Uppsala Computing Science
Research Report no. 156
December 22, 1998
ISSN 1100-0686

Expressive power and data complexity of nonrecursive logic programming

Evgeny Dantsin

Andrei Voronkov

Computing Science Department
Uppsala University
Box 311
S 751 05 Uppsala, Sweden
{dantsin,voronkov}@csd.uu.se

Abstract

We characterize the expressiveness and data complexity of nonrecursive logic programming. Our main results show that nonrecursive logic programming has the same expressive power as nonrecursive Datalog with negation (under a natural representation of inputs). Thus, the use of recursive data structures, namely trees, in nonrecursive $\mathrm{Datalog}^{\neg}$ gives no gain in the expressive power. It also follows from our results that nonrecursive logic programming, like nonrecursive $\mathrm{Datalog}^{\neg}$, has polynomial data complexity. This contrasts with a huge difference between these query languages in the program complexity.[*] [†]

[†]Evgeny Dantsin is partially supported by grants from TFR, INTAS and RFBR. Andrei Voronkov is partially supported by TFR grants.

Contents

---

1 Introduction

In the context of databases, logic programming is viewed as a query language. The study of the expressiveness and complexity of query languages is one of central topics of research in databases. In this report we characterize the expressive power and data complexity of the nonrecursive fragment of logic programming[3].

The *expressive power* of a query language is measured by the set of queries expressible in this language. The *complexity* of a query language is measured in time and space required to evaluate queries expressible in the language. The complexity of evaluating a query q on a database instance I has three forms, Vardi [1982]: (i) *data complexity* is the complexity expressed as a function of the size of I when q is fixed, (ii) *program complexity* (or *expression complexity*) is the complexity expressed as a function of the size of q when I is fixed, (iii) *combined complexity* is the complexity expressed as a function of the combined size of I and q. Since the size of q is normally much less than the size of I, the data complexity is considered to be most relevant to complexity issues in databases.

We compare nonrecursive logic programming with nonrecursive Datalog with negation[4]. It is well known that when only range-restricted rules are allowed, Datalog$^\neg$ has the same expressive power as the relational algebra. Vardi [1982] and Immerman [1987] characterized the complexity of nonrecursive Datalog$^\neg$: its data complexity is in AC_0, i.e. in the class consisting of languages decided by unbounded fan-in circuits of polynomial size and constant depth [see e.g. Papadimitriou 1994].

---

[3]In our terminology, logic programming includes negation.

[4]Datalog with negation is denoted by Datalog$^\neg$. Its rules are allowed to be not range-restricted.

Queries in nonrecursive Datalog$^\neg$ are easy to evaluate, however its expressiveness is not enough for many applications. There are two natural approaches to increasing its expressive power. The first is to allow recursive rules, which leads to full Datalog$^\neg$. The second is to allow more complex values, for example recursive data structures like sets or trees. From this standpoint, nonrecursive logic programming can be viewed as an example of the second approach, namely, as "Datalog$^\neg$ with trees". However, as our main results show, this extension does not increase the expressive power.

More precisely, we prove the following results:

1. Every query definable by a nonrecursive logic program can be evaluated in time polynomial in the size of a database instance. A more precise bound depends on how database instances are encoded and what model of computation is used.
2. A query is definable by a nonrecursive logic program if and only if this query is definable by a nonrecursive Datalog$^\neg$ program. Since logic programs and Datalog programs have different inputs, this result is proved with respect to a natural representation of logic programming inputs by Datalog inputs.
3. Since queries definable by nonrecursive Datalog$^\neg$ programs are in AC_0 [Immerman 1987], we obtain the following consequence: under a natural encoding of inputs, queries definable by nonrecursive logic programs are in AC_0.
4. We introduce a relational algebra of trees and prove that a query is definable in this algebra if and only if this query is definable by a nonrecursive range-restricted logic program.
5. A query is definable by a nonrecursive logic program if and only if this query is a boolean combination of queries definable by range-restricted nonrecursive logic programs.

