Reasoning about Strings in Databases

Gösta Grahne, Matti Nykänen and Esko Ukkonen
Department of Computer Science
P.O. Box 26, FIN–00014 University of Helsinki, FINLAND
{grahne,mnykanen,ukkonen}@cs.helsinki.fi

Abstract

In order to enable the database programmer to reason about relations over strings of arbitrary length we introduce alignment logic, a modal extension of relational calculus. In addition to relations, a state in the model consists of a two-dimensional array where the strings are aligned on top of each other. The basic modality in the language (a transpose, or "slide") allows for a rearrangment of the alignment, and more complex formulas can be formed using a syntax reminiscent of regular expressions, in addition to the usual connectives and quantifiers. It turns out that the computational counterpart of the string-based portion of the logic is the class of multitape two-way finite state automata, which are devices particularly well suited for the implementation of string matching. A computational counterpart of the full logic is obtained from relational algebra by extending the selection operator into filters based on these multitape machines. Safety of formulas in alignment logic implies that new strings generated from old ones have to be of bounded length. While an undecidable property in general, this boundedness is decidable for an important subclass of formulas. As far as expressive power is concerned, alignment logic includes previous proposals for querying string databases, and gives full Turing computability. The language can be restricted to define exactly regular sets and sets in the polynomial hierarchy.

1 Introduction

In this paper we focus on the problem of strings in databases. Our primary source of motivation is the storage and qualitative processing of genetic information. For instance, the theory of gene regulation explores the combinatorial or grammatical structure inherent in genetic data, as opposed to its statistical properties. This grammatical structure can be quite involved, since gene regulation involves non-context-free dependencies between different parts of a string [CoV91]. Such dependencies should be explicitly expressible, as they constitute knowledge about the family of strings that the current database represents.

For serving as a framework for databases containing stringological information we extend the relational model to include strings over some finite alphabet Σ. A relation of arity k in our model is then a finite subset of the k-fold Cartesian product of Σ^* with itself. In other words, each column in a relation can contain a string of arbitrary length, instead of just a single atomic value.

It is quite clear that a database language operating on string relations should have a pattern-matching ability in order to be able to express queries of the form "list all tuples of relation r, where the second component is of the form $(GC + A)^*$." However, in applications such as the aforementioned gene regulation the language needs to have expressive power beyond regular sets.

In addition to data extraction features, the string language also needs data restructuring constructs. For example, given two unary relations, one might want to concatenate each string from one of the relations with a string from the other relation, as opposed to merely taking the Cartesian product of the two relations. Or for a more involved example, one might want to shuffle two relations (we shall see in Section 2 how to express such a transformation).

How should one go about when building a database language having such features? From the literature we find the following types of proposals. On one hand, like in [PiT86, HeS93] one can add, say to relational algebra, a selection predicate for testing membership in a set specified by for instance a regular expression. This partially solves the data extraction problem but does not support restructuring operations very well. On the other hand, one can add a restructuring operator, such as the transducer mappings of Ginsburg and Wang [GiW92]. This gives good data restructuring abilities, but only rather implicit data extraction features, because (finite-state) transducers generate the regular sets only.

Then there is the avenue of a *declarative* approach. The idea is to design a language for expressing properties of strings. One such proposal can be found in [Ric92], which essentially suggests using the modalities of *temporal logic* for this purpose. Each successive position in a string is seen to be the timewise "next" instance of that string. The temporal modalities lend themselves naturally to reasoning about strings. But as shown by Wolper [Wol83], using only the modalities *next* and *until*, to say that a property holds in every even position of the string is not within the power of the language. Using Wolper's *extended* temporal logic would be a step in the right direction. Still, extended temporal logic cannot express for instance the two-place predicate of equalness between strings, not to mention predicates such as saying that one string is a manifold of the other.

We therefore define a logic in which we can express both properties of individual strings and properties relating strings to each other. The purely relational part of the logic is handled by relational calculus. The string part of the logic is *state based* (as opposed to second or higher order) like e.g. temporal and dynamic logic. A state of the strings is a structure where the strings are aligned on top of each other in a certain way, and a state change is obtained by sliding some of the strings.

The logic is defined in Section 2. In Section 3 we give an algebraic language and show that it has the same expressive power as the logic. The novel construct in the algebraic language is a selection operator based on certain finite state acceptors. In Section 4 we study the safety of formulas in the logic. The main tool for determining safety is the concept of limitation in the string formulas: does the boundedness of some variables imply boundedness of some other variables. We show that the limitation problem is undecidable in general, and decidable for an important subclass of formulas. In Section 5 we use the same subclass of formulas to obtain a characterization of the polynomial hierarchy. The components of our logic that the aforementioned acceptors are a counterpart of are called *string formulas*. The string formulas as such have the same expressive power as the multitape two-way automata. This alone is enough e.g. for string matching [GaS83]. The string formulas together with one projection operator yield the power of full Turing computability.

