UPMAIL Technical Report No. 149
November 27, 1997
Revised February 16, 1998
ISSN 1100-0686

Complexity of Query Answering in Logic Databases with Complex Values

Evgeny Dantsin*

Steklov Institute of Mathematics at St.Petersburg
Fontanka 27, St.Petersburg 191011
Russia

Andrei Voronkov†

Computing Science Department
Uppsala University
Box 311, S-751 05 Uppsala
Sweden

email: {dantsin,voronkov}@csd.uu.se

*Supported by grants from the Swedish Royal Academy of Sciences, the Swedish Institute and RFBR/INTAS.
Author's email address at St.Petersburg: dantsin@pdmi.ras.ru
†Supported by a TFR grant.

Abstract

This paper characterizes the computational complexity of nonrecursive queries in logic databases with complex values. Queries are represented by Horn clause logic programs. Complex values are represented by terms in equational theories (finite sets and multisets are examples of such complex values). We show that the problem of whether a query has a nonempty answer is NEXP-hard for nonrecursive range-restricted queries. We also show that this problem is in NEXP if complex values satisfy the following condition: the solvability problem for equations in the corresponding equational theory is in NP. Since trees, finite sets and multisets satisfy this condition, the query answering problem for logic databases with trees, finite sets and multisets is shown to be NEXP-complete.[2]

Contents

1 Introduction

Relational query languages typically deal with tuples of atomic values like strings or integers. Applications require to handle more complex values, for example sets or multisets. Various kinds of complex values in databases and logic programming have been considered in many papers, including [2, 3, 15, 6, 22, 20, 16, 5, 1, 14, 13, 10, 9, 23]. Our paper studies complexity aspects of databases with complex values.

There are two main kinds of complexity in databases: (1) *query evaluation* complexity that measures time or space needed to evaluate a query, and (2) *descriptive complexity* that characterizes the expressive power of query languages. In turn, query evaluation complexity is considered in three forms: *data complexity*, *program complexity* and *combined complexity.* The data complexity is expressed as a function of the size of data (represented by ground atoms) for a fixed query (represented by a logic program and a goal). The program complexity means that a query is variable, while data are fixed. The combined complexity considers both data and query to be variable.

By the *query answering problem* we mean the problem of whether a given goal succeeds with respect to a given logic program. In terms of logic databases, this problem is formulated as follows: given a database instance and a query, does the outputted relation contain at least one tuple? Thus, the *query answering complexity*, i.e. the complexity of the above problem, is a variant of the combined complexity, namely the combined complexity of Boolean queries. This kind of complexity seems to be especially relevant to databases with complex values since in this case an answer can be too large or even infinite.

For Horn clause logic programs, the query answering problem can be naturally formalized as the decision problem for corresponding fragments of first-order logic. For non-Horn clause logic programs, decidability and complexity have been mainly considered in the context of non-monotonic semantics ([7]). As for logic programs with complex values, no previous complexity results are known to us.

This paper characterizes the query answering complexity for nonrecursive Horn clause logic programs with complex values. Complex values are represented by terms in equational theories (see details in Section 2). Trees, finite sets and multisets are examples of such complex values.

We show that this problem for nonrecursive range-restricted logic programs is NEXP-hard. We also show that the query answering problem for nonrecursive logic programs is in NEXP if complex values satisfy the following condition: the solvability problem for equations in the corresponding equational theory is in NP. Since trees, finite sets and multisets satisfy this condition, the query answering problem for logic databases with trees, finite sets and multisets is shown to be NEXP-complete.

This paper is structured as follows. In Section 2 we define what we mean by complex values and define notation for logic programs with complex values. In Section 3 we prove NEXP-hardness of the query answering problem for nonrecursive range-restricted logic programs. In Section 4 we prove the inclusion in NEXP for complex values satisfying the condition mentioned above.

