On the Equivalence of Recursive and Nonrecursive Datalog Programs

Surajit Chaudhuri*
Database Tech. Department
Hewlett Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
E-mail: chaudhuri@hpl.hp.com

Moshe Y. Vardi†
Department of Computer Science
Rice University
Houston, TX 77005-1892
E-mail: vardi@cs.rice.edu
URL: http://www.cs.rice.edu/~vardi

Abstract: We study the problem of determining whether a given recursive Datalog program is equivalent to a given nonrecursive Datalog program. Since nonrecursive Datalog programs are equivalent to unions of conjunctive queries, we study also the problem of determining whether a given recursive Datalog program is contained in a union of conjunctive queries. For this problem, we prove doubly exponential upper and lower time bounds. For the equivalence problem, we prove triply exponential upper and lower time bounds.

1 Introduction

It has been recognized for some time that first-order database query languages are lacking in expressive power [AU79, GM78, Zl76]. Since then, many higher-order query languages have been investigated [AV89, CH80, Ch81, CH82, Im86, Va82]. A query language that has received considerable attention recently is *Datalog*, the language of logic programs (known also as Horn-clause programs) without function symbols [K90, Ul89], which is essentially a fragment of fixpoint logic [CH85, Mo74]. (See [Ul88] for a detailed discussion of Datalog.)

The gain in expressive power does not, however, come for free; evaluating Datalog programs is harder than evaluating first-order queries [Va82]. Recent works have addressed the problems of finding efficient evaluation methods for Datalog programs ([BR86] is a good survey on this topic) and developing optimization techniques for Datalog (see [MP91, Na89b, NRSU89]). The techniques to optimize evaluation of queries

*Work done while this author was at Stanford University and was supported by ARO grant DAAL03-91-G-0177, NSF grant IRI-87-22886, Air Force grant AFOSR-88-0266, and a grant of IBM Corp.
†Work done while this author was at the IBM Almaden Research Center.

are often based on the ability to transform a query into an equivalent one that can be evaluated more efficiently [RSUV93]. Therefore, determining equivalence of queries is one of the most fundamental optimization problems. Naturally, the problem of determining equivalence of Datalog programs has received attention. Unfortunately, Datalog program equivalence is undecidable [Shm87].

Since the source of the difficulty in evaluating Datalog programs is their recursive nature, the first line of attack in trying to optimize such programs is to eliminate the recursion. The following example is from [Na89a].

Example 1.1: Consider the following Datalog program Π_1:

$$\begin{array}{ll} buys(X,Y) & :-likes(X,Y). \\ buys(X,Y) & :-trendy(X), buys(Z,Y). \end{array}$$

It can be shown that Π_1 is equivalent to the following nonrecursive program.

$$\begin{array}{ll} buys(X,Y) & :-likes(X,Y). \\ buys(X,Y) & :-trendy(X), likes(Z,Y). \end{array}$$

Consider, on the other hand, the following Datalog program Π_2:

$$\begin{array}{ll} buys(X,Y) & :-likes(X,Y). \\ buys(X,Y) & :-knows(X,Z), buys(Z,Y). \end{array}$$

It can be shown that Π_2 is not equivalent to the following nonrecursive program:

$$\begin{array}{ll} buys(X,Y) & :-likes(X,Y). \\ buys(X,Y) & :-knows(X,Z), likes(Z,Y). \end{array}$$

In fact, Π_2 is inherently recursive, i.e., it is not equivalent to any nonrecursive program. ∎

Thus, a problem of special interest is that of determining the equivalence of a given recursive Datalog program to a given nonrecursive program, i.e., a Datalog program where the dependency graph among the predicates is acyclic.

This problem is the main focus of this paper. Note that this problem is different from that of determining whether a given recursive Datalog program is equivalent to *some* nonrecursive program. The latter problem, called the *boundedness* problem, is known to be undecidable [GMSV93] and has been studied extensively (see [KA89] for a survey and [HKMV91, HKMV95] for recent results)

A nonrecursive program can be rewritten as a union of conjunctive queries. Thus, containment of a nonrecursive program in a recursive program can be reduced to the containment of a conjunctive query in a recursive program. The latter problem was shown to be decidable; in fact it is EXPTIME-complete [CK86, CLM81, Sa88b]. Thus,

what was left open is the other direction, i.e., the problem of determining whether a recursive program is contained in a nonrecursive program. We attack this problem by investigating the containment of recursive programs in unions of conjunctive queries. Our main result is that containment of recursive programs in unions of conjunctive queries is decidable. Therefore, iIt follows that equivalence of two given programs is decidable when one is recursive and the other nonrecursive.

We first prove that the decidability of the containment problem follows from a powerful general decidability result due to Courcelle [Cou91]. Unfortunately, while Courcelle's result yields the decidability of the containment problem, it provides only nonelementary time-bounds [Cou90]. The main body of the paper is dedicated to a detailed study of the computational complexity of containment and equivalence.

For upper bounds, we use the automaton-theoretic approach advocated in [Va92]. The key idea is that a recursive program can be viewed as an infinite union of conjunctive queries. These conjunctive queries can be represented by proof trees, and the set of proof trees corresponding to a given recursive program can be represented by a tree automaton. This representation enables us to reduce containment of recursive programs in unions of conjunctive queries to containment of tree automata, which is known to be decidable in exponential time [Se90]. The size of the tree automata obtained in the reduction is exponential in the size of the input; as a result, we obtain a doubly-exponential time upper bound for containment in unions of conjunctive queries. These bounds turn out to be optimal; by a succinct encoding of alternating exponential-space Turing machines, we prove a matching doubly-exponential time lower bound. A case of special interest is that of *linear* programs, i.e., programs in which each rule contains at most one recursive subgoal [CK86, UV88]. In this case, the corresponding set of proof trees can be represented by word automata, for which containment is known to be decidable in polynomial space [MS72]. As a result, we obtain an exponential space upper bound for the containment problem for linear programs, which is also matched by a lower bound.[1]

We then note that expressing a nonrecursive program as a union of conjunctive queries may involve an exponential blow-up in size. Thus, our upper-bound technique for containment yields a triply-exponential time upper bound for containment in nonrecursive programs (doubly-exponential space upper bound for linear programs). We show that the succinctness of nonrecursive programs is inherent, by proving a matching triply-exponential time lower bound (doubly-exponential space lower bound for linear programs). Finally, we observe that these results also yield the same complexity bounds for equivalence to nonrecursive programs. Thus, while equivalence to nonrecursive programs is decidable, it is highly intractable. We note that one has to be careful in interpreting lower bounds for query containment. While containment of conjunctive queries in recursive program is complete for EXPTIME [CK86, CLM81, Sa88b], this complexity

[1] Our upper bounds follow also from van der Meyden's results on recursively indefinite databases [Mey93, Theorem 5.7].

is simply the expression complexity of evaluation Datalog programs [Va82]. In fact, if attention is restricted to programs of bounded arity, we get NP-completeness instead of EXPTIME-completeness. In contrast, our lower bounds here imply "real" intractability, and they hold even for programs of bounded arity.

2 Preliminaries

2.1 Conjunctive Queries and Datalog

A *conjunctive query* is a positive existential conjunctive first-order formula, i.e., the only propositional connective allowed is $\wedge$ and the only quantifier allowed is $\exists$. Without loss of generality, we can assume that conjunctive queries are given as formulas $\theta(x_1, \ldots, x_k)$ of the form $(\exists y_1, \ldots, y_m)(a_1 \wedge \ldots a_n)$ with free variables among $x_1, \ldots, x_k$, where the a_i's are atomic formulas of the form $p(z_1, \ldots, z_l)$ over the variables $x_1, \ldots, x_k, y_1, \ldots, y_m$. For example, the conjunctive query $(\exists y)(E(x,y) \wedge E(y,z))$ is satisfied by all pairs $\langle x, z \rangle$ such that there is a path of length 2 between x and z. The free variables are also called *distinguished variables*. We distinguish between variables and *occurrences* of variables in a conjunctive query, but we only consider occurrences of variables in the atomic formulas of the query. For example, the variables x and y have each two occurrences in $(\exists y)(E(x,y) \wedge E(y,z))$. An occurrence of a distinguished variable in a conjunctive query is called a *distinguished occurrence*. A union of conjunctive queries is a disjunction

$$\bigvee_{i=1}^{s} \theta_i(x_1, \ldots, x_k)$$

of conjunctive queries.

A union of conjunctive queries $\Theta(x_1, \ldots, x_k)$ can be applied to a database D. The result

$$\Theta(D) = \{(a_1, \ldots, a_k) | D \models \Theta(a_1, \ldots, a_k)\}$$

is the set of k-ary tuples that satisfy Θ in D. If Θ has no distinguished variables, then it is viewed as a Boolean query; the result is either the empty relation (corresponding to **false**) or the relation containing the 0-ary tuple (corresponding to **true**).

A (Datalog) program consists of a set of Horn rules. A Horn rule consists of a single atom in the head of the rule and a conjunction of atoms in the body, where an atom is a formula of the form $p(z_1, .., z_l)$ where p is a predicate symbol and $z_1..z_l$ are variables. The predicates that occur in head of rules are called *intensional* (IDB) predicates. The rest of the predicates are called *extensional* (EDB) predicates. Let Π be a Datalog program. Let $Q^i_\Pi(D)$ be the collection of facts about an IDB predicate Q that can be deduced from a database D by at most i applications of the rules in Π and let $Q^\infty_\Pi(D)$ be the collection of facts about Q that can be deduced from D by any number of applications of the rules in Π, that is,

$$Q^\infty_\Pi(D) = \bigcup_{i \geq 0} Q^i_\Pi(D).$$

We say that the program Π with goal predicate Q is *contained* in a union of conjunctive queries Θ if $Q^{\infty}_{\Pi}(D) \subseteq \Theta(D)$ for each database D. It is known (cf. [MUV84, Na89a]) that the relation defined by an IDB predicate in a Datalog program Π, i.e., $Q^{\infty}_{\Pi}(D)$, can be defined by an *infinite* union of conjunctive queries. That is, for each IDB predicate Q there is an infinite sequence $\varphi_0, \varphi_1, \ldots$ of conjunctive queries such that for every database D, we have $Q^{\infty}_{\Pi}(D) = \bigcup_{i=0}^{\infty} \varphi_i(D)$. The φ_i's are called the *expansions* of Q.

A predicate P *depends* on a predicate Q in a program Π, if Q occurs in the body of a rule r of Π and P is the predicate at the head of r. The *dependence graph* of Π is a directed graph whose nodes are the predicates of Π, and whose edges capture the dependence relation, i.e., there is an edge from Q to P if P depends on P. A program Π is *nonrecursive* if its dependence graph is acyclic, i.e., no predicate depends recursively on itself. It is well-known that a nonrecursive program has only finitely many expansions (up to renaming of variables). Thus, a nonrecursive program is equivalent to a union of conjunctive queries.

2.2 Containment of Conjunctive Queries

Let $\theta(x_1, \ldots, x_k)$ and $\psi(x_1, \ldots, x_k)$ are two conjunctive queries with the same vector of distinguished variables. We say that θ is *contained* in ψ if $\theta(D) \subseteq \psi(D)$ for each database D, i.e., if the following implication is valid

$$\forall x_1 \ldots \forall x_k(\theta(x_1, \ldots, x_k) \rightarrow \psi(x_1, \ldots, x_k))$$

Definition 2.1: A *containment mapping* from a conjunctive query ψ to a conjunctive query θ is a renaming of variables subject to the following constraints: (a) every distinguished variable must map to itself, and (b) after renaming, every literal in ψ must be among the literals of θ. ∎

Conjunctive-query containment can be characterized in terms of containment mappings (cf. [Ul89]).

Theorem 2.2: *A conjunctive query $\theta(x_1, \ldots, x_k)$ is contained in a conjunctive query $\psi(x_1, \ldots, x_k)$ iff there is a containment mapping from ψ to θ.*

It will be convenient to view a containment mapping h from ψ to θ as a mapping from occurrences of variables in ψ to occurrences to variables in θ. Such a mapping has the property that v_1 and v_2 are occurrences of the same variable in ψ, then $h(\mathsf{v}_1)$ and $h(\mathsf{v}_2)$ are occurrences of the same variable in θ.

Sagiv and Yannakakis [SY81] extended Theorem 2.2 to the case where queries are unions of conjunctive queries.

