_i)$ and a chain representing δ.

Corollary. Assume that $N_i \Rightarrow \delta$, where δ is a string of terminals and $i > 2$. Then, $\mathcal{P}$ generates a closed top-down expansion T generating a conjunctive query C, as follows. The body of C consists of the atoms $f(W), f(G), g(Z), g(N_1)$ and $g(N_2)$; atoms of the form $h(U, V)$, where U and V appear nowhere else in C; and either $g(N_i)$, or $f(N_i)$ and a binary chain representing δ.

Proof. Assume that $N_i \Rightarrow \delta$. Our proof proceeds by induction on n, the depth of the sentential forms derived by N_i in G. The basis ($n = 1$) follows by Lemma 4.29 (since the 6th position in any child of the root is G, if r is used to expand the root), and the induction also follows by Lemma 4.29.

The corollary is proved as in the previous lemma. □

4.6.2 Using the construction

Let G_1 be an arbitrary MCNF grammar over the alphabet $\Sigma = \{a_1, a_2\}$, with extensional nonterminals N_1 and N_2 representing a_1 and a_2 respectively, start symbol K_1 and nonterminals $K_1, \ldots, K_n$. Let G_2 be an arbitrary MCNF grammar over the same alphabet, with the same extensional nonterminals representing the same terminals, and with start symbol L_1 and nonterminals $L_1, \ldots, L_m$. Create a new start symbol S; construct the new grammar G_3 with start symbol S, with productions that are the union of the productions in G_1 and G_2, and with the new productions

$s_1 : S \rightarrow K_1$
$s_2 : S \rightarrow L_1$

Let G_4 be the grammar obtained from G_3 by deleting the production s_1, but with the same start symbol. Clearly, G_3 and G_4 have the same nonterminals, and *yield*(M) is the same in both grammars, for $M \neq S$. Also, $L(G_3) = L(G_1) \cup L(G_2)$ and $L(G_4) = L(G_2)$; hence, $L(G_4) \subset L(G_3)$, and $L(G_3) \subset L(G_4)$ iff $L(G_1) \subseteq L(G_2)$.

Apply Algorithm 4.2 to G_3 and G_4 to produce $\mathcal{P}$ and $\mathcal{Q}$ respectively, making sure that the order of nonterminals is preserved in the rule heads of $\mathcal{P}$ and $\mathcal{Q}$.

Lemma 4.32 $\mathcal{P} \subset \mathcal{Q}$ ($\mathcal{P} \equiv \mathcal{Q}$) iff $L(G_1) \subset L(G_2)$.
Proof. By Lemmas 4.30, 4.31 and 4.9 , and by construction. □

Theorem 4.3 Let $\mathcal{P}$ and $\mathcal{Q}$ be safe, single-IDB programs with a single recursive rule and nine initialisation rules. Then, the containment or equivalence of such programs is undecidable.
Proof. By Lemmas 4.32 and 4.4. □

Theorem 4.4 The sequencability of single-IDB programs is undecidable.
Proof. Let $\mathcal{P}$ and $\mathcal{Q}$ be as above. Assume that $\mathcal{P}$ has the recursive rule r_1 and the nine initialization rules $i_1 \ldots i_9$, and that $\mathcal{Q}$ consists of the recursive rule r_2 and the same nine initialisation rules. Add the atom $f(R)$ to the body of r_1 to obtain the rule r_1', and let $\mathcal{T}$ be the program consisting of $r_1', r_2, i_1, \ldots, i_9$. Now, since the "home" position for R is occupied by the persistent variable Z in every p-atom in either recursive rule, we may conclude that the recursive rule r_1' is used at most once in any top-down expansion that is not contained in the initialisation rule i_3, and that r_1' is used to expand the root in such a case[4]. Thus, the yield of S in the program represents $L(G_1) \cup L(G_2)$, but the yield of S in the sequenced program represents $L(G_2)$, and our result follows as above. □

[4]The body of the resulting expansion contains the atom $f(R)$, but this atom is irrelevant to the sequencability of $\mathcal{T}$.