2 Preliminaries

2.1 Complex values

Equational theories. In order to define what we mean by complex values, we first define the notion of an equational theory. Let $\mathcal{F}$ be a set of *function symbols.* As usual, a nonnegative arity is assigned to every function symbol. *Constants* are function symbols of arity 0. We assume that $\mathcal{F}$ contains at least one constant. The notion of a *term* is defined in the standard way: a constant is a term; a variable is a term; if $t_1, \ldots, t_n$ are terms and f is a function symbol of arity n then $f(t_1, \ldots, t_n)$ is a term. (Some variations are possible, for example variables may have sorts). A term without variables is called *ground.* By an *equation* we mean an expression $s = t$. An equation is said to be *ground* if s and t are ground. An *equational theory* E *over* $\mathcal{F}$ is a set of ground equations closed under the logical consequence relation, i.e. such that

1. E contains all ground equations of the form $t = t$.
2. If E contains $s = t$ then E contains $t = s$.
3. If E contains $r = s$ and $s = t$ then E contains $r = t$.
4. If E contains $s_1 = t_1, \ldots, s_n = t_n$ then E contains $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$ for each n-ary function symbol $f \in \mathcal{F}$.

Basically, each equational theory E defines a *domain* $\mathcal{D}_E$ of complex values: the elements of $\mathcal{D}_E$ are the equivalence classes of all ground terms over $\mathcal{F}$ with respect to the equality relation in E.

Examples of complex values. *Trees* can be viewed as a trivial example of complex values. Namely, trees are values represented by ground terms, and the equality relation on them is defined as identity. Thus, equational theories for trees consist of only equations $t = t$.

Lists are a special kind of trees. In logic programming, lists are constructed from the constant `[]` (called the *empty list*) by using the binary function symbol $[s \,|\, t]$ (called the *list constructor*). If a term t represents a list then a term $[s \,|\, t]$ represents the list obtained from the list t by adding the term s as the first element. Equational theories for lists are the same as for trees.

Bags (or *finite multisets* in another terminology) are lists in which the order of elements is immaterial. We denote bags similar to lists, but using the *empty bag* $\langle\rangle$ instead of the empty list `[]`, the *bag constructor* $\langle s \,|\, t\rangle$ instead of the list constructor $[s \,|\, t]$ and $\langle t_1, \ldots, t_n\rangle$ instead of $[t_1, \ldots, t_n]$. Bags can be axiomatized by ground equations $\langle r \mid \langle s \mid t\rangle\rangle = \langle s \mid \langle r \mid t\rangle\rangle$ and their logical consequences.

Sets can be viewed as bags in which the number of occurrences of an element is immaterial. Similar to bags, we use the *empty set* $\{\}$ and the *set constructor* $\{s \,|\, t\}$ that denotes the set obtained from the set t by adding s as an element. Sets can be axiomatized as follows: (1) in the axioms for bags, we replace the empty bag and the bag constructor by the empty set and the set constructor respectively, and (2) we add all ground equations $\langle s \mid t \rangle = \langle s \mid \langle s \mid t \rangle \rangle$. Note that we defined *hereditarily finite sets,* i.e. finite sets whose elements can be other (hereditarily finite) sets.

There are other ways of integrating finite sets into logic programming. For example, in [10] sets are represented by terms $\{s_1, \ldots, s_n \mid t\}$ where t can be any term not necessarily representing a set (the authors call such objects *colored sets*). Thus, a set is characterized not only by its elements but also by some "type" (color). Our approach is more traditional: every tail subterm t in $\{s_1, \ldots, s_n \mid t\}$ is required to represent some set.

2.2 Logic programs with complex values

Herbrand interpretations. Let Σ be a signature consisting of a set $\mathcal{F}$ of function symbols (defined as above) and a set of predicate symbols. We assume that Σ contains at least one constant and does not contain the equality symbol $=$. Atoms, substitutions, instances, variants etc. are defined in the standard way (see, for example, [4]). The *Herbrand universe* U_Σ for Σ is defined as the set of all ground terms over Σ. The set of all ground atoms over Σ is called the *Herbrand base* B_Σ for Σ. By a *Herbrand interpretation* $\mathcal{I}$ for Σ we mean any subset of B_Σ (all atoms in $\mathcal{I}$ are considered to be true, all other ground atoms are considered to be false).