Theorem 2.3: *If $\Phi = \cup_i \varphi_i$ and $\Psi = \cup_i \psi_i$ are union of conjunctive queries, then Φ is contained in Ψ (i.e., $\Phi(D) \subseteq \Psi(D)$ for every database D) iff each φ_i is contained in some ψ_j , i.e., there is a containment mapping from ψ_j to φ_i.*

2.3 Expansion Trees

Expansions can be described in terms of *expansion trees*. The nodes of an expansion tree for a Datalog program Π are labeled by pairs of the form (α, ρ), where α is an IDB atom and ρ is an instance of a rule r of Π such that the head of ρ is α. The atom labeling a node x is denoted α_x and the rule labeling a node x is denoted ρ_x. In an expansion tree for an IDB predicate Q, the root is labeled by a Q-atom. Consider a node x, where α_x is the atom $R(\mathbf{t})$, ρ_x is the rule

$$R(\mathbf{t}) : -R_1(\mathbf{t}^1), \ldots, R_m(\mathbf{t}^m),$$

and the IDB atoms in the body of the rule are $R_{i_1}(\mathbf{t}^{i_1}), \ldots, R_{i_l}(\mathbf{t}^{i_l})$. Then x has children $x1, \ldots, xl$ labeled with the atoms $R_{i_1}(\mathbf{t}^{i_1}), \ldots, R_{i_l}(\mathbf{t}^{i_l})$. In particular, if all atoms in ρ_x are EDB atoms, then x must be a leaf. The query corresponding to an expansion tree is the conjunction of all EDB atoms in ρ_x for all nodes x in the tree, with the variables in the root atom as the free variables. Thus, we can view an expansion tree τ as a conjunctive query. Let $trees(Q, \Pi)$ denote the set of expansion trees for an IDB predicate Q in Π. (Note that $trees(Q, \Pi)$ is an infinite set.) Then for every database D, we have

$$Q_{\Pi}^{\infty}(D) = \bigcup_{\tau \in trees(Q,\Pi)} \tau(D).$$

It follows that Π is contained in a conjunctive query θ if there is a containment mapping from θ to each expansion tree τ in $trees(Q, \Pi)$, i.e., a mapping, which maps distinguished variables to distinguished variables and maps the atoms of θ to atoms in the bodies of rules labeling nodes of τ.

Of particular interest are expansion trees that are obtained by "unfolding" the program Π.

Definition 2.4: An expansion tree τ of a Datalog program Π is an *unfolding expansion tree* if it satisfies the following conditions: (a) the atom labeling the root is the head of a rule in Π, and (b) if a node x is labeled by (α_x, ρ_x), then the variables in the body of ρ_x either occur in α_x or they do not occur in the label of any node above x. ∎

Intuitively, an unfolding expansion tree is obtained by starting with a head of a rule in Π as the atom labeling the root, and then creating children by unifying an atom labeling a node with a "fresh" copy of a rule in Π. Note that if a variable v occur in the atom labelling a node x but not in the atoms labeling the children of x, then v will not occur in the label of any descendant of x.

We denote the collection of unfolding expansion trees for an IDB predicate Q in a program Π by $u_trees(Q, \Pi)$. It is easy to see that every expansion tree can be obtained by renaming variables in an unfolding expansion tree. Thus, every expansion tree, viewed as a conjunctive query, is contained in an unfolding expansion tree.

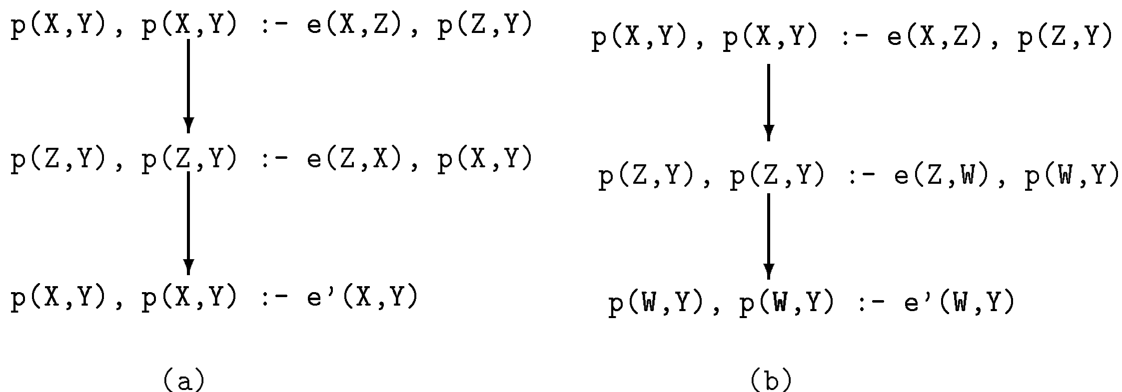

Figure 1: (a) Expansion Tree (b) Unfolding Expansion Tree

Example 2.5: Figure 1 shows expansion trees for the IDB predicate p in the following transitive closure program.

$$\begin{array}{llll} r1: & p(X,Y) & :- & e(X,Z), p(Z,Y) \\ r0: & p(X,Y) & :- & e'(X,Y) \end{array}$$

Note that the variable X is re-used in the child of the root of the expansion tree, while a new variable W is used instead of X in the child of the root of the unfolding expansion tree. ∎

The following proposition follows immediately.

Proposition 2.6: *Let Π be a program with a goal predicate Q. For every database D, we have*

$$Q_\Pi^\infty(D) = \bigcup_{\tau \in u_trees(Q,\Pi)} \tau(D).$$

3 Decidability

We can view a conjunctive query $\varphi(x_1, \ldots, x_k)$ with free variables among $x_1, \ldots, x_k$ as a 2-sorted relational structure $\mathbf{A}_\varphi$. The sorts V and F, denote the set of variables and atomic formulas in φ, respectively. For each l-ary predicate symbol P in the vocabulary of φ, we have a predicate symbol P' in the vocabulary of $\mathbf{A}_\varphi$ of type $F \times V^l$. The vocabulary of $\mathbf{A}_\varphi$ also has constant symbols $\mathbf{x}_1, \ldots, \mathbf{x}_k$. These constant and predicate symbols are interpreted in $\mathbf{A}_\varphi$ as follows. First, the constant symbol $\mathbf{x}_i$ is interpreted as x_i. Second, if the atomic formula a_i is $P(z_1, \ldots, z_l)$ in φ, then we have a tuple $\langle a_i, z_1, \ldots, z_l \rangle$ in the interpretation of P'. (Note that φ can have multiple occurrences of the same atomic formula, which explains why we need the sort F in $\mathbf{A}_\varphi$.)

Since a conjunctive query φ can be viewed as a 2-sorted relational structure $\mathbf{A}_\varphi$, we can view $u_trees(Q,\Pi)$ as a set of 2-sorted relational structures, which we denote as $str(Q,\Pi)$. If Q is k-ary, then we can assume that all conjunctive queries in $u_trees(Q,\Pi)$ have free variables among $x_1,\ldots,x_k$. Thus, all structures in $str(Q,\Pi)$ have the same vocabulary, denoted $vocab(Q,\Pi)$. We can now express properties of Datalog program in terms of properties of the associated collection of 2-sorted structures. If ψ is a 1st-order formula over $vocab(Q,\Pi)$, then we say that the program Π with goal predicate Q *satisfies* ψ if ψ holds in *all* structures in $str(Q,\Pi)$.

As an example, consider the property of *strong nonredundancy.* We say that a Datalog program Π with goal predicate Q is strongly nonredundant if no unfolding expansion tree contains two distinct occurrences of the same EDB atom. It is easy to see that this property can be expressed as a first-order property of the structures in $str(Q,\Pi)$. For simplicity assume that there is a single EDB predicate P, which happens to be k-ary. Then the desired property holds if the program Π with goal predicate Q satisfies the sentence

$$(\forall x_1,x_2\in F)(\forall y_1,\ldots,y_k\in V)(P'(x_1,y_1,\ldots,y_k) \\ \wedge P'(x_2,y_1,\ldots,y_k)\Rightarrow x_1=x_2).$$

First-order logic gives us a very powerful language to describe properties of Datalog queries in terms of the associated set of structures. It is not clear, a priori, whether such properties can be effectively tested. After all, to check whether a Datalog program Π with a goal Q satisfies a first-order sentence ψ we have to check in principle the infinitely many structures in $str(Q,\Pi)$. The following powerful result by Courcelle asserts that, nevertheless, first-order properties of Datalog programs can be effectively tested.[2]

Theorem 3.1: [Cou90, Cou91] *There is an algorithm to decide, given a Datalog program* Π *with goal predicate* Q *and a first-order sentence* ψ *over* $vocab(Q,\Pi)$*, whether* Π *satisfies* ψ*.*

The decidability of containment in nonrecursive programs follows now from Theorem 3.1.

Theorem 3.2: *Containment of recursive Datalog programs in nonrecursive Datalog programs is decidable.*

Proof: Let us assume that Π is a recursive Datalog program with the goal predicate Q. Let Θ be an arbitrary nonrecursive program. Assume that Θ has already been rewritten as a finite union

$$\bigvee_{i=1}^{s}\varphi_i(x_1,\ldots,x_k)$$

[2]Courcelle's result applies also to monadic second-order logic, which is a powerful extension of first-order logic.

of conjunctive queries. Let $\varphi_i(x_1,\ldots,x_k)$ be $(\exists y_1,\ldots,y_m)(a_1 \wedge \ldots \wedge a_n)$ with free variables among $x_1,\ldots,x_k$, where a_i is an atomic formula $p_i(z_1,\ldots,z_l)$ over the variables $x_1,\ldots,x_k,y_1,\ldots,y_m$. Define φ'_i to be the sentence $(\exists y_1,\ldots,y_m \in V)(\exists a_1,\ldots,a_n \in F)(a'_1 \wedge \ldots a'_n)$, where a'_i is the atomic formula $p'_i(a_i,z'_1,\ldots,z'_l)$, and $z'_1,\ldots,z'_l$ are obtained from $z_1,\ldots,z_l$ by substituting $\mathbf{x}_j$ for x_j.

We claim that Π is contained in Θ iff Π satisfies Θ', where Θ' is $\bigvee_{i=1}^{s} \varphi'_i$. Assume that Π is contained in Θ. Then, from Theorem 2.3, it follows that for every expansion $\tau(x_1,\ldots,x_k) \in u_trees(Q,\Pi)$, there exists some $\varphi_i = (\exists y_1,\ldots,y_m)(a_1 \wedge \ldots \wedge a_n)$ such that there is a containment mapping from φ_i to τ. Let $\tau(x_1,\ldots,x_k)$ be $\exists z_1,\ldots,z_r(b_1 \wedge \ldots \wedge b_s)$. Let the corresponding tuples in $\mathbf{A}_\tau$ be $b'_1,\ldots,b'_s$. Consider any a_q where $1 \leq q \leq n$. Since there is a containment mapping from φ_i to τ, it follows that a_q maps to some b_j where the distinguished variables are preserved. Therefore, there is a substitution for the variables in a'_q such that it corresponds to the literal b'_j. Therefore, φ'_i holds over $\mathbf{A}_\tau$. Thus, Π satisfies Θ'.

Let us now assume that Π satisfies Θ'. Let τ be an expansion of Π that corresponds to an unfolding expansion tree, and let $\mathbf{A}_\tau$ be the corresponding structure. Since Π satisfies Θ', it follows that some φ'_i must hold over $\mathbf{A}_\tau$. Therefore, there is an assignment of variables, such that a literal a'_q of φ'_i corresponds to a tuple b'_j. Moreover, such a mapping ensures that the distinguished variables map to themselves. Thus, τ is contained in Θ. This completes our proof. ∎

Corollary 3.3: *Equivalence of Datalog programs to nonrecursive programs is decidable.*

Unfortunately, Theorem 3.1 yields a very high upper bound; the algorithm described in [Cou90] is of nonelementary time complexity, i.e., its time complexity cannot be bounded by any finite stack of exponentials. There is, however, a possible way around this difficulty. While Courcelle's algorithm for arbitrary first-order properties of Datalog programs has a nonelementary time complexity, more efficient algorithms may exist for specific properties. The crux of Courcelle's result is the well known connection between monadic second-order logic and tree automata (cf. [TW68, Ra69]). It is conceivable that by using automata-theoretic techniques directly we might be able to obtain more feasible algorithms for the equivalence problem. A similar strategy of using automata theory directly rather than monadic second-order logic was demonstrated successfully for decision problems in the area of program verification (cf. [VW86, EJ88]).

4 Automata on Words and Trees

In this section, we review some of the relevant results from automata theory on emptiness and containment of automata. We will use these results for proving the upper-bound on the complexity of deciding containment of a Datalog predicate in a union of conjunctive queries. The material in this section is quoted from [Va92].

