Expressive power and data complexity of nonrecursive query languages for lists and trees

(Extended Abstract)

Evgeny Dantsin[*]

St.Petersburg Institute of Mathematics and University of Manchester

dantsin@cs.man.ac.uk

Andrei Voronkov[†]

University of Manchester

voronkov@cs.man.ac.uk

ABSTRACT

We extend the traditional query languages by primitives for handling lists and trees. Our main results characterize the expressive power and data complexity of the following extended languages: (1) relational algebra with lists and trees, (2) nonrecursive Datalog$^\neg$ with lists and trees, (3) nonrecursive Prolog with lists and trees, (4) first-order logic over lists and trees.

Languages (2)–(4) turn out to have the same expressive power; their range-restricted fragments have the same expressive power as (1). Every query in these languages is a boolean combination of range-restricted queries.

We also prove that these query languages have polynomial data complexity under any "reasonable" encoding of inputs. Furthermore, under a natural encoding of inputs, languages (2)–(4) have the same expressive power as first-order logic over finite structures, therefore their data complexity is in AC_0. Thus, the use of lists and trees in nonrecursive query languages gives no gain in the expressiveness. This contrasts with a huge difference between the nonelementary program complexity of extended languages (2)–(4) and the $PSPACE$ program complexity of their relational counterparts.

Our results partly explain why lists and trees are not so widely used in nonrecursive query languages as other collection types.

[*]Partially supported by grants from EPSRC, TFR and INTAS-RFBR.

[†]Partially supported by grants from EPSRC and the Faculty of Science and Technology, University of Manchester

Keywords

Data complexity, expressive power, lists, trees, term algebras

1. INTRODUCTION

It is well known that first-order logic and other nonrecursive query languages over finite structures cannot express many important relations, for example transitive closure, graph connectivity, acyclicity [3, 13, 15]. The standard way to increase their expressive power is to extend first-order logic by various recursion-like operators, such as fixpoint operators [3, 9, 2, 19, 30]. Is it possible to increase the expressive power by using recursive datastructures instead of recursion-like operators? For example, one could use lists or trees, probably the most natural recursive datastructures. In this paper we show that the use of lists and trees does not increase the expressiveness.

Namely, we extend the traditional query languages by primitives for handling lists and trees. Our main results characterize the expressive power and data complexity of the following extended languages: (1) relational algebra with lists and trees, (2) nonrecursive Datalog$^\neg$ with lists and trees, (3) nonrecursive Prolog with lists and trees, (4) first-order logic over lists and trees.

Languages (2)–(4) turn out to have the same expressive power; their range-restricted fragments have the same expressive power as (1). Every query in these languages is a boolean combination of range-restricted queries.

We also prove that these query languages have polynomial data complexity under any "reasonable" encoding of inputs. Furthermore, under a natural encoding of inputs, languages (2)–(4) have the same expressive power as first-order logic over finite structures, therefore their data complexity is in AC_0. Thus, the use of lists and trees in nonrecursive query languages gives no gain in the expressiveness. This contrasts with a huge difference between the nonelementary program complexity of extended languages (2)–(4) and the $PSPACE$ program complexity of their relational counterparts.

Our results are heavily based on the Safety Theorem (proved in Section 5.2) which plays the same role as quantifier elimination in term algebras [25, 24, 16]. This theorem reduces the evaluation of a formula on an infinite domain to evaluation of another formula on a finite domain. Similar techniques were used in [18] for relational data model and in [26, 5] for constraint databases.

A relevant subject is extensions of the relational model by complex values. Papers on this subject are too numerous to be mentioned here. Some approaches and results can be found in the recent survey [12]. The specifics of lists and trees as compared to other complex values is that elements in the domain of a given type may have "arbitrarily deep nesting depth of the constructs" [17]. Query languages for such values were considered in [17, 8]. The most known query language of this type is, evidently, Prolog.

We introduce lists and trees in the relational data model by using interpreted structures that extend structures of the relational model. Such approach is common in the area of constraint databases [21, 26, 5, 6, 29].

This paper is organized as follows. Section 2 introduces lists and trees. Section 3 defines query languages for lists and trees. Main results are formulated in Section 4 without proofs. The proofs are given in Section 5.

2. LISTS AND TREES

In this section we modify the standard definitions from database theory in order to include lists and trees. Let $\mathcal{U}$ be a finite set whose elements are called *urelements*. We call $\mathcal{U}$ the *domain of urelements*. The universe $\mathcal{U}$ plays the role of the (active) domain in the relational database model.

We will extend the domain $\mathcal{U}$ by lists and trees, obtaining an infinite domain. In defining this extended domain and functions on the domain, we will often blur the distinction between syntax and semantics, using the same names for the data constructors and for the function symbols denoting these constructors.

A *list* is any sequence $u_1, \ldots, u_n$ of urelements. We denote by `[]` the empty list, and by $[a\,|\,l]$ the list obtained from a list l by adding the element a as the first element. The function $[\_\,|\,\_]$ is called the *list constructor*. The set of all lists is denoted by $\mathcal{L}$. *Binary trees* (for brevity we will simply say *trees*) are defined inductively as follows:

1. The *empty tree*, denoted ε, is a tree.
2. If t_1, t_2 are trees and u is an urelement, then $\top(t_1, u, t_2)$ is a tree.