Let E be an equational theory. Let $\mathcal{I} \subseteq B_\Sigma$ be a Herbrand interpretation such that if $\mathcal{I}$ contains an atom $p(s_1, \ldots, s_n)$ and E contains equations

$$s_1 = t_1, \ldots, s_n = t_n,$$

then $\mathcal{I}$ contains $p(t_1, \ldots, t_n)$. In this case we call $\mathcal{I}$ a *Herbrand interpretation with respect to the equational theory E.*

Logic programs. We fix a signature Σ and assume that all terms and atoms below are terms and atoms over Σ. A *clause* is any formula of the form $A_1 \wedge \ldots \wedge A_n \supset A$, where $A_1, \ldots, A_n, A$ are atoms and $n \geq 0$. Such a clause is usually written as

$$A \texttt{ :- } A_1, \ldots, A_n.$$

and simply as A when $n = 0$. The atom A is called the *head*, and the sequence $A_1, \ldots, A_n$ is called the *body* of this clause. By the *length* of the body we mean the number n.

A *logic program* is a finite set of clauses. A *goal* is a conjunction $A_1 \wedge \ldots \wedge A_n$ of atoms, written as $A_1, \ldots, A_n$ where $n \geq 0$. When $n = 0$, it is called the *empty goal* and denoted by $\Box$.

By a *logic program with complex values* we mean a pair $(\mathcal{P}, E)$ where $\mathcal{P}$ is a logic program and E is an equational theory. We also call such a pair a *logic program over an equational theory.* In particular, we say that a logic program $\mathcal{P}$ over an equational theory E is a *logic program with sets* if E is the equational theory of sets over $\mathcal{F}$. Similarly, we use the terms *logic programs with bags, trees*, etc.

Semantics. The semantics of logic programs over equational theories can be defined in many ways by adapting the standard logic programming semantics [4, 17, 12] for the corresponding equational theory. For example, one can use the least fixpoint semantics or a generalization of SLD-resolution. We define the semantics by using the least Herbrand model.

Let $A \texttt{ :- } A_1, \ldots, A_n$ be a clause without variables. This clause is said to be *true* in a Herbrand interpretation $\mathcal{I}$ if A belongs to $\mathcal{I}$ whenever $A_1, \ldots, A_n$ belong to $\mathcal{I}$. We say that $\mathcal{I}$ is a *Herbrand model* for a logic program $\mathcal{P}$ if all ground instances of all clauses of $\mathcal{P}$ are true in $\mathcal{I}$. If, in addition,

$\mathcal{I}$ is a Herbrand interpretation with respect to an equational theory E then we call $\mathcal{I}$ a *Herbrand model for $\mathcal{P}$ over E.*

Like the case of ordinary logic programs, Herbrand models for logic programs over equational theories have the following property: the intersection of Herbrand models for $\mathcal{P}$ over E is again a Herbrand model for $\mathcal{P}$ over E. This property allows us to define the *least Herbrand model* for a logic program with complex values: it is the smallest Herbrand model for $\mathcal{P}$ over E (with respect to the subset relation $\subseteq$).

Let $\mathcal{P}$ be a logic program over an equational theory E and $\mathcal{G}$ be a goal. We say that $\mathcal{G}$ *succeeds with respect to* $\mathcal{P}$ over E if there exists a ground instance $A_1, \ldots, A_n$ of $\mathcal{G}$ such that all $A_1, \ldots, A_n$ belong to the least Herbrand model of $\mathcal{P}$ over E. In other words, the formula $\exists(A_1 \wedge \ldots \wedge A_n)$ is true in the least Herbrand model of $\mathcal{P}$ over E.

Nonrecursive programs. A logic program $\mathcal{P}$ is said to be *nonrecursive* if all predicate symbols in $\mathcal{P}$ can be numbered so that the following condition is satisfied for every clause in $\mathcal{P}$: the number of the predicate in the head is greater than the number of each predicate in the body.

