Complexity of Query Answering in Logic Databases with Complex Values

Evgeny Dantsin[1]* and Andrei Voronkov[2]**

[1] Steklov Institute of Mathematics at St.Petersburg
Fontanka 27, St.Petersburg 191011, Russia
[2] Computing Science Department, Uppsala University
Box 311, S-751 05 Uppsala, Sweden

Abstract. This paper studies the complexity of the query answering problem in logic databases with complex values. Logic databases are represented as Horn clause logic programs; complex values are described in a data model based on equational theories. As examples of complex values, we consider trees, bags and finite sets. We give a natural sufficient condition under which the query answering problem for non-recursive programs with complex values is in NEXP. In particular, logic programs with trees, bags and finite sets satisfy this condition. We also show that the query answering problem for non-recursive range restricted logic programs is NEXP-hard. Thereby, the query answering problem for logic databases with trees, bags and sets turns out to be NEXP-complete.

1 Introduction

The relational model of data deals with simple values, namely tuples consisting of atomic components. Various generalizations and formalisms have been proposed to handle more complex values like nested tuples, tuples of sets, etc. For example, Kuper [14] introduced logic databases with sets. A thorough discussion of such formalisms and their expressive power may be found in Abiteboul and Beeri [2]. Our paper studies complexity aspects of databases with complex values.

There are three main kinds of complexity in databases: the *data complexity*, the *program complexity* (both defined in Vardi [22]) and the *combined complexity* (see e.g. the survey [6]). The data complexity means that data (represented by the facts of a logic programs) are fixed, while a query (represented by the rules of a logic program and a goal) is variable. The program complexity (sometimes called the *expression complexity*) means that data are variable, while a query is fixed. The combined complexity means that both data and query are variable.

By the *query answering complexity* we mean the combined complexity of checking whether a given goal succeeds with respect to a given logic program. Imagine a database user asking a query. In the case of complex values, an answer

* Supported by grants from the Swedish Royal Academy of Sciences, the Swedish Institute and RFBR/INTAS.
** Supported by a TFR grant.

can be too large or even infinite. The query answering complexity measures how long the user should wait until the fact of the mere existence of a positive answer is established.

The query answering problem can be naturally formalized as the decision problem for fragments of first-order logic corresponding to Horn clause logic programs. Decidability and complexity of fragments of predicate calculus have been actively studied (see e.g. the recent monograph [5]). As for logic programming, the decidability and complexity have been basically considered in the context of non-monotonic semantics of propositional programs or programs without function symbols. Surprisingly, very little is known about the complexity of query languages for complex values.

This paper contains some complexity bounds for logic programs with complex values, including trees, bags and sets. Such results are useful for the design of future database query languages able to manage complex values. To the best of our knowledge, this is the first analysis of the complexity for databases with complex values. So far, the complexity was mainly analyzed for constraint satisfiability.

This paper is structured as follows. In Section 2 we introduce our data model and logic programs with complex values. We also give examples of complex values. In Section 3 we prove NEXP-hardness of the query answering problem for non-recursive range restricted logic programs. In Section 4 we consider complex values described by theories satisfying some very general assumption, called NP-solvability. We show that the query answering problem for logic programs with such complex values is in NEXP. In Section 5 we use NP-solvability for trees, bags and sets to obtain NEXP-completeness for a number of classes of non-recursive databases with complex values.

2 Preliminaries

2.1 Data model

In order to define what we mean by *complex values,* we shall introduce the corresponding data model. Since we consider the logical model of computation, databases are represented as logic programs. In logic programs, values are represented by ground terms, and two values are equal when the corresponding terms coincide. In our data model, values are also represented by ground terms, but the interpretation of equality can be arbitrary. To formalize this, we use the notion of *Herbrand interpretations with respect to equational theories*. Thus, our notion of logic databases with complex values is more general than *well-behaved equational logic programs* by Gallier and Raatz [10] but in some sense less general than that of constraint logic programs or constraint databases (see Jaffar and Maher [11]). The generalization of our results to constraint logic programs can also be done, but the notion of data becomes less transparent.