The set of all trees is denoted by $\mathcal{T}$.

We consider queries defined on the domain $\mathcal{U} \cup \mathcal{L} \cup \mathcal{T}$ regarded as a three-sorted structure. For technical convenience, we will sometimes refer to the domains $\mathcal{U}$, $\mathcal{L}$, $\mathcal{T}$ as *sorts*. A *relation schema* R is an n-tuple $(s_1, \ldots, s_n)$, where $n \geq 1$ and s_i's are sorts. A *relation instance* of R is a finite subset of $s_1 \times \ldots \times s_n$. A *database schema* D is a sequence $(P_1 : R_1, \ldots, P_m : R_m)$, where $m \geq 1$, P_i's are distinct predicate names and R_i's are relation schemas. A *database instance* d of D is a sequence $(P_1 : r_1, \ldots, P_m : r_m)$ where r_i is an instance of R_i, for all $i = 1, \ldots, m$.

A *query* with an *input database schema* D and an *output relation schema* $(s_1, \ldots, s_n)$ is any mapping from the set of all instances of D into the set of all subsets of $s_1 \times \ldots \times s_n$ (such subsets can be infinite). When we speak about queries, we will informally call relation instances of the input database schema *input relations*, and subsets of $s_1 \times \ldots \times s_n$ *output relations*. We do not require finiteness of the output for a technical reason: some introduced query languages may define infinite relations. A *boolean query* with an input database schema D is any mapping from the set of all instances of D into the set $\{\mathit{true}, \mathit{false}\}$.

2.1 Other models of trees

The results of this paper hold for various modifications of the notion of tree. Here we will briefly mention some of them.

1. We can consider trees generated by several tree constructors instead of $\top$.
2. We can consider nonsorted models, where elements of a list should not be of the same sort, and similar for trees.
3. We can consider embedded values, like trees of trees or lists of trees etc.
4. We can consider term algebras (see [16]) or term algebras with sorts [27]. In fact, the model for tree of this paper, and each models above can be formalized in a suitable term algebra with or without sorts.

All definitions given below and all results below can be generalized for these modifications.

3. QUERY LANGUAGES FOR LISTS AND TREES

In this section we introduce several query languages for lists and trees: relational algebra, Datalog$^{\neg}$, Prolog, and first-order logic.

3.1 Relational algebra

Relational algebra with lists and trees is obtained from standard relational algebra (see e.g. [28]) by using sorts and adding the following operations:

- The nullary operation $\{\texttt{[]}\} : 2^{\mathcal{L}}$ denoting the set consisting of the empty list.
- The operation $\mathit{head\_tail} : 2^{\mathcal{L}} \to 2^{\mathcal{U} \times \mathcal{L}}$ defined as follows:

$$\mathit{head\_tail}(S) = \{(h, t) \mid [h\,|\,t] \in S\}.$$

- The operation $\mathit{cons} : 2^{\mathcal{U} \times \mathcal{L}} \to 2^{\mathcal{L}}$ defined as follows:

$$\mathit{cons}(S) = \{[h\,|\,t] \mid (h, t) \in S\}.$$

- The nullary operation $\{\varepsilon\} : 2^{\mathcal{T}}$ denoting the set consisting of the empty tree.
- The operation *split* $: 2^{\mathcal{T}} \to 2^{\mathcal{T}\times\mathcal{U}\times\mathcal{T}}$ defined as follows:

$$split(S) = \{(t_1, n, t_2) \mid \top(t_1, n, t_2) \in S\}.$$

- The operation *tree* $: 2^{\mathcal{T}\times\mathcal{U}\times\mathcal{T}} \to 2^{\mathcal{T}}$ defined as follows:

$$tree(S) = \{\top(t_1, s, t_2) \mid (t_1, s, t_2) \in S\}.$$

Example 1 The query finding all lists whose first element is either a or b in a set S can be defined in relational algebra with lists and trees as

$$cons(\sigma_{\$1=a \vee \$1=b}(head\_tail(S))).$$

Example 2 The query finding all nonempty trees whose left subtree coincides with its right subtree in a set S can be defined in relational algebra with lists and trees as

$$tree(\sigma_{\$1=\$3}(split(S))).$$

3.2 Datalog$^\neg$

Datalog$^\neg$ with trees and lists is obtained from the standard Datalog$^\neg$ as defined in e.g. [1, 12] by adding sorts and adding the following interpreted relations:

$$\begin{array}{rcl} empty\_list(l) & \equiv & l = \texttt{[]}. \\ rel\_cons(l, h, t) & \equiv & l = \texttt{[}h\,|\,t\texttt{]}. \\ empty\_tree(l) & \equiv & l = \varepsilon. \\ rel\_tree(t, t_1, n, t_2) & \equiv & t = \top(t_1, n, t_2). \end{array}$$

Semantics of Datalog$^\neg$ with lists and trees can be easily defined as an adaptation of semantics of the standard Datalog$^\neg$ for the case of interpreted relations, e.g. in the style of [17, 26, 5]. In this paper we only deal with nonrecursive Datalog$^\neg$ whose semantics is unambiguous.

