Complexity of Nonrecursive Logic Programs with Complex Values*

Sergei Vorobyov
Max-Planck Institut für Informatik
Im Stadtwald, D-66123, Saarbrücken, Germany
sv@mpi-sb.mpg.de

Andrei Voronkov
Computing Science Department, Uppsala University
Box 311, S-75105, Uppsala, Sweden
voronkov@csd.uu.se

Abstract

We investigate complexity of the SUCCESS problem for logic query languages with complex values: check whether a query defines a nonempty set. The SUCCESS problem for recursive query languages with complex values is undecidable, so we study the complexity of nonrecursive queries. By complex values we understand values such as trees, finite sets, and multisets. Due to the well-known correspondence between relational query languages and datalog, our results can be considered as results about relational query languages with complex values. The paper gives a complete complexity classification of the SUCCESS problem for nonrecursive logic programs over trees depending on the underlying signature, presence of negation, and range restrictedness. We also prove several results about finite sets and multisets.

1 Introduction

A number of complexity results have been established for logic query languages. They are surveyed in [49, 18]. The major themes in these results are the complexity and expressive power of extensions of datalog: the logic query language for describing relations over tuples of simple, non-structured objects. Due to the well-known correspondence between datalog and relational algebra, the complexity results about nonrecursive datalog can be restated as results about relational query languages, for example, SQL-92 or its fragments.

New relational query languages, for example SQL-3, extend traditional languages in several directions. One of them is the introduction of complex values, like sets. There is no uniform convention on how complex values should be handled in relational query languages. Indeed, the introduction of any new type of values requires addition of new operations on the corresponding algebras.

In logic query languages, stemming from logic programming, there seems to be a uniform viewpoint on how complex values should be treated. There are two major approaches.

1. In *constraint logic programming* [41, 42] and *constraint databases* [33] any value is identified by the set of constraints true on this value. The addition of a new type of values requires the addition of new constraint predicates. A similar approach to relational query languages was also considered in [10].
2. Another approach to adding complex values, which can be called *structural*, requires that values be represented by means of their structure. For example, to represent sets one may enrich the language with constant $\emptyset$ to denote the empty set and the set constructor $\{s|t\}$ denoting the addition of an element s to the set t. Then the set $\{t_1, \ldots, t_n\}$ will be denoted by the term $\{t_1| \ldots \{t_n|\emptyset\} \ldots\}$. The only changes to the semantics of logic programming are the changes in the treatment of equality, since new predicate symbols are not free constructors. Such an approach is considered in a number of papers, for example [25, 37, 9, 50, 22, 21, 19].

Of course, a combination of the two approaches is also possible.

This paper studies complexity of nonrecursive query answering in logic databases with complex values. By complexity we mean expression complexity formalized as the following decision problem: given both the database and a query, whether the output relation is nonempty. We call this problem the *SUCCESS problem.* For the query languages that we study, there is no difference between the expression complexity and the combined complexity as defined in [55], since the input database can be considered as part of the query without changing the output relation. It is also easy to see that the complexity of the SUCCESS problem is exactly the expression (or the combined) complexity. Nonrecursive queries in logic databases are represented by nonrecursive logic programs. Among nonrecursive logic programs we distinguish range-restricted ones, as of special interest for databases. By complex values we understand various versions of trees, finite multisets and sets. Our main results are proved for trees represented as terms composed from free constructors. For example, we give a complete classification of complexity for nonrecursive logic programs with trees depending on the signature, range-restrictedness, and presence of negation. We also prove several results about logic programs over various domains of sets.

There is yet no generally accepted approach to using sets and multisets in deductive databases. Existing implementations use different approaches. Sets and multisets in deduc-

*Full version available as Research Report MPI-I-97-2-010 from `http://data.mpi-sb.mpg.de/internet/reports.nsf/NumberView/97-2-010`, [62]

In: Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'98), June 1-3, 1998, Seattle, WA.
Jan Paredaens, Latha Colby, editors.

tive databases should further be discussed, both from practical and theoretical viewpoints. Our results about unary signatures can be interpreted as results about lists, often used in deductive databases to replace sets and multisets.

It is possible to reformulate our complexity results in terms of some relational query languages with complex values, using suitable algebraic formalizations corresponding to nonrecursive logic programs, see, e.g., [1].

Results of this paper show that nonrecursive query languages for complex values are highly intractable. It will be interesting to investigate these classes in terms of fixed-parameter complexity similar to the analysis done in [46, 56, 29, 36].

We briefly mention some results on the complexity of recursive logic programs. These (and other results) are surveyed in [18].

Definite programs. For definite programs without function symbols, the SUCCESS problem is *DEXPTIME*-complete for recursive programs [55, 30] and *PSPACE*-complete for nonrecursive programs [55, 17, 31]. For definite programs, with function symbols, it is r.e.-complete [3, 53].

Normal programs. For logic programs with negation, several nonmonotonic semantics exist. In the case of *stratified* programs most of these semantics coincide with the *perfect model semantics*. For this semantics the following results are known. The SUCCESS problem for programs without function symbols is *PSPACE*-complete [55, 17, 31] in the case of nonrecursive programs and *NEXPTIME*-complete in the case of recursive programs [6]. In the case of arbitrary function symbols the SUCCESS problems for programs with n levels of stratified negation is Σ^0_n-complete [5]. The paper [13] addresses the expressive power of locally stratified recursive programs and shows that every Δ^1_1-set is definable by a stratified program over a perfect model.

In this paper we study the complexity of nonrecursive logic programs over complex values, such as trees, finite sets, and multisets. We do not study the expressive power of languages with complex values. We do not consider recursive query languages or aggregation.

2 Preliminaries

Logic and signatures. By first-order logic we mean first-order logic with equality. Formulas are constructed using all standard connectives, except the equivalence $\equiv$. The equality predicate is denoted $\approx$. We denote by $\Phi\left[\psi_i/\varphi_i\right]_{i=1}^{m}$ the formula obtained from Φ by simultaneous replacement of all occurrences of subformulas φ_i by formulas ψ_i.

Our results will depend on the signature in which programs are written. We will only consider *functional signatures*, consisting of finite or countable sets of function symbols.

If a signature Σ consists of k constants, l unary function symbols, and m function symbols of arity ≥ 2, we will denote such a signature by (k, l, m). We will also use ordinals and wildcards in the notation for signatures. For example, $(\omega, \_, 3)$ denotes any signature with infinitely many constants, any number of unary function symbols, and 3 function symbols of arity ≥ 2. Similarly, $(< \omega, 0, 0)$ denotes a signature with any finite number of constants and no function symbols of arity ≥ 1.

> **Proviso.** *We always assume that signatures have at least two symbols, including at least one constant.*

Other signatures can also be considered but are not interesting for our aims.