2 Alignment Logic

In the world of strings we distinguish as state an *alignment* of the strings. The following figure illustrates an alignment of the three strings abc, abb, and $cacd$.

$$\begin{array}{ccccc} & \mathtt{a} & \mathtt{b} & \mathtt{c} & \\ \mathtt{a} & \mathtt{b} & \mathtt{b} & & \\ & \mathtt{c} & \mathtt{a} & \mathtt{c} & \mathtt{d} \end{array}$$

Figure 1: An alignment of three strings

Properties of an alignment are expressed with respect to the vertical window. For instance, in the alignment above the following proposition is true: "window position of the topmost string equals a or the window position of the middle string is different from c." On the other hand, the following proposition is false in that alignment: "the window position of the middle and the bottom string of the alignment are equal."

Let Σ be a henceforth fixed finite alphabet. Formally an alignment is then a partial function $A : \mathbb{N} \times \mathbb{Z} \to \Sigma$, such that for all $i \in \mathbb{N}$, there is a $k \in \mathbb{Z}$ and an $m \in \mathbb{N}$, such that

$$A(i,j) = \begin{cases} a & \text{for some } a \in \Sigma, \\ & \text{if } j \in \{k, \ldots, k+m\} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The *window column* is numbered 0. Thus, if the alignment in Figure 1 represents the three first rows of A, then we have for instance $A(2,-1) = c, A(2,0) = a, A(2,1) = c, A(2,2) = d$, and $A(2,j)$ is undefined elsewhere.

Alignments are connected to each other through state transitions called transposes. A transpose says that certain strings in the alignment should be shifted one position to the left (or to the right). A *left transpose* is a construct of the form $[i_1, \ldots, i_k]_l$, where k and each i_j are in $\mathbb{N}$, and it represents a function on the set of all alignments.

This function is defined as[1]

$$[i_1,\ldots,i_k]_l A(i,j) = \begin{cases} A(i,j+1) \text{if } i \in \{i_1,\ldots,i_k\} \\ A(i,j) \quad \text{otherwise} \end{cases}$$

The *right transposes* are denoted $[i_1,\ldots,i_k]_r$, and are defined similarly to the left ones, except that the rows mentioned in the expression are transposed one position to the right, instead of to the left (e.g. $[3,5]_r A(3,j) = A(3,j-1)$, and $[3,5]_r A(5,j) = A(5,j-1)$, and all the other rows remain unchanged). Below we show some transposes of the alignment in Figure 1.

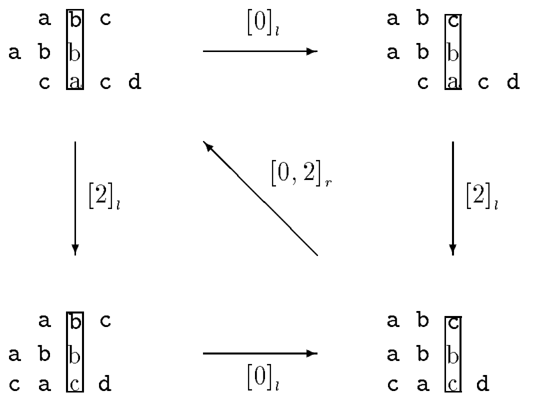


Figure 2: Transposing alignments

The propositions on the window of a particular alignment are expressed through *window formulas*, which are Boolean combinations of *atomic* window formulas of the form $x = \bot$, $x = a$, or $x = y$, where x and y are variables in an countably infinite set V, $\bot$ denotes "undefined," and a is a symbol in Σ.

The variables range over row numbers in alignments, and they are fixed through an *assignment*, which is a function $\theta : V \to \mathbb{N}$. Given a particular alignment A and a window formula ϕ, the definition below tells us when A *satisfies* ϕ *with assignment* θ, in symbolic notation $A \models \phi\theta$.

1. $A \models (x = \bot)\theta$ iff $A(\theta x, 0)$ is undefined.

2. $A \models (x = a)\theta$ iff $A(\theta x, 0) = a$.

3. $A \models (x = y)\theta$ iff $A(\theta x, 0) = A(\theta y, 0)$.

4. $A \models (\phi \wedge \psi)\theta$ iff $A \models \phi\theta$ and $A \models \psi\theta$.

[1] We also require that at least one of $A(i,j)$ and $A(i,j+1)$ is defined, i.e. the rows are never shifted more than one position into the undefined area.

5. $A \models (\neg\phi)\theta$ iff not $A \models \phi\theta$.

For instance, if the three first rows of A are as in Figure 1, and if θ maps x to 0, y to 1, and z to 2, then we have $A \models ((x = a) \vee \neg(y = c))\theta$, and $A \not\models (x = z)\theta$.

