Introduction

Considerable attention recently is directed to the undecidability of logic programs (known also as Horn-clause programs) without function symbols [RB86, CH85, CM78, HN84, U85]. This has been recognized for some time that first-order database query languages are lacking in

decidability for monadic programs is decidable.

theory of two-way alternating tree automata. We also use our techniques to show that

numbers, and logic. In particular, one of the tools we develop is a

complexity of logic programs (the non-reconstructive predicates can have arbitrary

the recursivity predicates are monadic (the non-reconstructive predicates are decidable for monadic programs) 'i.e., programs whose

show that both boundedness is decidable for monadic programs, i.e., programs whose

a given Datalog program is bounded is undecidable, even for binary programs. We

database query languages. It is possible to eliminate recursion from a Datalog program

Abstract

Ibm Almaden Research Center

Moshé Y. Vardi

Hebrew University of Jerusalem

Haim Gaifman

Brown University

Piers C. K. Cheung

Stavros S. Cosmadakis

Preliminary Report

for Database Logic Programs

Decidability Optimization Problems

Computer Science

RI 6195 (60985) 3/22/88
and hence, that the recursion is "bounded." More formally:

\[(\lambda \; Z) \; \text{ knows } (Z \cdot X) \quad \text{access} (X) \quad \text{source} (X) \quad \text{title} (X) \quad \text{tuple} (X)\]

on the other hand, the program:

\[(\lambda \; Z) \; \text{ knows } (Z \cdot X) \quad \text{access} (X) \quad \text{source} (X) \quad \text{title} (X) \quad \text{tuple} (X)\]

In this example, the query (\(\lambda \; Z\) Z) can be changed to (\(\lambda \; Z\) Z) P"y"s, yielding an equivalent recursion-free program. On the other hand, the program:

\[(\lambda \; Z) \; \text{ knows } (Z \cdot X) \quad \text{access} (X) \quad \text{source} (X) \quad \text{title} (X) \quad \text{tuple} (X)\]

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 "p" can be made in a single step, whereas in the latter, recursion is required.

Let us now illustrate recursion elimination using a second Datalog example, from [Nag86].

We refer to the literature for a detailed definition of Datalog [MVW88] and use the following example to illustrate its syntax and semantics:

\[(Z \cdot Y) \quad \text{access} (X) \quad \text{source} (X) \quad \text{title} (X) \quad \text{tuple} (X)\]

In this example, "title" and "source" are extensional datatypes over finite structures.

But this is essentially a fragment of existential logic [MOT74].
problem of database programming. Our results, in turn, are a consequence of the following

Theorem 1. The problem of database programming is NP-complete.

Proof. Let \( \mathcal{P} \) be the collection of facts about the

database. Let \( \mathcal{Q} \) be the collection of facts about the


database. Let \( \mathcal{R} \) be the collection of facts about the


database. Let \( \mathcal{S} \) be the collection of facts about the


database. Let \( \mathcal{T} \) be the collection of facts about the


database. Let \( \mathcal{U} \) be the collection of facts about the


database. Let \( \mathcal{V} \) be the collection of facts about the


database. Let \( \mathcal{W} \) be the collection of facts about the


database. Let \( \mathcal{X} \) be the collection of facts about the


database. Let \( \mathcal{Y} \) be the collection of facts about the


database. Let \( \mathcal{Z} \) be the collection of facts about the


database.

Theorem 2. The problem of database programming is

NP-complete.

Proof.

A database contains information about several domains of interest. To express this information, we

use a programming language. The language includes

operators for

selecting

sets

of objects

such

as

"{a, b, c}";

operators for

declaring

functions,

such

as

"f(x) = x + 1";

operators for

defining

predicates,

such

as

"(x > 0)\land (x < 10)";

and

operators for

defining

rules,

such

as

"if x > 0 then y = x + 1".

We can express a wide range of

database

queries using this language.

For example, to find all employees whose names start

with "A", we could write

\[ \text{select all names starting with "A" from employees} \]

or, equivalently,

\[ \text{find the set of all x such that \( x \) starts with "A" and \( x \) is an employee} \]

Using this language, we can define complex

data structures such as

"a set of all employees who have worked in the same department";

"a function that computes the average salary of all employees who work in the same department";

and

"a rule that enforces the necessity of having a job for all employees".

These examples illustrate the power and flexibility of

our language. With a little bit of creativity, we can express almost any

database query in this language.

However, there are some limitations to the language. For

example, it cannot directly express queries that involve

temporal information, such as

"find all employees who worked in the company before 1990";

"find all employees who have worked in the company for at least 5 years";

and

"find all employees who have worked in the company for at least 5 years and have a salary of at least $50,000".

Despite these limitations, the language is still quite powerful. It allows us to express a wide variety of

queries in a concise and readable way. Moreover, it is

relatively easy to implement the language in a computer

program.