4.1 Automata on Words

An automaton A is a tuple $(\Sigma, S, S_0, \delta, F)$, where Σ is a finite *alphabet*, S is a finite set of *states*, $S_0 \subseteq S$ is the set of *initial* states, $F \subseteq S$ is the set of *accepting* states, and $\delta : S \times \Sigma \rightarrow 2^S$ is a *transition function*. Note that the automaton is *nondeterministic*, since it may have many initial states and the transition function may specify many possible transitions for each state and letter.

A *run* r of A over a *word* $w = a_0, \ldots, a_{n-1} \in \Sigma^n$ is a sequence $s_0, \ldots, s_n \in S^{n+1}$ such that

- $s_0 \in S_0$,
- $s_{i+1} \in \delta(s_i, a_i)$ for $0 \leq i < n$.

The run r is *accepting* if $s_n \in F$. The word w is accepted by A if A has an accepting run over w. The *language* of A, denoted $L(A)$, is the set of words accepted by A.

An important property of automata is their closure under Boolean operations.

Proposition 4.1: [RS59] *Let A_1, A_2 be a automata over an alphabet Σ. Then there are automata A_3, A_4, and A_5 such that $L(A_3) = \Sigma^* - L(A_1)$, $L(A_4) = L(A_1) \cap L(A_2)$, and $L(A_5) = L(A_1) \cup L(A_2)$.*

The constructions for union and intersection involve only a polynomial blowup in the size of the automata. In contrast, complementation may involve an exponential blow-up in the size of the automaton [MF71].

The *nonemptiness problem* for automata is to decide, given an automaton A, whether $L(A)$ is nonempty.

Proposition 4.2: [Jo75, RS59] *The nonemptiness problem for automata is decidable in nondeterministic logarithmic space.*

Proof: Let $A = (\Sigma, S, S_0, \delta, F)$ be the given automaton. Let s, t be states of S. Say that s is *directly connected* to t if there is a letter $a \in \Sigma$ such that $t \in \delta(s, a)$. Say that s is connected to t if there is a sequence $s_1, \ldots, s_m$, $m \geq 1$, of states such that $s_1 = s$, $s_n = t$, and s_i is directly connected to s_{i+1} for $1 \leq i < m$. It is easy to see that $L(A)$ is nonempty iff there are states $s \in S_0$ and $t \in F$ such that s is connected to t. Thus, automata nonemptiness is equivalent to *graph reachability*, which can be tested in nondeterministic logarithmic space. ∎

A problem related to nonemptiness is the *containment problem*, which is to decide, given automata A_1 and A_2, whether $L(A_1) \subseteq L(A_2)$. Note that $L(A_1) \subseteq L(A_2)$ iff $L(A_1) \cap \overline{L(A_2)} = \emptyset$. Thus, by Proposition 4.1, the containment problem is reducible to the nonemptiness problem, though the reduction may be computationally expensive.

Proposition 4.3: [MS72] *The containment problem for automata is PSPACE-complete.*

4.2 Automata on Trees

Let N denote the set of positive integers. The variables x and y denote elements of N^*. A *tree* τ is a finite subset of N^*, such that if $xi \in \tau$, where $x \in N^*$ and $i \in N$, then also $x \in \tau$ and and if $i > 1$ then also $x(i-1) \in \tau$. The elements of τ are called *nodes*. If x and xi are nodes of τ, then x is the *parent* of xi and xi is the *child* of x. The node x is a *leaf* if it has no children. By definition, the empty sequence ϵ is a member of every tree; it is called the *root*.

A Σ-*labeled tree*, for a finite alphabet Σ, is a pair (τ, π), where τ is a tree and $\pi : \tau \rightarrow \Sigma$ assigns to every node a label. *Labeled trees* are often referred to as *trees*; the intention will be clear from the context. The set of Σ-labeled trees is denoted $trees(\Sigma)$.

A *tree automaton* A is a tuple $(\Sigma, S, S_0, \delta, F)$, where Σ is a finite alphabet, S is a finite set of states, $S_0 \subseteq S$ is a set of initial states, $F \subseteq S$ is a set of accepting states, and $\delta : S \times \Sigma \rightarrow 2^{S^*}$ is a transition function such that $\delta(s, a)$ is finite for all $s \in S$ and $a \in \Sigma$. A *run* $r : \tau \rightarrow S$ of A on a Σ-labeled tree (τ, π) is a labeling of τ by states of A, such that the root is labeled by an initial state and the transitions obey the transition function δ; that is, $r(\epsilon) \in S_0$, and if x is not a leaf and x has k children, then $\langle r(x1), \ldots, r(xk) \rangle \in \delta(r(x), \pi(x))$. If for every leaf x of τ there is a tuple $\langle s_1, \ldots, s_l \rangle \in \delta(r(x), \pi(x))$ such that $\{s_1, \ldots, s_l\} \subseteq F$, then r is *accepting*. A *accepts* (τ, π) if it has an accepting run on (τ, π). The *tree language* of A, denoted $T(A)$, is the set of trees accepted by A.

An important property of tree automata is their closure under Boolean operations.

Proposition 4.4: [Cos72] *Let A_1, A_2 be a automata over an alphabet Σ. Then there are automata A_3, A_4, and A_5 such that $L(A_3) = \Sigma^* - L(A_1)$, $L(A_4) = L(A_1) \cap L(A_2)$, and $L(A_5) = L(A_1) \cup L(A_2)$.*

As in word automata, the constructions for union and intersection involve only a polynomial blowup in the size of the automata, while complementation may involve an exponential blow-up in the size of the automaton.

The *nonemptiness problem* for tree automata is to decide, given a tree automaton A, whether $T(A)$ is nonempty.

Proposition 4.5: [Do70, TW68] *The nonemptiness problem for tree automata is decidable in polynomial time.*

Proof: Let $A = (\Sigma, S, S_0, \delta, F)$ be the given tree automaton. Let $accept(A)$ be the minimal set of states in S such that

- $F \subseteq accept(A)$, and
- if s is a state such that there are a letter $a \in \Sigma$ and a transition $\langle s_1, \ldots, s_k \rangle \in \delta(s, a) \cap accept(A)^*$, then $s \in accept(A)$.

It is easy to see that $T(A)$ is nonempty iff $S_0 \cap accept(A) \neq \emptyset$. Intuitively, $accept(A)$ is the set of all states that label the roots of accepting runs. Thus, $T(A)$ is nonempty precisely when some initial state is in $accept(A)$. The claim follows, since $accept(A)$ can be computed bottom-up in polynomial time. ▮

We note that using techinques such as in [Be80], the nonemptiness problem for tree automata is decidable in linear time.

A problem related to nonemptiness is the *containment problem*, which is to decide, given tree automata A_1 and A_2, whether $T(A_1) \subseteq T(A_2)$. As for word automata, the containment problem is reducible to the nonemptiness problem, though the reduction may be computationally expensive.

Proposition 4.6: [Se90] *The containment problem for tree automata is EXPTIME-complete.*

5 Containment in Union of Conjunctive Queries

5.1 Proof Trees

The basic idea behind *proof trees* is to describe expansion trees using a finite number of labels. We bound the number of labels by bounding the set of variables that can occur in labels of nodes in the tree. If r is a rule of a Datalog program Π, then let $var_num(r)$ be the number of variables occurring in IDB atoms in r (head or body). Let $var_num(\Pi)$ be twice the maximum of $var_num(r)$ for all rules r in Π. Let $var(\Pi)$ be the set $\{x_1, \ldots, x_{var_num(\Pi)}\}$. A proof tree for Π is simply an expansion tree for Π all of whose variables are from $var(\Pi)$. We denote the set of proof trees for a predicate Q of a program Π by $p_trees(Q, \Pi)$.

The intuition behind proof tree is that variables are re-used. In an unfolding expansion tree, when we "unfold" a node x we take a "fresh" copy of a rule r in Π. In a proof tree, we take instead an instance of r over $var(\Pi)$. Since the number of variables in $var(\Pi)$ is twice the number of variables in any rule of Π, we can instantiate the variables in the body of r by variables different from those in the goal α_x.

Example 5.1: Figure 2 describes an unfolding expansion tree and a proof tree for the IDB predicate p in the transitive-closure program of Example 2.5. In the proof tree, instead of using a new variable W, we re-use the variable X. ▮

A proof tree represents an expansion tree where variables are re-used. In other words, the same variable is used to represent a set of distinct variables in the expansion tree. Intuitively, to reconstruct an expansion tree for a given proof tree, we need to distinguish among occurrences of variables.

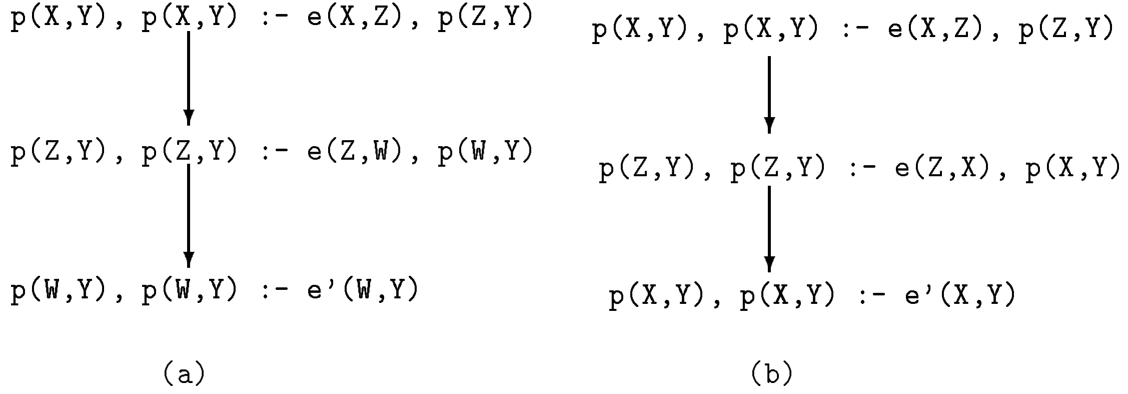

Figure 2: (a) Unfolding Expansion Tree (b) Proof Tree

Definition 5.2: Let x_1 and x_2 be nodes in a proof tree τ, with a lowest common ancestor x, and let v_1 and v_2 be occurrences, in x_1 and x_2, respectively, of a variable v. We say that v_1 and v_2 are *connected* in τ if the goal of every node, except perhaps for x, on the simple path connecting x_1 and x_2 has an occurrence of v. We say that an occurrence v of a variable v in τ is a *distinguished occurrence* if it is connected to an occurrence of v in the atom labeling the root of τ. ∎

From the definition above, it follows that connectedness is an equivalence relation and it partitions the occurrences of variables in the proof tree. We denote the equivalence class of an occurrence v of a variable v in a proof tree τ by $[\mathsf{v}]_\tau$. We will omit τ when it is clear from the context.

Example 5.3: Consider the proof tree in Figure 2. The occurrences of the variable Y in the root and in the interior node are connected. Both occurrences of Y are distinguished. The occurrences of the variable X in the root and in the leaf are not connected. The occurrence of X in the root is distinguished, but the occurrence of X in the leaf is not distinguished. ∎

Every proof tree corresponds to an expansion tree and hence to an expansion. We want to define containment mappings from conjunctive queries to proof trees such that there is a containment mapping from a conjunctive query to a proof tree iff there is a containment mapping from the conjunctive query to the expansion corresponding to the proof tree. The definition should force a variable in the conjunctive query to map to a unique variable in the expansion corresponding to the proof tree.

Definition 5.4: A *strong* containment mapping from a conjunctive query θ to a proof tree τ is a containment mapping h from θ to τ with the following properties:

- h maps distinguished occurrences in θ to distinguished occurrences in τ, and
- if v_1 and v_2 are two occurrences of a variable v in θ, then the occurrences $h(\mathsf{v}_1)$ and $h(\mathsf{v}_2)$ in τ are connected.

∎

We now relate containment of programs and strong containment mappings.

Proposition 5.5: *Let θ be a conjunctive query and and let Π be a program with goal predicate Q. If Π is contained in θ, then, for every proof tree $\tau \in p_tree(Q, \Pi)$, there is a strong containment mapping from θ to τ.*

Proof: Assume that Π is contained in θ, and let $\tau \in p_tree(Q, \Pi)$. Rename every occurrence v of a variable v in τ by an occurrence of a new variable $v_{[\mathsf{v}]}$, i.e., connected occurrences v_1 and v_2 of a variable v are replaced by occurrences v'_1 and v'_2 of $v_{[\mathsf{v}_1]}$ (note that $[\mathsf{v}_1] = [\mathsf{v}_2]$). Denote this renaming, which is a mapping on occurrences (not on variables), by Δ. It is easy to prove that the result of this renaming is an expansion tree; call it τ'. Since Π is contained in θ, there is a containment mapping h' from θ to τ'.

