How to make SQL stand for String Query Language.

Gösta Grahne[1] and Emmanuel Waller[2]

[1] Department of Computer Science
Concordia University
Montreal, Quebec, Canada H3G 1M8
grahne@cs.concordia.ca
[2] LRI, Université Paris-Sud
91405 Orsay Cedex, France
waller@lri.fr

Abstract. A *string database* is simply a collection of tables, the columns of which contain *strings* over some given alphabet. We address in this paper the issue of designing a simple, user friendly query language for string databases. We focus on the language $FO(\bullet)$, which is classical first order logic extended with a concatenation operator, and where quantifiers range over the set of all strings. We wish to capture all *string queries*, i.e., well-typed and computable mappings involving a notion of string genericity. Unfortunately, unrestricted quantification may allow some queries to have infinite output. This leads us to study the "safety" problem for $FO(\bullet)$, that is, how to build syntactic and/or semantic restrictions so as to obtain a language expressing only queries with finite output, hopefully all string queries. We introduce a family of such restrictions and study their expressivness and complexity. We prove that none of these languages express *all* string queries. We prove that a family of these languages is equivalent to a simple, tractable language that we call `SriQueL`, standing for *String Query Language*, which thus emerges a robust and natural language suitable for string querying.

1 Introduction

Current database management systems, especially those based on the relational model, have little support for string querying and manipulation. This can be a problem in several string-oriented application areas such as molecular biology (see e.g. [6, 31]) and text processing, the latter becoming crucial with the burst of the Web, XML and digital libraries, among others. In such a system, a string is one of the basic data types of Codd's relational model [5], which means that the strings are treated as atomic entities; thus a string can be accessed only as a whole and not on the level of the individual characters occuring within it. Modern object-oriented systems are usually alike in this sense. Although they offer support for complex objects, strings are usually treated as atomic. In SQL, the only non-atomic operator is the LIKE-operator which can be used in simple pattern matching tasks such as finding a substring in a field; however, the expressive power of the operator is limited. In this paper, we address the issue of designing a general purpose query language for string databases, based on first order logic.

Lately we have witnessed in the database community an increased research activity into databases having strings, tuples and sets as the principal datatypes. The area deserves to be regarded as a subfield of database theory. It is called "string databases" or "sequence databases," depending on the authors. Here we shall use the former name.

We extend the relational model to include finite strings over some given finite alphabet Σ as primary objects of information. A relation of arity k in our model is then a finite subset of the k-fold Cartesian product of Σ^*, the set of all finite strings over Σ, with itself. In other words, each position in a tuple of a relation contains a string of arbitrary length instead of just a single atomic value. This definition was essentially introduced in [14] an later used in e.g. [17] and [24]. However, a brief excursion into the history books reveals for instance that Stockmeyer [34] was familiar with string relations. Earlier still Quine, [26] showed that first order logic over strings is undecidable.

From the point of view of design, in addition to *data extraction* features, such as "retrieve all palindromes," the string language needs also *data restructuring* constructs [14, 17, 24, 35]. For example, given two unary relations, one might want to concatenate each string from one relation with a string from the other relation, as opposed to merely taking the Cartesian product of the two relations. The former returns a set of "new" strings, whereas the latter returns a set of pairs of strings previously existing in the input instance.

Adequate query languages for string databases have been the aim of several attempts, mainly [14, 17, 24] (see Section 6 for works less close to ours). These three provide full power languages, based on sophisticated primitives, namely transducers, datalog with or without negation, and an original logic equivalent to multi-tape automata. However, in addition to finding complete languages, there is a need to define, and understand in depth, languages *à la* `SQL`, user-friendly, and of low time complexity, without recursion and within LOGSPACE. The "user-friendly" aspect was part of the motivation of the theory of range restriction, inspiring numerous works (see e.g. [1], Chap. 5.3 and 5.4).

In this paper we return to the spirit of "the founding fathers" ([34, 26]). In other words, we will use as string-language relational calculus with an interpreted concatenation function. Our syntax will be called $FO(\bullet)$, where the symbol $\bullet$ stands for concatenation. We will define various semantics, one of them yielding our main language. Since according to a central authority in the field [37], `SQL` (i.e. FO) is "intergalactic dataspeak," we take a parsimonious stand to stick to FO as close as possible. In spite of its rather "formal" look, we believe the simplicity and low complexity of our main language to be well-suited for "everyday" database querying and programming. As a consequence we called it `StriQueL`, standing for *String Query Language*.

The contributions of this paper are the following. String queries are formally defined, with emphasis on string genericity. Given the syntax $FO(\bullet)$, we then consider first evaluating query formulas by making quantifiers range over the whole domain Σ^*. This yields the query language $[\![FO(\bullet)]\!]_{nat}$. The question is then: What are the relationships between $[\![FO(\bullet)]\!]_{nat}$ and string queries? Does the former capture all, or only some of the latter?

One can pursue a "top-down" or a "bottom-up" approach. The "top-down" approach would consist in taking the whole language $[\![FO(\bullet)]\!]_{nat}$, and restricting it syntactically without sacrificing expressive power. In this direction, we prove that the problem whether a formula expresses a string query is (not surprisingly) undecidable. We then undertake a "bottom-up" approach: designing a very simple language, obviously capturing only string queries, then empovering it little by little.

We first study three versions of a restricted semantics, where the quantifiers are not allowed to range over the full domain Σ^*, and we compare them to each other. In a parallel approach, we define syntactic restrictions yielding a language that, we believe, corresponds naturally in the string context to the intuitive relational "`SQL` level." That is, its comfort and expressivity are the ones we wish for "everyday" querying of the database by end-users. We call it $FO_{rr}(\bullet)$ (for *range-restricted*).

It turns out that $FO_{rr}(\bullet)$ is equivalent to one of our semantic restrictions. This result emphasises the robustness of the language. All these properties made us call it `StriQueL`—String Query Language.

Now, it is not surprising that we show that our `StriQueL` language does not express all string queries. More precisely, its complexity is shown to be of the same order of magnitude as its homologue the pure relational SQL, namely logarithmic space.

Although considered a quality for "everyday" use of string databases, these limitations in expressive power are not desirable for more advanced applications. We consider then how to overcome the limitations. We define a safe, non range-restricted, fragment of $FO(\bullet)$, through an operator schema Γ. It is based on an interesting extension, namely introducing an additional symbol in Σ along with particular constraints.

All languages introduced (except $[\![FO(\bullet)]\!]_{nat}$) are shown to compute only string queries, and to be actually evaluable (their semantics is constructive).