Herbrand interpretations. Let Σ be a signature, i.e. a set of function and predicate symbols containing at least one constant; constants are considered to be function symbols of arity 0. Terms, atoms, substitutions, instances, variants etc. are defined in the standard way (see, for example, Apt [3]). The *Herbrand universe* U_Σ for Σ is the set of all *ground terms over* Σ, that is terms without variables. The *Herbrand base* B_Σ for Σ is the set of all ground atoms of Σ. A *Herbrand interpretation for* Σ is any subset of B_Σ (all atoms in this set are considered to be true, all other ground atoms are considered to be false).

Equational theories. By an *equation* over Σ we mean an atom $s = t$, where the equality symbol $=$ does *not* belong to Σ. An equation is said to be *ground* if s and t are ground. An *equational theory* E *over* Σ 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 $s_1 = t_1, \dots, s_n = t_n$ then E contains $f(s_1, \dots, s_n) = f(t_1, \dots, t_n)$ for each n-ary function symbol $f \in \Sigma$.

The solvability problem for equational theories. We say that an equation $s = t$ has a *solution* σ in an equational theory 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. The *solvability problem* for an equational theory E is the following decision problem: given a system of equations, is the system solvable in E? We call an equational theory *NP-solvable* if the solvability problem for E is in NP. Similar definition can be given for other complexity classes (for example, P-solvable equational theories), but in this paper we consider only NP-solvable theories.

Herbrand interpretations with respect to equational theories. By a *Herbrand interpretation with respect to an equational theory* E we mean a Herbrand interpretation $\mathcal{I} \subseteq B_\Sigma$ such that if (i) an atom $p(s_1, \dots, s_n)$ belongs to $\mathcal{I}$ and (ii) the equational theory E contains equations $s_1 = t_1, \dots, s_n = t_n$, then $p(t_1, \dots, t_n)$ belongs to $\mathcal{I}$ too. This notion can be viewed as a formalization of our data model.

Examples of complex values. *Trees* are data represented by ground terms, and the equality relation on them is defined as identity. Thus, trees correspond to equational theories that consist of equations $t = t$.

Lists are a special kind of trees. In logic programming, lists are constructed from the empty list `[]` by using *the list constructor* $[s \mid t]$ whose second argument t is a list. The list $[t_1 \mid \dots [t_n \mid []] \dots]$ is denoted by $[t_1, \dots, t_n]$. The equational theory for lists contains no additional axioms in comparison with 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* $\langle s \mid t\rangle$ instead of the list constructor $[s \mid t]$ and $\langle t_1, \ldots, t_n\rangle$ instead of $[t_1, \ldots, t_n]$. Bags can be axiomatized by ground equations $\langle r \mid \langle s \mid t\rangle\rangle = \langle s \mid \langle r \mid t\rangle\rangle$, or alternatively by ground equations $\langle t_1, \ldots, t_m, t_{m+1}, \ldots, t_n\rangle = \langle t_1, \ldots, t_{m+1}, t_m, \ldots, t_n\rangle$.

Sets can be viewed as bags in which the number of occurrences of an element in immaterial. Similar to bags, we shall use the *set constructor* $\{s \mid t\}$ denoting the set obtained from the set t by adding s as an element. Sets can be axiomatized as follows. In the axioms for bags we replace the bag constructor by the set constructor and add all ground equations $\{s_1, s_1, \ldots, s_n\} = \{s_1, \ldots, s_n\}$. Note that we consider *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, Dovier et.al. [8] represent sets by terms $\{s_1, \ldots, s_n \mid t\}$ where t can be any term not 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". Our approach is simpler and 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

We fix a signature Σ and assume that all definitions are given with respect to 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 in the signature Σ. A *goal* is a set $\{A_1, \ldots, A_n\}$ of atoms, 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 logic program $\mathcal{P}$ *over an equational theory* E, i.e. in fact a pair $(\mathcal{P}, E)$. The semantics of such programs can be given in many ways by adapting standard logic programming semantics [3, 16, 10] for the corresponding equational theory. For example, one can define the least fixpoint semantics or a generalization of SLD-resolution. We define the semantics by using the notion of 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 $\subseteq$).

