RJ 6145 (60855) 3/22/88
Computer Science

Decidable Optimization Problems for Database Logic Programs

Preliminary Report

Stavros S. Cosmadakis*
IBM Watson Research Center

Haim Gaifman†
Hebrew University of Jerusalem

Paris C. Kanellakis‡
Brown University

Moshe Y. Vardi§
IBM Almaden Research Center

Abstract

Datalog is the language of logic programs without function symbols. It is used as a database query language. If it is possible to eliminate recursion from a Datalog program Π, then Π is said to be *bounded*. It is known that the problem of deciding whether a given Datalog program is bounded is undecidable, even for *binary* programs. We show here that boundedness is decidable for *monadic* programs, i.e., programs where the recursive predicates are monadic (the non-recursive predicates can have arbitrary arity). Underlying our results are new tools for the optimization of Datalog programs based on automata theory and logic. In particular, one of the tools we develop is a theory of two-way alternating tree automata. We also use our techniques to show that containment for monadic programs is decidable.

1 Introduction

It has been recognized for some time that first-order database query languages are lacking in expressive power [AU79, Fa75, GM78, Zl76]. Since then many higher-order query languages have been investigated [CH80, Ch81, CH82, Im86, Va82]. A language that has received considerable attention recently is *Datalog*, the language of logic programs (known also as Horn-clause programs) without function symbols [BR86, CH85, GM78, HN84, Ul85]. This

*Address: IBM Watson Research Center 36-209, P.O.Box 128, Yorktown Heights, NY 10598.

†Address: Inst. of Math. and Computer Science, Hebrew University, Givat, Ram, Jerusalem 91 904, Israel. Part of the research reported here was done while this author was visiting the Computer Science Department of Stanford University.

‡Address: Dept. of Computer Science, Brown University, P.O.Box 1910, Providence, RI 02912. The work of this author was supported partly by NSF grant IRI-8617344 and partly by an Alfred P. Sloan Foundation Fellowship.

§Address: IBM Almaden Research Center K53-802, 650 Harry Rd, San Jose, CA 95120-6099.

language is essentially a fragment of fixpoint logic [Mo74] interpreted over finite structures, i.e., databases.

We refer to the literature for a detailed definition of Datalog [MW88] and use the following example to illustrate its syntax and semantics:

$$access(X) \ :- source(X).$$
$$access(X) \ :- access(Y), access(Z), triple(X, Y, Z).$$

In this example, *triple* and *source* are ***extensional database (EDB) predicates***, i.e., representing basic facts stored in the database. That is, the input to this program is a database consisting of facts for predicates *source* and *triple*. The predicate *source* represents the set of source nodes, and the predicate *triple* represents accessibility conditions. For example, $source(1)$ is an EDB fact stating that 1 is one of the source nodes. Similarly, $triple(1, 2, 3)$ is an EDB fact stating that if 2 and 3 are accessible, then so is 1. The ***intensional database (IDB) predicate*** *access* represents facts deduced from the database via the two rules of the logic program above: the first rule says every source node is accessible, and the second rule tells how the accessibility predicate extends. Note that the first rule is nonrecurive and the second rule is recursive. We can now query, for instance, $access(7)$ to determine whether 7 is a accessible node. This example can be used to express the set of ***accessible*** nodes in a ***path system*** [Co74]. One can show that path system accessibility is not definable in first-order database query languages. The above program is also an example of a ***monadic*** program, i.e., its IDB predicates are monadic although the EDB predicates may have arbitrary arities.

The gain in expressive power does not, however, come for free; evaluating Datalog programs is harder than evaluating first-order ones [Va82]. Recent studies have addressed the problems of finding efficient evaluation methods and compile-time optimization techniques for Datalog programs; [BR86] is a good survey on this topic. Since the source of the difficulty in evaluating Datalog queries is their recursive nature, the first line of attack in trying to optimize such programs is to eliminate the recursion. A decision procedure for recursion elimination for ***monadic*** Datalog programs is the subject of this paper.

Let us now illustrate recursion elimination using a second Datalog example, (from [Na86]):

$$buys(X, Y) \ :- likes(X, Y).$$
$$buys(X, Y) \ :- trendy(X), buys(Z, Y).$$

In this example, $buys(Z, Y)$ can be changed to $likes(Z, Y)$, yielding an equivalent recursion-free program. On the other hand the program

$$buys(X, Y) \ :- likes(X, Y).$$
$$buys(X, Y) \ :- knows(X, Z), buys(Z, Y).$$

is inherently recursive, i.e., it is not equivalent to any recursion-free program. The difference is that in the former program deduction of a fact involving *buys* can be made in a number of steps that is independent of the number of facts of the input predicates *likes* and *trendy*, that is, the recursion is "bounded". More formally:

A Datalog program Π is a collection of function-free Horn clauses (rules). We distinguish between a program's extensional predicates (EDB), which are those predicates occurring only in the bodies of its rules, and the intensional predicates (IDB), which are all those occurring as heads of its rules. We say that program Π is *monadic* if all its IDB predicates are monadic. A database D is a *finite* collection of ground atomic formulas (facts) for the EDB predicates of a program Π. Let $Q^i_\Pi(D)$ be the collection of facts about an IDB predicate Q that can be deduced from 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 applications of the rules in Π, that is,

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

Let $\Pi^i(D)$ (resp. $\Pi^\infty(D)$) be the union of $Q^i_\Pi(D)$ (resp., $Q^\infty_\Pi(D)$) for all IDB predicates Q in Π.

Since D is finite, for each D there is some k, depending on D and Π, such that $Q^k_\Pi(D) = Q^\infty_\Pi(D)$. The IDB predicate Q is *bounded* (in Π) if there exists a constant c, depending only on Π, such that for any database D, we have $Q^c_\Pi(D) = Q^\infty_\Pi(D)$. The program Π is *bounded* if all its IDB predicates are bounded; that is, there exists a constant c, depending only on Π, such that for any database D, we have $\Pi^c(D) = \Pi^\infty(D)$.

The notion of boundedness has been introduced in the context of the universal relation database model [MUV84], and has been studied recently by a number of researchers, e.g., [CK86, GMSV87, Io85, Ka86, Na86, NS87, Sa85, Va88]. The distinction between predicate boundedness and program boundedness is one of the contributions of this paper. Note that a solution to the predicate boundedness problem implies a solution to the program boundedness problem but not vice versa.

At first results on detecting boundedness in Datalog were positive; [Io85, Na86, NS87, Sa85] describe algorithms for detecting boundedness in several specialized classes of Datalog programs. Unfortunately, the early optimism about detecting boundedness algorithmically was unjustified. In [GMSV87] it was shown that boundedness is undecidable even for programs that have a single IDB predicate (so predicate boundedness and program boundedness coincide) and that are also *linear*, i.e., all rule bodies have at most one occurrence of an IDB. This result was tightened recently in [Va88] to *binary linear* programs with a single IDB. Our main result in this paper is that predicate and program boundedness are decidable for *monadic* programs. Our result is in a sense syntactically tight. Not only is boundedness for binary programs undecidable but, as was shown in [GMSV87], extending monadic Datalog with "impure" features, such as *inequalities* in the rule bodies, also makes boundedness undecidable. Inequalities are a weak form of negative atomic formulas and it is somewhat surprising that their presence can have such a dramatic effect. In addition to proving upper bounds on the complexity of boundedness, we prove some lower bounds. Though our bounds are not in general tight, they are tight in the important case of monadic linear programs, where we prove that the problem is PSPACE-complete.