Example 3 The queries of Examples 1 and 2 can be expressed in nonrecursive Datalog$^\neg$ with lists and trees respectively as

$$\begin{array}{l} first\_a\_or\_b(x) \leftarrow S(x),\ rel\_cons(x, a, z). \\ first\_a\_or\_b(x) \leftarrow S(x),\ rel\_cons(x, b, z). \\ \\ same\_subtrees(x) \leftarrow S(x), rel\_tree(x, y, z, y). \end{array}$$

Note that in general we do not impose the safety or range restrictions on Datalog$^\neg$ clauses. To define the range-restricted fragment of Datalog$^\neg$ with lists and trees, we need to define range-restricted Datalog$^\neg$ clauses. The notion of a *safe symbol* (i.e. constant or variable) in a clause C is defined as follows.

1. Every constant is safe.
2. Every variable occurring in a positive literal L with the predicate symbol different from *rel_cons*, *rel_tree* in the body of C is safe.
3. If $rel\_cons(l, h, t)$ is a positive literal in C, then
   (a) If l is safe, then so are h, t.
   (b) If both h, t are safe, then so is l.
4. If $rel\_tree(t, t_1, n, t_2)$ is a positive literal in C, then
   (a) If t is safe, then so are t_1, n, t_2.
   (b) If all t_1, n, t_2 are safe, then so is t.

A clause is called *range-restricted* if every variable in it is safe. It is easy to see that both clauses of Example 3 are range-restricted.

By using the standard arguments (see e.g. [28]) one can prove the following statement.

Proposition 1 *A query q is expressible in relational algebra with lists and trees if and only if it is expressible in nonrecursive range-restricted Datalog$^\neg$ with lists and trees.*

3.3 Prolog

Prolog with lists and trees is obtained from the standard Datalog$^\neg$ by adding sorts, constants `[]` and ε, and the functions `[_|_]` (the list constructor) and $\top$ (the tree constructor). For the standard semantics of Prolog see e.g. [4]. Though there are several approaches to semantics of recursive Prolog, in the case of nonrecursive definitions all of them collapse to the so-called stratified semantics. The semantics of nonrecursive Prolog can also be found in [34].

Example 4 The queries of Examples 1 and 2 can be expressed in nonrecursive Prolog with lists and trees respectively as

$$\begin{array}{l} first\_a\_or\_b(\texttt{[}a\,|\,x\texttt{]}) \leftarrow S(\texttt{[}a\,|\,x\texttt{]}). \\ first\_a\_or\_b(\texttt{[}b\,|\,x\texttt{]}) \leftarrow S(\texttt{[}b\,|\,x\texttt{]}). \\ \\ same\_subtrees(\top(x, y, x)) \leftarrow S(\top(x, y, x)). \end{array}$$

The notion of range-restricted Prolog clauses is simpler than that of Datalog$^\neg$: a Prolog clause is called *range-restricted* if every variable in it occurs in a positive literal in the body. Evidently, both clauses of Example 4 are range-restricted.

It is not hard to give semantics-preserving translations from (range-restricted) Datalog$^\neg$ with lists and trees to (range-restricted) Prolog with lists and trees, and vice versa. As a consequence, we obtain

Proposition 2
A query q is expressible in (range-restricted) nonrecursive Datalog$^\neg$ with lists and trees if and only if it is expressible in (range-restricted) nonrecursive Prolog with lists and trees.

3.4 First-order logic

First-order logic with lists and trees, is obtained from first-order logic over finite structures by adding sorts and the same symbols as in Datalog$^\neg$, i.e. the predicate symbols *empty_list*, *rel_cons*, *empty_tree*, and *rel_tree* with their natural interpretation.

Example 5 The queries of Examples 1 and 2 can be expressed in first-order logic with lists and trees respectively as

$$\{x \mid S(x) \wedge (\exists z)(\exists u)(rel\_cons(x,u,z) \wedge (u = a \vee u = b))\}.$$
$$\{x \mid S(x) \wedge (\exists y)(\exists z) rel\_tree(x,y,z,y)\}.$$

Similarly to the definitions of range-restricted Datalog$^\neg$ and Prolog, we could define range-restricted formulas of first-order logic over lists and trees. However, such a definition would be too technical, and we do not present it in this paper.

By using the standard technique (see e.g. [34]) one can prove

Proposition 3
A query q is expressible in nonrecursive Prolog with lists and trees if and only if it is expressible in first-order logic with lists and trees.

4. MAIN THEOREMS

By summarizing Propositions 1–3 we obtain Theorems 1 and 2 below characterizing the relative expressive power of the query languages introduced in Section 3.

Theorem 1 *The following query languages have the same expressive power:*

1. *Nonrecursive Datalog$^\neg$ with lists and trees;*
2. *Nonrecursive Prolog with lists and trees;*
3. *First-order logic over lists and trees.*

Theorem 2 *The following query languages have the same expressive power:*

1. *Relational algebra with lists and trees;*
2. *Nonrecursive range-restricted Datalog$^\neg$ with lists and trees;*
3. *Nonrecursive range-restricted Prolog with lists and trees.*

Using the standard technique [1]see e.g., it is not hard to argue that the range-restricted fragments of the languages considered in this paper have the following standard properties:

Theorem 3 *Relational algebra with lists and trees is domain-independent and defines queries with finite output.*

The next theorems are much more difficult to prove. Theorem 4 compares the expressive power of two kinds of query languages: arbitrary and range-restricted ones.