All of our results are based on a technical theorem proved in Section 3. This theorem, called the Safety Theorem, plays the same role as quantifier elimination in term algebras [Hodges 1993].

There is a striking difference between the data complexity and program complexity of nonrecursive logic programs. [Vorobyov and Voronkov 1998] proved that (i) the program complexity of nonrecursive logic programming is not elementary recursive[5], (ii) range-restricted nonrecursive logic programming is program complete for a complexity class intermediate between NTIME($2^{O(n)}$) and DSPACE($2^{O(n)}$). At the same time, it follows from our report that both query languages have polynomial data complexity. Note that nonrecursive Datalog$^\neg$ is PSPACE-complete (with respect to the program complexity). Therefore, the huge gap in the program complexity between nonrecursive logic programming and nonrecursive Datalog$^\neg$ does not pay off by a gain in the expressive power.

2 Preliminaries

Nonrecursive logic programs define the same set of queries as first-order formulas over term algebras [see e.g. Vorobyov and Voronkov 1998]. Therefore, either of these notations can be used to define the set of queries that we study in this report. It is more convenient for us to define queries in terms of first-order formulas interpreted in a special class of structures.

By first-order logic we mean first-order logic with equality. Definitions of basic notions of first-order logic such as a structure, a term, a sentence, etc. are standard [e.g. Hodges 1993]. For a

[5]This result is a straightforward consequence of the nonelementary lower bound for the first-order theory of term algebras proved by Vorobyov [1996].

formula A, the set of all free variables of A is denoted by $free(A)$. If M is a structure and A is a formula or relation symbol, the interpretation of A in M is denoted by A^M.

DEFINITION 2.1 (term algebra) Let Δ be a signature containing at least one constant. The *term algebra* of the signature Δ is the Δ-structure denoted by $TA(\Delta)$ and defined as follows:

1. The domain of $TA(\Delta)$ is the set of all ground terms of Δ. This set is called the *Herbrand base* of Σ and denoted by $HB(\Delta)$.
2. Every ground term is interpreted by itself. Any two distinct ground terms are unequal.
3. Every relation symbol (if any) is interpreted by an empty relation.

Term algebras are also called *absolutely free algebras* [see Maĺcev 1961, Maĺcev 1971]. The decidability of term algebras in finite signatures was proved by Maĺcev [1961] by quantifier elimination. Later [Kunen 1987, Maher 1988, Hodges 1993] proved it for finite and infinite signatures, also by using quantifier elimination. Various complexity results for term algebras in different signatures were proved in [Stockmeyer and Meyer 1973, Stockmeyer 1977, Volger 1983, Vorobyov 1996].

DEFINITION 2.2 (finite-relation extensions) Let Δ be a signature containing at least one constant. We say that a Δ-structure M is a *finite-relation extension* of $TA(\Delta)$ if M satisfies conditions 1 and 2 in Definition 2.1 above and the following condition: every relation symbol is interpreted by a finite relation. The class of all finite-relation extensions of $TA(\Delta)$ is denoted by $TA^*(\Delta)$.

We prove our results for a fixed *signature* Σ consisting of one monadic *relation symbol* P, one binary *function symbol* f and one *constant* a. It is straightforward to generalize the results for arbitrary finite signatures.

Two formulas A and B over Σ are called *equivalent* if they are equivalent in $TA^*(\Sigma)$, i.e. in every finite-relation extension of $TA(\Sigma)$. Likewise, a formula is called *valid* if it is valid in $TA^*(\Sigma)$.

Note that any structure $M \in TA^*(\Sigma)$ is uniquely determined by the value P^M of the input predicate. For every finite subset T of $HB(\Sigma)$, we denote by M_T the structure $M \in TA^*(\Sigma)$ such that $P^M = T$.