We now define a containment mapping h from θ to τ as follows. Let u be an occurrence of a variable u in θ, and suppose that $h'(\mathsf{u})$ is $\Delta(\mathsf{v})$, where v is an occurrence of the variable v in τ, i.e., $h'(\mathsf{u})$ is an occurrence of $v_{[\mathsf{v}]}$ in τ'. Define $h(\mathsf{u}) = \mathsf{v}$.

We have to show that h is also a mapping on variables. Consider now two occurrences u_1 and u_2 of a variable u in θ. Then $h'(\mathsf{u}_1)$ and $h'(\mathsf{u}_2)$ are occurrences of some variables $v'_{[\mathsf{v}']}$ and $v''_{[\mathsf{v}'']}$ in τ', where v' and v'' are occurrences in τ. But $v'_{[\mathsf{v}']}$ and $v''_{[\mathsf{v}'']}$ must be the same variable, since h' is a containment mapping from θ to τ', so $v'_{[\mathsf{v}']}$ and $v''_{[\mathsf{v}'']}$ coincide, as well as $[\mathsf{v}']$ and $[\mathsf{v}'']$. It follows that $h(\mathsf{u}_1)$ and $h(\mathsf{u}_2)$ are connected occurrences of the same variable. A similar argument shows that h maps distinguished occurrences in θ to distinguished occurrences in τ. It follows that h is a strong containment mapping from θ to τ. ∎

Proposition 5.6: *Let θ be a conjunctive query and and let Π be a program with goal predicate Q. If, for every proof tree $\tau \in p_tree(Q, \Pi)$, there is a strong containment mapping from θ to τ, then Π is contained in θ.*

Proof: Assume that, for every proof tree $\tau' \in p_tree(Q, \Pi)$, there is a strong containment mapping from θ to τ'. Let τ be an unfolding expansion tree of Π. We obtain a proof tree τ' from τ by renaming of variables in a top-down fashion. Let x be a node of τ that was not yet relabeled and that is labeled in τ by (α_x, ρ_x). The variables in the body of ρ_x either occur in α_x or they do not occur in the label of any node above x. We rename the variables in the body of ρ_x that do not occur in α_x by variables from $var(\Pi)$ that do not occur in α_x; distinct variables in the body of ρ_x are renamed by distinct variables of

$var(\Pi)$. This renaming can be done, since the number of variables in $var(\Pi)$ is at least twice the number of variables in any rule of Π. Denote this renaming, which is a mapping, by Δ. Note that the distinguished variables of τ' are not renamed by this process. It is also easy to verify that if $\mathsf{v}_1 = \Delta(\mathsf{u}_1)$ and $\mathsf{v}_2 = \Delta(\mathsf{u}_2)$ are connected occurrences in τ', then u_1 and u_2 must be occurrences of the same variable in τ.

Since τ' is a proof tree, by assumption, there is a strong containment mapping h' from θ to τ'. We use h' to define a containment mapping h from θ to τ. We define h on occurrences of variables. Let u, v, and w be occurrences of variables u, v, and w in τ, τ', and θ, respectively. If $h'(\mathsf{w}) = \mathsf{v}$ and $\mathsf{v} = \Delta(\mathsf{u})$, then we take $h(\mathsf{w}) = \mathsf{u}$. We claim that h can also be viewed as a mapping on variables, i.e., $h(w) = u$. Indeed, suppose that w_1 and w_2 are both occurrences of w in θ. Since h' is a strong containment mapping, $h'(\mathsf{w}_1)$ and $h'(\mathsf{w}_2)$ must be connected in τ'. But then, as observed above, there are occurrences u_1 and u_2 of u in τ such that $h'(\mathsf{w}_1) = \Delta(\mathsf{u}_1)$ and $h'(\mathsf{w}_2) = \Delta(\mathsf{u}_2)$. A similar argument shows that h maps distinguished variables in θ to distinguished variables in τ. Thus, h is indeed a containment mapping. ∎

The propositions above yield the following characterization of containment.

Corollary 5.7: *Let Π be a program with goal predicate Q, and let θ be a conjunctive query. Then Π is contained in θ if and only if there are strong containment mappings from θ to all proof trees in $p_trees(Q, \Pi)$.*

Since we are also interested in containment in union of conjunctive queries, we need the following characterization

Theorem 5.8: *Let Π be a program with goal predicate Q, and let $\Theta = \cup_i \theta_i$ be a union of conjunctive queries. Then Π is contained in Θ if and only if for every proof tree $\tau \in p_trees(Q, \Pi)$ there is a strong containment mappings from some θ_i to τ.*

Proof: Theorem 2.3 tells us that if $\Phi = \cup_i \varphi_i$ and $\Psi = \cup_i \psi_i$ are union of conjunctive queries, then Φ is contained in Ψ (i.e., $\Phi(D) \subseteq \Psi(D)$ for every database D) iff each φ_i is contained in some ψ_j. It follows that Π is contained in Θ iff each expansion tree (resp. unfolding expansion tree) is contained in some θ_i. The claim now follows by repeating the arguments in the proofs of Propositions 5.5 and 5.6. ∎

We will use the characterization above to obtain optimal upper bound for containment of programs in conjunctive queries.

5.2 Upper Bounds

The main feature of proof trees, as opposed to expansion trees, is the fact that the numbers of possible labels is finite; it is actually exponential in the size of Π. Because the set of labels is finite, the set of proof trees $p_trees(Q, \Pi)$, for an IDB predicate Q in a program Π, can be described by a tree automaton.

Proposition 5.9: *Let* Π *be a Datalog program with a goal predicate* Q*. Then there is an automaton* $A^{p_trees}_{Q,\Pi}$*, whose size is exponential in the size of* Π*, such that* $T(A^{p_trees}_{Q,\Pi}) = p_trees(Q,\Pi)$*.*

Proof: We describe the construction of the automaton

$$A^{p_trees}_{Q,\Pi} = (\Sigma, \mathcal{I} \cup \{accept\}, \mathcal{I}_Q, \delta, \{accept\})$$

The state set $\mathcal{I}$ is the set of all IDB atoms with variables among $var(\Pi)$. The start-state set is the set of all atoms $Q(\mathbf{s})$, where the variables of $\mathbf{s}$ are in $var(\Pi)$. The alphabet $\Sigma = \mathcal{I} \times \mathcal{R}$ where $\mathcal{R}$ is the set of instances of rules of Π over $var(\Pi)$. The transition function δ is constructed as follows:

- Let ρ be a rule instance

  $$R(\mathbf{t}) : -R_1(\mathbf{t}^1), \ldots, R_m(\mathbf{t}^m),$$

  in $\mathcal{R}$, where the IDB atoms in the body of the rule are $R_{i_1}(\mathbf{t}^{i_1}), \ldots, R_{i_l}(\mathbf{t}^{i_l})$. Then $\langle R_{i_1}(\mathbf{t}^{i_1}), \ldots, R_{i_l}(\mathbf{t}^{i_l})\rangle \in \delta(R(\mathbf{t}), (R(\mathbf{t}), \rho))$.

- Let ρ be a rule instance

  $$R(\mathbf{t}) : -R_1(\mathbf{t}^1), \ldots, R_m(\mathbf{t}^m),$$

  in $\mathcal{R}$, where all atoms in the body of the rule are EDB atoms. Then $\langle accept\rangle \in \delta(R(\mathbf{t}), (R(\mathbf{t}), \rho))$.

It follows that $T(A^{p_trees}_{Q,\Pi}) = p_trees(Q,\Pi)$. It is easy to see that the number of states and transitions in the automaton is exponential in the size of Π. ∎

We now show that strong containment of proof trees in a conjunctive query can be checked by tree automata as well.

Proposition 5.10: *Let* Π *be a Datalog program* Π *with goal predicate* Q*, and let* θ *be a conjunctive query. Then there is an automaton* $A^{\theta}_{Q,\Pi}$*, whose size is exponential in the size of* Π *and* θ*, such that* $T(A^{\theta}_{Q,\Pi})$ *is the set of proof trees* τ *in* $p_trees(Q,\Pi)$ *where there is a strong containment mapping from* θ *to* τ*.*

Proof: We describe the construction of $A^{\theta}_{Q,\Pi}$, and then prove its correctness.

We view θ as a set of atoms. Every state of the automaton includes a subset of atoms of θ that have not yet been strongly mapped to τ. Such unmapped atoms may share variables with atoms that have already been mapped. Therefore, also included in the state description is a partial mapping that indicates the images of the mapped variables. A transition on an input symbol (α, ρ) results in mapping of zero or more unmapped

atoms to the body of ρ. The remainder of the unmapped atoms are partitioned among the sequence of states prescribed by the transition.

The automaton $A^{\theta}_{Q,\Pi}$ is $(\Sigma, S \cup \{accept\}, S_Q, \delta, \{accept\})$. The sets $\mathcal{I}$ and $\Sigma = \mathcal{I} \times \mathcal{R}$ are as in the proof of Proposition 5.9. We assume that the conjunctive query θ has a set of variables V_θ. The state set S is the set $\mathcal{I} \times 2^{\theta} \times 2^{V_\theta \times var(\Pi)}$. The second component in S represents the collection of subsets (of atoms) of θ and the final component contains the set of partial mappings from V_θ to $var(\Pi)$. The start-state set S_Q consists of all triples $(Q(\mathbf{s}), \theta, M_{\theta,\mathbf{s}})$, where the variable of $\mathbf{s}$ are in $var(\Pi)$ and $M_{\theta,\mathbf{s}}$ is a mapping of the distinguished variables of θ into the variables of $\mathbf{s}$. The transition function is constructed as follows:

- Let ρ be a rule instance

  $$R(\mathbf{t}) : -R_1(\mathbf{t}^1), \ldots, R_m(\mathbf{t}^m),$$

  in $\mathcal{R}$, where the IDB atoms in the body of the rule are $R_{i_1}(\mathbf{t}^{i_1}), \ldots, R_{i_l}(\mathbf{t}^{i_l})$. Then

  $$\langle (R_{i_1}(\mathbf{t}^{i_1}), \beta_1, M') \ldots, (R_{i_l}(\mathbf{t}^{i_l}), \beta_l, M') \rangle \in \delta((R(\mathbf{t}), \beta, M), (R(\mathbf{t}), \rho))$$

  if the following hold:

  1. β can be partitioned into $\beta', \beta_1, \ldots, \beta_l$, where β' is mapped to atoms in the body of ρ by a mapping $M_{\beta'}$ that is consistent with M,
  2. M' is a partial mapping that extends M and is consistent with $M_{\beta'}$.
  3. β_j and β_k can share a variable only if this variable is in the domain of M' and its image is in both $\mathbf{t}^{i_j}$ and $\mathbf{t}^{i_k}$.
  4. If a variable occurs in β_j and it is in the domain of M', then its image is in $\mathbf{t}^{i_j}$.

- Let ρ be a rule instance

  $$R(\mathbf{t}) : -R_1(\mathbf{t}^1), \ldots, R_m(\mathbf{t}^m),$$

  in $\mathcal{R}$, where all atoms in the body of the rule are EDB atoms. Then $\langle accept \rangle \in \delta((R(\mathbf{t}), \beta, M), (R(\mathbf{t}), \rho))$ if there is a mapping that extends M and maps all literals in β to atoms in the body of ρ.

It is easy to see that the number of states and transition in the automaton is exponential in the size of Π and θ. We now show the correctness of our construction. First, we show that if there is a strong containment mapping h from θ to τ, then τ is accepted by $A^{\theta}_{Q,\Pi}$. We prove acceptance by showing the existence of an accepting run r.

We show that our definition of r satisfies the inductive property that if $R(\mathbf{t})$ is the goal labeling a node x, then $r(x) = (R(\mathbf{t}), \beta, M)$, where M is consistent with h, and h maps β to atoms in bodies of rules labeling x or nodes below x.

The run starts with $r(\epsilon) = (Q(\mathbf{s}), \theta, M_{\theta,\mathbf{s}})$, where $Q(\mathbf{s})$ be the atom labeling the root of τ and $M_{\theta,\mathbf{s}}$ is the restriction of h to the distinguished variables of θ; the range of $M_{\theta,\mathbf{s}}$ are the variables of $\mathbf{s}$. Since h is a strong containment mapping from θ to τ, it follows that all literals in θ are mapped in τ. Thus, the specification of the root of τ satisfies the inductive property.