Logic programming. We assume elementary knowledge of standard facts about semantics of logic programming. They may be found in, e.g., [4] or [39]. *Clauses* will be written as

$$P(t_1, \ldots, t_n) \leftarrow L_1, \ldots, L_m.$$

We will consider both *normal clauses*, where each L_i is a literal, and *definite clauses*, where each L_i is an atomic formula. We assume that the equality predicate $\approx$ does not occur in clauses. A *normal (*respectively, *definite) program* is a finite set of normal (respectively, definite) clauses. A clause is called *range-restricted*, if all variables occurring in the clause also occur in a positive literal in the body.

Instead of considering a logic program $\mathcal{P}$, it will be convenient to work with *Clark's completion* [16] of $\mathcal{P}$, that is a set of *predicate definitions* of the form

$$\begin{array}{l} P_0(\bar{x}_0) \equiv \Phi_0, \\ \ldots \\ P_n(\bar{x}_n) \equiv \Phi_n, \end{array} \tag{1}$$

where all P_i are different predicates and each Φ_i is a first-order formula whose free variables are contained in $\bar{x}_i$.

For example, the set of two clauses

$$\begin{array}{l} P(a, x) \leftarrow R(x, z) \\ P(f(y), x) \leftarrow \neg R(x, z) \end{array}$$

can be rewritten as the following definition:

$$\begin{array}{rl} P(u, x) \equiv & \exists z(u \approx a \wedge R(x, z)) \vee \\ & \exists y \exists z(u \approx f(y) \wedge \neg R(x, z)) \end{array}$$

We only consider *nonrecursive* logic programs. In Clark's completion (1) of any nonrecursive logic program, each formula Φ_i may only contain predicate symbols $\approx, P_0, \ldots, P_{i-1}$. When a program is definite, the formulas Φ_i are constructed using only $\exists, \vee, \wedge$.

Semantics and term algebras. Semantics of normal logic programs is discussed in, e.g., [7]. In the nonrecursive case the semantics can be described by using the so-called term algebras.

Definition 2.1 The *term algebra* of a signature Σ, denoted $TA(\Sigma)$, is the algebra in which the carrier set is the set of ground terms of Σ, any ground term is interpreted by itself and any two distinct ground terms are unequal. □

Such algebras are also called *absolutely free algebras* [43]. In view of our proviso on signatures, term algebras considered in this paper have at least two elements.

> **Proviso.** *In this article we consider two formulas equivalent, if they are equivalent in any term algebra with at least two elements.*

By $Th(TA(\Sigma))$ we denote the *first-order theory* of $TA(\Sigma)$, i.e., the set of all sentences of the signature Σ with equality true in $TA(\Sigma)$. The decidability of $Th(TA(\Sigma))$ for finite signatures was proved in [44, 43] by quantifier elimination. Later [35, 40, 27] proved it again for the case of finite and infinite signatures, also by using quantifier elimination. The *PSPACE*-completeness of $Th(TA(\geq 2, 0, 0))$ is due to [52, 51]. The $LATIME(2^{O(n)})$-completeness[1] of $Th(TA(1, 2, 0))$ is due to [59]. For $Th(TA(\omega, \omega, \omega))$, [34] asserted *PSPACE*-completeness, but in [60] it was proved nonelementary recursive with a linearly growing stack of twos as a lower bound.

Nonrecursive definitions (1) can be regarded as explicit definitions of new predicates over a term algebra $TA(\Sigma)$, where Σ contains all symbols occurring in (1). The predicate P_0 is defined directly in terms of equality $\approx$, and each P_{i+1} is defined in terms of $P_0, \dots, P_i$. Thus, by eliminating explicit definitions, each P_i is explicitly defined by a formula of $Th(TA(\Sigma))$. This gives a straightforward semantics to nonrecursive logic programs. However, this semantics may depend on Σ (this fact is usually called *domain-dependence*). For example, the formula $\forall x(x \approx a \vee x \approx b)$ is true in $TA(\{a, b\})$ and false in $TA(\{a, b, c\})$. It is well known [2] that the semantics of P_i is domain-independent in the case of range-restricted clauses.

For any nonrecursive program $\mathcal{P}$, we will refer to the extension of $TA(\Sigma)$ by new predicates P_i explicitly defined by $\mathcal{P}$, as to the *perfect model* of $\mathcal{P}$ in Σ [47, 54, 38].

The SUCCESS problem. By the SUCCESS(Σ) *problem* for a class of logic programs $\mathcal{C}$ of the signature Σ we mean the following decision problem: given a logic program $\mathcal{P} \in \mathcal{C}$ and a goal G, does G succeed with respect to $\mathcal{P}$, where the definitions are understood as over $TA(\Sigma)$? When G succeeds with respect to $\mathcal{P}$ we can also say that the pair $(\mathcal{P}, G)$ defines a nonempty query.

Instead of checking whether G succeeds with respect to $\mathcal{P}$, we can introduce a new predicate `success`, add the clause `success` $\leftarrow G$ to $\mathcal{P}$, and ask whether `success` is true in the prefect model of the new program. This means that the last predicate P_n in (1) is `success` and we have to check whether the sentence defined by `success` with respect to (1) is true in $TA(\Sigma)$.

A standard form of logic programs. Since we assume at least two symbols in the signature, we can represent natural numbers less than or equal to some fixed number N by terms or sequences of terms of size $O(\log N)$. Let 0 and 1 be any two distinct ground terms. We encode any natural number $n < N$ as a sequence of terms 0 and 1 that forms the binary representation of n. For example, if $N = 6$, we need only 3 digits to represent numbers up to 6 and the number 2 can be represented as the sequence of arguments $(0, 1, 0)$ to some predicate P.

Now, we show that we can restrict ourselves to definitions (1) in which P_{i+1} is defined solely in terms of P_i, i.e., Φ_{i+1} uses only P_i and $\approx$ as predicate symbols. Without loss of generality we can assume that all P_i have the same number of arguments since we can always add dummy arguments to P_i. Consider predicates $Q_0, \dots, Q_n$ having the same number of arguments as P_i plus enough arguments to represent natural numbers up to n. Our intention is to make $Q_i(j, \bar{x})$ equivalent to $P_j(\bar{x})$ for all $j \leq i$.

Introduce formulas $\Psi_i = \Phi_i \left[Q_j(j, \bar{t}) / P_j(\bar{t}) \right]_{j=1}^{n}$ and define Q_i as follows:

$$
\begin{aligned}
& Q_0(u, \bar{x}) \equiv u \approx 0 \wedge \Psi_0; \\
& Q_{i+1}(u, \bar{x}) \equiv \\
& \quad (u \approx i+1 \wedge \Psi_{i+1}) \vee \\
& \quad (\textstyle\bigvee_{j=1}^{i} u \approx j \wedge Q_i(u, \bar{x})) \quad \text{for all } 0 < i < n; \\
& \texttt{success} \equiv \exists \bar{x} Q_n(n, \bar{x}).
\end{aligned}
$$

The equivalence of $Q_i(j, \bar{x})$ and $P_j(\bar{x})$ is straightforward. Therefore (see the definition of `success` in terms of Q_n), we conclude that the SUCCESS problem for the definitions of predicates P_i is equivalent to that for the definitions of predicates Q_i. Note that the definitions of Q_i's can be constructed from the definitions of P_i's in polynomial time (moreover, the reduction is log-space and is linearly bounded).