We have implemented several versions of the

language, each with different features and capabilities. In

general, the more powerful the language, the more difficult

it is to implement. However, even with some limitations,

we have been able to implement a reasonably effective

system.

The language is currently being used in a number of

research projects. It is also being used in a number of

commercial applications, including

"a database management system for a large

corporation";

"a knowledge base for an expert system";

and

"a rule-based system for a legal application".

Despite some limitations, the language is proving to be

a versatile and powerful tool for database programming.

It is an exciting field of study, and we look forward to

seeing what the future holds.
our presentation in Section 8. With a summary of our results. We close
used to show the decidability of containment problems for nominal programs. 

Section 7 we indicate how our techniques may be
upward bounds or the non-nominal case. In Section 7 we focus on the bounds, 
define an exponential space upper bound for the inner case: a linear-exponential-time 
which are tight in the inner case. In Section 6 we focus on the boundedness of the 
we reduce the non-nominal case to the nominal connected case. For connected non-nominal programs we derive a 
we prove lower bounds. In Section 2 we focus on the problem of boundedness and 
doubly exponential space lower bound. In Section 2 we focus on the problem of boundedness and 
are equivalent to the poly-time case. For non-nominal programs we derive a 
proven in this simpler context. For non-connected non-nominal programs we derive 
and language theory.

After proving some complexity results, we determine in Section 3 whether this theory applies

Declarative Programs. Concerning recognition of the applicability of language theory to the optimization of

Declarative Programs. Concerning recognition of the applicability of language theory to the optimization of

Declarative Programs. Concerning recognition of the applicability of language theory to the optimization of

Declarative Programs. Concerning recognition of the applicability of language theory to the optimization of

Declarative Programs. Concerning recognition of the applicability of language theory to the optimization of

Declarative Programs. Concerning recognition of the applicability of language theory to the optimization of

Declarative Programs. Concerning recognition of the applicability of language theory to the optimization of

Declarative Programs. Concerning recognition of the applicability of language theory to the optimization of
3 Linear Connected Programs

Atomic formulas of $\mathcal{C}$. $\mathcal{C}$ is a conjunction of atomic formulas such that the atomic formulas of $\mathcal{C}$ are mapped to variables. The empty set of variables is the empty mapping on the variables of $\mathcal{C}$.

A conjunctive query is a union of conjunctive queries. A conjunctive query is a conjunction of conjunctive queries.

The $\mathcal{C}$'s are called the $\mathcal{C}$-expressions of $\mathcal{II}$. It was shown in [Nag86] that $\mathcal{II}$ is bounded if and only if for each $\mathcal{C}$-predicate $\mathcal{C}$ with expressions $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{C}_3$, $\ldots$, there is some $N > 1$ such that for all $m < N$ the query $\mathcal{C}$ is contained in $\mathcal{C}_m$.

It is known that the predicates defined by a Datalog program $\mathcal{II}$ can be equivalently defined by an infinite union of conjunctive queries $\mathcal{C}_m$. The $\mathcal{C}_m$'s are defined by recursive rules of the form $\mathcal{C}_m = \mathcal{C}_m \cup \mathcal{C}_{m+1}$.

2 Preliminaries

Because of length limitations we omit most details of the proofs in this paper. They will be given in the full paper.
Let \( \alpha \) be an \( \mathcal{L} \)-DB in \( \Pi \), and let \( \alpha \in \mathcal{L} \) be a \( \mathcal{L} \)-closed \( \Pi \)-consistent sequence of \( \mathcal{L} \)-expressions. Let \( \alpha \) be \( \Pi \)-acceptable if there is a \( \mathcal{L} \)-closed \( \Pi \)-consistent sequence of \( \mathcal{L} \)-expressions \( \beta \) such that \( \beta \) is the \( \Pi \)-quotient of \( \alpha \). We say that \( \alpha \) is \( \Pi \)-acceptable by \( \Pi \). If \( \alpha \) is \( \Pi \)-acceptable by \( \Pi \), then there is some \( \Pi \)-expression \( \xi \) for a word of \( \Pi \)-expressions \( \alpha \) such that \( \xi \) is \( \Pi \)-acceptable by \( \Pi \)

\[ \alpha = \beta \]

or \( \alpha \) is \( \Pi \)-acceptable by \( \Pi \). Thus, \( \Pi \)-fragment \( \alpha \) is \( \Pi \)-acceptable by \( \Pi \) or \( \Pi \)-acceptable by \( \Pi \). So \( \alpha \) is \( \Pi \)-acceptable by \( \Pi \).

\[ \Pi \]

For example, let \( \alpha \) be a \( \Pi \)-closed \( \Pi \)-consistent sequence of \( \Pi \)-expressions. Then, \( \Pi \)-closed \( \Pi \)-consistent sequence of \( \Pi \)-expressions \( \Pi \) can be every \( \Pi \)-closed \( \Pi \)-consistent sequence of \( \Pi \)-expressions. This sequence \( \Pi \) is \( \Pi \)-acceptable by \( \Pi \) and \( \Pi \)-closed \( \Pi \)-consistent sequence of \( \Pi \)-expressions. The \( \Pi \)-closed \( \Pi \)-consistent sequence of \( \Pi \)-expressions is \( \Pi \)-acceptable by \( \Pi \).