Range-restricted logic programs. A logic program $\mathcal{P}$ is called *range-restricted* if every clause A `:-` $A_1, \ldots, A_n$ in $\mathcal{P}$ has the property: all variables of A occur in $A_1, \ldots, A_n$. Range-restricted queries play an important role in databases because they are domain independent (see [21] for example).

The query answering problem. Consider a set $\mathfrak{C}$ of logic programs over E. By the *query answering problem* for $\mathfrak{C}$ we mean the following decision problem: given a program $\mathcal{P} \in \mathfrak{C}$ and a goal $\mathcal{G}$, does $\mathcal{G}$ succeed with respect to $\mathcal{P}$ over E?

Note that the complexity with respect to the combined size of the program and the goal is equivalent to the complexity with respect to the size of the program. Let $\mathcal{P}$ be a program, $A_1, \ldots, A_n$ be a goal, and *yes* be a nullary predicate symbol foreign to Σ. Consider the program $\mathcal{P}' = \mathcal{P} \cup \{yes$ `:-` $A_1, \ldots, A_n\}$. Evidently, $\mathcal{G}$ succeeds with respect to $\mathcal{P}$ if and only if the goal consisting of *yes* succeeds with respect to $\mathcal{P}'$. This transformation of $\mathcal{P}$ into $\mathcal{P}'$ preserves the classes $\mathfrak{C}$ of logic programs considered in this paper (nonrecursive and/or range-restricted programs).

3 NEXP-hardness

Theorem 3.1 The query answering problem for the class of nonrecursive range-restricted logic programs with lists is NEXP-hard.

Proof. We reduce the TILING problem, which is known to be NEXP-complete (see for example [19], page 501), to the query answering problem for nonrecursive range-restricted logic programs with lists.

Informally, TILING is the problem of tiling the square of size $2^n \times 2^n$ by tiles (squares of size 1×1). More precisely, there is a finite set $\{t_0, \ldots, t_k\}$ of *tiles* and there are two binary relations `on` and `to` defined on the tiles. Tiles t_i and t_j are said to be *vertically compatible* if $\texttt{on}(t_i, t_j)$ holds and, similarly, *horizontally compatible* if $\texttt{to}(t_i, t_j)$ holds. A *tiling of the square of size* $2^n \times 2^n$ is a function f from $\{1, \ldots, 2^n\} \times \{1, \ldots, 2^n\}$ into $\{t_0, \ldots, t_k\}$ such that

1. $f(i,j)$ and $f(i+1,j)$ are vertically compatible, for all $1 \le i < 2^n$ and $1 \le j \le 2^n$;
2. $f(i,j)$ and $f(i,j+1)$ are horizontally compatible, for all $1 \le i \le 2^n$ and $1 \le j < 2^n$.

We also say that such f is a tiling *with t_i at the top left corner* if $f(1,1)$ is t_i. The TILING problem is defined as follows. Suppose that we are given a set $\{t_0, \ldots, t_k\}$ of tiles, compatibility relations `on` and `to`, and a number n (written in unary notation). The question we ask is whether there is a tiling of the square of size $2^n \times 2^n$ with t_0 at the top left corner.

The reduction we describe is a polynomial-time algorithm that transforms every instance I of the TILING problem into a logic program $\mathcal{P}$ and a goal $\mathcal{G}$ such that I has a tiling if and only if $\mathcal{G}$ succeeds with respect to $\mathcal{P}$. We think of tiles $t_0, \ldots, t_k$ in I as constants of the program $\mathcal{P}$. The compatibility relations `on` and `to` are represented in $\mathcal{P}$ by the corresponding predicates: namely, $\mathcal{P}$ contains clauses

$\mathtt{on}(t_i, t_j).$
$\mathtt{to}(t_l, t_m).$

for all pairs of compatible tiles.