Theorem 4 *A boolean query q is expressible in first-order logic over lists and trees if and only if q is a boolean combination of boolean queries expressible in relational algebra with lists and trees.*

The proof of this and the next theorem will be presented in Section 5.

Note that the complexity of first-order logic over trees is not elementary recursive[1] , while range-restricted nonrecursive Prolog can be decided in $DSPACE(2^{O(n)})$ [34]. Thus, Theorem 4 shows that the huge gap in the program complexity of the languages studied here and their range-restricted subsets does not pay off by a considerable gain in expressive power.

The complexity of first-order logic over lists (without trees) can be easily seen equivalent to the complexity of first-order theory of term algebras in which all function symbols have arity ≤ 1. The first-order theory of such term algebras is complete in LATIME($2^{O(n)}$): the class of problems solvable by alternating Turing machines running in exponential time but only with a linear number of alternations, for details see [14, 7, 32, 31, 34]. The range-restricted fragment of first-order logic over lists is PSPACE-complete, so there is also a gap between the complexities of the full and the range-restricted fragment.

The next two theorems characterize the data complexity. As usual in the case of the data complexity, we restrict ourselves to boolean queries [30]for details see. To define the data complexity, we need to measure the size of instances of the input database schema. The domain $\mathcal{U} \cup \mathcal{L} \cup \mathcal{T}$ on which queries are evaluated is infinite, but we will restrict it to the finite one using the following idea.

Every instance $d = (P_1 : r_1, \ldots, P_m : r_m)$ of an input database schema $(P_1 : R_1, \ldots, P_m : R_m)$ induces the sorted first-order structure M_d with the carrier $\mathcal{U} \cup \mathcal{L} \cup \mathcal{T}$ and signature

$$\{P_1, \ldots, P_m, empty\_list, rel\_cons, empty\_tree, rel\_tree\},$$

in which each P_i is interpreted as r_i, and the predicate symbols

$$empty\_list, rel\_cons, empty\_tree, rel\_tree$$

have their natural interpretation. Consider the substructure M_d^* of M_d whose carrier consists of all elements of $\mathcal{U}$ plus all sublists of the lists occurring in the input relations r_i, and all subtrees of the trees occurring in the input relations r_i. Since the input relations are finite, the structure M_d^*

[1]The nonelementary lower bound for the first-order theory of trees follows immediately from a result proved in [11]: the theory of any pairing function (i.e. binary one-to-one function) has a hereditary nonelementary lower bound.

is finite. Now we define the *size* of M_d as the size of M_d^* measured as in e.g. [20].

Theorem 5 (reduction of arbitrary queries to queries over finite structures)
For every query q expressible in first-order logic over lists and trees there exists a query q^* expressible in first-order logic over lists and trees such that $q(M_d) = q^*(M_d^*)$.

Since M_d^* is a finite structure, a first-order boolean query on M_d^* can be evaluated by a straightforward procedure polynomial in the size of M_d^*, therefore Theorem 5 implies

Theorem 6 (polynomial data complexity) *Every boolean query expressible in first-order logic over lists and trees can be evaluated in polynomial time.*

Remark. A more precise bound depends on how we specify a model of computation and a method of encoding of inputs. [20] proved that every first-order query on finite structures (even with order) is in AC_0 under a suitable encoding. If we use [20] encoding of inputs, then every boolean query expressible in first-order logic over lists and trees is also in AC_0. Furthermore, it is not hard to argue that this result also holds when the domain $\mathcal{U}$ of urelements is equipped with an order $<$.

[34] noted that it would be interesting to consider parametric complexity of nonrecursive query languages for lists and trees. We note that the standard notions of parametric complexity cannot be useful since the program complexity of the languages studied here is not elementary recursive. Of course we can measure complexity of the transformed query q^* by its number of quantifiers, but it will hardly give us any reasonable estimation of the original query q, since the size of q^* is, in general, nonelementary in the size of q.

Theorem 4 on the connection between arbitrary and range-restricted queries, Theorem 5 on reduction of arbitrary queries to queries over finite domains, and Theorem 6 on the polynomial data complexity are based on a technical result called the Safety Theorem and proved in Section 5.2.

5. PROOFS

This section contains proofs of Theorems 4 and 5. In Section 5.1 we will introduce term algebras as a simplified model for lists and trees, and in Section 5.2 prove the Safety Theorem from which both 4 and Theorems 5 will immediately follow.

5.1 A simplified domain for lists and trees

To simplify the presentation, we will make some changes in the definition of first-order logic over lists and trees. The reader can easily check that the proofs in this appendix can be immediately turned into the proofs for the original definition. Basically, the simplification is as follows. First, lists are omitted. Second, sorts are omitted as well. Third, the empty tree is omitted, and the tree constructor $\top$ is turned into a binary *pairing function* f. Fourth, we assume that the input database schema contains only one unary input predicate R. These restrictions amount to saying that we consider the *class* K of structures in the signature $\sigma = \{f, R\}$, where R is interpreted as a finite set, and f is a pairing function. The urelements are the elements that are not of the form $f(a, b)$.

Also for simplification, we say that two first-order formulas are *equivalent* if they are equivalent in all structures of K. Likewise, we call a first-order formula *valid* if it is valid in every structure of K.