Complex values. Here we briefly consider what kind of data are represented by first-order terms. The discussion of multisets and sets is postponed until Section 8. Terms of standard (unsorted) first-order logic represent *trees*. Trees may have an arbitrary depth.

Some formalizations of complex values in databases use *embedded tuples*. Embedded tuples correspond to sorted first-order logic, where every function symbol belongs to some sort. If all atomic domains are finite, every sort contains only a *finite* number of values, while trees allow one to construct an infinite number of values from a finite atomic domain. We will consider both typed and untyped versions.

We pay special attention to signatures consisting of unary function symbols. Terms in such signatures can be understood as representing *lists* of atomic values. For example, the term $f(g(g(h(a))))$ represents the list $[f, g, g, h]$.

3 Nonrecursive logic programs and term algebras

In this section we show that the SUCCESS problem for nonrecursive logic programs with negation is polynomial time equivalent to the underlying theory of term algebra. Thus, results on the complexity of $Th(TA(\Sigma))$ for various Σ are directly applicable to nonrecursive logic programming.

Theorem 3.1 For every signature Σ with at least two symbols, the theory $Th(TA(\Sigma))$ is polynomial-time equivalent to SUCCESS(Σ) (moreover, the reductions in both ways are log-space and linearly bounded).

Proof. A polynomial time reduction of $Th(TA(\Sigma))$ to the problem SUCCESS(Σ) is well-known and described in, e.g., [39]. It can be traced to Clark's completion. Thus, it remains to prove polynomial time reducibility of SUCCESS(Σ) to $Th(TA(\Sigma))$.

As described in Section 2, instead of a nonrecursive logic program we can consider a set of explicit definitions

$$
\begin{aligned}
& P_0(\bar{x}) \equiv \Delta_0, \\
& \dots \\
& P_k(\bar{x}) \equiv \Delta_k,
\end{aligned}
$$

such that each Δ_i contains only the predicate P_{i-1} and equality.

Denote by $\mathfrak{M}$ the perfect model of $\mathcal{P}$. We must demonstrate that one can construct in time polynomial in the size of $\mathcal{P}$ a sentence Φ such that

$$
TA(\Sigma) \vdash \Phi \;\Leftrightarrow\; \mathfrak{M} \models \exists \bar{x} P_k(\bar{x}). \tag{2}
$$

Since $\mathcal{P}$ is nonrecursive, the predicate P_k is *explicitly defined* in terms of $P_{k-1}, \dots, P_1$, and equality; thus, ultimately, can

[1] $LATIME(2^{O(n)})$ is the class of problems solvable by alternating Turing machines in linear exponential time with linear number of alternations.

be explicitly defined in terms of equality only. Therefore, we can write down an explicit definition for P_k and existentially quantify it. The resulting sentence Φ of $TA(\Sigma)$ will satisfy (2). The only drawback of this reduction is that it is *exponential*. This is because an explicit definition of P_i may contain several occurrences of the predicate P_{i-1} with different arguments, which gives an exponential blow-up. In [62] we show how this exponential blow-up can be avoided (recall that we do not allow equivalence $\equiv$ in formulas). $\square$

4 Programs without function symbols

For programs without function symbols it is not hard to establish that the SUCCESS problem for several classes of nonrecursive logic programs is *PSPACE*-complete (for details see [62]).

Theorem 4.1 SUCCESS$(_, 0, 0)$ is *PSPACE*-complete for the following classes of nonrecursive programs: (i) definite programs; (ii) normal programs; (iii) definite range-restricted programs; (iv) normal range-restricted programs. $\square$

5 Programs with binary function symbols

The complexity of the SUCCESS problem for definite programs (range-restricted or not) is characterized by the following theorem proved in [21, 20].

Theorem 5.1 SUCCESS$(_, _, \geq 1)$ for nonrecursive definite programs is *NEXPTIME*-complete. The same result holds for range-restricted nonrecursive definite programs. $\square$

To characterize the complexity of arbitrary nonrecursive logic programs, define functions $e_0(n) = n$, $e_{k+1}(n) = 2^{e_k(n)}$, and $e_\infty(n) = e_n(0)$. Recall that a problem is called *elementary recursive*, if it can be decided in time $\leq e_k(n)$ for some fixed $k \in \omega$. Denote by $NONELEMENTARY(f(n))$ the class of problems with lower and upper time bounds of the form $e_\infty(f(cn))$ and $e_\infty(f(dn))$ for some $c, d > 0$.

The following result is proved in [60]:

Theorem 5.2 The theories $Th(TA(\_, \_, \geq 1))$ are not elementary recursive with the lower bound $e_\infty(cn)$ for some $c > 0$. $\square$

The standard quantifier elimination for $TA(\_, \_, \geq 1)$ [44, 43, 27] gives the upper bound $e_\infty(dn)$ of the same kind. Hence, $Th(TA(\_, \_, \geq 1))$ is in $NONELEMENTARY(n)$.

Theorems 3.1 and 5.2 give

Theorem 5.3 The problem SUCCESS$(_, _, \geq 1)$ for nonrecursive normal programs is in $NONELEMENTARY(n)$. $\square$

6 Programs with unary function symbols

In this section we prove two complexity results for definite and normal programs in monadic signatures.

6.1 Definite programs

Surprisingly, the SUCCESS problem has the same complexity for nonrecursive definite programs also in the absence of binary function symbols.

It is easy to prove that SUCCESS(Σ) for definite programs does not depend on Σ, see, e.g., [23]. Hence, for definite programs we can always assume that Σ consists of the symbols occurring in the program (plus one constant if the program contains no constants).

Theorem 6.1 SUCCESS$(_, \geq 2, 0)$ for nonrecursive definite programs is *NEXPTIME*-complete.

Proof. *NEXPTIME*-hardness is proved by reduction of the TILING problem, see [62]. The technique of the proof essentially uses *difference lists*, known in logic programming. Inclusion in *NEXPTIME* follows from Theorem 5.1. $\square$

6.2 Normal programs

In this section we show that the complexity of the SUCCESS problem for nonrecursive logic programs with negation and function symbols of arity at most one is complete for $LATIME(2^{O(n)})$ — a class of problems solvable by alternating Turing machines in time $2^{O(n)}$ with linear number of alternations[2]. By [15], $LATIME(2^{O(n)}) \subseteq DSPACE(2^{O(n)})$. Also, obviously, $NTIME(2^{O(n)}) \subseteq LATIME(2^{O(n)})$. Both inclusions are *presumably* proper.