Although in molecular biology attempts of string manipulation have been proposed based on grammatical constructs, we believe first order logic over strings should provide an original and very flexible manipulation tool for this area. Moreover, it includes naturally some forms of pattern-matching capabilities. This issue is briefly discussed. Finally, the expressive power of the pure relational fragment of our languages is compared with pure relational FO.

The remainder of the paper is organised as follows. String queries and genericity, $[\![FO(\bullet)]\!]_{nat}$ and the discussion on the decidability of being a string query form Section 2. Section 3 introduces restrictions yielding our String Query Language, and contains a study of it. The Γ operator and the study of the corresponding language are presented in Section 4. In Section 5, we sketch the expressive power of our languages in terms of other formalisms, namely the pure relational and formal languages. Related works are detailed in Section 6, before concluding and presenting perspectives.

2 Definitions and Problems

In this section, we set up the basic definitions for our work. First, string queries are defined, emphasising string genericity. The syntax $FO(\bullet)$ and the semantics $[\![FO(\bullet)]\!]_{nat}$ are given. We then briefly discuss the difficulty of deciding whether a formula represents a string query. This difficulty is a strong motivation for our "bottom-up" approach to restricting the syntax or semantics for $FO(\bullet)$ (i.e., we shall begin with very simple syntax/semantics that we enrich little by little).

Throughout this paper we assume that a fixed finite alphabet Σ is given.

2.1 Queries and String Genericity

In our study of query languages for string databases, we first need to fix a definition of query. Although several definitions are possible, we choose in this paper one inspired from the traditional one in the relational model (see e.g. [1] for a definition and discussion of relational genericity). We adapt the traditional definition to string databases. Well-typedness and computability are essentially the same as in the relational case. String genericity is new. The idea of string genericity is to identify mappings that differ in only renaming of string symbols, i.e., letters of the fixed alphabet Σ.

A *relation* r of arity k in our model is a finite subset of the k-fold Cartesian product of Σ^*, the set of all finite strings over Σ, with itself. The arity of r, denoted $\alpha(r)$, is defined to be k. A *string database instance* I is a finite sequence of relations $(r_1, \dots, r_n)$. The *schema* of I is $(\alpha(r_1), \dots, \alpha(r_n))$.

We consider mappings from string databases to string relations. We'll sometimes call them also *string mappings* to emphasise the context. *Well-typedness* of a mapping simply says that the input and output schemas are fixed. Let us recall also computability [1], which immediately applies string mappings. A mapping h is *computable* if there exists a Turing machine M, such that for each instance I, given I encoded on the tape, M does the following. If $h(I)$ is defined, M computes the encoding of $h(I)$, writes it on the tape and stops. Otherwise M doesn't stop.

We now turn to the specificity of string mappings. First, recall that a mapping is *relational generic* if it is invariant under permutations of the domain (the domain being Σ^*). We will require in addition that permutations preserve the structure of strings. To this end we say that a permutation ρ of Σ^* is a *string morphism*, if for all u and v in Σ^*, $\rho(u.v) = \rho(u).\rho(v)$. In other words, symbols of Σ are "repainted" without changing the way they combine within strings. Now, a mapping h from string databases to string relations is *string generic* if for each permutation ρ of Σ^* which also is a string morphism, we have that for all instances I,

$$\rho(h(I)) = h(\rho(I)).$$

The concept of C-genericity for any finite subset C of Σ^* is defined as usual. C-genericity will be used to allow constants in query expressions.[1]

As stated in below, a relational generic mapping is also string generic, but some string generic mappings are not relational generic. This phenomenon is due to the fact that string isomorphisms of Σ^* are simply particular permutations of Σ^*. For instance, the mapping $R(u) \mapsto R(u.u)$ is string generic but not relational generic.

Fact 1 *Every relational generic mapping is string generic, but the converse does not hold.*

We now formally define string queries.

Definition 1 Let $(a_1, \ldots, a_k)$ be a database schema and b an arity. A *string query* of type $(a_1, \ldots, a_k) \to b$ is a partial string generic and computable mapping from the set of all database instances over $(a_1, \ldots, a_k)$ to the set of all relations of arity b.

2.2 A Family of Query Languages for the $FO(\bullet)$ Syntax

In this section, we assume classical notions and vocabulary (see e.g. the definition of FO in [1]) and focus on our new string primitive. The syntax $FO(\bullet)$ and a parameterised semantics $[\![FO(\bullet)]\!]_{\mathbf{d}}$ are presented. A difference with classical first order logic is that valuations and quantifiers only range over the domain $\mathbf{d}$ considered, where $\mathbf{d} \subseteq \Sigma^*$.

Syntax. We introduce here the syntax $FO(\bullet)$. It is an extension of first order predicate logic (FO) with a natural concatenation operator for strings, the *dot* operator, denoted $\bullet$. Terms are variables or constants from Σ^*, or of the form $t_1 \bullet t_2$, where t_1, t_2 are themselves terms. Formulas are built as usual from relation symbols and equality between terms, to give atoms; and inductively with the operators $\wedge, \vee, \neg, \exists, \forall$.

Semantics. Let be given terms t_1, t_2, an instance I and a valuation v for the variables appearing in t_1, t_2, i.e., a mapping from these variables to Σ^*. The *interpretation* of $t_1 \bullet t_2$ under I and v, denoted $I(t_1 \bullet t_2)$, is $I(t_1).I(t_2)$, where, in this expression, "." denotes the usual semantic concatenation of strings (associative, with neutral ϵ); and as usual $I(x) = v(x)$ for a variable x, and $I(u) = u$

[1] A way to define string genericity that might be considered more natural could be to define permutations ρ over Σ (instead of Σ^*), and then to extend ρ to strings in Σ^* and database instances in the straightforward way; this would avoid defining string morphisms.

for $u \in \Sigma^*$. Satisfaction of a formula φ of $FO(\bullet)$ is then inductively defined. Atoms and connectives $\wedge, \vee, \neg$ are as usual. The semantics of quantifiers $\exists, \forall$ is obtained by making variables range over some given domain $\mathbf{d} \subseteq \Sigma^*$ (see e.g. the definition of relativized interpretations in [1]).

Definition 2 Let $\varphi(\bar{x})$ be an $FO(\bullet)$ formula with $\bar{x}$ being a vector of its free variables. Given $\mathbf{d} \subseteq \Sigma^*$, the *mapping expressed by* φ *under the semantics* $\mathbf{d}$ is defined as

$$[\![\varphi]\!]_{\mathbf{d}}(I) = \{v(\bar{x}) : v \text{ is a valuation of } \bar{x} \text{ making } \varphi(\bar{x}) \text{ true in } I\},$$

where both v and the quantifiers of φ range over $\mathbf{d}$.