Consider the \( \Pi \)-closed \( \Pi \)-consistent sequence of \( \Pi \)-expressions. This \( \Pi \)-closed \( \Pi \)-consistent sequence of \( \Pi \)-expressions is \( \Pi \)-acceptable by \( \Pi \).

More formally, let \( \Pi \) be a \( \Pi \)-closed \( \Pi \)-consistent sequence of \( \Pi \)-expressions. Then, \( \Pi \)-closed \( \Pi \)-consistent sequence of \( \Pi \)-expressions is \( \Pi \)-acceptable by \( \Pi \).

Example 2.1: Let \( \Pi \) consist of the rules:

\[ (x) \theta : (x) \theta \\
(\lambda) \theta \vdash (\lambda) \theta \]

Inference rules are normally used to derive \( \Pi \)-closed \( \Pi \)-consistent sequences of \( \Pi \)-expressions. We shall show later how to remove these inference rules. The \( \Pi \)-closed \( \Pi \)-consistent sequence of \( \Pi \)-expressions is \( \Pi \)-acceptable by \( \Pi \).

Consider a \( \Pi \)-closed \( \Pi \)-consistent sequence of \( \Pi \)-expressions. This \( \Pi \)-closed \( \Pi \)-consistent sequence of \( \Pi \)-expressions is \( \Pi \)-acceptable by \( \Pi \).

Inference rules are normally used to derive \( \Pi \)-closed \( \Pi \)-consistent sequences of \( \Pi \)-expressions. We shall show later how to remove these
expand(II) = \{ f \in \mathcal{L}(\mathcal{A}) \mid \mathcal{A} \text{ accepts } f \}.

Proposition 3.4: Let II be a linear monadic program, and let \( \mathcal{A} \) be an IDB predicate that is regular. We first show that \( \text{expand}(\text{II}) \) is regular, and then we show that \( \text{expand}(\text{II}) \) is a regular language. Moreover, there is an automaton for \( \text{II} \) that accepts \( \mathcal{A} \). Two languages, \( \text{II} \) and \( \text{II}^{\text{non-c}} \), are regular. The following proposition shows that \( \text{expand}(\text{II}) \) is a linear language, and Proposition 3.3 shows that \( \text{II} \) and \( \text{II}^{\text{non-c}} \) are bounded.

Proposition 3.3: Given two linear monadic languages \( \mathcal{M}_{1} \) and \( \mathcal{M}_{2} \), we can prove the decidability of \( \mathcal{M}_{1} \cap \mathcal{M}_{2} \).

Proposition 3.2: A linear monadic finite automaton \( \mathcal{M}_{2} \) accepts a linear monadic language \( \mathcal{M}_{1} \) if and only if \( \mathcal{M}_{1} \subseteq \mathcal{M}_{2} \).

Proposition 3.1: Let \( \mathcal{M} \) be a linear monadic program, and let \( \mathcal{A} \) be an IDB predicate that is bounded. We say that \( \mathcal{M} \) has the unbounded prefix property with respect to \( \mathcal{A} \).

Proposition 3.1: Let \( \mathcal{A} \) be a linear monadic program, and let \( \mathcal{M} \) be a linear monadic language.

Connectedness.

Theorem: A linear monadic program \( \mathcal{M} \) is bounded if and only if there is a finite set \( S \) such that \( \mathcal{M} \subseteq \mathcal{S} \).

Proposition 3.2: Let \( \mathcal{M} \) be a linear monadic program, and let \( \mathcal{A} \) be an IDB predicate that is bounded.

Proposition 3.1: Let \( \mathcal{M} \) be a linear monadic program, and let \( \mathcal{A} \) be an IDB predicate that is bounded.

Theorem: A linear monadic program \( \mathcal{M} \) is bounded if and only if there is a finite set \( S \) such that \( \mathcal{M} \subseteq \mathcal{S} \).

Proposition 3.2: Let \( \mathcal{M} \) be a linear monadic program, and let \( \mathcal{A} \) be an IDB predicate that is bounded.

Proposition 3.1: Let \( \mathcal{M} \) be a linear monadic program, and let \( \mathcal{A} \) be an IDB predicate that is bounded.