First-order sentences of Σ can be viewed as a query language. Let A be such a sentence. Then a query Q_A is defined as a mapping from the set of finite subsets of $HB(\Sigma)$ to $\{\text{TRUE}, \text{FALSE}\}$ such that the following holds: a set $\{t_1, \ldots, t_n\}$ of ground terms is mapped to TRUE if and only if the sentence A is true in the structure in $TA^*(\Sigma)$ in which P is interpreted as $\{t_1, \ldots, t_n\}$. In other terms, $Q_A(\{t_1, \ldots, t_n\})$ is TRUE if and only if the following formula is valid in $TA^*(\Sigma)$:

$$\forall x(P(x) \equiv (x = t_1 \vee \ldots \vee x = t_n)) \supset A.$$

Using the fact that nonrecursive logic programs and first-order formulas over term algebras have the same expressiveness, we give the following definition.

DEFINITION 2.3 (queries definable by nonrecursive logic programs) We say that a query q is *definable* by a nonrecursive logic program if q is Q_A for some sentence A.

For our constructions, we will also consider the *extended signature* Σ^+ obtained from Σ by adding two unary *function symbols* l, r. We define l, r on structures in $TA^*(\Sigma)$ as follows:

$$l(a) = r(a) = a, \quad l(f(s,t)) = s, \quad r(f(s,t)) = t.$$

We will denote successive applications of l and r without parentheses, for example, we write lrt instead of $l(r(t))$. Any term of Σ^+ not using f, i.e. built up from variables and a by using zero or more applications of l and r, will be called an *$(l$-$r)$-term*.

3 Safety Theorem

Our main technical tool will be the notion of *safe formulas* introduced below. The purpose of this section is to prove that every sentence of Σ^+ is equivalent to a boolean combination of safe sentences.

First, we introduce some notation for terms and formulas of Σ^+. 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, $G(a, x, ly)$ may stand for the term $f(a, f(x, ly))$ or the term $f(f(ly, a), x)$. 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 $l(f(s_0, s_1))$ or $r(f(s_0, s_1))$.

Definition 3.1 For every free variable x in a formula A, 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 $y \neq lx \wedge P(f(llrx, f(rry, a)))$ is 3.

Definition 3.2 (safe formula) The class of *safe formulas* is defined by induction as follows:

1. For any term $G(t_1, \ldots, t_n)$ of the signature Σ^+, the formula $P(G(t_1, \ldots, t_n))$ is safe.
2. If a formula A is safe and the (l-r)-depth of x in A is 0 then $\exists x A$ is safe.
3. If formulas A and B are safe, then $A \wedge B$ is safe.
4. If A is a safe formula and s, t are (l-r)-terms whose variables are in *free*(A), then the formulas $A \wedge s = t$ and $A \wedge s \neq t$ are safe.
5. If A and B are safe formulas and *free*$(B) \subseteq$ *free*(A), then the formula $A \wedge \neg B$ is safe.

The intuition behind formulas is expressed by the following lemma.

Lemma 3.3 (finiteness) *Let $A(x)$ be a safe formula in which x is the only free variable and x has (l-r)-depth 0. Then for every structure M in $TA^*(\Sigma)$, the set of ground terms t such that $A(t)$ is true in M is finite.*

Proof. By straightforward induction on the definition of safe formulas, using the fact that P^M is finite in every structure M in $TA^*(\Sigma)$. □

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

Definition 3.4 (quasi-literal) A *quasi-literal* is either of the following formulas:

1. A safe formula or its negation.
2. A literal $s = t$ or $s \neq t$, where s and t are (l-r)-terms.

The following *safety theorem* is central to our results.