The *set of all mappings* expressed by query expressions in $FO(\bullet)$ under the semantics $\mathbf{d}$ is denoted $[\![FO(\bullet)]\!]_{\mathbf{d}}$.

At this point, we have in hand a rich and simple definition of a family of languages $[\![FO(\bullet)]\!]_{\mathbf{d}}$ based on the syntax $FO(\bullet)$ and parameterised by the domain $\mathbf{d}$. Our search for admissible semantic restrictions will go along the line of carefully defining more and more powerful domains $\mathbf{d}$.

2.3 Where and how to look for the right language—Issues and directions

In this section, we present our approach to a search for an admissible query languages for string databases.

Choosing a query language among ours now amounts to fixing the ranging domain $\mathbf{d}$. The natural choice for $\mathbf{d}$ would be to take the whole Σ^*. This is called the *natural semantics*, and defines a a query language $[\![FO(\bullet)]\!]_{nat}$ that we will consider now.

We want to write formulas φ in $FO(\bullet)$ such that the corresponding mapping $[\![\varphi]\!]_{nat}$ is a string query. The following fact is straightforward.

Fact 2 *There is an $FO(\bullet)$ formula φ such that $[\![\varphi]\!]_{nat}$ is not a string query.*

We are thus faced with the task of determining those formulas that express string queries (under the natural semantics). However, as stated below, a direct approach is doomed to fail.

Fact 3 *Given a formula φ in $FO(\bullet)$, it is undecidable whether $[\![\varphi]\!]_{nat}$ is a string query.*

Our purpose then becomes finding a syntactic fragment of $FO(\bullet)$ that captures *all and only* string queries under natural semantics. Our language `StriQueL` is a first step in this direction (see Section 3.3), but, due to its pragmatic low-complexity nature, it obviously does not express all string queries. The next step, beyond the scope of this paper, would be to design such a sound and complete language, or show that the class of string queries does not have an effective syntax.

As a starting point, one might look to the results of Makanin [8, 23], from which it follows that the satisfiability problem for *conjunctive* $FO(\bullet)$ queries under natural semantics is decidable. However it is not clear if Makanin's techniques can be extended to show that a conjunctive $FO(\bullet)$ formula expresses a string query.

3 Restricting the Language

3.1 Semantic restrictions

In this section, several progressively richer semantics, that is, semantics that allow expressing more and more string queries, are introduced. The expressivity of the query languages they generate are compared.

In the traditional relational setting, there are known ways to restrict the FO-language to only generate relational queries. The queries can then be evaluated using the so called active domain semantics (see e.g. [1]). This is due to the genericity property which enforces the output of queries to be within the active domain. In our setting string genericity allows to construct strings that are outside the active domain and an $FO(\bullet)$ formula can generate a string query even if the output is not included in the active domain. For instance take $\varphi(x) \equiv \exists y, z\,[(x = y \bullet z) \wedge R(y) \wedge R(z)]$, where the output will consist of concatenations of strings from the input instance. We therefore have to develop new techniques adapted to the string setting.

Before giving the formal definitions below, let us first give a flavour of the issue. The "bottom possibility," i.e., sticking to the relational concepts, consists in making the quantifiers range over the active domain, *adom*, (the strings in the input instances, but not their substrings). This use of the classical database notion of the active domain in the context of string queries appears in [15]. Unfortunately, extracting the square root of some strings in the input instance, as in e.g. $\varphi(x) \equiv R(x \bullet x)$, is possible only if the string x is itself in the input instance. Taking the extended active domain, *eadom*, (i.e., considering also substrings of string in the input instance) does the trick. The domain *eadom* was introduced in [24]. However, this domain does not allow to build up new strings using for instance the query $\varphi(x) \equiv \exists y\,[R(y) \wedge x = y \bullet y]$. If we in addition to substrings allow k concatenations of strings in the input instance we can handle the previous query. The corresponding domain is called $eadom^k$. This domain was not considered in previous works; however it can be seen as a combination of *eadom* and of the $adom^k$ used in [15].

The next step would be not bound the construction of new strings by some constant k. This would make the domain, and thus potentially the output, infinite. Consider for example the query $\varphi(x) \equiv \exists y, z\,[R(y) \wedge x = y \bullet z]$. Hence there is a need of a bound of some kind. But this bound could depend on the instance, as opposed to the query. Such a language is considered in Section 4.

In spite of its restrictions, used in an adequate manner, $eadom^k$ yields an appealing and useful language, as we shall see below.

We now proceed to the formal definitions. Given an instance I and a formula φ in $FO(\bullet)$, the *active domain of* φ *and* I, denoted *adom(*φ*,I)*, is the set of all strings occurring in φ or in the columns of the relations in I. The *extended active domain* of φ and I, denoted *eadom(*φ*,I)*, is *adom(*φ*,I)* closed under substrings. For each strictly positive integer k, the set of strings obtained by concatenating at most k strings from *eadom(*φ*, I)* gives us *eadom*$(\varphi, I)^k$.

We now consider the three above possibilities for the ranging domain **d**. We thus get three different string query languages, namely $[\![FO(\bullet)]\!]_{adom}$, $[\![FO(\bullet)]\!]_{eadom}$, and $[\![FO(\bullet)]\!]_{eadom^k}$. This gives us the following pleasant state of affairs.

Proposition 1 *For any* $FO(\bullet)$ *formula* φ*, and any* $k \geq 2$*,* $[\![\varphi]\!]_{adom}$*,* $[\![\varphi]\!]_{eadom}$*, and* $[\![\varphi]\!]_{eadom^k}$ *are string queries.*[2]

The issue in the next result (and again in Section 5), is whether enriching the semantics, and then restricting to mappings whose input/output is in the active domain, yields new string queries. In the following case it indeed does give more power.

Proposition 2 *Let* $[\![FO(\bullet)]\!]_{eadom} \cap$ adom *be the subset of those mappings in* $[\![FO(\bullet)]\!]_{eadom}$ *where the output contains only strings in* adom*. Then* $[\![FO(\bullet)]\!]_{adom}$ *is a proper subset of* $[\![FO(\bullet)]\!]_{eadom} \cap$ adom*.*

Crux. The proof uses the following lemma which is false for string queries in $[\![FO(\bullet)]\!]_{eadom}$. The lemma is proved using techniques from [3].

Lemma 1. On instances with only one element, Boolean $[\![FO(\bullet)]\!]_{adom}$ queries are constant.

The next results says that allowing concatenations in the domain strictly increases the expressive power of the corresponding query language.

Proposition 3 $[\![FO(\bullet)]\!]_{eadom^k}$ *is a proper subset of* $[\![FO(\bullet)]\!]_{eadom^{k+1}}$*, for each* $k \geq 1$.