Suppose now that x is not a leaf node and has l children. Assume that $\pi(x) = (R(\mathbf{t}), \rho)$. We know that ρ must be an instance of a recursive rule in $\mathcal{R}$:

$$R(\mathbf{t}) : -R_1(\mathbf{t}^1), \ldots, R_m(\mathbf{t}^m),$$

where the IDB atoms in the body of the rule are $R_{i_1}(\mathbf{t}^{i_1}), \ldots, R_{i_l}(\mathbf{t}^{i_l})$. Thus, x has l children, labeled by the IDB atoms in the body of ρ. By inductive hypothesis, let $r(x) = (R(\mathbf{t}), \beta, M)$ where h maps β to atoms in rules labeling nodes below x. We can partition β into subsets $\beta', \beta_1, \ldots, \beta_l$, where β' is mapped by h to atoms in the body of ρ, and β_j is mapped by h to atoms in bodies labeling the node xj or nodes below xj. We obtain M' from M by adding to M the pairs consisting of variables in β' and their corresponding images in h. Also, suppose that β_j and β_k share a variable. Since h is strong, it must maps the occurrences of this variable in β_j and β_k to occurrences in $\mathbf{t}^{i_j}$ and $\mathbf{t}^{i_k}$. In that case, we add the pair consisting of this variable and its image (in h) to M'. We now define $r(xj) = (R_{i_j}(\mathbf{t}_{i_j}), \beta_j, M')$. Note that our construction ensures that $r(\epsilon) \in S_0$ and if x is an internal node with children $x1, .., xl$, then $\langle r(x1), .., r(xl) \rangle \in \delta(r(x), \pi(x))$.

Finally, if x is a leaf-node, then $\pi(x) = (R(\mathbf{t}), \rho)$, where ρ is instance of a nonrecursive rule. Our inductive property ensures that $r(x) = (R(\mathbf{t}), \beta, M)$, where all literals in β map to literals in ρ that is consistent with h. Therefore, from the description of the automaton, it follows that $accept \in \delta(r(x), \pi(x))$. Thus, r is an accepting run.

Second, we show that if τ is accepted by $A^{\theta}_{Q,\Pi}$, then there is a strong containment mapping h from θ to τ. Let r be an accepting run. The proof is by bottom-up induction on the tree. The inductive hypothesis is that if $r(x) = (R(\mathbf{t}), \beta, M)$, then $R(\mathbf{t})$ is the goal labeling x, and there is a mapping h_x that is consistent with M and maps β to atoms in bodies of rules labeling x or nodes below x. Furthermore, h_x is a strong mapping; it maps occurrences of the same variable in β to connected occurrences in τ. Suppose first that x is a leaf. Since r is an accepting run, x is labeled by $(R(\mathbf{t}), \rho)$, where ρ is a rule instance

$$R(\mathbf{t}) : -R_1(\mathbf{t}^1), \ldots, R_m(\mathbf{t}^m),$$

in $\mathcal{R}$ and all atoms in the body of the rule are EDB atoms, and there is a mapping h_x that extends M and maps all literals in β to atoms in the body of ρ. Thus, the inductive hypothesis holds for leaves.

Suppose now that x is not a leaf, and let $x1, \ldots, xl$ be the children of x. Then x is labeled by $(R(\mathbf{t}), \rho)$, where ρ is a rule instance

$$R(\mathbf{t}) : -R_1(\mathbf{t}^1), \ldots, R_m(\mathbf{t}^m),$$

in $\mathcal{R}$, and the IDB atoms in the body of the rule are $R_{i_1}(\mathbf{t}^{i_1}), \ldots, R_{i_l}(\mathbf{t}^{i_l})$. Thus, we have $r(xj) = (R(\mathbf{t}^{i_j}), \beta_j, M')$, $1 \leq j \leq l$, where $\beta', \beta_1, \ldots, \beta_l$ is a partition of β that satisfies the conditions in the definition of δ. By the inductive hypothesis, for $j = 1, \ldots, l$, and there is a mapping h_j that is consistent with M' and maps β_j to atoms in bodies of rules labeling xj or nodes below xj. The definition of δ guarantees that the h_j's are consistent with each other and that there is a mapping $M_{\beta'}$ that maps β' to atoms in the body of ρ, and M' is an extension of $M_{\beta'}$. Thus, the union of the h_j's with $M_{\beta'}$ is a partial mapping h_x that is consistent with M and maps β to atoms in the bodies of rule labeling x or nodes below x. Furthermore, the definition of δ guarantees that h_x is strong.

Now let $r(\epsilon) = (Q(\mathbf{s}), \theta, M_{\theta,\mathbf{s}})$. By the induction hypothesis $Q(\mathbf{s})$ is the goal labeling the root of τ, and there is a strong mapping h that extends $M_{\theta,\mathbf{s}}$ and maps β to atoms in bodies of rules labeling nodes in τ. Thus, h is a strong containment mapping from θ to τ. ▮

We can now reduce the containment problem for Datalog programs in unions of conjunctive queries to an automata-theoretic problem.

Theorem 5.11: *Let Π be a program with goal predicate Q, and let $\Theta = \cup_i \theta_i$ be a union of conjunctive queries. Then Π is contained in Θ if and only if*

$$T(A_{Q,\Pi}^{p_trees}) \subseteq \bigcup_i T(A_{Q,\Pi}^{\theta_i}).$$

Proof: By Theorem 5.8, Π is contained in Θ if and only if for every proof tree $\tau \in p_trees(Q, \Pi)$ there is a strong containment mappings from some θ_i to τ. By Propositions 5.9 and 5.10, the latter condition is equivalent to

$$T(A_{Q,\Pi}^{p_trees}) \subseteq \bigcup_i T(A_{Q,\Pi}^{\theta_i}).$$

▮

Theorem 5.12: *Containment of a recursive Datalog program in a union of conjunctive queries is in 2EXPTIME (EXPSPACE for linear programs).*

Proof: By Propositions 4.1 and 4.4, we can obtain an automaton $A_{Q,\Pi}^{\Theta}$, whose size is exponential in the size of Π and Θ, such that

$$T(A_{Q,\Pi}^{\Theta}) = \bigcup_i T(A_{Q,\Pi}^{\theta_i}).$$

Thus, by Theorem 5.11, containment in a union of conjunctive queries can be reduced to containment of tree (resp. word) automata of exponential size. Since containment of tree automata can be decided in exponential time (Proposition 4.6), and containment of word automata can be decided in polynomial space (Proposition 4.3), the result follows. ▮

Remark 5.13: The automata-theoretic technique used here is closely related to the automata-theoretic techniques used in [CGKV88] to prove the decidability of boundedness of monadic programs. The result here, however, is more robust since it applies to programs of arbitrary arity. In contrast, boundedness is undecidable for binary programs [Va88, HKMV95]. ∎

Remark 5.14: So far, we have assumed that neither the recursive program nor the union of conjunctive queries contain constants. However, this restriction is easily relaxed by redefining the containment mapping (Defintion 2.1). The proof of Theorem 5.12 then extends in a straight-forward fashion. In the presence of constants, a *containment mapping* from a conjunctive query ψ to a conjunctive query θ is a renaming of variables subject to the following constraints: (a) every distinguished variable must map to itself, and (b) every nondistinguished variable must map to either a variable or a constant in θ and (c) after renaming, every literal in ψ must be among the literals of θ. ∎

5.3 Lower Bounds

Theorem 5.12 provides a doubly exponential time (resp., exponential space) upper bound for containment of (resp. linear) Datalog programs in a union of conjunctive queries. We now show that these bounds are optimal. We accomplish this via a succinct encoding of alternating (resp., deterministic) exponential-space Turing machines. It is known that alternating exponential-space machines have the same computational power as doubly-exponential-time Turing machines [CKS81]; thus, deciding if an alternating Turing machine accepts the empty tape using space 2^n is complete for doubly exponential time.

We focus first on linear programs and exponential-space Turing machines. A configuration of an exponential-space Turing machine M can be described by a string of length 2^n. The symbols of the string are either symbols of the input alphabet or *composite* symbols. A composite symbol is a pair (s, a), where s is a state of M and a is an input symbol. Such a composite symbol denotes the fact that M is in state s and is scanning the symbol a. An important feature of Turing machine computations is the *locality* of the transitions, i.e., the succession relation between configurations depends only on local constraints. We can associate with M a 4-ary relation R_M on symbols that characterizes the transitions of M. Suppose that $\mathbf{a} = a_1 \ldots a_m$ and $\mathbf{b} = b_1 \ldots b_m$ are two configurations, $m = 2^n$. Then $\mathbf{b}$ is a successor configuration of $\mathbf{a}$ only if $(a_{i-1}, a_i, a_{i+1}, b_i) \in R_M$ for $1 < i < m$. We also need to associate with M two 3-ary relations R^l_M and R^r_M on symbols that characterize the transitions at the left and right end of the configuration, i.e., $(a_1, a_2, b_1) \in R^l_M$ and $(a_{m-1}, a_m, b_m) \in R^r_M$.

The idea of our encoding is that the unfolding expansions of the recursive program Π correspond to a sequence of configurations ending with an accepting configuration. The role of the union Θ of conjunctive queries is to check whether the sequence corresponds to an accepting computation. If an expansion τ does not correspond to an accepting computation, then we will have that $\tau \subseteq \Theta$. Thus, we will have that $\Pi \subseteq \Theta$ if and

only if the machine M does not accept. In order to check that an expansion τ does not correspond to an accepting computation, we have to compare corresponding positions on successive configurations. To do that, we address each position in a configuration; we need n bits for each address. In our encoding, each rule unfolding will describe one address bit. Thus, each position in a configuration will be encoded by n rule unfoldings.

If $\mathbf{a} = \alpha_n \dots \alpha_1$ and $\mathbf{b} = \beta_n \dots \beta_1$ are two n-bit numbers, then $\mathbf{b} = \mathbf{a} + 1 \pmod{2^n}$ precisely when the following hold: for $1 < i \leq n$, we have that $\alpha_i = \beta_i$ iff $\alpha_j = 0$ for some $1 \leq j < i$. Since this condition is not local, we encode *carry* bits in addition to address bits. Now $\mathbf{b} = \mathbf{a} + 1 \pmod{2^n}$ if and only if there is an n-bit carry $\mathbf{c} = \gamma_n \dots \gamma_1$ such that $\gamma_1 = 1$, $\gamma_{i+1} = 1$ precisely when $\alpha_i = 1$ and $\gamma_i = 1$ for $1 \leq i \leq n-1$, and $\beta_i = 0$ precisely when either both $\alpha_i = 0$ and $\gamma_i = 0$ or both $\alpha_i = 1$ and $\gamma_i = 1$, for $1 \leq i \leq n$. Thus, succession of addresses also has the locality property if the carry bits are available.

We encode configurations in the following manner. Let $Bit_1, \dots, Bit_n$ be 5-ary IDB predicates and let $A_1, \dots, A_n$ be 8-ary EDB predicates:

- The first two arguments of A_i act as the constants 0 and 1,
- the 3rd and 4th arguments of A_i encode address and carry bits, respectively
- the 5th and 6th arguments of A_i link successive bits, and
- the 7th and 8th arguments of A_i link successive configurations.

For $1 \leq i \leq n-1$, we have in Π the following rules:

$$\begin{aligned}
Bit_i(x,y,z,u,v) &:- Bit_{i+1}(x,y,z',u,v), A_i(x,y,x,x,z,z',u,v),\\
Bit_i(x,y,z,u,v) &:- Bit_{i+1}(x,y,z',u,v), A_i(x,y,x,y,z,z',u,v),\\
Bit_i(x,y,z,u,v) &:- Bit_{i+1}(x,y,z',u,v), A_i(x,y,y,x,z,z',u,v),\\
Bit_i(x,y,z,u,v) &:- Bit_{i+1}(x,y,z',u,v), A_i(x,y,y,y,z,z',u,v).
\end{aligned}$$

The intuition is that each unfolding of a rule for a Bit_i predicate describes one address bit. The variable z can be thought as a pointer to an address bit, while z' points to the next bit. Note the four possible combinations of variables in the third and fourth arguments of body EDB predicate A. Each combination encodes two bits of information, an address bit and a carry bit. Intuitively, x and y, which are *persistent* variables, i.e., they appear both in the head atom and in the recursive body atom, act as the constants 0 and 1. That is, the third argument being x, or y corresponds to the address bit being 0 or 1, respectively. Similarly, the fourth argument being x, or y corresponds to the carry bit being 0 or 1, respectively. The carry bits encode the carry obtained when the *previous* address is incremented by 1. Note that the variables u and v are also persistent in the above rules; this persistence connects nodes that belong to the same configuration.