THEOREM 3.5 (safety theorem) *There is an algorithm that transforms any formula A of the signature Σ^+ into an equivalent boolean combination A' of quasi-literals. In particular, 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 3.6 (decreasing (l-r)-depth) *Let $A(x)$ be a quasi-literal containing 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.*

PROOF. Without loss of generality we assume that $A(x)$ does not contain trivial equalities $x = \alpha x$ or inequalities $x \neq \alpha x$. The formula $A(f(x_0, x_1))$ may not be a quasi-literal for two reasons. First, terms $l(f(x_0, x_1))$ and $r(f(x_0, x_1))$ may appear. In this case we replace them by x_0 and x_1 respectively. Second, f may appear in equalities and disequalities. Then we replace such equalities and disequalities using the following equivalences:

$$\begin{array}{rcl} f(x_0, x_1) = t & \equiv & (t \neq a \wedge x_0 = lt \wedge x_1 = rt), \\ f(x_0, x_1) \neq t & \equiv & (t = a \vee (t \neq a \wedge (x_0 \neq lt \vee x_1 \neq rt))). \end{array}$$

The second equivalence may lead to disjunctions that are not quasi-literals. In this case, we turn the formula into a boolean combination of quasi-literals by using the distributivity laws (induction on the definition of quasi-literals).

The assertion about the depth of x_0, x_1 is straightforward. □

LEMMA 3.7 (occurrences in 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 M be any structure in $TA^*(\Sigma)$. For every ground substitution $\bar{s}$ for $\bar{y}$, the set of ground terms t such that $D(t, \bar{s})$ is true in M is finite. Indeed, this is obvious for equalities $x = t_i$ and this follows from Lemma 3.3 for safe formulas B_j. Therefore, for every ground substitution $\bar{s}$ for $\bar{y}$, the set of ground terms t such that $\neg A(t, \bar{s})$ is true in M is finite. Or, equivalently, for every ground substitution $\bar{s}$ for $\bar{y}$, the set of ground terms t such that $A(t, \bar{s})$ is false in M is finite. Since $HB(\Sigma)$ contains infinitely many terms, one can choose a term t such that $A(t, \bar{s})$ is true in M. Hence, $\exists x A$ is valid. □

LEMMA 3.8 (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 3.6 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 kinds 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 an equality $x = t$, 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 3.7. 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 only members A_i of three kinds: $B(x)$, $\neg B(x)$ and $x \neq t$. We call the first kind of members *safe*, and two other kinds *unsafe*. A free variable u in the conjunction is called *unsafe* if it only occurs in 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

$$\exists x(C(x) \wedge \neg D_1(x, u) \wedge \ldots \wedge \neg D_k(x, u) \wedge x \neq \alpha_1 u \wedge \ldots \wedge x \neq \alpha_m u \wedge E(x)), \qquad (1)$$

where $C(x)$ is a safe formula and u does not occur in $C(x)$, $E(x)$. 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

$$\exists x(C(x) \wedge \forall x(C(x) \supset \neg D(x, u)) \wedge \neg D(x, u) \wedge E(x)) \qquad (4)$$
$$\vee$$
$$\exists x(C(x) \wedge \exists x(C(x) \wedge D(x, u)) \wedge \neg D(x, u) \wedge E(x)) \qquad (5)$$

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 can is equivalent to

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

that 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

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

This formula is equivalent to

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

In the last formula all occurrences of u are safe. This completes the proof. □

Proof (of Safety Theorem 3.5). 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 3.6. Likewise, we get rid of all subterms of the form $l(f(t_0,t_1))$ and $r(f(t_0,t_1))$. 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 3.8 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. This means that if some A_j is an equality $s=t$ or a disequality $s \neq t$, then s,t are ground. Therefore, such A_j can be replaced by TRUE or FALSE. After that we obtain a boolean combination of safe formulas. □

4 Query evaluation

The following lemma asserts the key property of safe formulas which will be used to evaluate safe sentences in structures of $TA^*(\Sigma)$.