The following key theorem is due to [24].

Theorem 6.2 $Th(TA(1, 2, 0))$ is $NTIME(2^{O(n)})$-hard w.r.t. loglin reducibility[3], and can be decided in $DSPACE(2^{O(n)})$.

[59, 58] improved it[4] to

Theorem 6.3 $Th(TA(1, 2, 0))$ is $LATIME(2^{O(n)})$-complete.

Theorem 6.3 and our Theorem 3.1 imply

Corollary 6.4 SUCCESS$(1, 2, 0)$ for nonrecursive normal programs is $LATIME(2^{O(n)})$-complete. $\square$

The lower bound for the nonrecursive SUCCESS$(\geq 1, \geq 2, _)$ with negation also follows immediately:

Corollary 6.5 SUCCESS$(_, \geq 2, _)$ for nonrecursive normal programs is $LATIME(2^{O(n)})$-hard. $\square$

The upper bound appears to be of the same kind:

Theorem 6.6 SUCCESS$(_, \geq 2, 0)$ for nonrecursive logic programs with negation is in the class $LATIME(2^{O(n)})$. $\square$

The proof may be found in [62]. We use the result of [24, 59] that the theory $Th(TA(1, 2, 0))$ of two successors is $LATIME(2^{O(n)})$-complete. In [62] we show how the results of [24, 59] generalize to any number of successors greater than 1.

Summarizing, the complexity of the SUCCESS problem in unary signatures is as follows:

Theorem 6.7 The SUCCESS$(_, \geq 2, _)$ problem for nonrecursive logic programs with negation is $LATIME(2^{O(n)})$-complete. $\square$

[2] [32] calls this class $TA(2^{O(n)}, n)$, which clashes with our usage of $TA()$ for 'term algebras'

[3] Hence, *NEXPTIME*-hard w.r.t. polynomial reducibility, [32].

[4] At the time when [24] was written, the complexity classes defined simultaneously in terms of time, space, and alternations were not yet well known. They first appeared in [11, 14, 12].

7 Range-restricted programs

We start with two general statements about range-restricted programs. The first one asserts that in the range-restricted case only symbols occurring in the program matter.

Lemma 7.1 Let $\mathcal{P}$ be a nonrecursive range-restricted program of signature Σ, P be a predicate different from equality, and $t_1, \ldots, t_n$ be ground terms of Σ such that $P(t_1, \ldots, t_n)$ is true in the perfect model of $\mathcal{P}$ in Σ. Then $t_1, \ldots, t_n$ are built of symbols occurring in $\mathcal{P}$. □

The proof of this theorem is straightforward.

The second statement bounds the depth of terms that may occur in ground clauses true in perfect models. The *depth* of a term t, denoted $depth(t)$, is defined in a standard way, so that the depth of a constant is 0.

Lemma 7.2 Let $\mathcal{P}$ be a nonrecursive range-restricted program in a signature Σ, P be a predicate different from equality, and $t_1, \ldots, t_n$ be ground terms of Σ such $P(t_1, \ldots, t_n)$ is true in the perfect model of $\mathcal{P}$ in Σ. Let K be the maximal number of occurrences of function symbols in clauses of $\mathcal{P}$, and N be the number or predicates defined in $\mathcal{P}$. Then $depth(t_i) \leq NK$ for all i.

Proof. See [62]. □

The case of programs without function symbols was already considered in Section 4. We now proceed to range-restricted programs in the case of unary and then arbitrary function symbols.

7.1 Programs with unary function symbols

The lower bound in Theorem 6.1 essentially depends on non-range-restrictedness. Interestingly, the SUCCESS problem for monadic signatures and range-restricted clauses is simpler, even in presence of negation.

Theorem 7.3 SUCCESS(_, $\geq 2, 0$) for nonrecursive range-restricted normal programs is *PSPACE*-complete.

Proof. *PSPACE*-hardness follows from Theorem 4.1. Inclusion in *PSPACE* is proved by an alternating polynomial time algorithm. The key idea is that we can restrict term depth using Lemma 7.2, and guess terms of the appropriate depth in nondeterministic polynomial time. □

7.2 Arbitrary range-restricted programs

In the rest of this section we consider range-restricted programs with negation. The complexity of this case is characterized by the following theorem.

Theorem 7.4 For nonrecursive range-restricted programs SUCCESS(_, _, ≥ 1) is $LATIME(2^{O(n)})$-complete.

Proof. Inclusion in $LATIME(2^{O(n)})$ is proved similarly to Theorem 7.3, using Lemma 7.2. The only difference is that when we guess ground terms of polynomial depth using nondeterministic OR-branching, we have to make an exponentially deep number of guesses. Thus, the algorithm runs in exponential time, but still with a linear number of alternations.

The proof $LATIME(2^{O(n)})$-hardness uses a reduction of the theory of *bounded concatenation*. We give the definition following [14].

Fix a finite alphabet A with at least two symbols. Let $L(A)$ be the first-order language with equality, with constants a for each $a \in A$, and whose only atomic formulas (other than equalities) are of the form $bcat(x, y, z, \mathbf{n})$, where $\mathbf{n}$ is the unary numeral for the nonnegative integer n. Then for any function $t : N \to N$, we define *t-bounded concatenation theory t-BCT(A)* as the set of true sentences in $L(A)$ under the following interpretation: the underlying domain is the set A^* of words over A, the constant symbols denote the elements $a \in A$, and for all words U, V, W, $bcat(U, V, W, \mathbf{n})$ is true if (i) U is the concatenation of V and W, and (ii) the length of U is at most $t(n)$.

Without loss of generality one may suppose that the alphabet A consists of two symbols, 0 and 1. [59] basing on [24] proved $LATIME(2^{O(n)})$-completeness of 2^n-$BCT(A)$.

Since formulas of t-$BCT(A)$ may contain equalities, one cannot directly claim that a sentence of t-$BCT(A)$ with arbitrarily quantified variables is equivalent to a boundedly quantified sentence. This is because the sizes of values of variables in equalities are unbounded. We need the following lemma allowing us to bound quantified variables in sentences of 2^n-$BCT(A)$. For $Q \in \{\forall, \exists\}$ let $Qx \leq 2^{cn}$ mean that a quantified variable ranges over words of length at most 2^{cn}. The following lemma is proved in [62].

Lemma 7.5 In 2^n-$BCT(A)$, given a quantified prenex sentence

$$F = Q_1 x_1 \ldots Q_k x_k \ \Phi$$

of length n one can construct in polynomial time a boundedly quantified sentence

$$F_b = Q_1 x_1 \leq 2^{cn} \ldots Q_r x_r \leq 2^{cn} \ \Theta$$

true iff F is true. □

To prove $LATIME(2^{O(n)})$-hardness we reduce 2^n-$BCT(A)$ to the SUCCESS problem for nonrecursive range-restricted programs in any signature Σ with a function symbol of arity ≥ 2. For simplicity, we assume that the signature Σ consists of three constants $0, 1, \varepsilon$ and a binary function symbol f.