Underlying our decidability and complexity results are two new tools for the optimization of Datalog programs: one is based on automata theory and the other on the compactness theorem for first-order logic.

Let us first outline the relationship with automata theory. A predicate defined by a Datalog program Π can instead be defined by an infinite union of *conjunctive queries*. Conjunctive queries constitute a fragment of first-order database queries, for which many optimization problems have been completely resolved [ASU79, CM77]. For monadic programs the infinite unions of conjunctive queries can be represented as a collection of trees over some finite alphabet, which we view as a tree language. We show that the boundedness condition can be equivalently phrased as a language-theoretic condition on tree languages. This naturally leads to reducing boundedness to an automata-theoretic problem. Resolving the decidability and complexity of this problem requires a development of a theory of *2-way alternating tree automata*. Using this theory we reduce boundedness to a condition on standard *top-down tree automata*.

Our automata-theoretic technique yields better upper bounds for a class of program that we call *connected*; the rule $Q(X) : -A(X,Y), P(Z)$ is not connected, because the variable Z is not related to the variable X. Using the compactness theorem we show that program boundedness (as opposed to predicate boundedness) for arbitrary monadic programs can be reduced to program boundedness for connected monadic programs. The compactness argument is actually applicable to Datalog programs in general and is therefore of independent interest.

The usefulness of language theory in the analysis of Datalog has been demonstrated before (cf. [AP87, BKBR87, UV86]). All these papers, however, considered the so called *chain* programs, which are programs where the rules correspond very naturally to productions of a context-free grammar. Since the class of chain programs is somewhat artificial, the general applicability of language theory to the analysis of Datalog programs was questionable. For instance, boundedness of binary recursion in Datalog is undecidable, but boundedness of binary chain programs is a direct encoding of the finiteness problem for context-free languages and therefore is easily decidable. The class of monadic programs is quite a natural, expressive and syntactically simple class. Furthermore, their arbitrary EDB predicates and rule bodies are combinatorially very different from grammatical production rules. Thus, we view our results (together with related results in [Va88]) as a convincing demonstration of the applicability of language theory to the optimization of Datalog programs.

After some preliminary definitions in Section 3, we examine in Section 3 boundedness of linear monadic programs. The relationship between monadic Datalog and language theory is established in this simpler context. For connected linear monadic programs we derive a polynomial space upper bound. Section 4 contains the generalization of these results to nonlinear monadic programs. For connected nonlinear monadic programs we derive a doubly exponential time upper bound. In Section 5 we focus on program boundedness and we reduce the monadic case to the monadic connected case. We also prove lower bounds, which are tight in the linear case. In Section 6 we focus on predicate boundedness. We derive an exponential space upper bound for the linear case and a triply exponential time upper bound for the nonlinear case. In Section 7 we indicate how our techniques may be used to show the decidability of containment problems for monadic programs. We close our presentation in Section 8 with a summary of our results.

Because of length limitations we omit most details of the proofs in this paper. They will be given in the full paper.

2 Preliminaries

Let F be a set of atomic formulas. The *variable graph* for F is an undirected graph $G_F = (V, E)$, where V is the set of variables occurring in F and $\{X, Y\} \in E$ if X and Y occur in the same atomic formula of F.

A monadic *recursive* rule r is of the form $Q(X) : -A, Q_1(Y_1), \ldots, Q_n(Y_n)$, where A is the set of atomic EDB formulas in the rule. (For simplicity of exposition in this abstract, we assume that X is different from $Y_1, \ldots, Y_n$. However, our results hold in general, i.e., they hold even if X is identical with any Y_i.) We say that r is *connected* if G_A is a connected graph and X, $Y_1, \ldots, Y_n$ are nodes in G_A.[1] A monadic *non-recursive* (*initialization*) rule r is of the form $Q(X) : -B$, where B is its set of EDB formulas. We say that r is *connected* if G_B is connected and X is a node in G_B. We call A a *recursive body* and B an *initialization body*. A program Π is *connected* if all its rules are connected. The *diameter* d of Π is the maximal distance between two connected variables in any of the variable graphs for rules of Π. *Linear* monadic programs consist of recursive rules of the form $Q(X) : -A, Q'(Y)$, and initialization rules of the form $Q(X) : -B$.

It is known that the predicates defined by a Datalog program Π can be equivalently defined by an infinite union of *conjunctive queries* [CM77].[2] That is, there is an infinite sequence $C_0, C_1, \ldots$ of conjunctive queries such that for every database D, we have

$$Q_\Pi^\infty(D) = \bigcup_{i=0}^{\infty} C_i(D).$$

The C_i's are called the *Q-expansions* of Π. It was shown in [Na86] that Π is bounded if and only if for each IDB predicate Q with expansions $C_0, C_1, \ldots$ there is some $N \geq 1$ such that for all $n > N$, the query C_n is contained in C_m for some $m \leq N$. Note that C_n is contained in C_m if there is a *containment mapping* from C_m to C_n [CM77].A containment mapping from C_m to C_n is a mapping from the variables of C_m to the variables of C_n that maps free variables to free variables such that the atomic formulas of C_m are mapped to atomic formulas of C_n.

3 Linear Connected Programs

In this and in the following section, we assume that we are dealing with connected programs. We also assume in this section that programs are *linear*, i.e., that every rule has at most one occurrence of an IDB predicate. (For example, $Q(X) : -A(X, Y), Q(Y)$ is linear, while $Q(X) : -A(X, Y, Z), Q(Y), Q(Z)$ is not linear.) These assumptions have both

[1] The notions of variable graph and connectivity of variables were introduced in [Ga82].

[2] A conjunctive query is an existentially quantified conjunction of atomic formulas. For example, $\exists x(P(x, z) \wedge Q(z, y))$ is a conjunctive query defining the set of pairs $\langle x, y\rangle$ such that for some z the pair $\langle x, z\rangle$ is in the relation P and the pair $\langle z, y\rangle$ is in the relation Q.

a technical reason, which will see shortly, and a practical reason, since "real-life" programs tend to be connected and linear (cf. [BR86]). We shall show later how to remove these assumptions.

Consider a linear monadic program Π with recursive bodies $\{A_1, \ldots, A_m\}$ and initialization bodies $\{B_1, \ldots, B_n\}$. Intuitively, we construct conjunctive queries as sets of bodies with the appropriate renaming of variables, but it is convenient to view this sets as sequences.

Example 3.1: Let Π consist of the rules:

$$Q(X) :- A(X, Y), Q(Y)$$
$$Q(X) :- B(X)$$

Then C_0 is $B(X^1)$ and C_1 is $A(X^1, Y^1), B(Y^1)$ and C_2 is $A(X^1, Y^1), A(Y^1, Y^2), B(Y^2)$, etc. In every one of these conjunctive queries C_i, all variables except X^1 are existentially quantified. The free variable X^1 is known in database terminology as the ***distinguished variable*** of C_i. ∎

More formally, let A_j^i (resp., B_j^i) be a variant of A_j (resp., B_j), where all the variables carry a superscript i and if $i > 1$ variable X^i is replaced by variable Y^{i-1}. First, view $A_{i_1}^1$ (resp., $B_{i_1}^1$) as a conjunctive query where all variables except for X^1 are existentially quantified and this one body is the *leaf*. Inductively, suppose that C is a conjunctive query built from bodies of Π, $A_{i_k}^k$ is a recursive body that is the leaf of C, and $A_{i_{k+1}}$ (resp., $B_{i_{k+1}}$) is a body of Π. Since $A_{i_k}^k$ is a recursive body it contains variable Y^k, which by our convention is also in $A_{i_{k+1}}^{k+1}$ (resp., $B_{i_{k+1}}^{k+1}$). Then $C, A_{i_{k+1}}^{k+1}$, (resp., $C, B_{i_{k+1}}^{k+1}$), can be viewed as a conjunctive query C', where all variables except for X^1 are existentially quantified and where $A_{i_{k+1}}^{k+1}$ (resp., $B_{i_{k+1}}^{k+1}$) is the leaf of C'. The Q-expansions of Π are a subset of the set of conjunctive queries constructed in this fashion whose leaf is an initialization body.