Our main technical tool is the notion of *safe formulas* introduced below, [28]cf.. The purpose of this section is to prove that every sentence is equivalent to a boolean combination of safe sentences. This statement (the *Safety Theorem*) plays the same role as quantifier elimination in term algebras, e.g. [16].

For a structure $S \in K$, we denote by R^S the interpretation of R in S, i.e. the set R^S. We will extend the signature σ by adding two unary function symbols l and r. Their interpretation in the structures of K is defined as follows: (i) $l(u) = r(u) = u$ for any urelement u, (ii) $l(f(t_1, t_2)) = t_1$ and $r(f(t_1, t_2)) = t_2$ for all t_1, t_2. The extended signature is denoted by σ^+.

We introduce some notation for terms and formulas of the extended signature. We will leave out parentheses in successive applications of l and r, for example, we write lrt instead of $l(r(t))$. Any term of σ^+ not using f, i.e. built up from variables and c by using zero or more applications of l and r, will be called an *$(l$-$r)$-term*. Lowercase Greek letters α, β, γ stand for sequences, maybe empty, of l's and r's. Such sequences denote compositions in $(l$-$r)$-terms. For example, if $\alpha = lrr$ then αx denotes the term $lrrx$. We write $G(t_1, \ldots, t_n)$ for any term obtained by zero or more applications of f to $(l$-$r)$-terms $t_1, \ldots, t_n$. For example, $G(x, ly)$ may stand for the term $f(x, f(x, ly))$ or the term $f(x, f(ly, x), x)$. We use this notation to emphasize that $t_1, \ldots, t_n$ are $(l$-$r)$-terms. Note that a term $G(t_1, \ldots, t_n)$ has no subterms of the form $lf(s_0, s_1)$ or $rf(s_0, s_1)$.

Definition 1 ($(l$-$r)$-depth) For every free variable x in a formula A of the signature σ^+, we define the *$(l$-$r)$-depth* of x in A as the maximal length of α such that αx appears in A.

For example, the $(l$-$r)$-depth of x in the formula $y \neq lx \wedge R(f(llrx, f(rry, x)))$ is 3.

We denote by $Ur(x)$ the formula $\neg\exists y \exists z(x = f(y, z))$. It is not hard to argue that this formula defines urelements.

Definition 2 (safe formula) The class of *safe formulas* of the signature σ^+ is defined by induction as follows:

1. For any term $G(t_1, \ldots, t_n)$ of σ^+, the formula $R(G(t_1, \ldots, t_n))$ is safe.
2. For any term $G(t_1, \ldots, t_n)$ of σ^+, the formula $Ur(G(t_1, \ldots, t_n))$ is safe.
3. If A is safe and the $(l$-$r)$-depth of x in A is 0, then $\exists x A$ is safe

4. If both A and B are safe, then $A \wedge B$ is safe.
5. If A is safe and t_1, t_2 are (l-r)-terms whose variables are free in A, then $A \wedge t_1 = t_2$ and $A \wedge t_1 \neq t_2$ are safe.
6. If both A and B are safe and every free variable in B is a free variable in A, then $A \wedge \neg B$ is safe.

The intuition behind safe formulas is expressed by the following lemma.

Lemma 1 (finiteness) *Let $A(\bar{x}, \bar{y})$ be a safe formula in which every variable of $\bar{x}$ has (l-r)-depth 0. Let $S \in K$ be a structure. Then for every valuation τ of $\bar{y}$, there exists only a finite number of valuations ν of $\bar{x}$ such that $S, \tau, \nu \models A(\bar{x}, \bar{y})$.*

PROOF. By straightforward induction on the definition of safe formulas, using the fact that both R^S and the set of urelements are finite in every structure $S \in K$. $\square$

It is convenient to think of safe formulas as some kind of atoms. More precisely, we generalize the notion of literals as follows.

Definition 3 (quasi-literal) A *quasi-literal* is either of the following formulas: (i) a safe formula or its negation, (ii) a literal $t_1 = t_2$ or $t_1 \neq t_2$, where t_1 and t_2 are (l-r)-terms.

5.2 Safety theorem

Theorem 7 (safety theorem) *There is an algorithm that transforms any formula A of the signature σ^+ into an equivalent boolean combination A' of quasi-literals. If A is a sentence then A' is a boolean combination of safe sentences.*

The proof will be given below, after a series of lemmas which show how to eliminate "unsafety" in subformulas of A.

Lemma 2 (decreasing (l-r)-depth) *Let $A(x)$ be a quasi-literal with a free occurrence of x. Then for every variables x_0 and x_1 not occurring in $A(x)$, the formula $A(f(x_0, x_1))$ is equivalent to a boolean combination B of quasi-literals. In addition, B has the following property: if x has a non-zero (l-r)-depth d in $A(x)$ then both (l-r)-depths of x_0 and x_1 in B are strictly less than d.*

Lemma 3 (negated formulas) *Let A be a formula*

$$x \neq t_1 \wedge \ldots \wedge x \neq t_n \wedge \neg B_1(x) \wedge \ldots \wedge \neg B_m(x),$$

where (i) the (l-r)-depth of x in A is 0, (ii) each t_i is an (l-r)-term different from x, (iii) each B_j is a safe formula containing a free occurrence of x. Then $\exists x A$ is valid.