Now comparing $[\![FO(\bullet)]\!]_{eadom}$ to $[\![FO(\bullet)]\!]_{eadom^k} \cap$ eadom, the latter equaling $[\![FO(\bullet)]\!]_{eadom^k}$ restricted to mappings that have output only containing strings from $[\![FO(\bullet)]\!]_{eadom}$, it turns out that $[\![FO(\bullet)]\!]_{eadom^k} \cap$ eadom is a succinct version of $[\![FO(\bullet)]\!]_{eadom}$.

Proposition 4 *For each* $FO(\bullet)$ *formula* φ *with* m *free variables* $\{x_1, \ldots, x_m\}$ *there exists a an* $FO(\bullet)$ *formula* ψ *with* $k \times m$ *free variables such that for all instances* I*, we have* $[\![\varphi]\!]_{eadom^k}(I) = [\![\psi]\!]_{eadom}(I)$*.*

As a consequence $[\![FO(\bullet)]\!]_{eadom}$ and $[\![FO(\bullet)]\!]_{eadom^k} \cap$ eadom has the same expressive power.

Proposition 5 $[\![FO(\bullet)]\!]_{eadom} = [\![FO(\bullet)]\!]_{eadom^k} \cap$ eadom*.*

[2] Note that $[\![FO(\bullet)]\!]_{eadom^1} = [\![FO(\bullet)]\!]_{eadom}$

3.2 Syntactic restrictions

We now define *range-restricted formulas* by restricting the syntax of the language, and evaluating queries under the natural semantics. Intuitively, the idea of range-restriction here, is to carefully track, throughout a formula, whether each variable will take its values among substrings of the instance (*eadom*), or a finite number of concatenations of them.

We use as a basis the method given in [1], Algorithm 5.4.3. The relational algorithm is extended by adding the following possibilities for φ.

Definition 3 *Let φ be a $FO(\bullet)$ formula. Then $rr(\varphi)$, the set of range-restricted variables of φ, is defined as follows:*

If φ is of the form $R(t_1, \ldots, t_n)$ then $rr(\varphi)$ = the set of all variables appearing in $t_1, \ldots, t_n$.
If φ is of the form $x = u$ or $u = x$, where $u \in \Sigma^*$ then $rr(\varphi) = \{x\}$.
If φ is of the form $u = v_1 \bullet x_1 \bullet \cdots \bullet v_n \bullet x_n \bullet v_{n+1}$, where u and the v_i's are in Σ^* (possibly ϵ) then $rr(\varphi) = \{x_1, \ldots, x_n\}$.
If φ is of the form $\varphi_1 \wedge x = v_1 \bullet x_1 \bullet \cdots \bullet v_n \bullet x_n \bullet v_{n+1}$ then $rr(\varphi) =$
if $x \in rr(\varphi_1)$ then $rr(\varphi_1) \cup \{x_1, \ldots, x_n\}$;
if all x_i's are in $rr(\varphi_1)$, then $rr(\varphi_1) \cup \{x\}$;
otherwise $rr(\varphi_1)$.
Negation and existential quantification are as in [1].

A formula φ is said to be *range-restricted* if $rr(\varphi)$ equals the set of free variables in φ. The set of all range-restricted $FO(\bullet)$ formulas is denoted $FO_{rr}(\bullet)$.

Proposition 6 *For any $FO_{rr}(\bullet)$ formula φ, the mapping $[\![\varphi]\!]_{nat}$ is a string query.*

Crux. The idea is given in the proof of Theorem 7 below.

3.3 StriQueL, the String Query Language—Expressivity and Complexity

Theorem 7 $FO_{rr}(\bullet) = \bigcup_{k \geq 1} [\![FO(\bullet)]\!]_{eadom^k}$

Crux. Given a formula φ in $FO_{rr}(.)$ we compute k with the same algorithm as $rr(\varphi)$ in the following manner.

If φ is of the form $R(x)$ then $k_x = 1$.
If φ is of the form $x = u$ $(u \in \Sigma^*)$ then $k_x = 1$.
If φ is of the form $x = y \bullet u$ then $k_x = k_y + 1$.
If φ is of the form $x = y_1 \bullet y_2$ then $k_x = 2 \times max(k_{y_1}, k_{y_2})$.

On the basis of the robustness emphasised by this equivalence, and on its comfort, we take $FO_{rr}(\bullet)$ as being our string query language: `StriQueL`.

Evidently the `StriQueL` language does not express all string queries, just as `SQL` does not express all relational queries.

Theorem 8 *There is a string query outside* $[\![FO_{rr}(\bullet)]\!]_{nat}$.

Crux. The idea is to find a $[\![FO(\bullet)]\!]_{nat}$ formula φ that computes the concatenation of all the strings in some input instance I. If this is the case, φ cannot be range-restricted, because from Proposition 4 and Theorem 7 we know that the output of range restricted formulas are at most k concatenations of strings in I, for a given k independent of I.

We slightly adapt now the usual relational definition of data complexity for strings. Given a instance I, its size $|I|$ corresponds to the sum of the lengths of all the strings that occur in I. Complexity measures are now given in terms of $|I|$.

Theorem 9 *Let* φ *be an* $FO(\bullet)$ *formula and* I *an instance. Then* $[\![\varphi]\!]_{adom}(I)$, $[\![\varphi]\!]_{eadom}(I)$, *and* $[\![\varphi]\!]_{eadom^k}(I)$ *are computable in logarithmic space.*

Crux. The idea of the proof is more intricate than in the relational case, although it is based on the same principle. For $[\![\varphi]\!]_{adom}(I)$ the proof goes as in the usual setting. For $[\![\varphi]\!]_{eadom}(I)$ the crux is that a string in *eadom* can be described using two pointers (one for the beginning, one for the end) on the input instance, it is therefore easy to adapt the proof of $[\![\varphi]\!]_{adom}(I)$ to this case. The same holds for $[\![\varphi]\!]_{eadom^k}(I)$, with $2k$ pointers.

4 Beyond constant bounds

`StriQueL`, our language that was presented in previous section is, we believe, well-suited to "everyday" database programming and querying by end-users. However, some more advanced applications and/or programmers may want more powerful languages. For all semantics in the previous section, the number of concatenations in the creation of new strings is bounded by a constant independent of the input. (Nevertheless, in our StriQueL this constant depends on the query, and is not only arbitrarily fixed for the whole language as in, e.g., $[\![FO(\bullet)]\!]_{eadom^k}$.) It is clear that this constraint should be relaxed for certain situations.