Viewing conjunctive queries as sequences of bodies suggests a language-theoretic notation. Consider the monadic program Π of the previous construction. The ***recursive alphabet*** $\Sigma_{\Pi,r}$ is the set $\{a_1, \ldots, a_m\}$, and ***initialization alphabet*** $\Sigma_{\Pi,i}$ is the set $\{b_1, \ldots, b_n\}$. The ***alphabet*** Σ_Π is the union of $\Sigma_{\Pi,r}$ and $\Sigma_{\Pi,i}$. We denote the conjunctive query that is the sequence of bodies $A_{i_1}^1, \ldots, A_{i_k}^k$, by the word $a_{i_1} \ldots a_{i_k}$, and we denote the conjunctive query that is a sequence of bodies $A_{i_1}^1, \ldots, A_{i_k}^k, B_{i_{k+1}}^{k+1}$, by the word $a_{i_1} \ldots a_{i_k} b_{i_{k+1}}$. That is, every conjunctive query we constructed in the previous paragraph is denoted by a word of $(\Sigma_{\Pi,r})^*$ or a word of $(\Sigma_{\Pi,r})^*(\Sigma_{\Pi,i})$. Thus, the set of Q-expansions of a program Π can be viewed as a language over (Σ_Π), denoted $expand(\Pi, Q)$ (see Proposition 3.4). In Example 3.1, $\Sigma_{\Pi,r} = \{a\}$ and $\Sigma_{\Pi,i} = \{b\}$. So C_0 is denoted by the word b, C_1 is denoted by the word ab, C_2 is denoted by the word aab, etc. Thus, $expand(\Pi, Q) = a^*b$.

Let Q be an IDB in Π, and let C be a conjunctive query denoted by a word of $(\Sigma_{\Pi,r})^*$ or a word of $(\Sigma_{\Pi,r})^*(\Sigma_{\Pi,i})$. We say that C is ***Q-accepted*** by Π, if there is some Q-expansion C_i such that there is a containment mapping from C_i to C. Equivalently, C is ***Q-accepted*** by Π, if $X^1 \in Q_\Pi^\infty(C)$, where C is viewed as a database [CM77]. Let $accept(\Pi, Q)$ be the language over alphabet Σ_Π consisting of the word which denote conjunctive queries

Q-accepted by Π. The language $notaccept(\Pi, Q)$ is the complement of $accept(\Pi, Q)$. Note that $expand(\Pi, Q) \subseteq accept(\Pi, Q)$.

The importance of these languages is made clear by the following language-theoretic characterization of unboundedness. We note that the proposition does not hold for disconnected programs.

Proposition 3.2: *Let Π be a linear connected monadic program, and let Q be an IDB predicate of Π. Then Q is unbounded in Π if and only if for every $k > 0$, there is a word w in $expand(\Pi, Q)$ such that $|w| > k$ and the prefix of w of length k is in $notaccept(\Pi, Q)$.*

The intuition behind Proposition 3.2 is as follows. Q is bounded in Π if and only if there exists some $k \geq 0$ such that for every Q-expansion C_n there is some Q-expansion C_i, $i \leq k$, and there is a containment mapping from C_i to C_n. If Π is connected, then all the Q-expansions are connected. Suppose that the diameter of Π is d. If $i \leq k$, $n > dk$, and there is a containment mapping from C_i to C_n, then there is also a containment mapping from C_i to the prefix of C_n that contains only dk bodies. This follows from the fact that the image of a connected conjunctive query under a containment mapping must also be connected.

Proposition 3.2 motivates the following definition. Let L_1 and L_2 be languages over some alphabet Σ. We say that L_1 has the ***unbounded prefix property with respect to*** L_2 if for every $k > 0$ there is a word $w \in L_1$ such that $|w| > k$ and the prefix of w of length k is in L_2. Thus, Π is unbounded if and only if for some IDB predicate Q of Π the language $expand(\Pi, Q)$ has the unbounded prefix property with respect to $notaccept(\Pi, Q)$. The following proposition states that the unbounded prefix property is decidable for regular languages.

Proposition 3.3: *Given two nondeterministic finite automata M_1 and M_2 over an alphabet Σ, determining whether $L(M_1)$ has the unbounded prefix property with respect to $L(M_2)$ can be tested in nondeterministic space logarithmic in the size of the input.*

It follows from Propositions 3.2 and 3.3 that we can prove the decidability of boundedness for connected linear monadic programs by showing that $expand(\Pi, Q)$ and $notaccept(\Pi, Q)$ are regular. We first show that $expand(\Pi, Q)$ is regular.

Proposition 3.4: *Let Π be a linear monadic program, and let Q be an IDB predicate of Π. Then $expand(\Pi, Q)$ is a regular languages. Moreover, there is an automaton for $expand(\Pi, Q)$ whose size is linear in the length of Π.*

Proof: Intuitively, the program Π can be viewed as a right-linear context-free grammar that generates the set of Q-expansions. Let $\mathcal{P}$ be the set of IDB predicates in Π. We take $S = \mathcal{P} \cup \{s_f\}$ to be the state set, Q be the start state, and s_f to be the acceting state. Finally, $\rho : S \times \Sigma_\Pi \rightarrow 2^S$ is defined such that $Q'' \in \rho(Q', a_i)$ if Π includes a recursive rule $Q''(X) : -A_i, Q'(Y)$, and $s_f \in \rho(Q', b_i)$ if Π include an initialization rule $Q'(X) : -B_i$. The automaton $(\Sigma_\Pi, S, \{Q\}, \rho, \{s_f\})$ accepts the language $expand(\Pi, Q)$. ∎

Showing that $notaccept(\Pi, Q)$ is also regular is not so easy. The trick is to consider a bigger alphabet. An *enriched body* of Π is a conjunction of atomic formulas of the form $A, Q_1(X_1), \ldots, Q_k(X_k)$, where A is a body of Π, $X_1, \ldots, X_k$ are variables of A, and $Q_1, \ldots, Q_k$ are IDB predicates of Π. We can build conjunctive queries from enriched bodies in a manner analogous to the manner we built conjunctive queries from bodies. This results in *enriched queries.* An enriched query is *legal* if it contains all IDB facts implied from the other facts by the rules of Π; that is, a legal enriched query is an enriched query closed under the rules of Π.

Example 3.5: Consider the program of Example 3.1. $A(X, Y)$ is a recursive body and $B(X)$ is an initialization body. $A(X, Y), Q(X)$ is an enriched body. $B(X^1), Q(X^1)$ is a legal enriched query and $A(X^1, Y^1), B(Y^1), Q(Y^1)$ is an illegal enriched query (in the latter query, $Q(X^1)$ is missing). ∎

For each body A_j (resp., B_j) we assume a fixed enumeration $A_j^1, \ldots$ (resp., $B_j^1, \ldots$) of its enrichments, and we associate letters with all of them. Thus, a_j^i (resp., b_j^i) corresponds to A_j^i (resp., B_j^i). The result is the *enriched alphabet* Σ_Π^e, and words over the enriched alphabet are *enriched words. Legal enriched words* are words that correspond to legal enriched queries. Note that the cardinality of the enriched alphabet is exponential [3] in the size of the program.