PROOF. Let $\bar{y}$ denote the list of all free variables of A different from x. Consider $\neg A$. Each member of this disjunction defines a finite set of values of x in the following sense. Let $D(x, \bar{y})$ denote such a member, i.e. either $x = t_i$ or $B_j(x)$. Let S be any structure in K. For every valuation τ of $\bar{y}$, the number of valuations ν of x such that $S, \tau, \nu \models D(x, \bar{y})$ is finite. Indeed, this is obvious for equalities $x = t_i$ and this follows from Lemma 1 for safe formulas B_j. Therefore, for every valuation τ of $\bar{y}$, the set of valuations ν of x such that $S, \tau, \nu \models \neg A(x, \bar{y})$ is finite. Or, equivalently, for every valuation τ of $\bar{y}$, the set of valuations ν of x such that $A(x, \bar{y})$ is false in S under the valuations τ and ν is finite. Since the universe of S contains infinitely many trees, one can choose a valuation ν of x such that $S, \tau, \nu \models A(x, \bar{y})$. Hence, $\exists x A$ is valid. $\square$

Lemma 4 (elimination of "unsafe" quantifiers) *Any formula $\exists x(A_1 \wedge \ldots \wedge A_k)$, where $A_1, \ldots, A_k$ are quasi-literals, is equivalent to a boolean combination of quasi-literals.*

PROOF. Using Lemma 2 on decreasing (l-r)-depth, we can reduce the (l-r)-depth of x to zero. Thus, we can assume that our conjunction consists of only four types of quasi-literals.

1. Equalities $x = t$. Since the (l-r)-depth of x is 0, we can assume that x does not occur in t.
2. Disequalities $x \neq t$. As above, we can assume that x does not occur in t.
3. Safe formulas $B(x)$, which contain a free occurrence of x and maybe other free variables.
4. Negated safe formulas $\neg B(x)$, which contain a free occurrence of x and maybe other free variables.

If the conjunction contains $x = t$, we eliminate $\exists x$ by using the equivalence $\exists x(x = t \wedge C(x)) \equiv C(t)$. Therefore, we assume that the conjunction contains no equalities. If the conjunction consists of only disequalities and negated safe formulas, the formula $\exists x(A_1 \wedge \ldots \wedge A_k)$ is valid by Lemma 3. It remains to consider the case where at least one safe formula $B(x)$ occurs in the conjunction.

Now our conjunction $A_1 \wedge \ldots \wedge A_k$ contains members A_i of only three types: $B(x)$, $\neg B(x)$ and $x \neq t$. We call members of the first type *safe*, and the other members *unsafe*. A free variable u in the conjunction is called *unsafe* if it occurs in only unsafe members. Evidently, if there are no unsafe variables, then the conjunction is safe and we are done. We show how to get rid of unsafe variables.

Let u be an unsafe variable. The formula $\exists x(A_1 \wedge \ldots \wedge A_k)$ has the form

$$\begin{aligned} \exists x(C(x) \wedge \neg D_1(x, u) \wedge \ldots \wedge \neg D_k(x, u) \wedge \\ x \neq \alpha_1 u \wedge \ldots \wedge x \neq \alpha_m u \wedge E(x)), \end{aligned} \quad (1)$$

where $C(x)$ is a safe formula and u has no occurrences in $C(x)$ or $E(x)$. Denoting by $D(x, u)$ the formula

$$D_1(x, u) \vee \ldots \vee D_k(x, u) \vee x = \alpha_1 u \vee \ldots \vee x = \alpha_m u,$$

we rewrite (1) as

$$\exists x(C(x) \wedge \neg D(x,u) \wedge E(x)). \qquad (2)$$

Using the tautology

$$\forall x(C(x) \supset \neg D(x,u)) \vee \exists x(C(x) \wedge D(x,u)), \qquad (3)$$

we turn formula (2) into the equivalent formula

$$\begin{array}{ll}\exists x(C(x) \wedge \forall x(C(x) \supset \neg D(x,u)) & \\ \quad \wedge \neg D(x,u) \wedge E(x)) & (4) \\ \vee & \\ \exists x(C(x) \wedge \exists x(C(x) \wedge D(x,u)) & \\ \quad \wedge \neg D(x,u) \wedge E(x)) & (5)\end{array}$$

First, consider subformula (4). Using Modus Ponens, we turn it into the equivalent formula $\forall x(C(x) \supset \neg D(x,u)) \wedge \exists x(C(x) \wedge E(x))$. Since $\forall x(C(x) \supset \neg D(x,u))$ is equivalent to $\neg\exists x(C(x) \wedge D(x,u))$, this formula is a boolean combination of $\exists x(C(x) \wedge E(x))$, in which u does not occur, and $\exists x(C(x) \wedge D(x,u))$. Using the definition of $D(x,u)$, we can rewrite the last formula to the equivalent formula

$$\begin{array}{l}\exists x(C(x) \wedge D_1(x,u)) \vee \ldots \vee \\ \exists x(C(x) \wedge D_k(x,u)) \vee \\ \exists x(C(x) \wedge x = \alpha_1 u) \vee \ldots \vee \\ \exists x(C(x) \wedge x = \alpha_m u).\end{array}$$

This formula is equivalent to

$$\begin{array}{l}\exists x(C(x) \wedge D_1(x,u)) \vee \ldots \vee \\ \exists x(C(x) \wedge D_k(x,u)) \vee C(\alpha_1 u) \vee \ldots \vee C(\alpha_m u),\end{array}$$

The latter is a boolean combination of formulas which either have less quantifiers than $\exists x(A_1 \wedge \ldots \wedge A_k)$ or in which u is safe.