First, we choose a representation of nonempty words on $\{0, 1\}$ by binary trees (or terms) constructed using $f, 0, 1$. For any such term t we define the corresponding word $w(t)$ as follows:

$$\begin{array}{lcl} w(0) & = & 0 \\ w(1) & = & 1 \\ w(f(s,t)) & = & w(s)w(t) \end{array}$$

Note that words do not have a unique representation.

Let us fix some positive integer N. Consider the following nonrecursive clauses defining predicates $\texttt{nonempty\_word}_n$:

$$\begin{array}{l} \texttt{nonempty\_word}_0(0) \leftarrow \\ \texttt{nonempty\_word}_0(1) \leftarrow \end{array}$$

and the clauses

$$\begin{array}{l} \texttt{nonempty\_word}_{n+1}(f(x,y)) \leftarrow \\ \qquad \texttt{nonempty\_word}_n(x),\ \texttt{nonempty\_word}_n(y) \\ \texttt{nonempty\_word}_{n+1}(x) \leftarrow \texttt{nonempty\_word}_n(x) \end{array}$$

for all $n < N$. Evidently, we have $\texttt{nonempty\_word}_n(t)$ if and only if t is a ground term of depth $\leq n$ built from $f, 0, 1$, encoding a nonempty word.

We recursively define the notion *a tree T' is obtained from a tree T by removing the leftmost leaf.* This holds if

1. $T = f(a,t)$, a is a constant and $T' = t$;

2. $T = f(t_1,t_2)$, t_1 is not a constant, t_1' is obtained from t_1 by removing the leftmost leaf and $T' = f(t_1',t_2)$.

The removal of the leftmost leaf is illustrated by the following example. Removing the leftmost leaf from the term $f(f(f(0,1),f(1,0)),f(0,1))$ yields $f(f(1,f(1,0)),f(0,1))$.

Now we define two other series of predicates:

- $\mathtt{weq}_n(x,y)$ for all $n \le N$ mean that x, y are terms of depth $\le n$ and $w(x) = w(y)$;

- $\mathtt{diff}_n(x,y,z)$ for all $n \le N$ mean that x, y, z are terms of depth $\le n$, $w(y)w(z) = w(x)$ and z is obtained from x by a sequence of removals of leftmost leaves.

The definition of $\mathtt{weq}_0$ is obvious:

$\mathtt{weq}_0(0,0) \leftarrow;$
$\mathtt{weq}_0(1,1) \leftarrow.$

The definition of $\mathtt{diff}_0$ is empty, because $\mathtt{diff}_0(x,y,z)$ is never true.

In order to define $\mathtt{weq}_{n+1}$ for trees $f(x,y)$ and $f(u,v)$ we should consider three possible cases: (i) $w(x) = w(u)$; (ii) $w(x)$ is a prefix of $w(u)$; and (iii) $w(u)$ is a prefix of $w(x)$. This gives rise to the following three clauses:

$\mathtt{weq}_{n+1}(f(x,y),f(u,v)) \leftarrow \mathtt{weq}_n(x,y), \mathtt{weq}_n(u,v);$
$\mathtt{weq}_{n+1}(f(x,y),f(u,v)) \leftarrow$
$\quad \mathtt{diff}_n(u,x,z), \mathtt{diff}_n(y,z,v'), \mathtt{weq}_n(v',v);$
$\mathtt{weq}_{n+1}(f(x,y),f(u,v)) \leftarrow$
$\quad \mathtt{diff}_n(x,u,z), \mathtt{diff}_n(v,z,y'), \mathtt{weq}_n(y',y).$

Finally, we add a self-explaining clause:

$\mathtt{weq}_{n+1}(x,y) \leftarrow \mathtt{weq}_n(x,y).$

In order to define $\mathtt{diff}_{n+1}$ for trees $f(x,y)$ and $f(u,v)$ we should consider five possible cases: (i) $w(x)$ is a prefix of $w(u)$; (ii) $w(x) = w(u)$; (iii) $w(u)$ is a prefix of $w(x)$ and $w(x)$ is a prefix of $w(u)w(v)$; (iv) $w(x) = w(u)w(v)$; and (v) $w(u)w(v)$ is a prefix of $w(x)$. This gives rise to the following five clauses.

$\mathtt{diff}_{n+1}(f(x,y),f(u,v),w) \leftarrow$
$\quad \mathtt{diff}_n(u,x,z), \mathtt{diff}_n(y,z,w)\ ;$
$\mathtt{diff}_{n+1}(f(x,y),f(u,v),w) \leftarrow$
$\quad \mathtt{weq}_n(x,u), \mathtt{diff}_n(y,v,w)\ ;$
$\mathtt{diff}_{n+1}(f(x,y),f(u,v),w) \leftarrow$
$\quad \mathtt{diff}_n(x,u,z_1), \mathtt{diff}_n(v,z_1,z_2), \mathtt{diff}_n(y,z_2,w)\ ;$
$\mathtt{diff}_{n+1}(f(x,y),u,y) \leftarrow$
$\quad \mathtt{weq}_n(x,u), \mathtt{nonempty\_word}_n(y)\ ;$
$\mathtt{diff}_{n+1}(f(x,y),u,f(z,y)) \leftarrow$
$\quad \mathtt{diff}_n(x,u,z), \mathtt{nonempty\_word}_n(y)\ .$

Note that the last two clauses also cover the case when the second argument of $\mathtt{diff}_n$ is a constant. As before, we have to add

$\mathtt{diff}_{n+1}(x,y,z) \leftarrow \mathtt{diff}_n(x,y,z).$

Before, we only encoded nonempty words by terms using $f, 0, 1$. Now we also assume that ε encodes the empty word, i.e. $w(\varepsilon)$ is the empty word, and define the 2^n-bounded concatenation:

- for each $n \le N$, $\mathtt{conc}_n(x,y,z)$ means that x, y, z are terms of depth $\le n$ and $w(x)w(y) = w(z)$.

First, we define predicates $\mathtt{word}_n$ defining terms of depth $\le n$ denoting (maybe empty) words:

$\mathtt{word}_n(\varepsilon) \leftarrow;$
$\mathtt{word}_n(x) \leftarrow \mathtt{nonempty\_word}_n(x);$

Then we use the clauses

$\mathtt{conc}_n(\varepsilon,x,x) \leftarrow \mathtt{word}_n(x);$
$\mathtt{conc}_n(x,\varepsilon,x) \leftarrow \mathtt{word}_n(x);$
$\mathtt{conc}_n(x,y,z) \leftarrow \mathtt{diff}_n(z,x,y'), \mathtt{weq}_n(y',y).$

Note that all these clauses are nonrecursive, range-restricted, and can be constructed in time polynomial in N.