Several possibilities arise thus for the domain, for instance the following: The domain could be $eadom^n$ (or more generally $(\Sigma|_I)^n \subseteq \Sigma^n$), for some n depending on I, e.g. $n = |I|$, or $n = max\{|u| \ : \ u \ \in \ I\}$, or any arbitrary total function, or family of functions (for instance, $n = |I|$ being fixed, $\bigcup_{p \text{ polynomial}} \Sigma^{p(n)}$). One of the more naive solutions yielding this might be for instance the following. Given I and $n = |I|$, take for domain $eadom^n$.

In this section, we provide a more powerful language, again a variant of $[\![FO(\bullet)]\!]_{nat}$. This language comes with a constructive semantics, i.e., an algorithm to compute the output of a query.

As we discussed above, such languages may range from ones capturing all string queries, down to *while/fixpoint*-like ones, or even simpler ones. We follow here again our "bottom-up" exploration and provide a rather simple one, though much more powerful than our previous `StriQueL`. Moreover, to explore a different direction than in previous sections, we introduce a slight variant in our semantics. We believe this variant also to be very promising for future work.

The language $[\![FO^{\#}(\bullet)]\!]_{nat}$. Until now, we proceeded as follows. Given the alphabet Σ, the language $[\![FO(\bullet)]\!]_{nat}$ over Σ was formally defined, allowing constants from Σ in the formulas (and in their inputs), and with quantifiers ranging over Σ^*. According to its definition, this language was given as input instances over Σ^*. Let now some new symbol not in Σ, say #, be given. Consider $FO(\bullet)$ over $\Sigma \cup \{\#\}$, but now, the input of a formula is an instance over Σ^* only, that is, an instance in which # does *not* appear. We denote this language $[\![FO^{\#}(\bullet)]\!]_{nat}$.

The language $FO_\Gamma(\bullet)$. What we shall do now, is to define a simple syntactic extension of $FO_{rr}(\bullet)$ (expressing a subset of $[\![FO^{\#}(\bullet)]\!]_{nat}$), in which we allow particular subformulas having the property that a few designated quantified variables are allowed to range *arbitrarily* over $(\Sigma \cup \{\#\})^*$. We show however (in the same spirit as for $FO_{rr}(\bullet)$), based on the particular structure of these subformulas, that, given I, only a finite number of valuations can satisfy the formula φ defining the query, thus yielding a finite evaluation and a finite output.

To simplify the presentation of the syntax, we use a "macro" that we call Γ. More precisely, a formula in the language may now contain as a subformula an expression $\Gamma(\cdots)$. These subformulas are added to $FO_{rr}(\bullet)$ to yield the language $FO_\Gamma(\bullet)$. Note that, again, as for $FO_{rr}(\bullet)$, the formal semantics is simply that of $[\![FO^{\#}(\bullet)]\!]_{nat}$ (i.e., with the full domain $(\Sigma \cup \{\#\})^*$). Formally, the $\Gamma(\cdots)$ expressions are replaced ("expanded") by their corresponding $FO^{\#}(\bullet)$ subformula, so that strictly speaking the syntax of $FO_\Gamma(\bullet)$ is a fragment of $FO^{\#}(\bullet)$.

Before introducing the formal details of the Γ operator, let us present its principles intuitively. In an expression $\Gamma(x, y, \mathit{one\_step})$, x and y are strings, and *one_step* is an ordinary first order formula. This formula *one_step* defines a relationship between x and y that will have to be satisfied. With these three arguments, Γ will be doing the following.

First, a string $x\#s_1\#s_2\#\ldots\#s_n\#y$ is built (call it z in the following). Then, it is verified whether $\mathit{one\_step}(x, s_1)$ holds, then $\mathit{one\_step}(s_1, s_2), \ldots$, until $\mathit{one\_step}(s_n, y)$. If all are true, then the formula $\Gamma(x, y, \mathit{one\_step})$ is true.

Now, how is it that the number of such "big" strings z, and the evaluation of the $\mathit{one\_step}(s_i, s_{i+1})$'s, are both bounded? Intuitively, to achieve this, Γ generates only s_i's of strictly increasing size (thus bounded by $|y|$), and forces also *one_step* to be finitely evaluable (in fact, it has even to be quantifier-free).

Strictly speaking, this mechanism is encoded in a big "parameterized" first order formula, that we call Γ. By "parameterized", we simply mean that some subformula in $\Gamma(\cdots)$ will be the "parameter" $\mathit{one\_step}(\cdots)$. (This kind of construction is analogous to axiom schemata in axiomatic logic.) For the sake of the presentation, we choose a very simple variant of Γ. Richer variants can easily be defined.

We define a formula schema $\Gamma(x, y, one\_step)$, where one_step has two free variables, and has to be built using only conjunctions and disjunctions of equality atoms (no negations, and no quantifiers),[3] and the free variables of Γ are x, y. The variable x represents the "initial step" and y the "final step" in Γ. Examples of how to use Γ to build a $FO_\Gamma(\bullet)$ formula are given below.

$$\begin{aligned}
&\Gamma(x, y, one_step) \equiv \\
&\ no_\#(x) \wedge no_\#(y) \wedge \\
&\ \exists z, u : (z = \# \bullet x \bullet \# \bullet u \bullet \# \bullet y \bullet \#) \wedge \\
&\ \forall z_1, v, w : (substring(z_1, z) \wedge z_1 = \# \bullet v \bullet \# \bullet w \bullet \# \ \wedge \ no_\#(v) \wedge no_\#(w)) \rightarrow \\
&\ (\exists u_1, u_2 \ (w = u_1 \bullet v \bullet u_2 \ \wedge \ (u_1 \neq \epsilon \vee u_2 \neq \epsilon)) \ \wedge \ one_step(v, w))
\end{aligned}$$

Let denote this $FO^{\#}(\bullet)$ formula φ_Γ, and call it the "expansion" of our $\Gamma(\cdots)$ subformula shorthand. In the expansion the predicate $no\_\#$ checks easily that its argument does not feature the # symbol, and *substring* is the obvious abbreviation. The beginning of the last line says that at each step, each v is a strict substring of w.

We extend the definition of range restriction to Γ, by adding one case to Definition 3 in Section 3.2. Notice that, with respect to range restriction, Γ is a "propagator" of range restriction, not a "generator". That is, it does not bound variables by itself, but it transmits bounds from y to x.

If $\varphi = \varphi_1 \wedge \Gamma(x, y, one\_step)$ then
if $y \notin rr(\varphi_1)$, then $rr(\varphi) = rr(\varphi_1)$;
if $y \in rr(\varphi_1)$, then $rr(\varphi) = rr(\varphi_1) \cup \{x\}$.