LEMMA 4.1 *Let $A(x_1,\ldots,x_n)$ be a safe formula of the signature Σ and $x_1,\ldots,x_n$ be all free variables in A. Let M be a structure in $TA^*(\Sigma)$. For any ground terms $t_1,\ldots,t_n$, if $A(t_1,\ldots,t_n)$ is true in M then $t_1,\ldots,t_n$ are subterms of terms belonging to P^M.*

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

THEOREM 4.2 (evaluation of safe sentences) *For every safe sentence A, there is a polynomial-time algorithm that takes as input a finite set T of ground terms and checks whether A is true in M, where M is the structure in $TA^*(\Sigma)$ specified by the condition $P^M = T$.*

PROOF. Since A is a safe sentence, A may contain occurrences of l and r only in ground terms αa which can be replaced by a. Therefore, we can assume that A is in the signature Σ. By Lemma 4.1, all quantifiers in A can be bounded by the set $\{s_1,\ldots,s_m\}$ consisting of all terms in T and their subterms. Thus, any quantified subformula $\exists x B(x)$ is equivalent to the disjunction $B(s_1) \vee \ldots \vee B(s_m)$. Using such equivalences, we can eliminate all quantifiers and turn A into an equivalent quantifier-free formula A'. The size of A' does not exceed $|A| \cdot |T|^{ck}$, where A denotes the size of A, $|T|$ denotes the total size of terms in T, k is the number of quantifiers in A, and c is a constant depending on how we encode terms and formulas. Given A' and T, we can evaluate A' in polynomial time. To obtain a more precise bound, we need to specify a model of computation and a method of encoding inputs. □

THEOREM 4.3 (polynomial computability) *Let A be any first-order sentence of the signature Σ. Then the query Q_A can be evaluated in time polynomial in the total size of input terms.*

PROOF. By Safety Theorem 3.5, A is equivalent to a boolean combination of some safe sentences $A_1,\ldots,A_m$. By Theorem 4.2, every query Q_{A_i} can be evaluated in time polynomial in the size of input terms. □

5 Nonrecursive logic programming and Datalog$^\neg$

Since logic programming is an extension of Datalog$^\neg$, the set of queries expressible in nonrecursive Datalog$^\neg$ is contained in the set of queries expressible in nonrecursive logic programming. To speak about the inverse inclusion, we need to represent inputs to logic programs by inputs to Datalog programs. In this section we introduce a natural representation of logic programming inputs by Datalog inputs. Under this representation, every query expressible by a nonrecursive logic program is shown to be expressible by nonrecursive Datalog$^\neg$. As a consequence, we show that every query expressible by a nonrecursive logic program is in AC_0.

As above, we assume that logic programs are written in the signature Σ. For Datalog$^\neg$ programs, a new function-free *signature* Σ' is introduced. This signature consists of the relation symbol P plus a unary relation symbol C and a ternary relation symbol F. The intuitive meaning of C is $\{a\}$

and of $F(x, y, z)$ is $x = f(y, z)$. The signature Σ' also contains other relation symbols for *output* and *intermediate* relations.

In view of the well-known equivalence between nonrecursive Datalog$^\neg$ and first-order logic [Vardi 1982], we will consider first-order formulas of the signature Σ'.

Let us define a mapping ρ from $TA^*(\Sigma)$ into the set of Σ'-structures. Every Σ-structure M in $TA^*(\Sigma)$ is mapped to a Σ'-structure $\rho(M)$ defined as follows:

1. The domain of $\rho(M)$ is the set D of all terms in P^M and their subterms.

2. The symbols P, C, F are interpreted as follows. For all $r, s, t \in D$

$$\begin{array}{rcl} t \in P^{\rho(M)} & \Leftrightarrow & t \in P^M, \\ (s, t, u) \in F^{\rho(M)} & \Leftrightarrow & s = f(t, u), \\ t \in C^{\rho(M)} & \Leftrightarrow & t = a. \end{array}$$

DEFINITION 5.1 Let q be a query, i.e. a mapping from the set of finite subsets of $HB(\Sigma)$ to $\{\text{TRUE}, \text{FALSE}\}$. Let A be a first-order formula of the signature Σ'. We say that A *defines the query q with respect to ρ* if the following statements are equivalent:

1. The query q maps a set T of ground terms to TRUE.

2. The formula A is true in the structure $\rho(M_T)$.

THEOREM 5.2 *Any query q definable by a nonrecursive logic program is definable with respect to ρ by a first-order formula.*

The proof of the theorem is heavily based on Safety Theorem 3.5 and Lemma 4.1. The main idea of the proof is that all quantifiers in safe formulas can be bounded by the set of subterms of the input. The proof will be given below after two lemmas.