Now we show how to reduce the theory of bounded concatenation to the SUCCESS problem for nonrecursive range-restricted clauses.

By Lemma 7.5 we can assume that all quantifiers in formulas of 2^n-$BCT(A)$ are bounded by 2^{cn}. Let φ be any formula of size $\le n$ whose free variables are $\bar{x}$. Define N as cn and build the definition of $\mathtt{conc}_m$ for all $m \le N$ as above. By induction on φ we construct a nonrecursive logic program $\mathcal{P}$ defining a predicate $P_\varphi(\bar{x})$ with the following property.

Lemma 7.6 For all ground terms $t_1, \ldots, t_n$ of depth $\le n$, the formula $P(t_1, \ldots, t_n)$ is true in the perfect model of $\mathcal{P}$ if and only if $\varphi(w(t_1), \ldots, w(t_n))$ is true in 2^n-$BCT(A)$.

We can assume that φ is constructed using only $\neg, \wedge$, and $\exists$. If φ is an atomic formula $bcat(x,y,z,\mathbf{n})$, we define P_φ as follows:

$P_\varphi(x,y,z) \leftarrow \mathtt{conc}_n(y,z,x)$

The cases of conjunction and existential quantifier are obvious:

$P_{\varphi_1 \wedge \varphi_2}(\bar{x}) \leftarrow$
$\quad P_{\varphi_1}(\bar{x}_1),$
$\quad P_{\varphi_2}(\bar{x}_2).$

$P_{\exists y \le 2^{cn} \varphi}(\bar{x}) \leftarrow$
$\quad P_\varphi(\bar{x},y),$
$\quad \mathtt{word}_N(y)$

Finally, the negation is handled in the following way:

$P_{\neg\varphi}(x_1, \ldots, x_m) \leftarrow$
$\quad \neg P_\varphi(x_1, \ldots, x_m),$
$\quad \mathtt{word}_N(x_1), \ldots, \mathtt{word}_N(x_m).$

The proof of Lemma 7.6 is straightforward. Note that the program defining P_φ is range-restricted, nonrecursive and can be constructed in time polynomial in the size of φ. Thus, 2^n-$BCT(A)$ is polynomial time reducible to the SUCCESS problem for nonrecursive range-restricted normal programs which completes the proof of Theorem 7.4. $\square$

8 Other kinds of complex values

In this section we briefly consider various formalizations of finite sets and multisets. The complexity of the corresponding nonrecursive fragment of logic programming with complex values depends on this formalization and the signature, i.e., the set of operations available on the values. We consider

1. *colored finite sets* [22];

2. *untyped finite sets* [19, 21];

3. *typed finite sets* [1].

The definitions and subsequent results can be straightforwardly generalized to finite multisets.

8.1 Formalizations of finite sets

Universe with colored sets. In order to construct the *domain of colored sets* [22], we need to add to a signature Σ a binary function symbol $\{\ldots \mid \ldots\}$, called the *set constructor*. A *color* is any term whose top function symbol is not $\{\mid\}$. A term $\{s_1 \mid \ldots \{s_n \mid t\}\}$, where t is a color, represents the *colored set* with elements $s_1, \ldots, s_n$ and the color t. Two colored sets are equal when they have the same elements and the same color. To represent the empty set, any color can be used. Note that a color may be a complex term whose subterms can be colored sets.

Universe with untyped sets. Untyped sets [19, 21] are defined similarly to colored sets, but the only color is the constant $\emptyset$, representing the empty set. In order to formalize untyped sets, a two-sorted signature must be used. (In fact, colored sets were introduced in order to have completely unsorted language.)

Universe with typed sets. We present a typed universe for complex values containing finite sets following [1].

We assume a set of *domain names* $\widehat{D_1}, \widehat{D_2}, \ldots$ and an infinite set of *attributes* $A_1, A_2, \ldots$ The domain names are associated with *domains* $D_1, D_2, \ldots$, i.e., non-empty sets whose elements are called *atomic values.*

Complex values are constructed from atomic ones using constructors. A type is associated with each value.

Definition 8.1 *Types* and *values* for this domain are defined as follows:

1. If $\widehat{D}$ is a domain name, then $\widehat{D}$ is an *atomic* type. For each $a \in D$, the element a is a value of this type.
2. If $T_1, \ldots, T_n$, where $n \geq 0$, are types and $A_1, \ldots, A_n$ are distinct attributes, then $[A_1 : T_1, \ldots, A_n : T_n]$ is a *tuple* type. If $v_1, \ldots, v_n$ are values of types $T_1, \ldots, T_n$, respectively, then $[A_1 : v_1, \ldots, A_n : v_n]$ is a value of the type.
3. If T is a type, then $\{T\}$ is a set type. Any finite set of values of type T is a value of type $\{T\}$.

Syntactic representation of values from the typed domain is similar to that for the untyped domain (in particular, we have the set constructor), but with the following changes.

1. Since the atomic domains are typed, the corresponding constants of the language are also typed.
2. We have *tuple terms* $[A_1 : t_1, \ldots, A_n : t_n]$ made of terms $t_1, \ldots, t_n$ with the natural interpretation.
3. The constants for the empty set and the set constructor satisfy the natural restrictions on their types.

In order to define semantics of nonrecursive logic programs over these universes, we need only one change. Instead of considering nonrecursive logic programs as sets of explicit definitions over the term algebra, we consider them over the corresponding universe.

The following theorem is proved in [19, 21].

Theorem 8.2 The SUCCESS problem for definite programs is *NEXPTIME*-complete for the following domains:

1. universe with colored sets and/or bags;
2. universe with untyped sets and/or bags;
3. universe with typed sets and/or bags.

The same holds for range-restricted definite programs. □

In the remaining part of the paper we address the complexity of nonrecursive logic programs over the three domains defined in this section.

8.2 Typed universe

Evidently, universes with typed and untyped sets correspond to two versions, typed and untyped, of the domain of hereditarily finite sets, see e.g., [8]. In this section we settle complexity bounds for the typed universe. We need some background material on the theory of typed hereditarily finite sets.

Type theory Ω is a rudimentary fragment of L. Henkin's theory of propositional types [26]. The language of Ω is a language of set theory, where every variable has a natural number type (written as a binary superscript) and there are two constants **0**, **1** of type 0. The theory has a countable number of predicate symbols $\in^n$, for every natural number n. The interpretation of Ω is as follows: **0** denotes 0, **1** denotes 1, elements of type 0 are 0 and 1 and elements of types $n+1$ are sets of elements of type n.

For complexity considerations let us fix any reasonable encoding of formulas of Ω as binary strings and agree that a variable of Ω be represented by its type and its identification number within a type, both written in binary.