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. Similarly, we use the terms *logic programs with bags, trees* etc. To illustrate this notions, we shall define a simple logic program over sets and bags (at the same time), which defines the predicate `set_to_bag`. This predicate can be used to transform a set into a corresponding bag and conversely, bag into the corresponding set.

```
set_to_bag({},⟨⟩).
set_to_bag({x | y},⟨x | z⟩) :-
  set_to_bag(y,z).
```

We give no other examples of such programs here, there is a vast literature on this subject (e.g. [12, 14, 1, 4, 18, 20, 21, 2, 23]).

We introduce the following requirement concerning signatures and equational theories of bags and sets. Let E be an equational theories over a signature Σ. If Σ contains the bag constructor $\langle \ldots \mid \ldots \rangle$, then Σ contains the constant $\langle\rangle$ and E is required to include the axioms for bags. Similarly, if E contains the set constructor $\{\ldots \mid \ldots\}$, then Σ contains the constant $\{\}$ and E is required to include the axioms for sets.

Special kinds of logic programs. For every logic program $\mathcal{P}$, we define a directed graph $G_{\mathcal{P}}$ as follows. The set of vertices of $G_{\mathcal{P}}$ is the set of predicates symbols of $\mathcal{P}$; there is an edge from a vertex p_2 to a vertex p_1 if and only if there is a clause A :- $A_1, \ldots, A_n$ in $\mathcal{P}$ such that p_1 is the predicate symbol of A and p_2 occurs in $A_1, \ldots, A_n$. A logic program $\mathcal{P}$ is said to be *non-recursive* if the graph $G_{\mathcal{P}}$ contains no cycles.

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 appear in $A_1, \ldots, A_n$.

Range restricted logic programs have many desirable features, mainly related to *domain independence.* Range restricted non-recursive logic programs containing only constants and predicate symbols are equivalent to positive relational algebra. These classes of logic programs have been actively studied in database theory.

2.3 Query answering complexity

The SUCCESS problem. 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 $A_1, \ldots, A_n$ belong to the least Herbrand model of $\mathcal{P}$ over E. Let an equational theory E be fixed and $\mathfrak{C}$ be a set of logic programs (over E). By the *SUCCESS 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?

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 (as for polynomial-time reduction). 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 $\mathcal{P}' = \mathcal{P} \cup \{yes \texttt{ :- } A_1, \ldots, A_n\}$. 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 (non-recursive and/or range restricted programs).

3 NEXP-hardness for non-recursive range restricted logic programs

Theorem 1 *Let $\mathfrak{C}$ be the set of non-recursive range restricted logic programs with lists. Then the SUCCESS problem for $\mathfrak{C}$ is NEXP-hard.*

Proof. We reduce the TILING problem, which is known to be NEXP-complete (see e.g. [19], page 501), to the SUCCESS problem for $\mathfrak{C}$.