There is a natural mapping δ from Σ_Π^e to Σ_Π that maps a_j^i (resp., b_j^i) to a_j (resp., b_j). A word w^e over Σ_Π^e is an *enrichment* of a word w over Σ_Π if $w = \delta(w^e)$. The following proposition establishes a connection between enrichments and $notaccept(\Pi, Q)$.

Proposition 3.6: *Let Π be a linear monadic program, and let Q be an IDB predicate of Π. Then w is in $notaccept(\Pi, Q)$ if and only if there is a legal enrichment w^e of w such that the corresponding enriched query C^e does not contain $Q(X^1)$.*

The idea is that in order to decide if a word w is $notaccept(\Pi, Q)$ an automaton guesses a legal enrichment w^e of w and checks whether the condition of Proposition 3.6 holds. To do this we need the following result.

Proposition 3.7: *Let Π be a linear monadic program. Then the set of legal enriched words is regular. Moreover, there is an automaton whose size is exponential in the length of Π that accepts precisely the legal enriched words.*

To see why Proposition 3.7 is true it is illuminating to consider the case that the program Π is connected and has diameter d. It is not hard to see that an enriched word w^e is legal if all its subwords of length at most $2d + 1$ are legal. Clearly, this can be checked by deterministic automaton with $O(|\Sigma_\Pi^e|^{2d})$ states. This idea is the basis of the argument for disconnected programs.

[3] By exponential in n we mean $2^{p(n)}$ for some polynomial p.

Corollary 3.8: *Let Π be a linear monadic program and let Q be an IDB predicate of Π. Then notaccept(Π, Q) is regular. Moreover, there is an automaton for notaccept(Π, Q) whose size is exponential in the length of Π.*

From Propositions 3.2, 3.3, and 3.4 and Corollary 3.8 it follows that linear connected program unboundedness is decidable in polynomial space.

Theorem 3.9: *The predicate and program boundedness problem for linear connected monadic programs are solvable in polynomial space.*

We note that for the case that the program contains only one recursive rule, boundedness is known to be NP-complete [Va88].

4 Connected Programs

In this section we remove the assumption that we are dealing with linear programs. We consider monadic programs with recursive rules of the form

$$Q(X) : -A, Q_1(Y_1), \ldots, Q_n(Y_n)$$

where A is a set of EDB formulas. We say here that the *branching* of A is n. In this case, the Q-expansions are conjunctive queries which are constructed naturally as *trees* of bodies rather than sequences of bodies. Thus, to handle this case, we have to use tree languages and tree automata.

Example 4.1: Let Π consist of the rules:

$$Q(X) : -A(X, Y, Z), Q(Y), Q(Z)$$
$$Q(X) : -B(X)$$

Then C_0 is $B(X^1)$ and C_1 is a tree with $A(X^1, Y^1)$ at the root and $B(Y^1)$ and $B(Z^1)$ as leaves. ∎

Let N denote the set of positive integers. We use the variables x and y to 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 $x \in \tau$. 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. x is a *leaf* if it has no children. By definition, the empty sequence λ is a member of every tree; it is called the *root*. The *depth* of a node x is its length (as a sequence). The depth of a tree τ, denoted *depth*(τ), is the maximal depth of a node of τ. The *k-prefix* of a tree is the subtree that consists of all nodes whose depth is at most k.

A *tree alphabet* is an alphabet Σ where every letter has a nonnegative *arity*. A *Σ-labelled tree* is a pair (τ, π), where τ is a tree and $\pi : \tau \rightarrow \Sigma$ assigns to every node a label, such that if $\pi(x) = a$ and a is k-ary and x is not a leaf, then the children of x are precisely $x1, \ldots, xk$. In particular, if $\pi(x)$ is 0-ary, then x must be a leaf. A labelled tree is *full*

if all its leaves have arity 0. The *k-prefix* of a labelled tree is defined in the obvious way. We often refer to *labelled trees* as *trees*. Our intention will be clear from the context. We denote by $tree(\Sigma)$ the set of all Σ-labelled trees.

Consider a monadic program Π with recursive and initialization bodies $\{A_1, \ldots, A_m\}$. Then these bodies can be used to build conjunctive queries in a manner that is an obvious extension of the linear case. Let $A_j^{i,j}$ be a variant of A_j, where all the variables carry superscripts i, j. Then the singleton tree $\{\lambda\}$ labelled by $A_{i_1,1}^{1,1}$ can be viewed as a conjunctive query C, where all variables except for $X^{1,1}$ are existentially quantified. Inductively, suppose that C is a conjunctive query built from bodies of Π, $A_{i_{j,k}}^{j,k}$ is a recursive body with branching l that is the label of a leaf x of C, $A_{i_{1,k+1}}, \ldots, A_{i_{l,k+1}}$ are bodies of Π, and θ is the substitution where $X^{p,k+1}$ is replaced by $Y_p^{j,k}$ for $1 \leq p \leq l$. Then the conjunctive query $\theta(C_k), A_{i_{1,k+1}}^{1,k+1}, \ldots, A_{i_{l,k+1}}^{l,k+1}$ can be viewed as a tree obtained from C in the following way: we add $x1, \ldots, xl$ as children of x and label them by $A_{i_{1,k+1}}^{1,k+1}, \ldots, A_{i_{l,k+1}}^{l,k+1}$, and then we replace $X^{p,k+1}$ by $Y_p^{j,k}$ for $1 \leq p \leq l$. All the expansions of Π have this form, where the leaves are initialization bodies.

As in the previous section, we define an alphabet Σ_Π by associating a letter with the body of every rule of a program Π. The arity of the letter is the branching of the corresponding IDB predicate; in particular, with initialization bodies we associate 0-ary letters. The language $expand(\Pi, Q)$ over alphabet Σ_Π consists of the full trees which denote Q-expansions of Π. The language $accept(\Pi, Q)$ over alphabet Σ_Π consists of the trees which denote conjunctive queries Q-accepted by Π. The language $notaccept(\Pi, Q)$ is the complement of $accept(\Pi, Q)$. Again, $expand(\Pi, Q) \subseteq accept(\Pi, Q)$.

The following result generalizes Proposition 3.2. We note that the proposition does not hold for disconnected programs.

Proposition 4.2: *Let Π be a connected monadic program. Then Π is unbounded if and only if for some IDB predicate Q of Π and for every $k > 0$, there is a tree (τ, π) in $expand(\Pi, Q)$ such that $depth(\tau) > k$ and the k-prefix of (τ, π) is in $notaccept(\Pi, Q)$.*

Proposition 4.2 motivates the following definition. Let L_1 and L_2 be tree languages over some alphabet Σ. We say that L_1 has the *unbounded prefix property with respect to* L_2 if for every $k > 0$ there is a tree $t \in L_1$ such that $depth(t) > k$ and the k-prefix of t is in L_2. Thus, Π is unbounded if and only if for every IDB predicate Q of Π the language $expand(\Pi, Q)$ has the unbounded prefix property with respect to $notaccept(\Pi, Q)$.