Theorem: A linear monadic program \( \mathcal{M} \) is bounded if and only if there is a finite set \( S \) such that \( \mathcal{M} \subseteq \mathcal{S} \).

Proposition 3.2: Let \( \mathcal{M} \) be a linear monadic program, and let \( \mathcal{A} \) be an IDB predicate that is bounded.

Proposition 3.1: Let \( \mathcal{M} \) be a linear monadic program, and let \( \mathcal{A} \) be an IDB predicate that is bounded.

Theorem: A linear monadic program \( \mathcal{M} \) is bounded if and only if there is a finite set \( S \) such that \( \mathcal{M} \subseteq \mathcal{S} \).

Proposition 3.2: Let \( \mathcal{M} \) be a linear monadic program, and let \( \mathcal{A} \) be an IDB predicate that is bounded.

Proposition 3.1: Let \( \mathcal{M} \) be a linear monadic program, and let \( \mathcal{A} \) be an IDB predicate that is bounded.

Theorem: A linear monadic program \( \mathcal{M} \) is bounded if and only if there is a finite set \( S \) such that \( \mathcal{M} \subseteq \mathcal{S} \).

Proposition 3.2: Let \( \mathcal{M} \) be a linear monadic program, and let \( \mathcal{A} \) be an IDB predicate that is bounded.

Proposition 3.1: Let \( \mathcal{M} \) be a linear monadic program, and let \( \mathcal{A} \) be an IDB predicate that is bounded.

Theorem: A linear monadic program \( \mathcal{M} \) is bounded if and only if there is a finite set \( S \) such that \( \mathcal{M} \subseteq \mathcal{S} \).

Proposition 3.2: Let \( \mathcal{M} \) be a linear monadic program, and let \( \mathcal{A} \) be an IDB predicate that is bounded.

Proposition 3.1: Let \( \mathcal{M} \) be a linear monadic program, and let \( \mathcal{A} \) be an IDB predicate that is bounded.
Proposition 3.7: Let II be a linear monadic program, and let O be an IDP predicate of (Ω, II). Then II is linear in order to decide if a word w is not accessible, an automaton whose size is exponential in the length of II. Moreover, there is an automaton whose size is exponential in the length of II, when w is not accessible, and only if there is a legal enrichment of w such that the corresponding enriched query does not contain (Ω, II).

Proposition 3.6: Let II be a linear monadic program, and let O be an IDP predicate of (Ω, II). Then w is an enrichment of a word w over Z^n if and only if there is a legal enrichment of w of such that the corresponding enriched query does not contain (Ω, II).

The idea is that in order to decide if a word w is not accessible, an automaton whose size is exponential in the length of II. Moreover, there is an automaton whose size is exponential in the length of II, when w is not accessible, and only if there is a legal enrichment of w such that the corresponding enriched query does not contain (Ω, II).

Proposition 3.5: Consider the program of Example 3.1. A recursive body and (\(\ell_X\) ) for such an enriched query is in an initial enriched query, and in the latter for each body (\(\ell_X\) ) is in an initial enriched query, and in the latter for each body (\(\ell_X\) ) is in an initial enriched query, and in the latter for each body (\(\ell_X\) ) is in an initial enriched query, and in the latter for each body (\(\ell_X\) ) is in an initial enriched query, and in the latter for each body (\(\ell_X\) ) is in an initial enriched query, and in the latter.
A tree is a partial order \( (V, E) \) where \( E \) is the set of edges and \( V \) is the set of nodes. The depth of a node \( v \) is the length of the longest path from the root to \( v \). The depth of a tree is the maximum depth of any node in the tree.

Let \( \mathcal{N} \) denote the set of positive integers. We use the variables \( x \) and \( y \) to denote leaves.

Then \( C_0 = (1) \) and \( \forall \) is a tree with \( \forall \) at the root and \( \forall \) and \( \forall \) at the root and \( \forall \) and \( \forall \) at the root.

\[(x) \forall - : (x) \forall \]

\[(z) \forall - \exists (x, y) \forall - : (x) \forall \]

**Example 4.1:** Let \( \mathcal{I} \) consist of the rules:

and tree automata.

rather than sequences of steps. Thus, to handle this case, we have to use tree languages.

the expressions are constructed from variables and syntactically correct strings of symbols.

where \( \forall \) is a set of EDB formulas. We say here that the branching of \( \forall \) is \( n \). In this case,

\[(\forall) \forall - \exists (x) \forall - : (x) \forall \]

Consider monadic programs with recursive rules of the form

In this section we remove the assumption that we are dealing with linear programs.

**4 Connected Programs**

It is known to be NP-complete [Yag89].

We note that for the case that the program contains only one recursive rule, bounded-

monadic programs are solvable in polynomial space.

**Theorem 3.9:** The predicate and program boundedness problem for linear connected program unboundedness is decidable in polynomial space.