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

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

$\mathtt{tiling}_1(\mathtt{[}X_1, X_2, X_3, X_4\mathtt{]}, X_1)$ `:-`
  $\mathtt{on}(X_1, X_3),$
  $\mathtt{on}(X_2, X_4),$
  $\mathtt{to}(X_1, X_2),$
  $\mathtt{to}(X_3, X_4).$

To define $\mathtt{tiling}_{i+1}$, 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

$$\begin{array}{|c|c|c|c|}\hline X_1 & X_2 & Y_1 & Y_2 \\ \hline X_3 & X_4 & Y_3 & Y_4 \\ \hline Z_1 & Z_2 & U_1 & U_2 \\ \hline Z_3 & Z_4 & U_3 & U_4 \\ \hline \end{array}
\qquad
\begin{array}{ccc}
\begin{array}{|c|c|}\hline X_1 & X_2 \\ \hline X_3 & X_4 \\ \hline \end{array} &
\begin{array}{|c|c|}\hline X_2 & Y_1 \\ \hline X_4 & Y_3 \\ \hline \end{array} &
\begin{array}{|c|c|}\hline Y_1 & Y_2 \\ \hline Y_3 & Y_4 \\ \hline \end{array} \\
\begin{array}{|c|c|}\hline X_3 & X_4 \\ \hline Z_1 & Z_2 \\ \hline \end{array} &
\begin{array}{|c|c|}\hline X_4 & Y_3 \\ \hline Z_2 & U_1 \\ \hline \end{array} &
\begin{array}{|c|c|}\hline Y_3 & Y_4 \\ \hline U_1 & U_2 \\ \hline \end{array} \\
\begin{array}{|c|c|}\hline Z_1 & Z_2 \\ \hline Z_3 & Z_4 \\ \hline \end{array} &
\begin{array}{|c|c|}\hline Z_2 & U_1 \\ \hline Z_4 & U_3 \\ \hline \end{array} &
\begin{array}{|c|c|}\hline U_1 & U_2 \\ \hline U_3 & U_4 \\ \hline \end{array}
\end{array}$$

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:

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

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

Theorem 2 *The SUCCESS problem for the following classes of non-recursive range restricted logic programs is NEXP-hard:*

1. *logic programs with trees in any signature with at least one function symbol of arity* ≥ 2*;*
2. *logic programs with bags;*
3. *logic programs with sets;*
4. *logic programs with any combination of the above data (for example, logic programs with trees and bags).*

Proof. The program $\mathcal{P}$ from the proof of Theorem 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 non-recursive 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

Theorem 3 *Let E be an NP-solvable equational theory. The SUCCESS problem for the set of non-recursive 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. We have

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

if the following conditions hold:

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 from 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 *SLD-derivation with constraints* we mean a sequence of pairs

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

such that $(\mathcal{G}_i, \mathcal{E}_i) \rightarrow (\mathcal{G}_{i+1}, \mathcal{E}_{i+1})$ holds for all $1 \leq i \leq N-1$. By the standard technique (see e.g. [17, 9]) one can prove that a goal $\mathcal{G}$ succeeds with respect $\mathcal{P}$ if and only if there is an SLD-derivation with constraints such that (i) $\mathcal{G}_1 = \mathcal{G}$; (ii) $\mathcal{E}_1 = \emptyset$; (iii) $\mathcal{G}_N = \Box$; and (iv) $\mathcal{E}_N$ is solvable in E.

It is easy to define a nondeterministic Turing machine M that decides the SUCCESS problem by generating SLD-derivations with constraints 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 $\rightarrow$). 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 nondeterministically checks whether $\mathcal{E}_i$ is solvable in E. If it 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 $\mathcal{G}$.

Consider the graph $G_{\mathcal{P}}$ defined in Subsection 2.2 (recall that we identify its vertices with predicate symbols of $\mathcal{P}$). For every predicate symbol p in $\mathcal{P}$, we define its *height* as the number of edges in the longest path in $G_{\mathcal{P}}$ coming from p. Since $\mathcal{P}$ is non-recursive, the height is bounded by the number of predicate symbols in $\mathcal{P}$. Let m be the maximum length of clause bodies in $\mathcal{P}$. We define the *weight* of an atom $p(t_1, \ldots, t_k)$ as the number $(m+1)^{h+1}$, where h is the height of p. The weight of a goal is the sum of weights of its atoms.