To generalize Proposition 3.3, we have to define *regular tree languages*: these are the tree language accepted by *tree automata*. A tree automaton M is a tuple $(\Sigma, S, S_0, \rho, F)$, where Σ is a tree alphabet, S is a set of states, $S_0 \subseteq S$ is a set of starting states, $F \subseteq S$ is a set of accepting states, and $\rho : S \times \Sigma \rightarrow 2^{S^*}$, is a transition function such that if $a \in \Sigma$ has arity l, then $\rho(s, a) \subseteq S^l$. A *run* $r : \tau \rightarrow S$ of M on a Σ-labelled tree (τ, π) is a labeling of τ by states of M, such that the root is labelled by a start state and the transitions obey the transition function ρ; that is, $r(\lambda) \in S_0$, and if x is not a leaf and $\pi(x)$ is k-ary, then $\langle r(x1), \ldots, r(xk) \rangle \in \rho(r(x), \pi(x))$. If for every leaf x of τ there is a tuple

$\langle s_1, \ldots, s_l \rangle \in \rho(r(x), \pi(x))$. such that $\{s_1, \ldots, s_l\} \subseteq F$, then r is *accepting*. M *accepts* (τ, π) if it has an accepting run on (τ, π).

Proposition 4.3: *Given two tree automata M_1 and M_2 over an alphabet Σ, determining whether $L(M_1)$ has the unbounded prefix property with respect to $L(M_2)$ can be tested in polynomial time in the size of the input.*

It follows from Propositions 4.2 and 4.3 that we can prove the decidability of boundedness for connected monadic programs by showing that $expand(\Pi, Q)$ and $notaccept(\Pi, Q)$ are regular tree languages. This can be done by extending the techniques developed for linear programs.

Proposition 4.4: *Let Π be a monadic program, and let Q be an IDB predicate of Π. Then $expand(\Pi, Q)$ and $notaccept(\Pi, Q)$ are regular tree languages. Moreover, there is a tree automaton for $expand(\Pi, Q)$ whose size is linear in the length of Π, and there is a tree automaton for $notaccept(\Pi, Q)$ whose size is doubly exponential in the length of Π.*

Propositions 4.2, 4.3, and 4.4 yield a doubly exponential upper bound for boundedness of monadic programs.

Theorem 4.5: *The predicate and program boundedness problems for connected monadic programs are solvable in doubly exponential time.*

5 Program Boundedness

We first show by way of an example that program boundedness and predicate boundedness indeed differ.

Example 5.1: Let Π consist of the rules:

$$Q'(X) :- A(X, Y), Q(Z)$$
$$Q(X) :- A(X, Y), Q(Y)$$
$$Q(X) :- B(X)$$

It is easy to see that Q' is bounded, while Q is unbounded. ∎

In this section we show how to extend the results of the previous sections by removing the assumption of connectedness. We first prove a general result, which characterizes boundedness of arbitrary programs (not necessarily monadic) in terms of boundedness of connected programs. This characterization, together with the techniques developed in the previous section, gives us a decision procedure for boundedness of general monadic programs.

Consider a (recursive or non-recursive) rule r of the form $Q(X_1, \ldots, X_k) :- R$, where R is a set of EDB and IDB formulas. Let $G_r = (V, E)$ be an undirected graph, where V is the set of all variables appearing in either side of the rule, and $\{X, Y\} \in E$ iff X, Y occur

in the same formula of R. Let $G_0 = (V_0, E_0)$ be the subgraph of G_r induced by the set V_0 of nodes *reachable* from $X_1, \ldots, X_k$, and let $G_i = (V_i, E_i)$, where $i = 1, \ldots, m$, $m \geq 0$, be the *connected components* of $G_r - G_0$. We now rewrite the rule r as $Q(X_1, \ldots, X_k) : -R_0, R_1, \ldots, R_m$, where an (EDB or IDB) formula $E(Z_1, \ldots, Z_l)$ is in R_i iff $Z_1, \ldots, Z_l \subseteq V_i$. We call $R_1, \ldots, R_m$ the *triggers*. Note that it is possible that R_0 is empty and that it is also possible that there are no triggers.

Let Π be a program, H a collection of ground atomic formulas or facts (over the EDB and the IDB symbols) and T a trigger (of some rule) of Π. We say that H *realizes* the trigger T iff the variables in T can be instantiated so that all the formulas of T become facts in H.

Let $S = (S_1, S_2)$ be a partition of the triggers of Π into two sets S_1, S_2. We construct a program Π_S as follows. For each rule $Q(X_1, \ldots, X_k) : -R_0, R_1, \ldots, R_m$ of Π (where $R_1, \ldots, R_m$ are the triggers) such that $R_1, \ldots, R_m \subseteq S_1$, we put in program Π_S the rule $Q(X_1, \ldots, X_k) : -R_0$. Thus, if any of the triggers is not in S_1, then the rule is discarded.

The new program Π_S is related to Π as follows:

Lemma 5.2: *Let D be a database and $m \geq 0$ such that $\Pi^m(D)$ realizes every trigger in S_1 and $\Pi^\infty(D)$ realizes no trigger in S_2. Then for every i we have $\Pi^i(D) \subseteq \Pi^i_S(D) \subseteq \Pi^{i+m}(D)$.*

Notice that the program Π_S may contain rules with empty bodies. We now derive another program Π'_S by eliminating such rules: that is, whenever $Q(X_1, \ldots, X_k) : -$ is a rule, we eliminate it, and we eliminate all formulas of the form $Q(Z_1, \ldots, Z_k)$ in all the remaining rules. The following result relates boundedness of Π to boundedness of Π'_S.

Theorem 5.3: *Let Π be a program. Π is unbounded iff there are a constant m, a partition $S = (S_1, S_2)$ of the triggers of Π, and an infinite sequence of databases D_k, $k = 1, \ldots,$ such that:*

1. *For all k, $\Pi^m(D_k)$ realizes all triggers in S_1, $\Pi^\infty(D_k)$ realizes no trigger in S_2.*
2. *Π'_S is unbounded on the set of databases $\{D_k\}$.*

The "if" direction of this result follows easily from Lemma 5.2. The "only-if" direction is nontrivial. We show by a *compactness* argument that, if Π is unbounded, then there are $S, m, \{D_k\}$ such that condition (1) is satisfied and Π is unbounded on $\{D_k\}$. From this it follows by Lemma 5.2 that Π'_S is unbounded on $\{D_k\}$.

Now observe that, if Π is a monadic program, then Π'_S is in fact *connected*. This makes it possible to use the techniques of Section 3 to prove the following generalization of Proposition 4.2.

Proposition 5.4: *Let Π be a monadic program. Then Π is unbounded iff there is a partition $S = (S_1, S_2)$ of the triggers of Π such that:*

1. *For some IDB predicate Q of Π'_S and for every $k > 0$, there is a tree (τ, π) in $expand(\Pi'_S, Q)$ such that $depth(\tau) > k$ and the k-prefix of (τ, π) is in $notaccept(\Pi'_S, Q)$; moreover, $\Pi^\infty(\tau, \pi)$ realizes no trigger in S_2.*

2. *There is a database D such that $\Pi^\infty(D)$ realizes all the triggers in S_1 and no trigger in S_2.*

The conditions of Proposition 5.4 can be tested by appropriately extending the automata-theoretic tools of the previous section. We thus show:

Theorem 5.5: *The program boundedness problem for monadic programs is solvable in doubly exponential time and is EXPTIME-hard. The program boundedness problem for linear monadic programs is PSPACE-complete.*

Proof: It remains to prove lower bounds. We focus here on the linear case. The proof is by generic reduction from polynomial-space Turing machines. Given a machine M and input x, we construct a program Π with a single IDB predicate $FING$ such that $FING$ is bounded precisely when M accepts x. We assume without loss of generality that if M does not accept x, then M diverges on x. Let M be a $SPACE(p(n))$-bounded Turing machine, for some polynomial p, with an alphabet Γ and set S of states. The set $\Delta = \Gamma \cup (S \times \Gamma)$ is called the *extended* alphabet of M. It is well-known that configurations of M can be described by words in Δ^*. Let $\Delta' = \Delta \cup \#$, where $\#$ is a new symbol. We can encode a computation of M as a word in Δ', where $\#$ symbols mark the beginning of new configurations.