From Propositions 3.2, 3.3, and 3.4 and Corollary 3.8, it follows that linear connected programs whose size is exponential in the length of \( \forall \).

Then monadic \((\forall) \) is regular. Moreover, there is an automaton for \((\forall) \) (\( \forall \)).

**Corollary 3.8:** Let \( \mathcal{I} \) be a linear monadic program and let \( \mathcal{O} \) be an IDP predicate of \( \mathcal{I} \).
10

\( (x) \neq \exists y \in \mathbb{R} \setminus \{0, 1\} \) if and only if there is a tuple of the form \((x, y, 0, 1, 2, 3, 4, 5) \in \mathbb{R}^7 \) such that for every \( \phi \in \mathcal{L}(x) \), \( \forall x \in \mathbb{R} \setminus \{0, 1\} \). Therefore, for the transition function \( \mathcal{T} \), whose role is to abstract away the structure of a set of states and their labels, we have to define regular expressions that are regular expressions with respect to a set of states.

To generalize Proposition 3, we have to define regular expressions that are regular expressions with respect to a set of states.

Proposition 4.2: Let \( \mathcal{L} \) be a connected monadic program. Then \( \mathcal{L} \) is unboundedly if and only if for some IDB predicate \( \mathcal{O} \), \( \mathcal{L} \) is unboundedly regular with respect to \( \mathcal{O} \).

Proposition 4.3: Let \( \mathcal{L} \) be a connected monadic program. Then \( \mathcal{L} \) is unboundedly regular with respect to \( \mathcal{O} \).

The following result generalizes Proposition 3. We note that the proposition does not hold for disconnected programs.

The following result generalizes Proposition 3. We note that the proposition does not hold for disconnected programs.
Consider a (recursive or non-recursive) rule of the form $\forall X \exists X'$, where $X'$ is a set of EDB and IDB formulas. Let $\mathcal{O}$ be an undirected graph, where $X$ is a set of all variables appearing in either side of the rule, and $X \subseteq \{X, X'\}$.

In this section we show how to extend the results of the previous sections by removing programs.

In this section, we show how to extend the results of the previous sections by removing programs.

**Example 5.1:** Let $\mathcal{O}$ consist of the rule:

$$(X) \overrightarrow{\mathcal{O}} (Y)$$

Indeed, $\mathcal{O}$.

We first show by way of an example that programs boundingness and predicative boundingness of programs are solvable in doubly exponential time.

**Theorem 4.5:** The predicate and program boundingness problems for connected monadic programs of monadic programs.

**Proposition 4.2:** A set $\mathcal{O}$ is a doubly exponential upper bound for program boundingness if and only if $\mathcal{O}$.

**Linear Programs:**

If follows from Propositions 4.2 and 4.3 that we can prove the decidability of bounded-

polynomial time in the size of the input.

**Proposition 4.3:** Given two automata $M$ and $M'$ over an alphabet $\Sigma$, determine whether $L(M')$ has the unbounded prefix property with respect to $L(M)$, can be decided in polynomial time in the size of the input.

**Proposition 1:** If $M$ is an accepting run on $(x)$, then $x$ is accepting.
Proposition 5.4. Let II be a monadic program. Then II is unbounded iff there is a partition $S = (S_1, S_2)$ of the integers of $\mathbb{Z}$, where $S_1 \cup S_2 = \mathbb{Z}$, such that

$$\forall x \in S_1 \exists y \in S_2 (x < y \land \forall z (x < z \land z < y \rightarrow x < z \land y < z))$$

Moreover, II is unbounded in $\mathbb{Z}$ if and only if for some $n$, there is a tree $T$ in which $n$ is in the definition $\mathcal{D}(n)$. For some $n$, for every $k > 0$, there is a tree $T'$ in which $k$ is in the definition $\mathcal{D}(k)$.

For some IDP program $\mathcal{A}$, let $\mathcal{A}$ be the subgraph of $\mathcal{G}$ induced by the set of nodes reachable from $X$, and let $\mathcal{C}(\mathcal{A})$ be the connected component of $\mathcal{G}$ containing $\mathcal{A}$.
W reflects that there are arbitrarily long legal computations, but that is possible precisely when only if there are arbitrarily long legal computations. Thus, when the program will be unbounded if and only if there are unbounded paths. A legal computation's termination by replacing some rules that move the letter to the next position. The only way the letter can be replaced is to 

Theorem 5. The program boundedness problem for

The conditions of Proposition 5.2. can be tested by appropriately extending the automaton-

Theorem 5.6: The program boundedness problem for monadic programs is decidable in

doubly exponential time and is EXPSPACE-hard. The program boundedness problem for monadic programs is decidable in

In any database D such that all \( T_\alpha超强 (D) \) reaches all the targets in \( S \), and no triggers
In any of the variable graphs, for rules II.