The rules for Bit_n encode also the symbol pointed to by the n-bit address. For each symbol a of the machine M we have a unary EDB predicate Q_a. The symbol in a configuration position is encoded by rules of the form:

$$Bit_n(x,y,z,u,v) \ :- Bit_1(x,y,z',u,v), A_n(x,y,x,x,z,z',u,v), Q_a(z).$$

(The 3rd and 4th arguments of A_n could also be the pair x, y, the pair y, x, or the pair y, y.)

So far the rules encode a sequence of address bits and tape symbols. To encode the start of the computation, we use the 0-ary *goal predicate* C, a 1-ary EDB predicate $Start$, and the rule

$$C :- Bit_1(x, y, z, u, v), Start(z).$$

To encode the end of the computation, we use rules of the form:

$$Bit_n(x, y, z, u, v) \ :- A_n(x, y, x, x, z, z', u, v), Q_a(z),$$

for symbols a that correspond to accepting states. (The 3rd and 4th arguments in A_n could also be the pair x, y, the pair y, x, or the pair y, y.)

Finally, to encode the transition from configuration to configuration, we use rules of the form

$$Bit_n(x, y, z, u, v) \ :- Bit_1(x, y, z', u', u), A_n(x, y, x, x, z, z', u, v), Q_a(z).$$

(The 3rd and 4th arguments in A_n could also be the pair x, y, the pair y, x, or the pair y, y.) Notice that u persists but changes position, but v does not persist (only x and y persist along nonsuccessive configurations). Intuitively, u's role is to connect successive configurations.

We now have to show how the conjunctive queries in Θ find errors in encoding of computations. The queries in Θ have no distinguished variables. We describe each disjunct of Θ by listing its atomic formulas. We use dots to denote variables with unique occurrences.

The first thing that we need to check is that the address bits indeed act as an n-bit counter. That is, the first address is $0, \ldots, 0$ and two adjacent addresses are successive. Thus, one possible error is that the first address is not $0, \ldots, 0$. Such an error can be found by the following conjunctive query:

$$Start(z_1), A_1(x, y, ., ., z_1, z_2, u, v), \ldots, A_i(x, y, y, ., z_i, z_{i+1}, u, v).$$

Here the third argument of A_i is y, expressing the fact that the i-th address bit is 1.

The other errors that can prevent adjacent addresses from being successive are:

1. the first carry bit is 0,
2. the i-th address bit and the i-th carry bit are 1, but the $(i+1)$-st carry bit is 0,
3. the i-th address bit or the i-th carry bit is 0, but the $(i+1)$-st carry bit is 1,
4. the i-th address bit and the i-th carry bit are 0, but the $(i+1)$-st address bit is 1,
5. the i-th address bit and the i-th carry bit are 1, but the $(i+1)$-st address bit is 1,

6. the i-th address bit is 1 and the i-th carry bit is 0, but the $(i+1)$-st address bit is 0,

7. the i-th address bit is 0 and the i-th carry bit is 1, but the $(i+1)$-st address bit is 0,

We show how errors of, for example, type (2) can be discovered; the other errors can be handled similarly. Such errors are found by the following conjunctive query:

$$\begin{array}{l} A_i(x, y, y, ., z_i, z_{i+1}, ., .), \\ A_{i+1}(x, y, ., ., z_{i+1}, z_{i+2}, ., .), \ldots, A_n(x, y, ., ., z_n, z_{n+1}, ., .), \\ A_1(x, y, ., ., z_{n+1}, z_{n+2}, ., .), \ldots \\ A_i(x, y, ., y, z_{n+i}, z_{n+i+1}, ., .), \\ A_{i+1}(x, y, ., x, z_{n+i+1}, z_{n+i+2}, ., .). \end{array}$$

The y's in the third argument on the first A_i atom and the fourth argument of the second A_i atom mean that the i-th address bit and the i-th carry bit are 1. The x in the fourth argument of the A_{i+1} atom means that the $(i+1)$-st carry bit is 0. Note that when the first n atoms refer to an address, the following $i+1$ atoms refer to the *next* address.

Next, we note that that every sequence of 2^n addresses starting with $0, \ldots, 0$ has to describe a single configuration, that is, we have to ensure that configuration change exactly when the address is $1, \ldots, 1$. Thus, there are two types of error here: (1) a configuration change when the address is not $1, \ldots, 1$,, and (2) a configuration does not change when the address is $1, \ldots, 1$. For example, error of the first type are found by the following conjunctive query:

$$\begin{array}{l} A_i(x, y, x, ., z_i, z_{i+1}, u, u), \ldots, \\ A_n(x, y, ., ., z_n, z_{n+1}, u, v), \\ A_1(x, y, ., ., z_{n+1}, z_{n+2}, u', u). \end{array}$$

The y's in the third argument of the first A_i atom means that the i-th address bit is 0. Thus, the address is not $1, \ldots, 1$. The fact that the variable u moved from the 7th position in the A_n atom to the 8th position in the second A_i atom indicates a configuration change.

We have so far ensured that we have a sequence of configurations of length 2^n with the proper sequence of addresses. We now have to enure that these sequence of configuration indeed represent a legal computation of the machine M. The first type possible error here is when the first configuration does not correspond to the empty tape with the head scanning the leftmost symbol. If $-$ is the blank symbol and s is the initial state, then the first configuration is $\langle s, - \rangle -^{2^n - 1}$. To ensure that the first symbol is indeed $\langle s, - \rangle$, the query

$$\begin{array}{l} Start(z_1), A_1(x, y, ., ., z_1, z_2, u, v), \ldots, \\ A_n(x, y, ., ., z_n, z_{n+1}, u, v), Q_a(z_n) \end{array}$$

checks whether the first symbols is not a, for each $a \neq \langle s, -\rangle$. Note that the variables $z_1, \ldots, z_n$ ensure that this query checks the first symbol in the first configuration. Similarly, to ensure that the rest of the symbols in the first configuration are blank, the query

$$\begin{array}{l} Start(z), A_1(x, y, ., ., z, ., ., u, v), \\ A_i(x, y, y, ., z_i, z_{i+1}, u, v), \ldots, \\ A_n(x, y, ., ., z_n, z_{n+1}, u, v), Q_a(z_n), \end{array}$$

checks that the first symbols is not a, for each $a \neq -$. Because of the variable z, the first A_1 atom must map to the first configuration. The variables u, v then ensures that the atoms $A_1, \ldots, A_n$ also map to the first configuration. The fact that the 3rd argument of A_i is y means that the query does not check the first symbol, since one of the address bits is 1.

Another type of errors is between corresponding symbols in two successive configurations, i.e., when such symbols do not obey the restrictions imposed by the relations R_M, R^l_M, and R^r_M. For example, a violation of R_M will be found by conjunctive queries of the following form:

$$\begin{array}{l} A_1(x, y, s_1, ., z_1, z_2, u, v), \ldots, \\ A_n(x, y, s_n, ., z_n, z_{n+1}, u, v), Q_a(z_n), \\ A_1(x, y, s_{n+1}, ., z_{n+1}, z_{n+2}, u, v), \ldots, \\ A_n(x, y, s_{2n}, ., z_{2n}, z_{2n+1}, u, v) Q_b(z_{2n}), \\ A_1(x, y, s_{2n+1}, ., z_{2n+1}, z_{2n+2}, u, v), \ldots, \\ A_n(x, y, s_{3n}, ., z_{3n}, z_{3n+1}, u, v) Q_c(z_{3n}), \\ A_1(x, y, s_{n+1}, ., z_{4n+1}, z_{4n+2}, u', u), \ldots, \\ A_n(x, y, s_{2n}, ., z_{5n}, z_{5n+1}, u', u), Q_d(z_{5n}). \end{array}$$

The pattern of the z_i's variables and the u, v variables ensures that the first three blocks of $A_1, \ldots, A_n$ atoms are mapped to three successive positions on the same configuration. The patern of the variables u, v, u' enures that the last block of $A_1, \ldots, A_n$ atoms is mapped to the next configuration. Finally, the reuse of the variables $s_{n+1}, \ldots, s_{2n}$ ensures the the second block and the last block refer to the same address and therefore are mapped to corresponding positions on successive configurations. Here a, b, c, d are such that $(a, b, c, d) \notin R_M$.

By adding to Θ conjunctive queries corresponding to all possible errors in the expansions of Π – there are $O(n)$ such errors, we reduce the acceptance problem for exponential-space Turing machines to containment of linear programs in unions of conjunctive queries.

We now sketch how this encoding can be extended to alternating exponential-space machines. An alternating machine M has existential and universal states. Without loss of generality, we can assume that (1) the machine always alternates between existential and universal states and (2) every configuration of M have two possible successors, a *left* successor and a *right* successor. The latter can be captured by M having two transition relations, one for left successors and one for right successors. An accepting computation of M is a tree of configurations, where each configuration is a successor of its parent, a

universal configuration (i.e., a configuration on which M is in a universal state) has both its successors as children, and all leaves are accepting configurations.

To encode a computation tree, we add to the Bit_i and A_i predicates two additional arguments. The rule

$$Bit_i(x, y, z, u, v) \; :- Bit_{i+1}(x, y, z', u, v), A_i(x, y, x, x, z, z', u, v)$$

will be replaced by the rule

$$Bit_i(x, y, z, u, v, w, t) \; :- Bit_{i+1}(x, y, z', u, v, w, t), A_i(x, y, x, x, z, z', u, v, w, t)$$

The intuition is that t is either x, when the configuration is existential, or y, when the configuration is universal. The pair u, v was replaced by the triple u, v, w to account for the fact that universal configuration has two successors. The other rules for Bit_i are replaced analogously.

We assume that the starting state is existential, so the rule for C is:

$$C :- Bit_1(x, y, z, u, v, w, x), Start(z).$$

The rules that encode transitions between configurations have to check whether the source configuration is existential or universal. For existential configurations we have rules such as:

$$\begin{array}{ll} Bit_n(x, y, z, u, v, w, x) & :- Bit_1(x, y, z', u', u, w', y), A_n(x, y, x, x, z, z', u, v, w, x), Q_a(z) \\ Bit_n(x, y, z, u, v, w, x) & :- Bit_1(x, y, z', u', v', u, y), A_n(x, y, x, x, z, z', u, v, w, x), Q_a(z) \end{array}$$

The 7th argument of Bit_n is x here, since these rules are for existential configurations. Here u migrates either to the 5th argument or the 6th argument of Bit_1. Migration to the 5th argument corresponds to a transition to a left successor, while migration to the 6th argument corresponds to a transition to the right successor.

For universal configurations we have rules such as:

$$\begin{array}{ll} Bit_n(x, y, z, u, v, w, y) & :- Bit_1(x, y, z', u', u, w', x), Bit_1(x, y, z', u', v', u, x), \\ & \quad A_n(x, y, x, x, z, z', u, v, w, y), Q_a(z) \end{array}$$

Here the 7th argument of Bit_n is y, since this rule is for universal configurations. This rule is nonlinear; the two occurrences of Bit_1 in the body correspond to transitions to both the left successor and the right successor.

The conjunctive queries in Θ also have to be revised to account for the additional arguments of the A_i's. There are also additional errors to capture. For example, if a is a composite symbol with a universal state then we have the query:

$$A_n(x, y, ., ., z_n, z_{n+1}, ., ., x), Q_a(z_n)$$

This query finds universal configurations marked as existential.

The queries that capture transition errors, have to distinguish between left successors and right successors. For example, the following query is directed at transitions to left successors:

$$\begin{array}{l} A_1(x,y,s_1,.,z_1,z_2,u,v,w,t),\ldots, \\ A_n(x,y,s_n,.,z_n,z_{n+1},u,v,w,t),Q_a(z_n), \\ A_1(x,y,s_{n+1},.,z_{n+1},z_{n+2},u,v,w,t),\ldots, \\ A_n(x,y,s_{2n},.,z_{2n},z_{2n+1},u,v,w,t)Q_b(z_{2n}), \\ A_1(x,y,s_{2n+1},.,z_{2n+1},z_{2n+2},u,v,w,t),\ldots, \\ A_n(x,y,s_{3n},.,z_{3n},z_{3n+1},u,v,w,t)Q_c(z_{3n}), \\ A_1(x,y,s_{n+1},.,z_{4n+1},z_{4n+2},u',u,w',t'),\ldots, \\ A_n(x,y,s_{2n},.,z_{5n},z_{5n+1},u',u,w',t'),Q_d(z_{5n}). \end{array}$$

Here u migrates one position to the right. For right successors, we need queries where u migrates two positions to the right.