The idea is to let the database encode the word encoding a prefix of the computation of M. Every element in the database encodes a letter in that word. We have an EDB unary predicate q_a for every letter $a \in \Delta$; an element x encodes the letter a precisely when $x \in q_a$ holds. We have a unary relation *first* that encodes the first letter, and a binary relation *succ* that encodes the adjacency relation between letters. Thus, to encode the word *abc*, the database need to contain elements x, y, z such that $q_a = \{x\}$, $q_b = \{y\}$, $q_c = \{z\}$, $first = \{x\}$, and $succ = \{(x, y), (y, z)\}$. Of course, not all databases would indeed constitute a meaningful encoding. This is a problem we will have to deal with.

The program P will have one monadic IDB predicate $FING$. The idea is that to check that a word encodes a legal computation of M it suffice to check triples of letters in corresponding positions in successive configurations. It is well-known that there is a relation $R_M \subseteq \Delta'^6$ such that two configurations are successive if and only for every pair of corresponding triples *abc* and *def* we have $(a, b, c, d, e, f) \in R_M$. One could think of $FING$ as a finger pointing at a position of the computation to check whether the three letters in that position are consistent with the three letter that are $p(n)$ letters to the right.

The program Π has several types of rules: ***encoding rules*** that check that no element of the database encode more than one letter, ***halting rules*** that check whether the computation reaches a halting state, ***error detecting rules*** that check whether the computation encoded by the database is legal, a ***finger pointing rule*** that initializes the pointing finger, and ***finger moving rules*** that move the finger to the next position. The only way the finger can keep being moved is along a legal computation. Thus, the program will be unbounded if and only if there are arbitrarily long legal computations. But that is possible precisely when M rejects x. Details will be given in the full paper.

For the nonlinear case, an exponential lower bound can be obtained by extending the technique developed for the linear case (the reduction is from *alternating* polynomial-space Turing machines [CKS81]). ∎

We note that our lower bounds use disconnected programs. The only known lower bound for connected programs is an NP-hardness bound proved in [Va88].

6 Predicate Boundedness

In the previous sections we have established upper bounds for program boundedness and for predicate boundedness of connected programs. In this section we solve the general predicate boundedness problem.

The key to our result is an alternative (to Propositions 3.2 and 4.2) language-theoretic characterization of predicate boundedness. We first need some technical definitions. Recall that a conjunctive query C is *Q-accepted* by Π, if there is some Q-expansion C_i such that there is a containment mapping from C_i to C. Equivalently, C is *Q-accepted* by Π, if $X^1 \in \Pi_Q(C)$, where C is viewed as a database. In Sections 2 and 3 we constructed conjunctive queries as structured (sequences or trees) sets of bodies. If C and C' are conjunctive queries, then C' is a *proper subquery* of C if C' is a proper subset of C when both C and C' are viewed as sets of bodies. We say C is *properly* Q-accepted by Π if there is a proper subquery C' of C that is Q-accepted by Π. Let $proper(\Pi, Q)$ be the language (resp., tree languages) over Σ_Π consisting of the words (resp., trees) that denote conjunctive queries properly Q-accepted by Π.

Proposition 6.1: *Let Π be a monadic program (resp., linear monadic program) and let Q be an IDB predicate of Π. Then Q is bounded if and only if there are only finitely many trees (resp., words) in $accept(\Pi, Q) - proper(\Pi, Q)$.*

Since the finiteness problem is decidable for regular languages and regular tree languages (the problem can be solved in nondeterministic logarithmic space for regular lanaguages and in polynomial time for regular tree languages), our strategy is to prove that $accept(\Pi, Q)$ and $proper(\Pi, Q)$ are regular, and then apply Proposition 6.1. Consider first $accept(\Pi, Q)$. It is regular by Propositions 3.8 and 4.4. This, however, yields automata that are doubly exponential for linear programs, and are triply exponential for nonlinear programs. A better approach is to use *adorned* words and 2-way automata.

We focus here on the linear case. Let Σ be an alphabet and let $k > 0$. We use the notation $\Sigma^{(k)}$ to denote the set of all words of length at most k, i.e, $\bigcup_{i=1}^{i=k} \Sigma^i$. Let $w = a_1, \ldots, a_n$ be a word over Σ. The *d-neighborhood* of a letter a_i, for $1 \leq i \leq n$, in w is the subword $a_j, \ldots, a_i, \ldots, a_k$, where $j = max(1, i - d)$ and $k = min(n, i + d)$. Now *d-adorned* words over Σ are actually words over $\Sigma^{(2d+1)}$. A word $w' = a'_1, \ldots, a'_n$ over $\Sigma^{(2d+1)}$ is a d-adorned word if there is a word $w = a_1, \ldots, a_n$ over Σ such that a'_i is the d-neighborhood of a_i in w for $1 \leq i \leq n$. We also say that w' is the d-adornment of w. Recall that the *diameter d* of Π be the maximal distance between two connected variables in any of the variable graphs for rules of Π.

Lemma 6.2: *Let Π be a linear monadic program with diameter d, and let Q be an IDB predicate in Π. Then there is a 2-way automaton M over $\Sigma_{\Pi}^{(2d+1)}$ such that M accepts the d-adornment w' of w if and only if $w \in accept(\Pi, Q)$. Moreover, the number of states in M is cubic in the length of Π.*

Proof: Let w denote the conjunctive query C. Recall that C is Q-accepted by Π if there is some Q-expansion C_i such that there is a containment mapping from C_i to C. The automaton M guesses C_i, one body at a time, and checks that it maps to C. Since the input to M is the adornment of w, M needs only to carry with it a small amount of information. It turns out that a cubic number of states suffices. ∎

Corollary 6.3: . *Let Π be a linear monadic program and let Q be an IDB predicate of Π. Then, there is an automaton for $accept(\Pi, Q)$ whose size is exponential in the length of Π.*

Proof: We first translate the 2-way automaton M of Lemma 6.2 to a 1-way automaton M'. This involves an exponential blow-up. This automaton still reads adorned words as input. Now we construct an automaton M'' that guesses the neighborhoods, checks that they constitute an adornment, and simulates M'. M'' accepts precisely $accept(\Pi, Q)$ and its size is exponential in the length of Π. ∎

To deal with nonlinear programs two extensions are needed. First, neighborhoods are not subwords any more but subtrees. Note, however, that while the number of word neighborhoods in exponential, the number of tree neighborhoods is doubly exponential. Secondly, we need to use *2-way alternating tree automata.* Two-way alternating tree automata generalize the alternating tree automata of [MS87], which in turn generalize tree automata and the alternating automata of [BL80,CKS81]. We define 2-way alternating tree automata in the appendix, and we prove that they can be converted to tree automata with an exponential blow-up. Using this result we obtain the following:

Proposition 6.4: *Let Π be a monadic program and let Q be an IDB predicate of Π. Then, there is an tree automaton for $accept(\Pi, Q)$ whose size is doubly exponential in the length of Π.*

Finally we need automata for $proper(\Pi, Q)$. For this we use automata that simulate the automata of Corollary 6.3 and Proposition 6.4 after deciding nondeterministically to ignore part of the input. This construction only causes a constant blow-up in the number of states. Note, however, that to apply Proposition 6.1 we need an automaton for the complement of $proper(\Pi, Q)$. This costs us another exponential!