Consider the diagrams of II by the maximal distance between two connected variables. We also say that "is the d-adjunction of.

d-neighborhood of at in w for 1 ≤ i ≤ j ≤ m. A word of length n = 4 + C(d+1)d2 + C(d+2)d + 2 + C(2d+2) over Σ is a d-adjunction word if there exists a word w = a1 a2 ... ad over Σ, such that a1 = a2 = ... = ad = a in Σ. Now is the superset of a word over Σ, the d-adjunction of a letter, a, for 1 ≤ i ≤ j ≤ m, and let be a word over Σ.

The notation (d) denotes the set of all words of length at most n, i.e., let (d). Let the notation (d) denote the set of all words of length at most n, i.e., (d).

We focus here on the linear case. Let (d) in an alphabet and let k < 0. We use a better approach in the restricted case.

Exponential bound for linear programs is highly exponential for nonlinear programs. A

It is known that Propositions 3.2 and 4.1. For instance, yields automata that are doubly exponential.

If is regular by Propositions 3.2 and 4.1. For instance, yields automata that are doubly exponential.

If and Propositions 3.2 and 4.1 are regular, then apply Proposition 6.1. Consider first (d).

Since the problem can be solved in nondeterministic logarithmic space for regular languages, the determinism problem is decidable for regular languages and regular tree languages.

Theorem (3.2): Let (d) be a regular program (3.2). Let (d) be a nondeterministic program (3.2). If (d) is bounded and only if there are only finitely many

Proposition 6.1: Let (d) be a nondeterministic program (3.2) and (d) be an NP problem of (d). Then (d) is bounded if and only if there are only finitely many

Conjunctive graphs property (d) accepted by (d). Let (d) be the formula (d) accepted by (d). Let (d) be the formula (d) accepted by (d). If (d) is a proper subgraph of (d) then (d) is a proper subgraph of (d). If (d) is a proper subgraph of (d) then (d) is a proper subgraph of (d).

If there is a conjunctive mapping of (d) to (d). Equivalently, (d) is accepted by (d). If there is a conjunctive mapping of (d) to (d). Equivalently, (d) is accepted by (d). If there is a conjunctive mapping of (d) to (d). Equivalently, (d) is accepted by (d). If there is a conjunctive mapping of (d) to (d). Equivalently, (d) is accepted by (d). If there is a conjunctive mapping of (d) to (d). Equivalently, (d) is accepted by (d). If there is a conjunctive mapping of (d) to (d). Equivalently, (d) is accepted by (d). If there is a conjunctive mapping of (d) to (d). Equivalently, (d) is accepted by (d). If there is a conjunctive mapping of (d) to (d). Equivalently, (d) is accepted by (d). If there is a conjunctive mapping of (d) to (d). Equivalently, (d) is accepted by (d). If there is a conjunctive mapping of (d) to (d). Equivalently, (d) is accepted by (d). If there is a conjunctive mapping of (d) to (d). Equivalently, (d) is accepted by (d). If there is a conjunctive mapping of (d) to (d). Equivalently, (d) is accepted by (d). If there is a conjunctive mapping of (d) to (d). Equivalently, (d) is accepted by (d). If there is a conjunctive mapping of (d) to (d). Equivalently, (d) is accepted by (d). If there is a conjunctive mapping of (d) to (d). Equivalently, (d) is accepted by (d). If there is a conjunctive mapping of (d) to (d). Equivalently, (d) is accepted by (d). If there is a conjunctive mapping of (d) to (d). Equivalently, (d) is accepted by (d). If there is a conjunctive mapping of (d) to (d). Equivalently, (d) is accepted by (d). If there is a conjunctive mapping of (d) to (d). Equivalently, (d) is accepted by (d). If there is a conjunctive mapping of (d) to (d). Equivalently, (d) is accepted by (d). If there is a conjunctive mapping of (d) to (d). Equivalently, (d) is accepted by (d). If there is a conjunctive mapping of (d) to (d). Equivalently, (d) is accepted by (d). If there is a conjunctive mapping of (d) to (d). Equivalently, (d) is accepted by (d). If there is a conjunctive mapping of (d) to (d). Equivalently, (d) is accepted by (d). If there is a conjunctive mapping of (d) to (d). Equivalently, (d) is accepted by (d). If there is a conjunctive mapping of (d) to (d). Equivalently, (d) is accepted by (d). If there is a conjunctive mapping of (d) to (d). Equivalently, (d) is accepted by (d). If there is a conjunctive mapping of (d) to (d). Equivalently, (d) is accepted by (d).

The key to our result is an alternative (to Propositions 3.2 and 4.1) language-theoretic

6 Predicates

For the predicate boundedness of linear programs. In this section we solve the problem of establishing upper bounds for program boundedness and in the previous sections we have established lower bounds for program boundedness.