To proceed the construction of $\mathcal{P}$ and $\mathcal{G}$, we generalize tiles in a natural way. A *hypertile of rank i* is defined by induction as a square of size $2^i \times 2^i$ consisting of tiles. A hypertile of rank 0 is any tile of the set $\{t_0, \ldots, t_k\}$. Let H_1, H_2, H_3 and H_4 be hypertiles of rank i, then the quadruple (H_1, H_2, H_3, H_4) is a hypertile of rank $i+1$. We think of this hypertile as the square

H_1	H_2
H_3	H_4

and we represent it in $\mathcal{P}$ by the list $\mathtt{[}H_1, H_2, H_3, H_4\mathtt{]}$ where H_1, H_2, H_3 and H_4 are lists that represent the corresponding hypertiles of rank i. Here is an example of a hypertile of rank 2 (formed from tiles a, b and c) and its representation by a list:

b	a	c	a
c	a	b	b
c	a	a	a
b	b	a	c

[[b,a,c,a],[c,a,b,b],[c,a,b,b],[a,a,a,c]]

Clearly, we can identify hypertiles of rank i with functions from $\{1, \ldots, 2^i\} \times \{1, \ldots, 2^i\}$ into $\{t_0, \ldots, t_k\}$. We call a hypertile a *tiling* if the corresponding function is a tiling.

In addition to the predicates `on` and `to`, the program $\mathcal{P}$ defines n binary predicates $\mathtt{tiling}_1$, $\ldots$, $\mathtt{tiling}_n$.

A statement $\mathtt{tiling}_i(H, T_0)$ is intended to hold if and only if (i) H is a hypertile of rank i; (ii) H is a tiling with the tile T_0 at the top left corner. Thus, the goal $\mathcal{G}$ consisting of one atom $\mathtt{tiling}_n(H, t_0)$ will succeed with respect to $\mathcal{P}$ if and only if there is a tiling for given $\{t_0, \ldots, t_k\}$, `on`, `to` and n.

The predicate $\mathtt{tiling}_1$ is defined in the following obvious way:

X_1	X_2
X_3	X_4

$$
\begin{array}{l}
\texttt{tiling}_1(\texttt{[}X_1,X_2,X_3,X_4\texttt{]},X_1)\ \texttt{:-}\\
\quad \texttt{on}(X_1,X_3),\\
\quad \texttt{on}(X_2,X_4),\\
\quad \texttt{to}(X_1,X_2),\\
\quad \texttt{to}(X_3,X_4).
\end{array}
$$

To define $\texttt{tiling}_{i+1}$, we use the following observation. Consider a hypertile H

$$\texttt{[[}X_1,X_2,X_3,X_4\texttt{]},\texttt{[}Y_1,Y_2,Y_3,Y_4\texttt{]},\ \texttt{[}Z_1,Z_2,Z_3,Z_4\texttt{]},\texttt{[}U_1,U_2,U_3,U_4\texttt{]}\ \texttt{]}$$

of rank $i+1$ and 9 its subsquares, hypertiles of rank i, shown below

X_1	X_2	Y_1	Y_2
X_3	X_4	Y_3	Y_4
Z_1	Z_2	U_1	U_2
Z_3	Z_4	U_3	U_4

X_1	X_2
X_3	X_4

X_2	Y_1
X_4	Y_3

Y_1	Y_2
Y_3	Y_4

X_3	X_4
Z_1	Z_2

X_4	Y_3
Z_2	U_1

Y_3	Y_4
U_1	U_2

Z_1	Z_2
Z_3	Z_4

Z_2	U_1
Z_4	U_3

U_1	U_2
U_3	U_4

It is easy to see that H is a tiling if and only if these 9 hypertiles are tilings too. Thus, $\texttt{tiling}_{i+1}$ can be defined as follows:

$$
\begin{array}{l}
\texttt{tiling}_{i+1}(\texttt{[[}X_1,X_2,X_3,X_4\texttt{]},\ \texttt{[}Y_1,Y_2,Y_3,Y_4\texttt{]},\\
\qquad\qquad \texttt{[}Z_1,Z_2,Z_3,Z_4\texttt{]},\ \texttt{[}U_1,U_2,U_3,U_4\texttt{]]},T_0)\ \texttt{:-}\\
\quad \texttt{tiling}_i(\texttt{[}X_1,X_2,X_3,X_4\texttt{]},T_0),\\
\quad \texttt{tiling}_i(\texttt{[}X_2,Y_1,X_4,Y_3\texttt{]},\_),\\
\quad \texttt{tiling}_i(\texttt{[}Y_1,Y_2,Y_3,Y_4\texttt{]},\_),\\
\quad \texttt{tiling}_i(\texttt{[}X_3,X_4,Z_1,Z_2\texttt{]},\_),\\
\quad \texttt{tiling}_i(\texttt{[}X_4,Y_3,Z_2,U_1\texttt{]},\_),\\
\quad \texttt{tiling}_i(\texttt{[}Y_3,Y_4,U_1,U_2\texttt{]},\_),\\
\quad \texttt{tiling}_i(\texttt{[}Z_1,Z_2,Z_3,Z_4\texttt{]},\_),\\
\quad \texttt{tiling}_i(\texttt{[}Z_2,U_1,Z_4,U_3\texttt{]},\_),\\
\quad \texttt{tiling}_i(\texttt{[}U_1,U_2,U_3,U_4\texttt{]},\_).
\end{array}
$$

The program $\mathcal{P}$ described above is nonrecursive and range-restricted. Clearly, $\mathcal{P}$ and the goal $\mathcal{G}$ are constructed from the instance I of TILING in polynomial time.

□

Theorem 3.2 The query answering problem is NEXP-hard for the following classes of nonrecursive range-restricted logic programs:

- logic programs with trees in any signature with at least one function symbol of arity ≥ 2;
- logic programs with bags;
- logic programs with sets;
- logic programs with any combination of the above types of complex values (for example, logic programs with trees and bags).

Proof. The program $\mathcal{P}$ from the proof of Theorem 3.1 contains the list constructor, but instead we could have used any function symbol of arity ≥ 2 if this symbol is different from the bag and set constructors. Thus, NEXP-hardness holds for nonrecursive logic programs with trees.

How to construct $\mathcal{P}$ in the case when Σ contains only the bag constructor and/or the set constructor? We can simulate lists $[s_1, s_2, s_3, s_4]$ by means of bags or sets, i.e. we can use bags and sets to define predicates equivalent to the predicates of $\mathcal{P}$. For example, a list $[s_1, s_2, s_3, s_4]$ can be represented by the set of pairs $(s_1, 1), \ldots, (s_4, 4)$ where $1, \ldots, 4$ are, in turn, represented by sets of cardinalities $1, \ldots, 4$. □

4 Inclusion in NEXP

Let E be an equational theory. We say that an equation $s = t$ has a *solution* σ in E if σ is a substitution such that the equation $s\sigma = t\sigma$ belongs to E (where $t\sigma$ and $s\sigma$ denote the results of application of σ to s and t respectively). A *system of equations* is a finite set of equations. A system of equations is said to be *solvable* in E if there exists a substitution σ such that σ is a solution for each equation in the system. By the *solvability problem* for E we mean the following decision problem: given a system of equations, is the system solvable in E?

Theorem 4.1 Let E be an equational theory such that the solvability problem for E is in NP. Then the query answering problem for nonrecursive logic programs over E is in NEXP.

Proof. Let $\mathcal{P}$ be a logic program over E. We define a binary relation $\rightarrow$ on pairs $(\mathcal{G}, \mathcal{E})$, where $\mathcal{G}$ is a goal and $\mathcal{E}$ is a system of equations. Namely, we have

$$(\mathcal{G}, \mathcal{E}) \rightarrow (\mathcal{G}', \mathcal{E}')$$

if the following conditions are satisfied:

1. $\mathcal{G}$ contains an atom $p(t_1, \ldots, t_n)$.
2. Some clause in $\mathcal{P}$ has a variant $p(s_1, \ldots, s_n)$:- $B_1, \ldots, B_m$ whose variables are disjoint with the variables of $(\mathcal{G}, \mathcal{E})$.
3. $\mathcal{G}'$ is $(\mathcal{G} \setminus \{p(t_1, \ldots, t_n)\}) \cup \{B_1, \ldots, B_m\}$;
4. $\mathcal{E}'$ is $\mathcal{E} \cup \{s_1 = t_1, \ldots, s_n = t_n\}$.