Chapter 5

Concluding remarks

In this report, we investigate opportunities to optimize recursive Horn-clause programs through transformations to simpler, more efficiently evaluable programs. We focus our attention on optimizations that may be described in terms of normal forms for the proof trees generated by the program in question. We introduce the idea of subtree eliminations as a way to describe normal forms, and present a uniform approach to the development of sufficient conditions for the detection of the applicability of a normal form to the program. We then illustrate this approach on the detection of one-boundedness, basis-linearizability and sequencability, and show how the sufficient conditions that are generated may be tuned to the desired complexity. We also investigate the complexity of these three optimization problems; our investigation yields a characterization of the complexity of conjunctive query containment, and tight undecidability results for the detection of program equivalence. Our results are contained in Table 5.1. The programs considered are all Datalog. The expression $\leq i$ *reps* means that each recursive rule in the program has at most i occurrences of any EDB predicate in the body of each recursive rule.

Whether sequencability is decidable in any interesting case is an open question. However,

	Polynomial time	$\mathcal{NP}$-hard	Undecidable
One-boundedness	linear sirup, ≤ 1 reps	linear sirup, ≤ 4 reps	never
Basis-linearizability	bilinear sirup, ≤ 1 reps	bilinear sirup, ≤ 4 reps	1 nonlinear rule, 5 basis rules.
Sequencability	???	2 recursive rules (both linear), ≤ 3 reps, 1 basis rule	2 recursive rules, 9 basis rules.

Table 5.1: Complexity results.

current work that the author is undertaking in conjunction with Tomás Feder seems to indicate a positive answer to that question.

Bibliography

[1] Abiteboul, S. Boundedness is undecidable for Datalog programs with a single recursive rule. *Information Processing Letters 32*, 281-287, 1989.

[2] Aho, A. V. and J. D. Ullman. Optimal partial-match retrieval when the fields are independently specified. *ACM Transactions on Database Systems* 4:2, 168-179, 1979.

[3] Aho, A. V., Y. Sagiv and J. D. Ullman. Equivalences among relational expressions. *SIAM J. Comp. 8*, 218-246, 1979.

[4] Aho, A. V., Y. Sagiv and J. D. Ullman. Efficient optimization of a class of relational expressions. *ACM Trans. Database Systems 4*, 435-454, 1979.

[5] Bancilhon, F. Naive evaluation of recursively defined relations. In Brodie and Mylopoulos (eds.), *On Knowledge Base Management Systems - Integrating Database and AI Systems*, Springer-Verlag.

[6] Bancilhon, F., D. E. Maier, Y. Sagiv and J. D. Ullman. Magic sets and other strange ways to implement logic programs. *Proc. Fifth ACM Symposium on Principles of Database Systems*, 1-15, 1986.

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

[8] Beeri, C. and R. Ramakrishnan. On the power of magic. *Proc. Sixth ACM Symposium on Principles of Database Systems*, 269-283, 1987.

[9] Borodin, A. On relating time and space to size and depth. *SIAM J. Compul.* **6**:4, 733-744, 1977.

[10] Chandra, A.K. and P.M. Merlin. Optimal implementation of conjunctive queries in relational databases. *Proc. Ninth Annual ACM Symposium on the Theory of Computing*, 77-90, 1977.

[11] Codd, E. F. Relational completeness of data base sublanguages. In *Data Base Systems* (R. Rustin, ed.) Prentice-Hall, Englewood Cliffs, New Jersey, 65-98, 1972.

[12] Gaifman, H., H. Mairson, Y. Sagiv and M.Y. Vardi. Undecidable optimization problems for database logic programs. *Proc. Second ACM Symposium on Logic in Computer Science*, 106-115, 1987.

[13] Helm, A.R. Detecting and eliminating redundant derivations in logic knowledge bases. *Proc. First International Conference on Deductive and Object-Oriented Databases*, 1989, 247-263.

[14] Henschen, L. and S. Naqvi. On compiling queries in recursive first-order databases. *JACM* **31:1**, 47-85, 1984.

[15] Hopcroft, J.E. and J. D. Ullman. *Introduction to automata theory, languages and computation*, Addison-Wesley, 1979.

[16] Ioannidis, Y. E. *Commutativity and its role in the processing of linear recursion.* Tech. Report 804, University of Wisconsin-Madison, 1989.