The language obtained is called $FO_\Gamma(\bullet)$. Its syntax is the syntax of $FO(\bullet)$ extended with the Γ operator, and its semantics is that of $[\![FO^{\#}(\bullet)]\!]_{nat}$, when Γ is expanded.[4]

We give below the queries $same\_length$ and $parity$ in $FO_\Gamma(\bullet)$. Note that strictly speaking, the symbol ¢ has to be simulated using only #, making the real formula heavier.[5]

$$\begin{aligned}
same_length(s_1, s_2) \equiv\ & x = car(s_1) \bullet ¢ \bullet car(s_2) \ \wedge \ y = s_1 \bullet ¢ \bullet s_2 \ \wedge \ \Gamma(x, y, \psi), \\
\text{where } \ \psi(v, w) \equiv\ & v = u_1 \bullet ¢ \bullet u_2 \wedge \ w = u_3 \bullet ¢ \bullet u_4 \ \wedge \\
& (\bigvee_{a \in \Sigma} u_3 = u_1 \bullet a) \ \wedge \ (\bigvee_{a \in \Sigma} u_4 = u_2 \bullet a),
\end{aligned}$$

[3] We could also for instance define it such that all new variables introduced by one_step be forced to be substrings of x or y. Also, one_step could be allowed to be any safe formula, e.g. range-restricted.

[4] To be more precise, a formula of $FO_\Gamma(\bullet)$ is one of $FO^{\#}(\bullet)$: all $\Gamma(\ldots)$ subformulas used above as shorthands are replaced by the actual $\varphi_{\Gamma(\ldots)}$ subformula. For instance our notation $\exists x, y : R(x, y) \wedge \Gamma(x, y, \psi)$ denotes the $FO^{\#}(\bullet)$ formula $\exists x, y : R(x, y) \wedge \varphi_{\Gamma(x,y,\psi)}$.

[5] This is done essentially by using two # symbols concatenated.

and the function *car* returns the first letter of its argument.

$$parity(x) \;\equiv\; \Gamma(\epsilon, x, \psi), \text{ where } \psi(v,w) \;\equiv \bigvee_{\{a,b\}\subseteq\Sigma} w = v \bullet a \bullet b.$$

The following theorem characterises $FO_\Gamma(\bullet)$.

Theorem 10 *Let φ be a formula in $FO_\Gamma(\bullet)$. Then*

1. *$[\![\varphi]\!]_{nat}$ is a string query.*
2. *For any instance I, $[\![\varphi]\!]_{nat}(I)$ can be evaluated in space polynomial in the size of the longest string in the extended active domain $eadom(I,\varphi)$.*

Crux. We briefly consider point 1. Let s be the longest string in $eadom(I,\varphi)$. Let φ be of the form $\psi(\ldots,y,\ldots) \wedge \Gamma(x,y,one\_step)$, where $\psi(\ldots,y,\ldots)$ is an $FO_{rr}(\bullet)$ formula[6]. As in the proof of Theorem 8 we will get a bound, say k, on the variable y. Then, we have that $[\![\varphi]\!]_{nat}(I) \subseteq eadom^n$, where $n \leq 1 + 2 + \ldots + |s|^k$. This is because of the following: The only "dangerous" variables are the quantified ones in the expansion of Γ. Now y is the "output" of Γ, i.e., the last word concatenated in the existentially quantified long derivation $\exists z : z = x \bullet \# \bullet s_1 \bullet \# \bullet s_2 \bullet \# \bullet \ldots \bullet \# \bullet y$. As $|s_1| < |s_2| < \ldots < |y|$ (because of the constraint imposed by Γ on *one_step*), we get the maximal size of n for any string in the range of $[\![\varphi]\!]_{nat}(I)$. This shows that if y is bound to $eadom^k$ for some constant k depending only on φ, then Γ is safe because quantifiers need only to range over $eadom^n$.

Note that, once replaced by its expansion, Γ may not be range restricted. However, we just showed above its actual safety (provided y is itself range restricted). In other words, we mix here a syntactic means (range restriction), and a semantically safe operator (Γ).

5 Expressive power—`StriQueL` *vs* others

In this section, we compare our languages with formalisms outside string query languages. We first compare the pure relational fragment of our languages with pure relational FO. We end this section by a note on the difficulty of the comparison of $FO(\bullet)$ languages with formal languages. We however believe that such comparisons are a promising direction of research.

String genericity *vs* relational genericity. The issue here is to determine (1) whether our string languages express more queries than relational FO, and (2) whether we can compute purely relational generic queries that are not in relational FO (recall Definition 1 and definitions of both genericity concepts in Section 2.1). The answer to question (1) is yes: we already saw in Section 2.1 that there are string generic mappings that are not relational generic. In addition

[6] Note that y might be quantified in $\psi(\ldots,y,\ldots)$

we now have that such mappings can be expressed in our simplest semantics, namely the active domain semantics. The answer to question (2) is no. This last result has to be put in contrast with the case of $FO(<)$, for which the order *did* bring new generic relational queries [16, 1]. We denote by $\Downarrow M$ the subset of mappings in M that are relational generic.

Proposition 11 1. There is a formula φ in $FO(\bullet)$, such that $[\![FO(\bullet)]\!]_{adom}$ is not a relational generic query.
2. $[\![FO]\!]_{adom} = \Downarrow[\![FO(\bullet)]\!]_{adom} = \Downarrow[\![FO(\bullet)]\!]_{eadom^k}$

Crux. Given $FO(\bullet)$ formula φ such that $[\![\varphi]\!]_{adom}$ is a relational generic query we construct a FO formula ψ, such that $[\![\psi]\!]_{adom}(I) = [\![\varphi]\!]_{adom}(I)$, for any instance I. A case analysis shows that ψ and φ agree on instances where the active domain is composed of words of size 1. From this and the fact that φ is relational generic, it follows that they must also agree on all models. This proof is inspired by techniques from [3].

On the relationship with formal languages. A string query is called *context-free* if there exists a context-free language $L \subseteq \Sigma^*$ such that the following holds: When given as input a unary relation r, the query returns $r \cap L$. The same definition extends to other classes of languages. Here we only give a conjecture grouping several representative situations.

Conjecture 1 *1. The query parity(x) is rational but not in $[\![FO(\bullet)]\!]_{nat}$.*
2. The query $\varphi(x) \equiv x = a^i \bullet b^j \bullet c^k$, where a and b are in Σ and $i \neq j$ or $j \neq k$, is context-free but not in $[\![FO(\bullet)]\!]_{nat}$.