We note that our lower bounds use disconnected programs. The only known lower bound for connected linear programs is an NP-hardness bound provided in [Yu88].
Theorem 6.5: The predicate boundedness problem for monadic programs is decidable in exponential time in the general case.

complement of Proposition 6.1. This consists of a single exponential
function of the length of the automaton, and the argumentation of
Blak, which in turn guarantees the decidability of the problem.
Secondly, we need to ensure that the automaton is 2-way.
To deal with monadic programs, two extensions are needed. First, neighborhoods are

Proposition 6.4: Let II be a monadic program, and let the 2DP predicate of II

Corollary 6.3: Let II be a monadic program, and let the 2DP predicate of II

Lemma 6.2: Let II be a monadic program with diameter d, and let C be an IDP

complete for ZF. (Cf. [GMSY87], we can show that predicate boundedness of monadic programs is complete
remark: however, that which program boundedness is internal whereas the predicate boundedness.
We do not know whether predicate boundedness is PSPACE in the internal case, and exponential time in general. We do
previous section, that is, PSPACE in the internal case, and exponential time in general. We do

The only lower bounds we have for predicate boundedness are those we gave in the

exponential space in the internal case and in doubly exponential time in the general case.

Theorem 4.9: There is a free automaton for accept(II, D) whose size is doubly exponential in the length of II.

Proposition 6.1: Let II be a monadic program, and let the 1DP predicate of II.

Corollary 4.8: There is a 2-way alternating 2-way alternating two-way alternating free automaton, the number of
neighborhoods in exponential space.

To deal with monadic programs, two extensions are needed. First, neighborhoods are
basic tools in the optimization of database logic programs. We hope that these techniques will become results for boundedness of such programs. We believe that these techniques apply also to nondeterministic programs and can yield decidable

are the famous gaps between the lower and upper bounds that need to be closed.

In conclusion, we remark that even though we have focused here on monadic programs,

We have presented results for boundedness and containment of monadic programs. To-

8 Concluding Remarks

integer programs is PSPACE-complete.

doubly exponential time and is EXPSPACE-hard. The predictive containment problem for

Proposition 7.1. Let II be programs with the same EDB predicates, and let \( \sigma \) be an

Proposition 7.2. The predicative containment problem for

Using our automata-theoretic techniques we can now obtain:

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

The first step is to prove that predicative containment is decidable.

Our automata-theoretic techniques, as it turns out, are applicable to another question

Application to Containment

<table>
<thead>
<tr>
<th>Program</th>
<th>Containment</th>
<th>Boundedness</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-EXPTIME</td>
<td>PSPACE</td>
<td>disconnected</td>
</tr>
<tr>
<td>2-EXPTIME</td>
<td>PSPACE</td>
<td>connected</td>
</tr>
<tr>
<td>EXPSPACE</td>
<td>PSPACE</td>
<td>predicable</td>
</tr>
<tr>
<td>2-EXPTIME</td>
<td>EXPSPACE</td>
<td>non-bound</td>
</tr>
<tr>
<td>1-EXPTIME</td>
<td>EXPSPACE</td>
<td>upper bound</td>
</tr>
<tr>
<td>EXPSPACE</td>
<td>EXPSPACE</td>
<td>lower bound</td>
</tr>
<tr>
<td>EXPSPACE</td>
<td>EXPSPACE</td>
<td>lower bound</td>
</tr>
<tr>
<td>EXPSPACE</td>
<td>EXPSPACE</td>
<td>lower bound</td>
</tr>
</tbody>
</table>

Table 1: The complexity of boundedness and containment for monadic programs.
A Appendix: Two-way Alternating Tree Automata

If $S$ is a set, then $L(S)$ denotes the free distributive lattice of $S$, that is, the closure of $S$ under the binary operations $\lor$ and $\land$. We can view $L(S)$ as the set of positive Boolean formulas over $S$. We say that an element $e \in L(S)$ is true with respect to a set $T \subseteq S$, if $e$ evaluates to 1. Otherwise, $e$ is false with respect to $T$.

Fix a set of states $S$. For each $s \in S$, we take the upward direction, 0 denotes staying in place, and $1, \ldots, k$ denote the $k$ downward directions in $S$. The boolean value $x$ evaluates to 1 if $j = 1$ and $0$ otherwise. $e$ is true with respect to $T$.

Let $[k]$ denote the set $\{-1, 0, \ldots, k\}$. We think of $[k]$ as a set of directions. Let $d$ denote the direction of the root point on the $i$th child of a node. Then whenever the automaton $M$ is in state $s$ and it reads the letter $a$, then it calls itself recursively either in state $s$ on the left child and state $t_a$ on the right child or in state $s$ on the left child and state $t_a$ on the right child. Two-way alternating tree automata can also be viewed as recursive processes, but they have more powerful recursive cells.