Theorem 5.15: *Containment of a recursive Datalog program in a union of conjunctive queries is complete for 2EXPTIME (EXPSPACE for linear programs).*

6 Equivalence to Nonrecursive Programs

In the previous section we studied the complexity of containment of recursive programs in unions of conjunctive queries. In this section we focus on containment of recursive programs in nonrecursive programs, i.e., we are given a recursive program Π and a nonrecursive program Π' and we have to decide whether Π is contained in Π'. The straightforward approach is to rewrite Π' as a union of conjunctive queries and apply the results of the previous section. Unfortunately, rewriting nonrecursive programs as unions of conjunctive queries may involve an exponential blow-up in size. We now show several examples; the queries defined in this example will be used later in the lower-bound proof.

Example 6.1: Let E be a binary EDB predicate, i.e., the database is a directed graph. Consider the nonrecursive program consisting of the rules, for $0 < i \leq n$:

$$dist_i(x,y) \ :- dist_{i-1}(x,z), dist_{i-1}(z,y)$$

and the rule

$$dist_0(x,y) \ :- E(x,y)$$

Clearly, $dist_i(x,y)$ holds precisely when there is a path of length 2^i between x and y. It is easy to see that the smallest conjunctive query equivalent to $dist_n$ is of exponential size. ∎

Example 6.2: This example is a variant of the previous example. Consider the nonrecursive program consisting of the rules, for $0 < i \leq n$:

$$\begin{array}{ll} dist_{\leq i}(x,y) & :- dist_{\leq i-1}(x,z), dist_{\leq i-1}(z,y) \\ dist_{<i}(x,y) & :- dist_{<i-1}(x,z), dist_{\leq i-1}(z,y) \end{array}$$

and the rules

$$\begin{array}{ll} dist_{\leq 0}(x,y) & :-E(x,y) \\ dist_{\leq 0}(x,x) & :- \\ dist_{<0}(x,x) & :- \end{array}$$

(Note the empty bodies in the last two rules; the convention is that an empty body is equivalent to **true**.) Here, $dist_{\leq i}(x,y)$ holds precisely when there is a path of length at most 2^i between x and y, and $dist_{<i}(x,y)$ holds precisely when there is a path of length at most $2^i - 1$ between x and y. ∎

Example 6.3: To the EDB predicate E of the previous example, add unary EDB predicates $Zero$ and One, i.e., the database is a node-labeled directed graph. Consider the nonrecursive program consisting of the rules, for $0 < i \leq n$:

$$equal_i(x,y,u,v) \ :-equal_{i-1}(x,x',u,u'), equal_{i-1}(x',y,u',v),$$

and the rules

$$\begin{array}{ll} equal_0(x,y,u,v) & :-E(x,y), E(u,v), Zero(x), Zero(u), \\ equal_0(x,y,u,v) & :-E(x,y), E(u,v), One(x), One(u). \end{array}$$

Here $equal_i(x,y,u,v)$ holds precisely when there are paths length 2^i between x and y and between u and v, respectively, and the paths have the same labels, with the possible exception of the last points. ∎

In general, the upper bounds of Theorem 5.12 together with the exponential blow-up involved in rewriting nonrecursive programs as union of conjunctive queries imply upper bounds of triply exponential time for containment of recursive programs in nonrecursive programs and doubly exponential space for containment of linear recursive programs in nonrecursive programs. It turns out that the succinctness of nonrecursive programs is inherent, and these bounds are optimal. We can encode doubly-exponential space-bounded computation by using 2^n-bit counters. We can discover errors among bits that are doubly exponentially far from each other using programs such as those in the above examples.

We focus first on linear programs and doubly exponential-space Turing machines. A configuration of such a machine M can be described by a string of length 2^{2^n}. Thus, we need 2^n address bit to describe a position in the configuration.

As in Section 5.3, each rule unfolding in our encoding will describe one address bit. Thus, each position in a configuration will be encoded by 2^n rule unfoldings. In Section 5.3, we had the predicates $Bit_1, \ldots, Bit_n$ to denote the n address bits. We cannot do this here, since we have 2^n address bits. Instead, we use one ternary IDB predicate Bit and one binary EDB predicate A. In addition we use several unary EDB predicates.

The recursive program Π contains the following rules:

$$\begin{array}{ll} Bit(z,u,v) & :-Bit(z',u,v), A(z,u,v), Address(z), E(z,z'), Zero(z), Carry0(z) \\ Bit(z,u,v) & :-Bit(z',u,v), A(z,u,v), Address(z), E(z,z'), Zero(z), Carry1(z) \\ Bit(z,u,v) & :-Bit(z',u,v), A(z,u,v), Address(z), E(z,z'), One(z), Carry0(z) \\ Bit(z,u,v) & :-Bit(z',u,v), A(z,u,v), Address(z), E(z,z'), One(z), Carry1(z) \end{array}$$

The difference from the rules for Bit_i in Section 5.3 is that we have additional EDB atoms $Address(z)$, $E(z,z')$, $Zero(z)$, $One(z)$, Carry0(z), and $Carry1(z)$. Here the variable z points to an address bit or to a symbol. So we call these variables *points*. The atom $Address(z)$ says that these rules describe address points, so we call these rules *address rules*. The atoms $E(z,z')$ connects adjacent points. The rest of the atoms encode the address and carry bits. Notice that these atoms encode the information that was previously carried by the first four arguments of the A_i predicates.

We have separate rules in Π for encoding configuration symbols:

$$Bit(z,u,v) \ :-Bit(z',u,v), A(z,u,v), E(z,z'), Symbol(z), Q_a(z).$$

The atom $Symbol(z)$ says that the rule describes a symbol, so we call this rule a *symbol rule*.

To encode the start of the computation, we use the 0-ary goal predicate C, a 1-ary EDB predicate $Start$, and the rule

$$C \ :-Start(z), Bit(z,u,v), A(z,u,v), Address(z), Zero(z), Carry1(z)$$

which means that the first bit of the first configuration is the address bit 0.

To encode the end of the computation, we put in Π rules of the form:

$$Bit(z,u,v) \ :-A(z,u,v), Symbol(z), Q_a(z),$$

for symbols a that correspond to accepting states. That is, the last symbol of the last configuration is an accepting symbol.

Finally, to encode the transition from configuration to configuration, we put in Π rules of the form

$$Bit(z,u,v) \ :-Bit(z',u',u), A(z,u,v), E(z,z'), Symbol(z), Q_a(z).$$

We now describe the construction of Π'. As in the previous lower-bound proof, the idea of our encoding is that the unfolding expansions of the recursive program Π correspond to a sequence of configurations, ending with an accepting configuration of the machine M. The role of the nonrecursive program Π' is to check whether the sequence corresponds to an accepting computation. If an expansion τ does not correspond to an accepting computation, then we will have that $\tau \subseteq \Pi'$. Thus, we will have that Π is contained in Π' if and only if the machine M does not accept.

In Section 5.3, unfolding expansions of the recursive program were guaranteed to correspond to sequences of n-bit addresses, with a symbol attached to each n-th bit. Here we want unfolding expansions that corresponds to sequences of 2^n-bit address points followed by a symbol point. This, however, is not guaranteed by the program, since we have a single Bit predicate. Instead, we have rules in Π' that "filter out" expansions that are not of this format.

All expansions start with the unfolding of the start rule. We need to verify that this is followed by $2^n - 1$ unfoldings of address rules, and then an unfolding of the symbol rule. This is verified by putting the following rule in Π':

$$C \quad :-Start(z), dist_{<n}(z, z'), Symbol(z')$$

and the rule

$$C \quad :-Start(z), dist_n(z, z'), Address(z')$$

The former rule finds expansions where one of the first 2^n unfoldings is of a symbol rule. The latter rule finds expansion where the $2^n + 1$-st unfolding is of an address rule.

The above queries take care of the first $2^n + 1$ unfoldings. To make sure that the expansions has the right format we also need the rule

$$C \quad :-Symbol(z), dist_{\geq n}(z, z'), Symbol(z'')$$

and the rule

$$C \quad :-Symbol(z), dist_n(z, z'), E(z', z''), Address(z'')$$

This makes sure that a symbol point is followed by 2^n address points followed by a symbol point.

So far we have ensured that that we have filtered away all expansions to do not correspond to sequences of blocks of 2^n address points followed by a symbol point. Analogously to Section 5.3, we need to check is that the address bits indeed act as an 2^n-bit counter. That is, the first address is $0, \ldots, 0$ and two adjacent addresses are successive. As in Section 5.3, there are 7 possible errors in the counter. For example, a possible error is that the first address is not $0, \ldots, 0$. Such an error can be found by the following query:

$$C \quad :-Start(z), dist_{<n}(z, z'), One(z').$$

Another possible addressing error is when the i-th address bit and the i-th carry bit are 1, but the $(i + 1)$-st carry bit is 0. Such an error is found by the following rule:

$$\begin{aligned} C \quad :-&Address(z), One(z), dist_n(z, z'), E(z', z''), \\ &Carry1(z''), E(z'', z'''), Carry0(z''') \end{aligned}$$

This query is using the fact that we need only consider expansions in which the distance between corresponding address points is precisely $2^n + 1$. Thus, z and z'' point to address points in corresponding positions of successive addresses, and z''' is the next address point.

So far we have ensured that our addresses indeed act as a 2^n-bit counter. We now have to ensure that every sequence of 2^{2^n} addresses starting with $0,\ldots,0$ describe a single configuration, that is, we have to ensure that configuration change exactly when the address is $1,\ldots,1$. Thus, there are two types of error here: (1) a configuration change when the address is not $1,\ldots,1$,, and (2) a configuration does not change when the address is $1,\ldots,1$. For example, error of the first type are found by the following rule in Π':

$$C \ :- Address(z), A(z,u,v), Zero(z), dist_{\leq n}(z,z'), Symbol(z'), E(z',z''), A(z'',u',u)$$

The atom $Zero(z)$ indicates that we are referring to an the address that is not $1,\ldots,1$. The fact that the variable u moved from the 2nd position in the first A atom to the 3th position in the second A atom indicates a configuration change.

We have so far ensured that we have a sequence of configurations of length $2^{2^n}+1$ with the proper sequence of addresses. We now have to enure that this sequence of configurations indeed represent a legal computation of the machine M. For example, we have to detect errors between corresponding symbols in two successive configurations, i.e., when such symbols do not obey the restrictions imposed by the relations R_M, R^l_M, and R^r_M. A violation of R_M will be found by rules in Π' of the following form:

$$\begin{aligned} C \ :- & A(z_1,,u,v), Q_a(z_1), E(z_1,t_1),\\ & A(t_1,u,v), dist_n(t_1,z_2),\\ & A(z_2,u,v), Q_b(z_2), dist_n(z_2,z_3'), E(z_2',z_3),\\ & A(z_3,u,v), Q_c(z_3),\\ & A(t_2,.,u), dist_n(t_2,z_4),,\\ & A_n(z_4,.,u), Q_d(z_4),\\ & equal_n(t_1,z_2,t_2,t_4). \end{aligned}$$

Here, the variables z_1, z_2, and z_3 point to three consecutive symbols a, b, and c. The variables t_1 points to the first address point preceding z_2. The variable z_4 points to the symbols d in the successor configuration (notice that u migrates one position to the right), and t_2 points to the first address point preceding z_4. The atom $equal_n(t_1,z_2,t_2,z_4)$ guarantees that z_2 and z_4 have the same addresses, so they point to symbols in corresponding positions.

By adding to Π' rules corresponding to all possible errors in the expansions of Π – there are $O(n)$ such errors, we reduce the acceptance problem for doubly exponential-space Turing machines to containment of linear programs in nonrecursive programs. We deal with nonlinear programs as in Section 5.3, that is, by adding arguments to Bit and A and using nonlinear rule we can force universal configurations to have two successors.

Theorem 6.4: *Containment of a recursive Datalog program in a nonrecursive Datalog program is complete for 3EXPTIME (2EXPSPACE for linear programs).*

Before stating our final result, we note that if Π is a recursive program and Π' is a nonrecursive program such that both Π and Π' have the same goal predicate Q, and Q occurs only in heads of rules, than $\Pi \subseteq \Pi'$ if and only if $\Pi \cup \Pi' \equiv \Pi'$. Thus, containment in nonrecursive programs is reducible to equivalence to nonrecursive programs.