Let us introduce bounded quantifiers. We extend the signature Σ by a new predicate symbol S and modify the class of structures $TA^*(\Sigma)$ as follows. For every such structure M, we let S^M be the set of all subterms of P^M. We also introduce *bounded quantifiers* of the form $(\forall x \in S)$ and $(\exists x \in S)$ with the natural interpretation: $(\forall x \in S)A$ is equivalent to $\forall x(S(x) \supset A)$ and $(\exists x \in S)A$ is equivalent to $\exists x(S(x) \wedge A)$. For every formula A of the extended signature, denote by A^S the formula obtained from A by replacement of all quantifiers $\forall x$ and $\exists x$ by the corresponding bounded quantifiers $(\forall x \in S)$ and $(\exists x \in S)$.

LEMMA 5.3 *Every safe formula A is equivalent to A^S.*

PROOF. By induction on A. The only case we have to consider is when A has the form $\exists x B$, B is safe and the $(l\text{-}r)$-depth of x in B is 0. By Lemma 4.1, $\exists x B$ is equivalent to $(\exists x \in S)B$. By the induction hypothesis, B is equivalent to B^S. Therefore $A = \exists B$ is equivalent to $(\exists x \in S)B^S$. By the definition of A^S, the last formula is A^S. Therefore, A is equivalent to A^S. □

LEMMA 5.4 *Every safe sentence A is equivalent to a sentence B such that (i) all quantifiers in B are bounded, (ii) all occurrences of f in B are in atomic formulas of the form $x = f(y, z)$, where x, y, z are variables, and (iii) all occurrences of a in B are in atomic formulas of the form $x = a$.*

Proof. By Lemma 5.3, we can assume that all quantifiers in A are bounded. Note that in safe sentences, f can only occur in atomic subformulas $P(G(\bar{x}))$. We demonstrate elimination of such occurrences of f by an example. For example, suppose that this atomic formula is $P(f(x, f(x, y))$. Introduce new variables for all non-variable subterms of this formula. In our case we introduce u for $f(x, f(x, y))$ and v for $f(x, y)$. Now we can replace the formula $P(f(x, f(x, y))$ by $(\exists u \in S)(\exists v \in S)(P(u) \wedge u = f(x, v) \wedge v = f(x, y))$. The elimination of occurrences of a is similar. □

Now we can prove Theorem 5.2.

Proof. Let q be a query defined by a nonrecursive logic program. By Theorem 3.5, q can be defined by a boolean combination A of safe sentences. By Lemma 5.4 we can assume that all quantifiers in A are bounded, all occurrences of f in A are in atomic formulas of the form $x = f(y, z)$, where x, y, z are variables, and all occurrences of a in A are in atomic formulas of the form $x = a$. We turn A into a formula A' of the signature Σ' as follows:

1. Replace all subformulas $x = f(y, z)$ by $F(x, y, z)$.
2. Replace all subformulas $x = a$ by $C(x)$.
3. Replace every bounded quantifier $(\forall x \in S)$ or $(\exists x \in S)$ by the corresponding unbounded quantifier $\forall x$ or $\exists x$.

It is not hard to argue that for every finite subset T of $HB(\Sigma)$, the formula A is true in M_T if and only if A' is true in $\rho(M_T)$. □

Immerman [1987] shows that every query definable in first-order logic over finite structures is in AC_0 (the class AC_0 consists of languages decided by unbounded fan-in circuits of polynomial size and constant depth [see e.g. Papadimitriou 1994]). Since nonrecursive Datalog$^\neg$ expresses the same queries as first-order logic, Immerman's [1987] result is often cited as a result about the expressive power of nonrecursive Datalog$^\neg$. To prove a similar result for nonrecursive logic programs, we define a specific encoding of inputs.