By an *constraint SLD-derivation* we mean a sequence of pairs

$$(\mathcal{G}_1, \mathcal{E}_1), \ldots, (\mathcal{G}_N, \mathcal{E}_N) \tag{1}$$

such that $(\mathcal{G}_i, \mathcal{E}_i) \to (\mathcal{G}_{i+1}, \mathcal{E}_{i+1})$ holds for all $1 \leq i \leq N-1$. By the standard technique (see for example [18, 11]) one can prove that a goal $\mathcal{G}$ succeeds with respect $\mathcal{P}$ if and only if there is a constraint SLD-derivation (1) such that (1) $\mathcal{G}_1 = \mathcal{G}$; (2) $\mathcal{E}_1 = \emptyset$; (3) $\mathcal{G}_N = \Box$; and (4) $\mathcal{E}_N$ is solvable in E.

It is easy to define a nondeterministic Turing machine M that decides the query answering problem by generating constraint SLD-derivations for every input $\mathcal{P}$ and $\mathcal{G}$. Namely, this machine nondeterministically makes transitions from $(\mathcal{G}_i, \mathcal{E}_i)$ to $(\mathcal{G}_{i+1}, \mathcal{E}_{i+1})$ (connected by the relation $\to$). When $\mathcal{G}_i$ is not empty and it is impossible to apply this step, M rejects the input. When $\mathcal{G}_i$ is empty, M should check whether $\mathcal{E}_i$ is solvable in E. By the theorem condition, this can be done in nondeterministic polynomial time (in the size of $\mathcal{E}_i$). If $\mathcal{E}_i$ is solvable then M accepts the input, otherwise the input is rejected. We show that M works within time $2^{n^{O(1)}}$ where n is the combined size of $\mathcal{P}$ and $\mathcal{G}$.

Since $\mathcal{P}$ is nonrecursive, by definition there is a numbering $p_1, \ldots, p_r$ of all predicates in $\mathcal{P}$ such that for every clause, if p_i occurs in the head and p_j occurs in the body then $i > j$. Let m be the maximum length of clause bodies in $\mathcal{P}$. We define the *weight* of an atom $p_i(t_1, \ldots, t_k)$ as the number $(m+1)^{i+1}$. The weight of a goal is the sum of weights of its atoms.

Consider any constraint SLD-derivation generated by M. Evidently, the weight of the initial goal $\mathcal{G}$ is exponential in n. For each transition $(\mathcal{G}_i, \mathcal{E}_i) \to (\mathcal{G}_{i+1}, \mathcal{E}_{i+1})$, the weight of $\mathcal{G}_{i+1}$ is less than the weight of $\mathcal{G}_i$. Therefore, the total size of all goals and the total size of all systems of equations is exponential in n.

The running time of M consists of the time of generating a constraint SLD-derivation and the time of checking solvability of the final system of equation. Since the size of the constraint SLD-derivation is exponential in n, the time of generating is exponential too. The time of checking solvability is $2^{n^{O(1)}}$ because the final system of equations has the size of the same order as the constraint SLD-derivation, i.e. exponential size, and the solvability problem for E is in NP by the condition of the theorem. $\Box$

Theorem 4.2 The query answering problem is NEXP-complete for the following classes of nonrecursive logic programs:

- logic programs with trees in any signature with at least one function symbol of arity ≥ 2;
- logic programs with bags;
- logic programs with sets;
- logic programs with any combination of the above types of complex values.

Proof. NEXP-hardness was proved in Theorem 3.2. The paper [8] describes a nondeterministic polynomial-time unification algorithm for terms representing any combination of trees, bags and sets. Thereby, the solvability problem for corresponding equational theories is in NP. Therefore, the inclusion in NEXP immediately follows from Theorem 4.1 above and [8]. $\Box$

References

[1] S. Abiteboul and C. Beeri. The power of languages for the manipulation of complex values. *VLDB Journal*, 4:727–794, 1995.