Note that if we are allowed to use additional "marker-symbols," as in the query language $[\![FO^{\#}(\bullet)]\!]_{nat}$, then computing the parity query is easy. These examples suggest that putting in correspondence fragments of $[\![FO(\bullet)]\!]_{nat}$ and classes of formal languages may not be easy.

6 Related works

Strings have been studied in databases, logic, formal language theory and computational complexity. In this section, we survey the main works in relationship with ours. We begin with databases, then turn to the others. Some definitions we used were introduced by other authors, as was pointed out in the text above. All the theorems that were explicitly stated above are novel.

Databases. Present-day database query languages offer little support for string relations. For example, the *Sequence Retrieving System (SRS)* [10], which has gained popularity in molecular biology, does not allow the database administrator to draw links from one preformatted data file to another [11], but only on its atomic non-sequence fields. Because the majority of current relational database management systems do not support application-specific data types such as sequences, some molecular biology database designers have begun to move towards object-oriented database technology [12]. Another solution, strongly advocated in for instance [7], is to introduce such types as relational domains, as we have done. In any case, the string handling concepts introduced in this work are in no way specific to the relational model; indeed, they are being applied for querying sequences of complex objects from object-oriented databases as well [2].

Our language differs from such languages as SEQ [32, 33], where sequence elements and their underlying order type are distinct domains, in that queries in our application areas are more oriented towards parsing-type tasks than for example computing moving averages on temporally sequenced data [32].

One of the early declarative database languages for string databases was Richardsons [27]. This language used the modalities of temporal logic for expressing properties of strings. Each successive position in a string is seen to be the time wise "next" instance of that string. The temporal modalities lend themselves naturally to reasoning about strings. There are however well-known restrictions [38] to basic temporal logic. In particular, no recursion or iteration is achieved.

The works closer to ours begin with Wang [15], and Ginsburg and Wang [14]. They extended the relational calculus with interpreted *regular transducer functions*. Such a function has n input strings, and the result is a regular finite state transduction on the inputs. The resulting language captures all r. e. sets (and sets in the arithmetical hierarchy) [15]. The drastic expressive power of the language results from the use of powerful interpreted functions. Of course, the transducer mappings of Wang and Ginsburg are intended to serve as semantic programming primitives, but using them may turn rather heavy from the point of view of programming comfort.

Later, Grahne, Nykänen, and Ukkonen [17] proposed a string-extension of relational calculus based on modal logic. Their language has basically the same expressive power as that of Ginsburg and Wang. It has however the advantage that the string primitives do not have to be "plugged in," they have a declarative semantics that can be coupled with relational structures. The string programming primitive is a declarative specification of a multitape finite state automaton. The multitape automaton have proven to be useful in pattern matching problems [13, 17]. The language has also been implemented [19]. A detailed study of safety and evaluability is done in [18].

Still later, Mecca and Bonner [24] used the interpreted functions of Ginsburg and Wang in a datalog-like language. Since Prolog with one function symbol already yields all r. e. sets [21], the Mecca-Bonner language has full Turing expressivity. In a series of sophisticated constructions [24] Mecca and Bonner restrict the language to capture sets in for instance PTIME, and hyper-exponential time. In the Mecca-Bonner language the programming primitive for recursion is datalog rules. String operations are expressed through the transducer functions, and if the programmer wants to iterate or recurse over strings, she has to mix the datalog rules and transducer functions, and stay within the syntactic restrictions given by Bonner and Mecca. W.r.t. expressivity, Bonner and Mecca consider computing string functions, i.e., mappings from a single relation containing a single string, and returning a single string.

In brief, compared to [14, 17, 24], we go back to the spirit of the "founding fathers". For the string programming primitive, we return to using only $\bullet$, simpler than all previous string programming primitives. And for the "host language", we return to FO, simpler than [17, 24]. We study in depth a low-complexity language, whereas the abovementioned authors explore very powerful ones (except some PTIME fragment with transducers in [24]).

Logic, formal languages and computational complexity. The main direction in logic seems to have been in showing undecidability of various fragments of the theories of concatenation. In the case of first order, this amounts to variants of our $[\![FO(\bullet)]\!]_{nat}$. It initiated with Quine [26] (see also [36]). Makanin [23] shows the decidability of the existential fragment of the theory of concatenation, with consequences on word equations. We used the recent presentation in [8], which contain other references; however, nothing seems to have been done w.r.t. computing the set of all solutions [30] (not only a yes/no answer), which is the output of our queries.

From a pure formal languages point of view, Salomaa [29], Chap. III, presents an apparently isolated result around characterising type-0 (i.e., recursively enumerable) and type-1 languages.

In the classical marriage between strings and formal language theory [28, 9], a model of a formula is a single string, not a whole collection of relations as in our study. As a consequence, issues, techniques and results are quite different.

In computational complexity, Stockmeyer [34] shows that the polynomial time hierarchy can be characterised in terms of (polynomially bounded) quantification over string relations.

In brief, in logic and related areas, to our knowledge, no study of a low power fragment as ours was undertaken.

7 Conclusion

Querying string databases has essentially focused until now on very powerful languages, i.e., variants of $[\![FO(\bullet)]\!]_{nat}$. Our main point in this paper was the proposal and in-depth study of a low complexity fragment of $[\![FO(\bullet)]\!]_{nat}$. Its simplicity being, we believe, well-suited for "everyday" database querying and programming, we called it `StriQueL`, standing for *String Query Language*.

More precisely, after having defined string queries, given the $FO(\bullet)$ syntax, we considered the general language $[\![FO(\bullet)]\!]_{nat}$ and defined our goal: all string queries. Discouraged by an undecidability result we obtained, we proceeded "bottom-up". Were successively introduced: semantics restrictions of $[\![FO(\bullet)]\!]_{nat}$ (i.e., $[\![FO(\bullet)]\!]_{adom}$, $[\![FO(\bullet)]\!]_{eadom}$, and $[\![FO(\bullet)]\!]_{eadom^k}$); then a syntactic restriction, $FO_{rr}(\bullet)$. We argued that this last one might deserve the name `StriQueL`—String Query Language. Its was shown that its complexity is in logarithmic space, and that it does not express all string queries. To get more expressive power, $FO_\Gamma(\bullet)$ a fragment of $[\![FO^\#(\bullet)]\!]_{nat}$, was introduced. Some comparisons with formal languages and pure relational languages were also proven.

All languages considered (except $[\![FO(\bullet)]\!]_{nat}$ and $[\![FO^\#(\bullet)]\!]_{nat}$) were shown to express only string queries. It remains open whether there is a string query not in $[\![FO(\bullet)]\!]_{nat}$.

Perspectives. First, an efficient operational semantics for `StriQueL` was not considered in this paper: the design of an adequate algebra is definitely relevant.