Let T be a finite set of ground terms and $t_0, \ldots, t_{n-1}$ be all terms occurring in T as subterms, in an arbitrary order. We encode the structure M_T as a sequence of bits $B_T = b_0 \ldots b_{n^3+2n-1}$ as follows. The first n^3 bits of B_T encode the relation $x = f(y, z)$ on $\{t_0, \ldots, t_{n-1}\}$ as follows: $t_i = f(t_j, t_k)$ if and only if $b_{in^2+jn+k} = 1$. The next n bits encode the input relation P on $\{t_0, \ldots, t_{n-1}\}$ as follows: $t_i \in T$ if and only if $b_{n^3+i} = 1$. The last n bits encode a as follows: $t_i = a$ if and only if $b_{n^3+n+i} = 1$. Note that the encoding B_T can be obtained in polynomial time from any standard representation of T as a sequence of terms, and vice versa. Therefore, Theorem 4.3 on polynomial data complexity of nonrecursive logic programming holds when the input is represented as B_T.

Theorem 5.5 *Under the encoding B_T, every query definable by a nonrecursive logic program is in AC_0.*

Proof. Note that B_T is the encoding used to represent first-order structures in [Immerman 1987]. Then apply Theorem 5.2 and [Immerman 1987]. □

We can use any total order $<$ on $HB(\Sigma)$ as a built-in relation in nonrecursive logic programs and change the encoding B_T to the encoding $B_T^<$, where the sequence of terms $t_0, \ldots, t_{n-1}$ respects the order $<$, i.e. $t_0 < \ldots < t_{n-1}$. Then every structure in M_T has a unique encoding $B_T^<$. The same proof shows that any nonrecursive logic programs with the order $<$ defines a query in AC_0.

6 Relational algebra of trees and safe queries

In this section we define the relational algebra of trees and prove that this algebra has the same expressive power as nonrecursive range-restricted logic programming. In addition, we characterize queries defined by nonrecursive logic programs in terms of queries defined by nonrecursive range-restricted logic programs. As before, we restrict ourselves to trees built from the empty tree a by the application of the binary tree constructor f. Formally, we define a *tree* as any ground term of the signature Σ.

DEFINITION 6.1 (relational algebra of trees) The *relational algebra of trees* consists of the standard five relational algebra operations $\cup, \times, \sigma, \pi, \setminus$ plus the following three operations A, LR, Tr defined as follows. For all sets of tuples S of arity 1 and Q of arity 2,

$$\begin{array}{rcl} A & = & \{a\}, \\ LR(S) & = & \{(s_0, s_1) \mid f(s_0, s_1) \in S\}, \\ Tr(Q) & = & \{f(s_0, s_1) \mid (s_0, s_1) \in Q\}. \end{array}$$

Of course, we can define relational algebras of trees not only for a and f but also for arbitrary sets of tree constructors.

We will use the standard definition of *range-restricted* logic programs [e.g. Lloyd 1987, Abiteboul, Hull and Vianu 1995], also called *safe* [e.g. Ullman 1988].

THEOREM 6.2 *A query q is definable in the relational algebra of trees if and only if q is definable by a range-restricted nonrecursive logic program.*

PROOF. The proof is standard and similar to the proof of the equivalence between the relational algebra and range-restricted nonrecursive Datalog$^\neg$ [e.g. Abiteboul et al. 1995]. □

THEOREM 6.3 *A query q is definable by a nonrecursive logic program if and only if q is a boolean combination of queries definable by range-restricted nonrecursive logic programs. Therefore, a query q is definable by nonrecursive logic program if and only if q is a boolean combination of queries definable in the relational algebra of trees.*

PROOF. Suppose that q is defined by a nonrecursive logic program, and hence by a sentence. By Safety Theorem 3.5, q is a boolean combination of queries definable by safe sentences. Using induction on the definition of safe formulas, it is straightforward to show that every query definable by a safe sentence, can be defined by a range-restricted nonrecursive logic program. For the second assertion, we use Theorem 6.2. □

Note that the "small" gain in expressive power of arbitrary vs range-restricted nonrecursive logic programs contrasts the gap in their program complexity which is not elementary recursive for arbitrary nonrecursive logic programs and is in DSPACE($2^{O(n)}$) for range-restricted nonrecursive logic programs.