[2] S. Abiteboul and S. Grumbach. Col: A logic-based language for complex objects. In J. Schmidt, S. Ceri, and M. Missikoff, editors, *Advances in Database Technology - EDBT'88. Proceedings of the International Conference on Extending Database Technology*, volume 303 of *Lecture Notes in Computer Science*, pages 271–293, Venice, Italy, March 1988. Springer Verlag.

[3] S. Abiteboul and S. Grumbach. A rule-based language with functions and sets. *ACM Transactions on Database Systems*, 16(1):1–30, 1991.

[4] K.R. Apt. Logic programming. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Methods and Semantics, chapter 10, pages 493–574. Elsevier Science, Amsterdam, 1990.

[5] C. Beeri and Y. Kornatzky. A logical query language for hypermedia systems. *Information Sciences*, 77:1–38, 1994.

[6] C. Beeri, S. Naqvi, O. Schmueli, and S. Tsur. Set constructors in a logic database language. *Journal of Logic Programming*, 10:181–232, 1991.

[7] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. In *Proceedings Twelfth Annual IEEE Conference on Computational Complexity*, pages 82–101, Ulm, Germany, June 1997.

[8] E. Dantsin and A. Voronkov. Bag and set unification. UPMAIL Technical Report 150, Uppsala University, Computing Science Department, September 1997.

[9] E. Dantsin and A. Voronkov. Complexity of query answering in logic databases with complex values. In S. Adian and A. Nerode, editors, *Logical Foundations of Computer Science. 4th International Symposium, LFCS'97*, volume 1234 of *Lecture Notes in Computer Science*, pages 56–66, Yaroslavl, Russia, July 1997.

[10] A. Dovier, E.G. Omodeo, E. Pontelli, and G. Rossi. {log}: A language for programming in logic with finite sets. *Journal of Logic Programming*, 28(1):1–44, 1996.

[11] M. Gabbrielli, G.M. Dore, and G. Levi. Observable semantics for constraint logic programs. *Journal of Logic and Computation*, 5(2):133–171, 1995.

[12] J.H. Gallier and S. Raatz. Extending SLD-resolution to equational Horn clauses using E-unification. *Journal of Logic Programming*, 6(3):3–44, 1989.

[13] S. Grumbach and V. Vianu. Tractable query languages for complex object databases. *Journal of Computer and System Sciences*, 51(2):149–167, 1995.

[14] P. Kanellakis, G.M. Kuper, and P.Z. Revesz. Constraint query languages. *Journal of Computer and System Sciences*, 51:26–52, 1995.

[15] G.M. Kuper. Logic programming with sets. *Journal of Computer and System Sciences*, 41:44–64, 1990.

[16] N. Leone and P. Rullo. Ordered logic programming with sets. *Journal of Logic and Computation*, 3(6):621–642, 1993.

[17] J.W. Lloyd. *Foundations of Logic Programming (2nd edition)*. Springer Verlag, 1987.

[18] M.J. Maher. A CLP view of logic programming. In *Proc. Conf. on Algebraic and Logic Programming*, volume 632 of *Lecture Notes in Computer Science*, pages 364–383, October 1992.

[19] C.H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[20] O. Shmueli, S. Tsur, and C. Zaniolo. Compilation of set terms in the logic data language (LDL). *Journal of Logic Programming*, 12(1):89–119, 1992.

[21] R.W. Topor. Domain independent formulas and databases. *Theoretical Computer Science*, 52(3):281–306, 1987.

[22] K. Vadaparty. On the power of rule-based languages with sets. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 26–36, 1991.

[23] A. Voronkov. Logic programming with bounded quantifiers. In A. Voronkov, editor, *Logic Programming*, volume 592 of *Lecture Notes in Artificial Intelligence*, pages 486–514. Springer Verlag, 1992.