Another direction will be the design and study of a more powerful variant of the Γ operator. A different direction, more in the "systematic" spirit of our `StriQueL` (or $FO_{rr}(\bullet)$), will be around the following notion of domain independence ($\Sigma^{\leq l}$ represents all string of size at most l):

> A string mapping (defined by φ in the semantics $[\![FO(\bullet)]\!]_{nat}$) is said to be *domain independent* if, for each instance I, there exists a constant l, such that $[\![\varphi]\!]_{nat}(I) = [\![\varphi]\!]_{\Sigma^{\leq l}}(I)$.

Of course, one issue is to compute the constant l. In addition, some aspects of domain independence are rather involved in the relational case [20] so here, because of the nature of string genericity and the infinite domain Σ^*, new techniques have to be developed.

A parallel step will be to define analogous to *fixpoint* and *while* dealing with strings. One of our purposes in doing the present work was to gain a deeper understanding of the notion of string query; we believe these recursive languages can now be addressed.

Another interesting further question is how our results correlate with query languages for lists. Is it possible to apply some of them to lists with arbitrary element type (typically having an infinite domain)? How can query languages for lists be adopted for strings?

To close this perspective section, we believe the most interesting issue at this point is this: Is there an effective syntactic restriction of $[\![FO(\bullet)]\!]_{nat}$ yielding exactly the class of all string queries?

Acknowledgements. The authors whish to thanks Luc Segoufin for fruitful discussions on the subject, and Leonid Libkin and Victor Vianu for comments on an earlier version of this paper.

References

1. S. Abiteboul, R. Hull and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.
2. N. Balkir, E. Sukan, G. Ozsoyoglu and Z. Oszoyoglu. VISUAL : A graphical icon-based query language. In *IEEE International Conferecne on Data Engineering*, 1996.
3. M. Benedikt, L. Libkin. Languages for Relational Databases over Interpreted Structures. In *Proc. ACM Symp. on Principles of Database Systems*, 1997.

4. M. Benedikt, L. Libkin. Safe Constraint Queries. In *Proc. ACM Symp. on Principles of Database Systems*, 1998.
5. E.F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of ACM*, 13:6, 1970
6. J. Collado-Vides. The search for a grammatical theory of gene regulation is formally justified by showing the inadequacy of context-free grammars. *Computer applications in the Biosciences*, 7(3):321-326, 1991.
7. C. Date. *An introduction to database systems.* Addison-Wesley, 1994.
8. V. Diekert, Makanin's algorithm. Available at http://www.lri.fr/~rtaloop/92.html
9. T. Eiter, G. Gottlob, Y. Gurevich. Existential Second-Order Logic over Strings. In *Proc. Symp. on Logic in Computer Science*, 1998.
10. T. Etzold and P. Argos. SRS - an indexing and retrieval tool for flat file data libraries. *Computer applications in the Biosciences*, 9(1):49-57, 1993.
11. T. Etzold and P. Argos. Transforming a set of biological flat file libraries to a fast access network. *Computer applications in the Biosciences*, 9(1):59-64, 1993
12. European Molecular Biologiy Network (EMBNET). *Strategies in Bioinformatics: a european perspective*, 1994.
13. Z. Galil, J.I. Seiferas. Time-Space-Optimal String Matching. *Journal of Computer and System Sciences* 26(3): 280-294, 1983.
14. S. Ginsburg, X.S. Wang. Pattern Matching by Rs-Operations: Toward a Unified Approach to Querying Sequenced Data. In *Proc. ACM Symp. on Principles of Database Systems*, 1992.
15. S. Ginsburg and W. Wang. Regular Sequence Operations and Their Use in Database Queries. *Journal of Computer and System Science* 56(1): 1-26 (1998)
16. Y. Gurevich. in [1], exercice 17.27.
17. G. Grahne, M. Nyknen, E. Ukkonen. Reasoning about Strings in Databases. In *Proc. ACM Symp. on Principles of Database Systems*, 1994.
18. G. Grahne, M. Nykänen. Safety, translation and evaluation of Alignment calculus. *First East-European Symposium on Advances in Database Information Systems (ABDIS'97)*, St-Petersburg, Russia, 1997.
19. G. Grahne, R. Hakli, M. Nykanen and E. Ukkonen. AQL: An alignment based language for querying string databases. *Proc. of the 9TH International Conference on Management of Data (Comad '98)*, Hyderabad, India, December 1998.
20. R. Hull, J. Su. Domain independence and the relational calculus. *Acta informatica*, 31:6, 1994.
21. J.W. Lloyd *Foundations of logic programming.* Springer, 1984.
22. J. Barwise (Ed). *Handbool of mathematical logic.* North-Holland, 1991.
23. G.S. Makanin. The problem of solvability of equations in a free semi-group. english translation in *Math. USSR Sbornik*, 32, 129-198, 1977.
24. G. Mecca, A.J. Bonner. Sequences, Datalog and Transducers. In *Proc. ACM Symp. on Principles of Database Systems*, 1995.
25. C.H. Papadimitriou. *Computational complexity.* Addison-Wesley, Chap. 17, 1994.
26. W. V. Quine. Concatenation as a basis for arithmetic. *Journal of Symbolic Logic*, 11(4):105-114, 1946
27. J. Richardson. Supporting lists in a data model (a timely approach). *Proc. of Intl. Conf. on Very Large Data Bases*, 1992.
28. G. Rozenberg, A. Salomaa (Eds). *Handbook of formal languages.* Springer, 1997.
29. A. Salomaa. *Formal languages*, Chap. 3, Academic Press, 1973.
30. K. Schulz. Personnal communication.
31. D. Searls. String variable grammar: a logic grammar formalism for the biological language of DNA. *Journal of Logic Programming*, pp. 73–102, 1995.

32. P. Seshadri, M. Livny and R. Ramakrishnan. SEQ: a model for sequence databases. *IEEE International Conference on Data Engineering*, pp. 232–239, 1995.
33. P. Seshadri, M. Livny and R. Ramakrishnan. The design and implementation of a sequence database system. *Proc. of Intl. Conf. on Very Large Data Bases*, 1996.
34. L.J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3:1-22, 1977
35. B. Subramanian, T. Leung, S. Vandenberg and S. Zdonik. The AQUA appraoch to querying lists and trees in objsct-oriented databases. *IEEE International Conference on Data Engineering*, pp. 80–89, 1995.
36. R. Treinen. A new method for undecidability proofs of first order theories. *J. Symbolic Computation*, 14:437-457, 1992.
37. J. Ullman. *Principles of database and knowledge-base systems, Volume 1.* Computer Science Press, 1988.
38. P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56:72-99, 1983.