7 Related works

Results on the expressiveness and complexity of query languages are too numerous to be mentioned here, see, e.g. the recent survey [Dantsin, Eiter, Gottlob and Voronkov 1997] for relevant results on Datalog and logic programming. At the same time, there are only few results on the expressive power and complexity of nonrecursive logic programs. First results on the data complexity are due to Vardi [1982] and Immerman [1987]. The full classification of the program complexity is presented in [Dantsin and Voronkov 1997, Vorobyov and Voronkov 1998].

Results about the expressive power and complexity of recursive logic programming are proved in [Andréka and Németi 1978, Tärnlund 1977, Blair 1982, Apt and Blair 1988, Blair, Marek and Schlipf 1995, Schlipf 1995]. In these papers the expressive power of logic programming under various semantics is defined as the class of predicates over the Herbrand base defined in these semantics.

There are many results on extensions of Datalog by complex values different from trees, in particular by finite sets and multisets [e.g. Kuper 1990, Abiteboul and Grumbach 1991, Vadaparty 1991, Abiteboul and Beeri 1995, Grumbach and Vianu 1995, Kanellakis, Kuper and Revesz 1995, Dantsin and Voronkov 1997].

References

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

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

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

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

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

Blair H. [1982], 'The recursion-theoretical complexity of the semantics of predicate logic as a programming language', *Information and Computation* **54**(1/2), 25–47.

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

Dantsin E., Eiter T., Gottlob G. and Voronkov A. [1997], Complexity and expressive power of logic programming, *in* 'Proceedings Twelfth Annual IEEE Conference on Computational Complexity', Ulm, Germany, pp. 82–101.

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

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

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

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

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

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

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

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

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

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

MAĹCEV A. [1971], Axiomatizable classes of locally free algebras of various types, *in* B. Wells III, ed., 'The Metamathematics of Algebraic Systems. Anatoliĭ Ivanovič Maĺcev. Collected Papers: 1936–1967', Vol. 66, North Holland, chapter 23, pp. 262–281.

PAPADIMITRIOU C. [1994], *Computational Complexity*, Addison-Wesley.

SCHLIPF J. [1995], 'The expressive power of the logic programming semantics', *Journal of Computer and System Sciences* **51**(1), 64–86. Abstract in Proc. PODS 90, pp. 196–204.

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

STOCKMEYER L. AND MEYER A. [1973], Word problems requiring exponential time: Preliminary report, *in* 'Proc. ACM STOC', pp. 1–9.

TÄRNLUND S.-A. [1977], 'Horn clause computability', *BIT* **17**, 215–216.

ULLMAN J. [1988], *Database and Knowledge-Base Systems*, Vol. I, Computer Science Press.

VADAPARTY K. [1991], On the power of rule-based languages with sets, *in* 'ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems', pp. 26–36.

VARDI M. [1982], The complexity of relational query languages, *in* 'Proc. 14th ACM STOC', San Francisco, pp. 137–146.

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

VOROBYOV S. [1996], An improved lower bound for the elementary theories of trees, *in* M. McRobbie and J. Slaney, eds, 'Automated Deduction — CADE-13', Vol. 1104 of *Lecture Notes in Computer Science*, New Brunswick, NJ, USA, pp. 275–287.

VOROBYOV S. AND VORONKOV A. [1998], Complexity of nonrecursive logic programs with complex values, *in* 'PODS'98', ACM Press, Seattle, Washington, pp. 244–253.

Index