The validity problem for Ω is trivially decidable, because every quantifier runs over a *finite* domain, but *is not elementary recursive.* Even stronger, [61]:

Theorem 8.3 Every Turing machine deciding Ω makes a number of steps exceeding

$$\exp_\infty(2^{c|\Phi|}) = 2^{2^{\cdot^{\cdot^{2}}}} \Big\} \text{ height } 2^{c|\Phi|}$$

for some constant $c > 0$ and infinitely many sentences Φ of Ω. □

This improves the previously known lower bound [45, pp. 478–479], which was a logarithmically growing tower of twos.

Reduction of Ω to nonrecursive logic programs over the typed universe of sets. Here we apply the strong lower bound from Theorem 8.3 by polynomially reducing Ω to the SUCCESS problem for the typed universe. It follows that the latter problem has the same lower complexity bound.

We consider the following typed universe $\mathcal{S}$ of finite sets. There is only one atomic type with constants 0 and 1. We construct all set types as in Section 8.1, i.e., using constants for empty sets and set constructors for all set types.

For a sentence Φ of the theory Ω its polynomial-time translation to the SUCCESS problem for nonrecursive logic programs over $\mathcal{S}$ is defined exactly as in the proof of Theorem 3.1, except for the case of atomic formulas. The membership predicates $\in^n$ can be defined by the following clause:

$\mathtt{member}_n(x, \{x \mid y\}) \leftarrow .$

Henceforth, Theorem 8.3 yields

function symbols	no	unary	any
	not range-restricted		
no negation	*PSPACE*	*NEXPTIME*	*NEXPTIME*
with negation	*PSPACE*	$LATIME(2^{O(n)})$	*NONELEMENTARY*(n)
	range-restricted		
no negation	*PSPACE*	*PSPACE*	*NEXPTIME*
with negation	*PSPACE*	*PSPACE*	$LATIME(2^{O(n)})$

Table 1: Summary of results

Theorem 8.4 The SUCCESS problem for nonrecursive logic programs over $\mathcal{S}$ is not elementary recursive. Even stronger: any algorithm solving the problem should necessarily spend time exceeding

$$2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\Big\}\text{ height } 2^{c|\mathcal{P}|}$$

for some constant $c > 0$ and infinitely many instances $\mathcal{P}$ of the problem. □

It is not difficult to show that the SUCCESS problem is decidable, and the upper bound is of the same kind. Summarizing, we get

Theorem 8.5 The SUCCESS problem for nonrecursive logic programs over $\mathcal{S}$ is in $NONELEMENTARY(2^n)$. □

To our knowledge, this is the most complicated (in complexity-theoretic sense) of all decidable problems for which any upper bound is currently known, cf., [61].

It is easy to see that the complexity remains the same for any typed universe, where atomic domains contain a finite number of elements, since in this case all types will still have a finite number of elements.

Consider now the typed universe $\mathcal{S}'$ defined as $\mathcal{S}$, but with an infinite atomic domain. In this case we have the following fact.

Theorem 8.6 The SUCCESS problem for nonrecursive logic programs over $\mathcal{S}'$ is undecidable. □

We leave a detailed proof to the reader. The idea is that it is possible to interpret the theory of a finite binary relation (Kalmar-Traktenbrot-Vaught-Rabin), cf., [57, 48]. Indeed, using the infinite atomic domain D we can express a finite set of any size (as any finite set S of elements of D) and interpret an arbitrary binary relation as a set of pairs $\{a, b\}$ with elements in D.

Analogously, if the atomic domain is finite, but variables and the membership predicate are untyped (polymorphic or interpreted over arbitrary strata) the theory and the problem are also undecidable.

[29, 36] investigate the expressive power of database query languages allowing for the *powerset* constructor, along with *explicit* boolean connectives and quantifiers[5]. The results of [29, 36] show that the expressive power of such query languages *strictly increases* as one allows for a deeper nesting of the powerset operator. In the context of typed set universes [29, 36] demonstrate that the towers of the hyperexponential functions in the lower and upper complexity bounds of a query language with k nestings grow *linearly* with k, and thus cover the class of all *elementary* time computable queries. Note that our set-based logic programming language is exponentially more succinct than the languages considered in [29, 36]. In fact, in writing formulas we use the *binary* notation for types and iterative applications of the powerset operator. E.g., $\texttt{member}_{11111111}(x, y)$ means that x is a member of the set obtained as a result of 255 applications of the powerset constructor. Expressing the latter by using the unary notation for *powerset*, as done in [29, 36], requires an exponentially longer expression. This partially explains why our lower and upper complexity bounds are *exponentially* growing stacks of twos (see Theorems 8.4 and 8.5), as compared to *linearly* growing stacks of twos in [29, 36].

[5] Recall that in LP we do not have, nor need boolean connectives and quantifiers.

8.3 Untyped universe

Similarly, we obtain

Theorem 8.7 The SUCCESS problems for nonrecursive logic programs over the untyped universe and the colored universe are undecidable. □

This follows from the undecidability of the theory of a binary relation (Kalmar, Trakhtenbrot, Vaught, Rabin), cf. [57, 48], or from Gandy's theorem [8].

Our Theorems 8.6 and 8.7 on undecidability of the SUCCESS problem for nonrecursive logic programs over untyped sets should be contrasted to the result of [28], who show that their set-based query language over untyped sets *without* `while` is expressively equivalent to typed language, and is thus within the class of *elementary time* computable queries.

9 Summary of results

The complexity of the success problem for nonrecursive logic programs over trees are summarized in Table 1. In all cases we have completeness in the corresponding complexity class, except for *NONELEMENTARY* (in this case both lower and upper bounds are linearly growing towers of twos).

We briefly discuss the obtained complexity results, by comparing results in all signatures with the simplest class: definite range-restricted programs. In the case without function symbols (corresponding to nonrecursive datalog) the complexity does not change when we add negation and remove range-restriction. For signatures with unary function symbols, the complexity grows when range restrictedness is dropped, and grows even further when negation is additionally allowed. Without negation, the absence of range restriction allows one to use difference lists and the complexity grows from *PSPACE* to *NEXPTIME*. For arbitrary signatures, we have a "small" increase in complexity when we either remove range restriction or add negation. However if we do both, the complexity becomes nonelementary. In all cases except datalog, it is the combination of negation and non-range restrictedness that gives a big jump in complexity.

We also proved several complexity results about nonrecursive logic programs over sets. However, the complexity of such programs over sets should be further investigated. For example, it is not known what is the complexity of the range-restricted fragment. Another interesting question is the complexity for programs with a fixed finite number of negations, both with and without range-restriction.

Acknowledgements. We thank Evgeny Dantsin, Thomas Eiter, Georg Gottlob, Gabriel Kuper, Moshe Vardi, and PODS'98 referees.

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, R. Hull, and V. Vianu. *Foundation of Databases.* Addison-Wesley Publishing Co., 1995.

[3] Andréka and Németi. A generalized completeness of Horn clause logic seen as a programming language. *Acta Cybernetica*, 4:3–10, 1978.