Theorem 6.5: *Equivalence of recursive Datalog programs to nonrecursive Datalog programs is complete for 3EXPTIME (2EXPSPACE for linear recursive programs).*

We note that the blow-up in the translation from nonrecursive programs to unions of conjunctive queries is caused by the nonlinearity of the nonrecursive programs. For linear nonrecursive programs the translation yields a union of conjunctive queries, where each conjunctive query is of size that is linear in the size of the nonrecursive program (though the number of terms in the union could be exponential).

Example 6.6: Again we use the binary EDB predicate E and the unary EDB predicates $Zero$ and One. Consider the nonrecursive program consisting of the rules, for $1 < i \leq n$:

$$\begin{aligned} word_i(x,y) &\ :-word_{i-1}(x,x'), E(x',y), Zero(y) \\ word_i(x,y) &\ :-word_{i-1}(x,x'), E(x',y), One(y) \end{aligned}$$

and the rules

$$\begin{aligned} word_1(x,y) &\ :-E(x,y), Zero(x) \\ word_1(x,y) &\ :-E(x,y), One(x) \end{aligned}$$

It is easy to see that by unfolding $word_n$ to a union of conjunctive queries we get exponentially many terms in the union, but each term is of size $O(n)$. ∎

Nevertheless, the proof of Theorem 5.12 does go through, and the bounds of the theorem hold for linear nonrecursive programs.

Theorem 6.7: *Equivalence of recursive Datalog programs to nonrecursive, linear Datalog programs is complete for 2EXPTIME (EXPSPACE for linear recursive programs).*

Proof: Let Π be a program with goal predicate Q and let Π' be a nonrecursive, linear program with goal Q. As observed above, Π' can be unfolded to an exponential union $\Theta = \cup_i \theta_i$ of conjunctive queries, each of size linear in the size of Π'. By Theorem 5.11, Π is contained in Θ if and only if

$$T(A_{Q,\Pi}^{p_trees}) \subseteq \bigcup_i T(A_{Q,\Pi}^{\theta_i}).$$

By Propositions 4.1 and 4.4, we can obtain an automaton $A_{Q,\Pi}^{\Theta}$, whose size is exponential in the size of Π and Π', such that

$$T(A_{Q,\Pi}^{\Theta}) = \bigcup_i T(A_{Q,\Pi}^{\theta_i}).$$

The upper bound then follows as in Theorem 5.12. ∎

7 Concluding Remarks

Courcelle's Theorem yields the decidability of equivalence to nonrecursive program, but with very weak upper bounds. Using the automaton-theoretic technique we found tighter upper bounds, triply-exponential time in general and doubly-exponential space for linear programs, and proved matching lower bounds. This shows that the problem is intractable; in fact, to our knowledge this is the most intractable decidable problem is database theory. We note that one has to be careful in interpreting lower bounds for query containment. While containment of conjunctive queries in recursive program is complete for EXPTIME [CK86, CLM81, Sa88b], this complexity is simply the expression complexity of evaluation Datalog programs [Va82]. In fact, if attention is restricted to programs of bounded arity, we get NP-completeness instead of EXPTIME-completeness. In contrast, our lower bounds here imply "real" intractability, and they hold even for programs of bounded arity.

For certain classes of recursive Datalog programs, such as monadic linear programs and single-rule programs, the equivalence problem is less intractable than the general case. We discuss these cases in another paper [CV94].

8 Acknowledgement

We thank Shuki Sagiv for discussions on the complexity of query containment and Jeff Ullman for useful comments that helped improve a previous draft of the paper.

References

[AU79] Aho, A.V., Ullman, J.D.: Universality of data retrieval languages. *Proc. 6th ACM Symp. on Principles of Programming Languages*, 1979, pp. 110–117.

[AV89] Abiteboul, S., Vianu, V.: Fixpoint Extensions of First-Order Logic and Datalog-like languages. *Proc. 4th IEEE Symp. on Logic in Computer Science*, 1989, pp. 71–79.

[Be80] Beeri, C.: On the membership problem for functional and multivalued dependencies in relational databases. *ACM Trans. on Database Systems*, 5(1980), pp. 241–259.

[BR86] Bancilhon, F., Ramakrishnan, R.: An amateur's introduction to recursive query processing strategies. *Proc. ACM Conf. on Management of Data*, Washington, 1986, pp. 16–52.

[Ch81] Chandra, A.K.,: Programming primitives for database languages. *Proc. 8th ACM Symp. on Principles of Programming Languages*, Williamsburg, 1981, pp. 50–62.

[CH80] Chandra, A.K., Harel, D.: Computable queries for relational databases. *J. Computer and Systems Sciences* 21(1980), pp. 156–178.

[CH82] Chandra, A.K., Harel, D.: Structure and Complexity of Relational Queries. *J. Computer and Systems Sciences* 25(1982), pp. 99–128.

[CH85] Chandra, A.K., Harel, D.: Horn Clause Queries and Generalizations. *J. Logic Programming*, **2** (1985), 1–15.

[CKS81] Chandra, A., Kozen, A., Stockmeyer, L.: Alternation, *J. ACM* 28(1981), pp. 114–133.

[CLM81] Chandra, A. K., H. R. Lewis, and J. A. Makowsky, Embedded implicational dependencies and their inference problem. *Proc. 13th ACM Symp. on Theory of Computing*, 1981, pp. 342–354.

[CV94] Chaudhuri, S., Vardi, M.Y.: On the complexity of equivalence between recursive and nonrecursive Datalog programs. *Proc. 13th ACM Symp. on Principles of Database Systems*, 1994, pp. 107–116.

[CGKV88] Cosmadakis, S.S., Gaifman, H., Kanellakis, P.C., Vardi, M.Y.: Decidable optimization problems for database logic programs. *Proc. 20th ACM Symp. on Theory of Computing*, 1988, pp. 477-490.

[CK86] Cosmadakis, S.S., Kanellakis, P.: Parallel evaluation of recursive rule queries. *Proc. 5th ACM Symp. on Principles of Database Systems*, Cambridge, 1986, pp. 280–293.

[Cos72] Costich, O.L.: A Medvedev characterization of sets recognized by generalized finite automata. *Math. System Theory* 6(1972), pp. 263–267.

[Cou90] Courcelle, B.: The monadic second-order theory of graphs I – Recognizable sets of finite graphs. *Information and Computation* 85(1990), pp. 12–75.

[Cou91] Courcelle, B.: Recursive queries and context-free graph grammars. *Theoretical Computer Science* 78(1991), pp. 217–244.

[Do70] Doner, J.E.: Tree acceptors and some of their applications. *J. Computer and System Sciences* 4(1971), pp. 406–451.

[EJ88] Emerson, E.A., Jutla, C.S.: Complexity of tree automata and modal logics of programs. *Proc. 29th IEEE Symp. on Foundations of Computer Science*, 1988, pp. 328–337.

[GM78] Gallaire, H., Minker, J.: *Logic and Databases*. Plenum Press, 1978.

[GMSV93] Gaifman, H., Mairson, H., Sagiv, Y., Vardi M.Y.: Undecidable optimization problems for database logic programs. *J. ACM* 40(1993), pp. 683–713.

[HKMV91] Hillebrand, G.G., Kanellakis, P.C., Mairson, H.G., Vardi, M.Y.: Tools for Datalog boundedness. *Proc. 10th ACM Symp. on Principles of Database Systems*, May 1991, pp. 1–12.

[HKMV95] Hillebrand, G.G., Kanellakis, P.C., Mairson, H.G., Vardi, M.Y.: *Undecidable boundedness problems for Datalog programs.* To appear in *J. Logic Programming.*

[Im86] Immerman, N.: Relational queries computable in polynomial time. *Information and Control* 68(1986), pp. 86–104.

[Jo75] Jones, N.D.: Space-bounded reducibility among combinatorial problems. *J. Computer and System Sciences* 11(1975), pp. 68–85.

[K90] Kanellakis P.C.: Elements of Relational Database Theory. *Handbook of Theoretical Computer Science*, Vol. B, Chapter 17, J. van Leeuwen, A.R. Meyer, N. Nivat, M.S. Paterson, D. Perrin ed., North-Holland 1990.

[KA89] Kanellakis, P., Abiteboul, S.: Deciding bounded recursion in database logic programs. *SIGACT News* 20:4, Fall 1989.

[MUV84] Maier, D., Ullman, J.D., Vardi, M.Y.: On the foundations of the universal relation model. *ACM Trans. on Database Systems* 9(1984), pp. 283–308.

[Mey93] van der Meyden, R.: Recursively indefinite databases. *Theoretical Computer Science*, 116(1993), pp. 151–194.

[MF71] Meyer, A.R., Fischer, M.J.: Economy of description by automata, grammars, and formal systems. *Proc. 21st IEEE Symp. on Switching and Automata Theory*, 1971, pp. 188–191.

[MS72] Meyer, A.R., Stockmeyer, L.J.: The equivalence problem for regular expressions with squaring requires exponential time. *Proc. 13th IEEE Symp. on Switching and Automata Theory*, 1972, pp. 125–129.

[MP91] Mumick, I.S.,Pirahesh, H.: Overbound and right-linear queries. *Proc. 10th ACM Symp. on Principles of Database Systems*, 1991, pp. 127–141.

[Mo74] Moschovakis, Y.N.: *Elementary Induction on Abstract Structures.* North Holland, 1974.

[Na89a] Naughton, J.F.: Data independent recursion in deductive databases. *J. Computer and System Sciences*, 38(1989), pp. 259–289.

[Na89b] Naughton, J.F.: Minimizing function-free recursive definitions. *J. ACM* 36(1989), pp. 69–91.

[NRSU89] Naughton, J.F., Ramakrishnan, R., Sagiv, Y., Ullman, J.D.: Efficient evaluation of right , left , and multilinear rules. *Proc. ACM-SIGMOD Int'l Conf. on Management of Data*, 1989, pp. 235–242.

[Ra69] Rabin, M.O.: Decidability of second-order theories and automata on infinite trees. *Trans. AMS* 141(1969), pp. 1–35.

[RS59] Rabin, M.O., Scott, D.: Finite automata and their decision problems. *IBM J. Research and Development*, 3(1959), pp. 114–125.

[RSUV93] Ramakrishnan, R., Sagiv, Y., Ullman, J.D., Vardi, M.Y.: Logical query optimization by proof-tree transformation. *J. Computer and System Sciences* 47(1993), pp. 222–248

[SY81] Sagiv,Y., Yannakakis M.: Equivalences among Relational Expressions with the union and difference operators. *JACM*, 27:4, pp 633-655.

[Sa88b] Sagiv, Y.: Optimizing Datalog programs. In *Foundations of Deductive Databases and Logic Programming*, J. Minker (ed.), Morgan Kaufmann Publishers, 1988, pp. 659–698.

[Se90] Seidl, H.: Deciding equivalence of finite tree automata. *SIAM J. Computing* 19(1990), pp. 424–437.

[Shm87] Shmueli, O.: Decidability and expressiveness aspects of logic queries. *Proc. 6th ACM Symp. on Principles of Database Systems*, 1987, pp. 237–249.

[TW68] Thatcher, J.W., Wright, J.B.: Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical System Theory* 2(1968), pp. 57–81.

[Ul88] Ullman, J.D.: *Principles of Database and Knowledge Base Systems*, Vol. 1, Computer Science Press, 1988.

[Ul89] Ullman, J.D.: *Principles of Database and Knowledge Base Systems*, Vol. 2, Computer Science Press, 1989.

[UV88] Ullman, J.D., Van Gelder, A.: Parallel complexity of logical query programs. *Algorithmica* 3(1988), pp. 5–42.

[Va82] Vardi, M.Y.: The complexity of relational query languages. *Proc. 14th ACM Symp. on Theory of Computing*, San Francisco, 1982, pp. 137–146.

[Va88] Vardi, M.Y.: Decidability and undecidability results for boundedness of linear recursive queries. *Proc. 7th ACM Symp. on Principles of Database Systems*, 1988, pp. 341–351.

[Va92] Vardi, M.Y.: Automata theory for database theoreticians. In *Theoretical Studies in Computer Science* (J.D. Ullman, ed.), Academic Press, 1992, pp. 153-180.

[VW86] Vardi, M.Y., Wolper, P.: Automata-theoretic techniques for modal logic of programs. *J. Computer and System Sciences* 32(1986), pp. 183–221.

[Zl76] Zloof, M.; *Query-by-Example: Operations on the Transitive Closure*. IBM Research Report RC5526, 1976.