Second, consider subformula (5). Using the definition of $D(x,u)$, we rewrite (5) as

$$\begin{array}{ll} & \exists x(C(x) \wedge \exists x(C(x) \wedge D_1(x,u)) \\ & \quad \wedge \neg D(x,u) \wedge E(x)) \;\; \vee \\ \ldots & \\ \vee & \exists x(C(x) \wedge \exists x(C(x) \wedge D_k(x,u)) \\ & \quad \wedge \neg D(x,u) \wedge E(x)) \\ \vee & \exists x(C(x) \wedge \exists x(C(x) \wedge x = \alpha_1 u) \\ & \quad \wedge \neg D(x,u) \wedge E(x)) \;\; \vee \\ \ldots & \\ \vee & \exists x(C(x) \wedge \exists x(C(x) \wedge x = \alpha_m u) \\ & \quad \wedge \neg D(x,u) \wedge E(x)).\end{array}$$

This formula is equivalent to

$$\begin{array}{ll} & \exists x(C(x) \wedge \exists x(C(x) \wedge D_1(x,u)) \\ & \quad \wedge \neg D(x,u) \wedge E(x)) \;\; \vee \\ \ldots & \\ \vee & \exists x(C(x) \wedge \exists x(C(x) \wedge D_k(x,u)) \\ & \quad \wedge \neg D(x,u) \wedge E(x)) \\ \vee & \exists x(C(x) \wedge C(\alpha_1 u) \\ & \quad \wedge \neg D(x,u) \wedge E(x)) \;\; \vee \\ \ldots & \\ \vee & \exists x(C(x) \wedge C(\alpha_m u) \\ & \quad \wedge \neg D(x,u) \wedge E(x))\end{array}$$

in which all occurrences of u are safe. This completes the proof. □

5.3 Proof of Theorem 7

The input formula A is transformed as follows. First, we eliminate all occurrences of $\forall x$ in A by replacing them by $\neg\exists x\neg$. Next, we eliminate all occurrences of f in equalities and disequalities as in the proof of Lemma 2. Likewise, we get rid of all subterms of the form $lf(t_0,t_1)$ and $rf(t_0,t_1)$. Now, each atomic subformula of A is a quasi-literal.

We continue the proof by induction on A. The cases when A is a conjunction, disjunction, or negation are obvious. It remains to consider the case when A is $\exists x B$, where B is a boolean combination of quasi-literals. By moving negations inwards and using the distributivity laws for conjunction and disjunction, we can turn B into a formula

$$\bigvee_i \bigwedge_j B_{i,j},$$

where each $B_{i,j}$ is a quasi-literal. Then we can replace $\exists x B$ by the equivalent disjunction

$$\bigvee_i \exists x \bigwedge_j B_{i,j}.$$

Now we apply Lemma 4 to turn every subformula $\exists x \bigwedge_j B_{i,j}$ into a boolean combination of quasi-literals.

The second part of the theorem is much easier. Suppose that A is a sentence. Apply the above algorithm to turn A into a boolean combination of quasi-literals $A_1,\ldots,A_k$. Since the algorithm introduces no new free variables, every A_i is closed, therefore is a safe sentence or its negation (because an equality $s = t$ or a disequality $s \neq t$ cannot be closed).

5.4 Proof of Theorem 5

In this section we prove Theorem 5 or, more precisely, its reformulation for the simplified model of lists and trees.

For every structure $S \in K$, by D^S we denote the union of two sets: the set of all urelements of S and the set of all subtrees of the trees belonging to R^S. According to our definition of structures with trees, D^S is finite.

Lemma 5 (bounded quantifiers) *Let S be a structure in K. Let $A(\bar{x})$ be a safe formula of the signature σ in which*

$\bar{x}$ *are all free variables in* A. *For any valuation* ν *of* $\bar{x}$, *if* $S, \nu \models A(\bar{x})$ *then* ν *maps* $\bar{x}$ *to* D^S.

PROOF. By straightforward induction on the definition of safe formulas. □

One can easily see that this lemma, when reformulated for the original definitions of lists and trees, shows that safe formulas can be evaluated on M_d^* instead of M_d. To this end, we extend σ by a new monadic relation symbol D with the following interpretation: for every structure S, the symbol D is interpreted by the set D^S of trees. *Bounded quantifiers* $(\forall x \in D)A$ and $(\exists x \in D)A$ denote $\forall x(D(x) \supset A)$ and $\exists x(D(x) \wedge A)$ respectively. For every formula A, we denote by $A|_D$ the formula obtained from A by replacement of all quantifiers $\forall x$ and $\exists x$ by the respective bounded quantifiers $\forall x \in D$ and $\exists x \in D$.

Lemma 6 *If* A *is a safe formula of the signature* σ, *then* A *is equivalent to* $A|_D$.

PROOF. By induction on the definition of safe formulas. The only non-trivial case is when A is $\exists x B$, where B is safe. By Lemma 5, $\exists x B$ is equivalent to $(\exists x \in D)B$. By the induction hypothesis, the latter formula is equivalent to $(\exists x \in D)B|_D$, i.e. to $A|_D$. □

This lemma immediately implies Theorems 5.