[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] K.R. Apt and H.A. Blair. Arithmetic classification of perfect models of stratified programs. In R. Kowalski and K.A. Bouwen, editors, *Proceedings of the Fifth Joint International Conference and Symposium on Logic Programming (JICSLP-88)*, pages 766–779. The MIT Press, 1988.

[6] K.R. Apt and H.A. Blair. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, 1988.

[7] K.R. Apt and R.N. Bol. Logic programming and negation: a survey. *Journal of Logic Programming*, 19,20:9–71, 1994.

[8] J. Barwise. *Admissible Sets and Structures.* Springer Verlag, 1975.

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

[10] M. Benedikt and L. Libkin. Languages for relational databases over interpreted structures. In *PODS 1997. Proceedings of the Sixteenth ACM-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 87–98, Tucson, Arizona, 1997.

[11] L. Berman. Precise bounds for Presburger arithmetic and the reals with addition: preliminary report. In *International Conference of Foundations of Computer Science*, pages 95–99, 1977.

[12] L. Berman. The complexity of logical theories. *Theoretical Computer Science*, 11:71–77, 1980.

[13] H. Blair, V.W. Marek, and J.S. Schlipf. The expressiveness of locally stratified programs. *Annals of Mathematics and Artificial Intelligence*, 15(3/4):209–229, 1995.

[14] A. R. Bruss and A. R. Meyer. On time-space classes and their relation to the theory of real addition. *Theoretical Computer Science*, 11:59–69, 1980.

[15] A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *Journal of the Association for Computing Machinery*, 28:114–133, 1981.

[16] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Base*, pages 293–322. Plenum Press, New York, 1978.

[17] E. Dantsin. The complexity of Prolog without loops (in Russian). In *Proceedings of the 4th Soviet Conference "Application of the Mathematical Logic Methods"*, volume 2, pages 112–113, Tallinn, 1986.

[18] 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.

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

[20] E. Dantsin and A. Voronkov. Complexity of query answering in logic databases with complex data. UPMAIL Technical Report 149, Uppsala University, Computing Science Department, September 1997.

[21] 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.

[22] 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.

[23] M. Falashi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of the operational behavior of logic languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

[24] J. Ferrante and C. W. Rackoff. *The computational complexity of logical theories*, volume 718 of *Lecture Notes in Mathematics.* Springer-Verlag, 1979.

[25] 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.

[26] L. Henkin. A theory of propositional types. *Fundamenta Mathematicae*, 52:323–344, 1963.

[27] W. Hodges. *Model theory.* Cambridge University Press, 1993.

[28] R. Hull and J. Su. Untyped sets, invention, and computable queries. In *PODS'89*, pages 347–359. ACM, 1989.

[29] R. Hull and J. Su. On the expressive power of database queries with intermediate types. *Journal of Computer and System Sciences*, 43:219–267, 1991.

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

[31] Neil Immerman. Languages that capture complexity classes. *SIAM Journal of Computing*, 16:760–778, 1987.

[32] D. S. Johnson. A catalog of complexity classes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 8, pages 67–161. Elsevier Science, 1990.

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

[34] K. Kunen. Answer sets and negation as failure. In *4th International Conference on Logic Programming*, volume 1, pages 219–228. The MIT Press, 1987.

[35] K. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4:289–308, 1987.

[36] G. Kuper and M.Y. Vardi. On the complexity of queries in the logical data model. *Theoretical Computer Science*, 116:33–57, 1993.

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

[38] V. Lifschitz. On the declarative semantics of logic programs with negation. In J. Minker, editor, *Deductive Databases and Logic Programming*. Morgan Kaufmann, 1988.

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

[40] M. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proc. IEEE Conference on Logic in Computer Science (LICS)*, pages 348–357, 1988.

[41] 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.

[42] M.J. Maher. A logic programming view of CLP. In *International Conference on Logic Programming*, pages 737–753, 1993.

[43] A. Maĺcev. Axiomatizable classes of locally free algebras of various types. In B.F. Wells III, editor, *The Metamathematics of Algebraic Systems. Anatoliĭ Ivanovič Maĺcev. Collected Papers: 1936–1967*, pages 262–281. North Holland, 1961.

[44] A. Maĺcev. On the elementary theories of locally free universal algebras. *Soviet Mathematical Doklady*, 2(3):768–771, 1961.

[45] A.R. Meyer. The inherent computational complexity of theories of ordered sets. In *International Congress of Mathematicians*, pages 477–482, 1974.

[46] C.H. Papadimitriou and M. Yannakakis. On the complexity of database queries. In *PODS'97*, pages 12–19, 1997.

[47] Przymusinski. On the declarative semantics of deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, 1988.

[48] M. O. Rabin. A simple method for undecidability proofs and some applications. In Y. Bar-Hillel, editor, *Proc. Intern. Congress on Logic, Methodology and Philosophy of Science*, Studies in Logic and the Foundations of Mathematics, pages 58–68, 1964.

[49] J.S. Schlipf. Complexity and undecidability results in logic programming. *Annals of Mathematics and Artificial Intelligence*, 15(3/4):257–288, 1995.

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

[51] L. J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3:1–22, 1977.

[52] L.J. Stockmeyer and A.R. Meyer. Word problems requiring exponential time: Preliminary report. In *Proc. ACM STOC*, pages 1–9, 1973.

[53] S.-Å. Tärnlund. Horn clause computability. *BIT*, 17:215–216, 1977.

[54] A. Van Gelder. Negation as failure using tight derivations for general logic programs. In J. Minker, editor, *Deductive Databases and Logic Programming*, pages 149–177. Morgan Kaufmann, 1988.

[55] M. Vardi. The complexity of relational query languages. In *Proc. 14th ACM STOC*, pages 137–146, San Francisco, 1982.

[56] M.Y. Vardi. On the complexity of bounded-variable queries. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 266–276, 1995.

[57] R. L. Vaught. Sentences true in all constructive models. *Journal of Symbolic Logic*, 25(1):39–53, 1960.

[58] H. Volger. A new hierarchy of elementary recursive decision problems. *Methods of Operations Research*, 45:509–519, 1983.

[59] H. Volger. Turing machines with linear alternation, theories of bounded concatenation and the decision problem of first order theories (Note). *Theoretical Computer Science*, 23:333–337, 1983.

[60] S. Vorobyov. An improved lower bound for the elementary theories of trees. In M.A. McRobbie and J.K. Slaney, editors, *Automated Deduction — CADE-13*, volume 1104 of *Lecture Notes in Computer Science*, pages 275–287, New Brunswick, NJ, USA, 1996.

[61] S. Vorobyov. The "hardest" natural decidable theory. In *Proc. IEEE Conference on Logic in Computer Science (LICS)*, pages 294–305, 1997.

[62] S. Vorobyov and A. Voronkov. Complexity of nonrecursive logic programs with complex values. Technical Report MPI-I-97-2-010, Max-Planck Institut für Informatik, Saarbrücken, November 1997.