Theorem 6.5: *The predicate boundedness problem for monadic programs is solvable in exponential space in the linear case and in triply exponential time in the general case.*

The only lower bounds we have for predicate boundedness are those we gave in the previous section, that is, PSPACE in the linear case and exponential time in general. We do not know whether predicate boundedness is indeed harder than program boundedness. We remark, however, that while program boundedness of non-monadic programs is complete for Σ_1^0 (r.e.) [GMSV87], we can show that predicate boundedness of general programs is complete for Σ_2^0.

Table 1: The complexity of boundedness and containment for monadic programs

		linear		non-linear	
		upper bound	lower bound	upper bound	lower bound
Program Boundedness		PSPACE	PSPACE	2-EXPTIME	1-EXPTIME
Predicate Boundedness	connected	PSPACE	NP	2-EXPTIME	NP
	disconnected	EXPSPACE	PSPACE	3-EXPTIME	1-EXPTIME
Containment		PSPACE	PSPACE	2-EXPTIME	1-EXPTIME

7 Application to Containment

Our automata-theoretic techniques, as it turns out, are applicable to another question related to optimization of logic programs, namely the question of *containment*. Let Π, Π' be programs with the same EDB predicates, and let Q be an IDB predicate in both Π and Π'. We say that Π *is contained in* Π' with respect to Q if for every database D we have $Q^{\infty}_{\Pi}(D) \subseteq Q^{\infty}_{\Pi'}(D)$. We say that Π *is contained in* Π' if Π is contained in Π' with respect to all IDB predicates of Π. It is shown in [Sh87] that predicate containment for binary programs is undecidable. This result can be strengthened to show that program containment is also undecidable. Using our automata-theoretic techniques we can show that predicate and program containment of monadic programs are decidable. Note that it suffices to prove that predicate containment is decidable.

The first step is to give a language-theoretic characterization of containment.

Proposition 7.1: *Let Π, Π' be programs with the same EDB predicates, and let Q be an IDB predicate in both Π and Π'. Then Π is contained in Π' with respect to Q if and only if $expand(\Pi, Q) \subseteq accept(\Pi', Q)$).*

Using our automata-theoretic techniques we can now obtain:

Theorem 7.2: *The predicate containment problem for monadic programs is solvable in doubly exponential time and is EXPTIME-hard. The predicate containment problem for linear monadic programs is PSPACE-complete.*

8 Concluding Remarks

We have presented results for boundedness and containment of monadic programs. To obtain these results we devised a language-theoretic approach to the problems and developed automata-theoretic tools. Table 1 summarizes our results. As can be seen in the table there are several gaps between the lower and the upper bounds that need to be closed.

In conclusion, we remark that even though we have focused here on monadic programs, several of our techniques apply also to nonmonadic programs and can yield decidability results for boundedness of such programs. We hope that these techniques will become basic tools in the optimization of database logic programs.

A Appendix: Two-way Alternating Tree Automata

If S is a set, then $\mathcal{L}(S)$ denotes the free distributive lattice of S, that is, the closure of S under the binary operations $\vee$ and $\wedge$. We can view $\mathcal{L}(S)$ as the set of positive Boolean formulas over S. We say that an element e of $\mathcal{L}(S)$ is *true* with respect to a set $T \subseteq S$, if e evaluates to 1 when all the elements in T are assigned 1 and all the elements in $S - T$ are assigned 0. Otherwise, e is *false* with respect to T.

Let $[k]$ denote the set $\{-1, 0, \ldots, k\}$. We think of $[k]$ as a set of directions; -1 denote the upward direction, 0 denotes staying in place, and $1, \ldots, k$ denote the k downward directions in a k-ary point on the tree. If $x \in N^*$, $i \in N$, and $j \in [k]$, then we take the convention that $xij = x$ if $j = -1$ and $xij = xi$ if $j = 0$.

An intuitive way to view tree automata is as recursive processes. Suppose $M = (\Sigma, S, s_0, \rho, F)$ and $\rho(s, a) = \{(t_1, t_2), (t_3, t_4)\}$. Then whenever the automaton M is in state s and it reads the letter a, then it calls itself recursively either in state t_1 on the left child and state t_2 on the right child or in state t_3 on the left child and t_4 on the right child. Two-way alternating tree automata can also be viewed as recursive processes, but they have more powerful recursive calls.

A 2-way alternating tree automaton M is a tuple $(\Sigma, S, S_0, \rho, F)$, where Σ is a tree alphabet, S is a set of states, $S_0 \subseteq S$ is the starting state set, $F \subseteq S$ is the accepting state set, and ρ is a transition function defined on $S \times \Sigma$ such that if $a \in \Sigma$ has arity l, then $\rho(s, a) \in \mathcal{L}(S \times [l])$. An element (s, i) of $S \times [l]$ is a *transition* into a state s in direction i. Suppose $\rho(s, a) = ((t_1, 1) \wedge (t_3, 1)) \vee (t_3, -1) \wedge (t_4, 2))$. Then whenever the automaton M is in state s and it reads the letter a, then it calls itself recursively either in states t_1 and t_2 on the left child or in state t_3 on the parent and t_4 on the right child. More formally, a run r of M over a Σ labelled tree (τ, π) is a tree labelled by elements of $S \times (\{-1, 0\} \cup N) \times N^*$. Intuitively, a label (s, j, x) denotes a transition into a state s in direction i onto a node x. The requirement from r are as follows:

1. The root of r is labelled by (s_0, i, λ) with $s_0 \in S_0$.
2. If a node α is labelled by (s, i, x) and x is not a member of τ, then α is a leaf of r.
3. If a node α is label by (s, i, x), $x \in \tau$, $\pi(x)$ is k-ary, and a child of α is labelled by (t, j, y), then $j \in [k]$ and $y = xj$.
4. If a node α is labelled by (s, i, x), $x \in \tau$, and the children of α are labelled by $(s_1, j_1, y_1), \ldots, (s_k, j_k, y_k)$, then $\rho(s, \pi(x))$ is true with respect to $\{(s_1, j_1), \ldots, (s_k, j_k)\}$.

Note that the tree r is distinct from the tree τ; it is the *computation* tree of the automaton on (τ, π). The run r is accepting if whenever a leaf α is labelled by (s, i, x), then $s \in F$.[4] M accepts (τ, π) if it has an accepting run. The set of trees accepted by M is denoted $trees(M)$.

Two-way alternating tree automata are not more expressive than tree automata. We first, show that we can build a tree automaton that is the complement of a given 2-way alternating tree automaton.

[4] We could also require that x is a leaf of τ. Since this requirement does not have an effect on our results, we chose not to assume it here.