[17] Ioannidis, Y. E. *Towards an algebraic theory of recursion.* Tech. Report 801, University of Wisconsin-Madison, 1989.

[18] Ioannidis, Y.E. and E. Wong. Transforming nonlinear recursion to linear recursion. *Proc. Second International Conference on Expert Database Systems*, 187-207, 1988.

[19] Johnson, D. S. and A. Klug. Optimizing conjunctive queries that contain untyped variables. *SIAM J. Comp.* **12**, 616-640, 1983.

[20] Kanellakis, P.C. Logic programming and parallel complexity. In J.Minker (ed.) *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, Los Altos, 1988.

[21] Morris, K., J. F. Naughton, Y. Saraiya, J. D. Ullman and A. Van Gelder. YAWN! (yet another window on NAIL!). *Data Engineering* **10**:4, 28-43, 1987.

[22] Morris, K., J. D. Ullman and A. Van Gelder. Design overview of the NAIL! system. *Proc. Third International Conf. on Logic Programming*, 554-568, 1986.

[23] Naqvi, S. A. and S. Tsur. *A Logic Language for Data and Knowledge Bases.* Computer Science Press, Rockville, Md., 1988.

[24] Naughton, J. F. Compiling separable recursions. *ACM SIGMOD Intl. Conf. on the Management of Data*, 312-319, 1988.

[25] Ramakrishnan, R., Y. Sagiv, J.D. Ullman and M.Y. Vardi. Proof-tree transformation theorems and their applications. *Proc. Eighth ACM Symposium on Principles of Database Systems*, 172-181, 1989.

[26] Sacca, D. and C. Zaniolo. On the implementation of a simple class of logic queries for databases. *Proc. Fifth ACM Symposium on Principles of Database Systems*, 16-23, 1986.

[27] Sacca, D. and C. Zaniolo. Magic counting methods. *Proc. ACM SIGMOD International Conference on the Management of Data*, 49-59, 1987.

[28] Sagiv, Y. Optimizing Datalog programs. *Proc. Sixth ACM Symposium on Principles of Database Systems*, 349-362, 1987.

[29] Sagiv, Y. and M. Yannakakis. Equivalence among relational expressions with the union and difference operators. *J. ACM 27*, 633-655, 1981.

[30] Saraiya, Y. Linearizing nonlinear recursions in polynomial time. *Proc. Eighth ACM Symposium on Principles of Database Systems*, 182-189, 1989.

[31] Saraiya, Y. Polynomial-time program transformations in deductive databases. *Proc. Ninth ACM Symposium on Principles of Database Systems*, 1990.

[32] Saraiya, Y. Hard problems for simple logic programs. *Proc. ACM SIGMOD International Conference on Management of Data*, 64-73, 1990.

[33] Shmueli, O. Decidability and expressiveness aspects of logic queries. *Proc. Sixth ACM Symposium on Principles of Database Systems*, 237-249, 1987.

[34] Ullman, J. D. Implementation of logical query languages for databases. *ACM Transactions on Database Systems* **10**, 289-321, 1985.

[35] Ullman, J. D. *Principles of Database and Knowledge-base Systems, Vol. I.* Computer Science Press, Rockville, Md., 1988.

[36] Ullman, J. D. *Principles of Database and Knowledge-base Systems, Vol. II.* Computer Science Press, Rockville, Md., 1989.

[37] Ullman, J. D. and A. van Gelder. Parallel complexity of logical query programs. *Proc. IEEE Symposium on Foundations of Computer Science*, 1986.

[38] van Emden, M. H. and R. A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM* **23**, 733-742, 1986.

[39] Vieille, L. Recursive axioms in deductive databases: the Query-Subquery approach. *Expert Database Systems*, 179-193, 1986.

[40] Zhang, W., C.T. Yu and D. Troy. *A necessary and sufficient condition to linearize doubly recursive programs in logic databases.* Manuscript, Dept. of EECS, University of Illinois at Chicago, 1988.

[41] Zhang, W., C.T. Yu and D. Troy. Necessary and sufficient conditions to linearize doubly recursive programs in logic databases. *ACM Transactions on Database Systems* **15**:*3*, 459-482, 1990.