Consider any SLD-derivation generated by M. Evidently, the weight of the initial goal $\mathcal{G}$ is exponential in n. For each transition $(\mathcal{G}_i, \mathcal{E}_i) \rightarrow (\mathcal{G}_{i+1}, \mathcal{E}_{i+1})$, the weight of $\mathcal{G}_{i+1}$ is strictly 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 an SLD-derivation and the time of checking solvability of the final system of equation. Since the size of the SLD-derivation is exponential in n, the time of generating is exponential too. The time of checking solvability is exponential because the final system of equations has the size of the same order as the SLD-derivation, i.e. $2^{n^{O(1)}}$, and the theory E is NP-solvable by the condition of the theorem.

5 NEXP-completeness

Let Σ be a signature containing the bag and set constructors. Let E_Σ be the equational theory over Σ that consists of only the axioms for bags and sets and their logical consequences. We call E_Σ a *combined theory of trees, bags and sets.*

Lemma 1 *Any combined theory of trees, sets and bags is NP-solvable.*

The detailed proof may be found in our technical report [7].

Theorem 4 *The SUCCESS problem for the following classes $\mathfrak{C}$ of non-recursive logic programs is NEXP-complete:*

1. *logic programs with trees in any signature with at least one function symbol of arity ≥ 2;*
2. *logic programs with bags;*
3. *logic programs with sets;*
4. *logic programs with any combination of the above data (for example, logic programs with trees and bags).*

The same holds for the subclass of range restricted programs in $\mathfrak{C}$.

Proof. Immediate from Theorems 2 and 3 and Lemma 1.

References

1. S. Abiteboul and S. Grumbach. A rule-based language with functions and sets. *ACM Transactions on Database Systems*, 16(1):1–30, 1991.
2. S. Abiteboul and C. Beeri. The power of languages for the manipulation of complex values. *VLDB Journal*, 4:727–794, 1995.

3. 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.
4. C. Beeri, S. Naqvi, O. Schmueli, and S. Tsur. Set constructors in a logic database language. *Journal of Logic Programming*, 10:181–232, 1991.
5. E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Springer Verlag, 1997.
6. E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. In *CCC'97*, 1997. To appear.
7. E. Dantsin and A. Voronkov. Complexity of query answering in logic databases with complex values. UPMAIL technical report, Uppsala University, Computing Science Department, 1997. `http://www.csd.uu.se/papers/reports.html`
8. 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.
9. M. Gabbrielli, G.M. Dore, and G. Levi. Observable semantics for constraint logic programs. *Journal of Logic and Computation*, 5(2):133–171, 1995.
10. 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.
11. J. Jaffar and M. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19,20:503–581, 1994.
12. B. Jayaraman and D.A. Plaisted. Programming with equations, subsets and relations. In *Proc. NACLP'89*, Cleveland, 1989. MIT Press.
13. D. Kapur and P. Narendran. NP-completeness of the set unification and matching problems. In J. Siekmann, editor, *Proc. 8th CADE*, volume 230 of *Lecture Notes in Computer Science*, 1986.
14. G.M. Kuper. Logic programming with sets. *Journal of Computer and System Sciences*, 41:44–64, 1990.
15. G.M. Kuper and M.Y. Vardi. The logical data model. *ACM Transactions on Database Systems*, 18(3):379–413, 1993.
16. J.W. Lloyd. *Foundations of Logic Programming (2nd edition)*. Springer Verlag, 1987.
17. 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.
18. T. Munakata. Notes on implementing sets in Prolog. *Communications of the ACM*, 35(3):112–120, 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. G. Smolka and R. Treinen. Records for logic programming. *Journal of Logic Programming*, 18:229–258, 1994.
22. M. Vardi. The complexity of relational query languages. In *Proc. 14th ACM STOC*, 1982.
23. A. Voronkov. Logic programming with bounded quantifiers revisited. UPMAIL Technical Report, Uppsala University, Computing Science Department, 1997, to appear.