A 2-way alternating tree automaton $M$ is a tuple $(S, \Sigma, S_0, \delta, \rho, F)$, where $S$ is a tree alphabet, $\Sigma$ is a set of states, $S_0 \subseteq S$ is the starting state set, $\delta \subseteq S \times \Sigma \times [k]$ is the accepting state set, and $\rho : S \times \Sigma \times \{0\} \to \{1, 0\}$ is a transition function defined on $S \times \Sigma$ such that, if $a \in \Sigma$ has arity $t$, then $\rho(a, s, \delta(a), t_a) \in L(S \times \{0\})$. Suppose $\rho(a, s, \delta(a), t_a) \in L(S \times \{0\})$. Then whenever the automaton $M$ is in state $s$ and it reads the letter $a$, then it calls itself recursively either in state $s$ on the left child or in state $t_a$ on the right child or in state $s$ on the left child and state $t_a$ on the right child. Two-way alternating tree automata can also be viewed as recursive processes, but they have more powerful recursive cells.

The requirement from $r$ are as follows:

1. The root of $r$ is labelled by $(s_0, i, \lambda)$ with $s_0 \in S_0$.

2. If a node $\alpha$ is labelled by $(s,i,z)$, $x \in \sigma$, and the children of $\alpha$ are labelled by

   (i,j,\bar{y}), \text{ then } j \in [k] \text{ and } y = z.

3. If a node $\alpha$ is labelled by $(s,i,z)$, $x \in \sigma$, and the children of $\alpha$ are labelled by

   (i,j,\bar{y}), \text{ then } j \in [k] \text{ and } y = z.

4. If a node $\alpha$ is labelled by $(s,i,z)$, $x \in \sigma$, and the children of $\alpha$ are labelled by

   (s,i,j,\bar{y}), \text{ then } j \in [k] \text{ and } y = z.

Note that the tree is distinct from the tree $r$ if it is the computation tree of the automaton on $(s,i,z)$. The run $r$ is accepting if whenever a leaf $\alpha$ is labelled by $(s,i,z)$, then $s \in F^s$. We could also require that $x$ is a leaf of $r$. Since this requirement does not have an effect on our results, we chose not to assume it here.
A.8. The reader should note that the class 2-way alternating automata on words is a subclass of 4-way alternating automata, which are more expressive than 1-way alternating automata. However, in mRNA [95], it is shown that 2-way alternating automata are not more expressive than 1-way alternating automata. This, therefore, generalizes the result of the class of 2-way alternating automata.

Remark A.8. The existence of such a structure can be checked by a 1-way alternating automaton.

Lemma (GH82). The existence of such a structure is a consequence of the following proposition:

Let $A_n$ be a $n$-way alternating automaton. Then there is an exponential in the size of $A_n$ such that for any exponential $W$ whose size is exponential in the size of $A_n$ there is a $1$-way alternating automaton $A_n$ with the following properties:

1. There is a set $S$ of nodes of $A_n$ such that $|S| \geq n$.
2. For all $x \in S$ and $s \in S$, there is a node $y$ such that $S \subseteq y$. (x, y)
3. For all $x \in S$, there is a node $y$ such that $S \subseteq y$. (x, y)
4. For all $x \in S$, there is a node $y$ such that $S \subseteq y$. (x, y)
5. For all $x \in S$, there is a node $y$ such that $S \subseteq y$. (x, y)
6. For all $x \in S$, there is a node $y$ such that $S \subseteq y$. (x, y)

Proof: We first define a tree $T$ in the following way: $T$ includes all the nodes of $\tau$, and for each node $x$ in $\tau$, the subtree of $T$ rooted at $x$ is the tree $T_x$.

There is a tree $T$ in the following way:

1. The tree $T$ is a union of $\tau$ and $\bigcup_{x \in \tau} T_x$.
2. The tree $T$ is a union of $\tau$ and $\bigcup_{x \in \tau} T_x$.
3. The tree $T$ is a union of $\tau$ and $\bigcup_{x \in \tau} T_x$.
4. The tree $T$ is a union of $\tau$ and $\bigcup_{x \in \tau} T_x$.
5. The tree $T$ is a union of $\tau$ and $\bigcup_{x \in \tau} T_x$.
6. The tree $T$ is a union of $\tau$ and $\bigcup_{x \in \tau} T_x$.

Theorem A.1: Let $W = \langle \Sigma, \Gamma, \delta \rangle$ be a 4-way alternating automaton. Then there is a 2-way alternating automaton $W'$ such that $W' \equiv W$.
References


Ullman, J.D.: Implementation of Logical Query Languages for Databases. ACM