Theorem A.1: *Let $M = (\Sigma, S, S_0, \rho, F)$ be a 2-way alternating tree automaton. Then there is a tree automaton M' whose size is exponential in the size of M such that $trees(M') = trees(\Sigma) - trees(M)$.*

Proof: We first define a tree τ' in the following way: τ' includes all the nodes of τ, and if x is leaf of τ and $\pi(x)$ is l-ary, then add to τ' the children $x1, \ldots, xl$. The claim in the theorem follows from the fact M does not accept a tree (τ, π) if and only if there exists a labelling $\xi : \tau' \rightarrow 2^S$ such that the following conditions hold:

1. $S_0 \subseteq \xi(\lambda)$.
2. For all $x \in \tau'$ we have that $\xi(x) \cap F = \emptyset$.
3. For all $x \in \tau$ and $s \in \xi(X)$, if $\pi(x)$ is l-ary, then there is a set $T \subseteq S \times [l]$ such that $\rho(s, \pi(x))$ is false with respect to $(S \times [l]) - T$ and $t \in \xi(xj)$ for all $\langle t, j \rangle \in T$.

The tree automaton M' "guesses" a labelling $\xi : \tau' \rightarrow 2^S$ and checks that is satisfies the above conditions. ∎

Since it is known that tree automata can be complemented, it follows that every 2-way alternating tree automaton has an equivalent tree automaton. Since, however, the complementation of tree automata involves an exponential blow up, a translation from 2-way alternating tree automata to tree automata that requires two complementations would involve a doubly exponential blow up. Fortunately, we can do better.

Theorem A.2: *Let $M = (\Sigma, S, S_0, \rho, F)$ be a 2-way alternating tree automaton. Then there is an tree automaton M'' whose size is exponential in the size of M such that $trees(M'') = trees(M)$.*

To prove Theorem A.2 we use a game-theoretic interpretation of acceptance (cf. [MS87]). Given a 2-way alternating tree automaton M and an input tree (τ, π), we define a game $G(M, \tau, \pi)$ between players P_a and P_r. Intuitively, P_a plays for acceptance, while P_r plays for rejection. M accepts (τ, π) precisely when P_a has a winning strategy in $G(M, \tau, \pi)$. It can be shown that if P_a has a winning strategy, then it has "forgetful" strategy, i.e., a strategy that depends only on the current position of the game and not the history of the game (cf. [GH82]). The existence of such a strategy can be checked by a tree automaton. ∎

Remark A.3: The reader should note that the class 2-way automata on words is a subclass of the class of 2-way alternating tree automata. Thus, Theorem A.2 generalizes the result in [RS59,Sh59] that 2-way automata are not more expressive than 1-way automata. Also, Theorem A.1 tells us that it is possible to obtain a 1-way automaton that is the complement of a given 2-way automaton with only an exponential blow-up (a naive approach would first translate the 2-way automaton to a 1-way automaton and then apply complementation, causing a doubly exponential blow-up). ∎

Acknowledgements. We'd like to thank Ron Fagin for helpful comments on a previous draft of this paper.

B References

[AP87] Afrati, F., Papadimitriou, C.H.: The parallel complexity of simple chain queries. *Proc. 6th ACM Symp. on Principles of Database Systems*, San Diego, 1987, pp. 210–213.

[ASU79] Aho, V.H., Sagiv, Y., Ullman, J.D.: Efficient optimization of a class of relational expressions. *ACM Trans. on Database Systems* 4(1979), pp. 435–454.

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

[BKBR87] Beeri, C., Kanellakis, P.C., Bancilhon, F., Ramakrishnan, R.: Bounds on the propagation of selection into logic programs. *Proc. 6th ACM Symp. on Principles of Database Systems*, San Diego, 1987, pp. 214–226.

[BL80] Brzozowski, J.A., Leiss, E.: On equations for regular languages, finite automata, and sequential networks. *Theoretical Computer Science* 10(1980), pp. 19–35.

[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* 1(1985), pp. 1–15.

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

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

[CM77] Chandra, A.K., Merlin, P.M.: Optimal implementation of conjunctive queries in relational databases. *Proc. 9th ACM Symp. on Theory of Computing*, Boulder, 1977, pp. 77–90.

[Co74] Cook, S.A.: An observation on time-storage trade off. *J. Computer and System Sciences* 9(1974), pp. 308–316.

[Fa75] Fagin, R.: Monadic generalized spectra. *Zeitschr. f. math. Logik und Grunlagen d. Math.* 21(1975), pp. 89–96.

[Ga82] Gaifman, H.: On local and non-local properties. *Proc. Logic Colloquium* (J. Sterne, ed.), North-Holland, 1981, pp. 105–132.

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

[GMSV87] Gaifman, H., Mairson, H., Sagiv, Y., Vardi M.Y.: Undecidable optimization problems for database logic programs. *Proc. 2nd IEEE Symp. on Logic in Computer Science*, Ithaca, 1987, pp. 106–115.

[GR82] Gurevich, Y., harrington, L.: Trees, automata, and games. *Proc. 14th ACM Symp. on Theory of Computing*, San Francisco, 1982, pp. 60–65.

[HN84] Henschen, L.J., Naqvi, S.A.: On compiling queries in recursive first-order databases. *J. ACM* 31(1984), pp. 47–85.

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

[Io85] Ioannidis, Y.E.: A time bound on the materialization of some recursively defined views. *Proc. 11th Int'l Conf. on Very Large Data Bases*, Stockholm, 1985, pp. 219–226.

[Ka86] Kanellakis, P.C.: Logic programming and parallel complexity. in *Foundations of Deductive Databases and Logic Programming*, J. Minker ed., (to appear).

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

[MS87] Muller, D.E., Schupp, P.E.: Alternating automata on infinite trees. *Theoretical Computer Science* 54(1987), pp. 267–276.

[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.

[MW88] Maier, D., Warren, D.S.: *Computing with Logic: Logic Programming with Prolog*, Benjamin Cummings, 1988.

[Na86] Naughton, J.F.: Data independent recursion in deductive databases. *Proc. 5th ACM Symp. on Principles of Database Systems*, Cambridge, 1986, pp. 267–279. Full version - Stanford University Technical Report STAN-CS-86-1102, to appear in J. Computer and System Sciences.

[NS87] Naughton, J.F., Sagiv, Y.: A decidable class of bounded recursions. *Proc. 6th ACM Symp. on Principles of Database Systems*, San Diego, 1987, pp. 227–236.

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

[Sa85] Sagiv, Y.: On computing restricted projections of representative instances. *Proc. 4th ACM Symp. on Principles of Database Systems*, Portland, 1985, pp. 171–180.

[Sh59] Shepherdson, J.C.: The reduction of two-way automata to one-way automata. *IBM J. Research and Development*, 3(1959), pp. 199–201.

[Sh87] Shmueli, O.: Decidability and expressiveness aspects of logic queries. *Proc. 6th ACM Symp. on Principles of Database Systems*, San Diego, 1987, pp. 237-249.

[St82] Streett, R.S.: Propositional dynamic logic of looping and converse is elementarily decidable. *Information and Control*, 54(1982), pp. 121–141.

[Ul85] Ullman, J.D.: Implementation of logical query languages for databases. *ACM Trans. on Database Systems* 10(1985), pp. 289–321.

[UV86] Ullman, J.D., Van Gelder, A.: Parallel complexity of logical query programs. *Proc. 27th IEEE Symp. on Foundations of Computer Science*, Toronto, 1986, pp. 438–454.

[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 Undecidablity Results for Boundedness of Linear Recursive Queries. *Proc. 7th ACM Symp. on Principles of Database Systems*, Austin, March 1988.

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