5.5 Proof of Theorem 6

Lemma 7 (evaluation of safe sentences) *For any safe sentence* A *of the signature* σ, *the problem of checking whether* $S \models A$ *holds or not, where* $S \in K$, *can be solved in time polynomial in the size of* S.

PROOF. Since A is a safe sentence, l and r do not occur in A, i.e. A is in the signature σ. By Lemma 5, all quantifiers in A can be bounded by the set D^S. Therefore, the value of A' on S can be computed in time polynomial in the size of D^S. To give a more precise bound, we would need to specify a model of computation and a method of encoding structures with trees. □

Theorem 6 immediately follows from this lemma.

5.6 Proof of Theorem 4

Lemma 8 *Every safe formula of the signature* σ *can be defined by a range-restricted nonrecursive Prolog query. In particular, every safe sentence of the signature* σ *can be defined by a range-restricted nonrecursive boolean Prolog query.*

PROOF. By induction on the definition of safe formulas, straightforward but tedious. □

Now we prove Theorem 4. Suppose that A is a first-order sentence of the signature σ defining a boolean query q. We have to show how to represent q by a boolean combination of range-restricted nonrecursive Prolog queries. By the Safety Theorem, A is a boolean combination of safe sentences. By Lemma 8, every safe sentence can be defined by a range-restricted nonrecursive boolean Prolog query, therefore A is a boolean combination of such queries.

6. QUANTIFIER ELIMINATION IN TERM ALGEBRAS WITH FINITE DOMAINS

We would like to note that our results can be reformulated in their most general form for term algebras instead of the domain for lists and trees.

In this section we reformulate the main results for terms algebras. Their proof are similar to the proofs given above. We will use the notions introduced above (e.g. query) for term algebras. Note that every term algebra is induced by its signature, so we formulate the results for arbitrary term algebras (i.e. term algebras in arbitrary signatures).

Theorem 8 *A boolean query* q *is expressible in the first-order theory of a term algebra if and only if* q *is a boolean combination of boolean first-order queries over this term algebra.*

Theorem 5 becomes a statement that generalises the quantifier eleimination theorem for term algebras.

Theorem 9 *For every first-order query* q *over a term algebra there exists a first-order query* q^* *such that the value of* q *on arbitrary input relations (evaluated over the term algebra) coincides with the value of* q^* *on the same input but evaluated over the finite subset of the term algebra consisting of the subterms of terms in the input relations.*

Theorem 6 holds for arbitrary term algebras:

Theorem 10 *Every first-order boolean query over a term algebra can be evaluated in polynomial time.*

7. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundation of Databases.* Addison-Wesley Publishing Co., 1995.

[2] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, 43:62–124, 1991.

[3] A. Aho and J. Ullman. Universality of data retrieval languages. In *ACM Symposium on Principles of Programming Languages*, pages 110–117, 1979.

[4] K. 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] 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.

[6] M. Benedikt and L. Libkin. Safe constraint queries. In *PODS'98*, pages 99–108, Seattle, Washington, 1998.

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

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

[9] A. Chandra and D. Harel. Structure and complexity of relational queries. *Journal of Computer and System Sciences*, 25:99–128, 1982.

[10] L. Colby, E. Robertson, L. Saxton, and D. V. Gucht. A query language for list-based complex objects. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 179–189, 1994.

[11] K. Compton and C. Henson. A uniform method for proving lower bounds on the computational complexity of logical theories. *Annals of Pure and Applied Logic*, 48:1–79, 1990.

[12] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. Technical Report INFSYS RR-1843-99-05, Institut für Informationssysteme, Technische Universität Wien, Feb. 1999.

[13] R. Fagin. Monadic generalized spectra. *Zeitschrift für Mathematical Logic und Grundlagen der Mathematik*, 21:89–96, 1975.

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

[15] H. Gaifman. On local and nonlocal properties. In J. Stern, editor, *Logic Colloquium'81*, pages 105–135. North Holland, 1982.

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

[17] R. Hull and J. Su. Algebraic and calculus query languages for recursively defined complex objects. *Journal of Computer and System Sciences*, 47:121–156, 1993.

[18] R. Hull and J. Su. Domain independence and the relational calculus. *Acta Informatica*, 31:513–524, 1994.

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

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

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

[22] G. Kuper and M. Vardi. The logical data model. *ACM Transactions on Database Systems*, 18(3):379–413, 1993.

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

[24] 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.

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

[26] J. Paredaens, J. Van den Bussche, and D. Van Gucht. First-order queries on finite structures over the reals. In *LICS'95*, pages 79–87, 1995.

[27] T. Rybina and A. Voronkov. A decision procedure for term algebras with queues. Preprint CSPP-2, Department of Computer Science, University of Manchester, Feb. 2000.

[28] J. Ullman. *Database and Knowledge-Base Systems*, volume I,II. Computer Science Press, 1988,1989.

[29] L. Vandeurzen, M. Gyssens, and D. Van Gucht. An expressive language for linear spacial database queries. In *PODS'98*, pages 109–118, Seattle, Washington, 1998.

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

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

[32] 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.

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

[34] S. Vorobyov and A. Voronkov. Complexity of nonrecursive logic programs with complex values. In *PODS'98*, pages 244–253, Seattle, Washington, 1998. ACM Press.