Transposes are expressed as indexed modalities in the language, and these modalities can be applied to window formulas. Formally, if ϕ is a window formula, and $\{x_1,\ldots,x_k\} \subset V$, then $[x_1,\ldots,x_k]_l\phi$ and $[x_1,\ldots,x_k]_r\phi$ are *atomic string formulas*. For example, $[x]_l(x = c \wedge y = b)$, $[z]_l(x = c)$, and $[x,z]_r(z = a \vee y = b)$ are all atomic string formulas. The definition of satisfaction for atomic string formulas in alignment A with assignment θ is

6. $A \models [x_1,\ldots,x_k]_l\phi\theta$ iff $[\theta x_1,\ldots,\theta x_k]_l A \models \phi\theta$.

7. $A \models [x_1,\ldots,x_k]_r\phi\theta$ iff $[\theta x_1,\ldots,\theta x_k]_r A \models \phi\theta$.

As an illustration, let A be the top-left alignment, and A' the bottom right alignment in Figure 2, and let θ map x to 0, y to 1, and z to 2. Then we have $A \models ([x]_l(x = c \wedge y = b))\theta$, $A \not\models ([z]_l(x = c))\theta$, and $A' \models ([x,z]_r(z = a \vee y = b))\theta$.

Using the concatenation operation we can compose atomic string formulas. We define a *formula word* to be an expression of the form $\phi_1\phi_2\ldots\phi_k$, where each ϕ_j is an atomic string formula. For instance, $([x,z]_r(z = a \vee y = b))([x]_l(x = c \wedge y = b))$ and $([x,z]_r(z = a \vee y = b))([z]_l(x = c))$ are formula words.

Before we proceed we shall introduce some notation for convenience: Atomic string formulas, like $[x,z]_r(z = a \vee y = b)$, will sometimes be denoted $\tau\alpha$, where in the above case τ stands for $[x,z]_r$, and α stands for $(z = a \vee y = b)$. If θ is an assignment, and $\tau\alpha$ as above, then $\tau\theta$ denotes $[x\theta, z\theta]_r$, and $\alpha\theta$ denotes $(z = a \vee y = b)\theta$.

Now we can denote formula words generically by $\tau_1\alpha_1\tau_2\alpha_2\ldots\tau_k\alpha_k$ and define

8. $A \models (\tau_1\alpha_1\tau_2\alpha_2\ldots\tau_k\alpha_k)\theta$ iff $\tau_1\theta A \models (\alpha_1 \wedge \tau_2\alpha_2\ldots\tau_k\alpha_k)\theta$.

Once again, let A' be the bottom right alignment in Figure 2, and let θ map x to 0, y to 1, and z to 2. Then we have the following: $A' \models (([x,z]_r(z = a \vee y = b)).([x]_l(x = c \wedge y = b)))\theta$, and $A' \not\models (([x,z]_r(z = a \vee y = b)).([z]_l(x = c)))\theta$.

Sets of formula words can be represented by expressions called *string formulas*: all atomic string formulas are string formulas, and if ϕ and ψ are string formulas, then so are $\phi.\psi$, $\phi + \psi$ and $(\phi)^*$.

A string formula can be seen as a regular expression over the alphabet of atomic string formulas. The set of formula words it thus defines is denoted $L(\phi)$. For example, if ϕ is the string formula

$$([x,z]_r(z=a \vee y=b)) \,.\, (([x]_l(x=c \wedge y=b)) + ([z]_l(x=c)))$$

then $L(\phi)$ is

$$\{([x,z]_r(z=a \vee y=b))([x]_l(x=c \wedge y=b)), ([x,z]_r(z=a \vee y=b))([z]_l(x=c))\}.$$

We now define

9. $A \models \phi\theta$, for string formula ϕ, iff there is a formula word $\tau_1\alpha_1\tau_2\alpha_2\ldots\tau_k\alpha_k$ in $L(\phi)$, such that $A \models (\tau_1\alpha_1\tau_2\alpha_2\ldots\tau_k\alpha_k)\theta$.

If ϕ is the string formula given in the preceding example, A' is the bottom right alignment in Figure 2, and θ maps x to 0, y to 1, and z to 2, then the reader can easily verify that indeed $A' \models \phi\theta$. If the fourth row in alignment A' were *abababa*, with the first a positionend in the window, and θ mapped variable u to 3, then we would have $A' \models (([u]_l(u=b)).([u]_l(u=a)))^*\theta$, and $A' \not\models ((([u]_l(u=a)).([u]_l(u=b)))^*\theta$.

A *database db* is a tuple $\langle r_1,\ldots,r_n\rangle$, where each r_i is a finite subset of the k_i-fold Cartesian product of Σ^* with itself, with k_i being the arity of r_i.

Given an alignment A, the *string represented by i in A* is the string obtained by concatenating the symbols in the defined portion of row i. This string is denoted by $\sigma_A(i)$, or simply by $\sigma(i)$, when A is clear from the context. Thus we can say for instance that the tuple whose first component is the string represented by the third row in A, and whose second component is the string represented by the sixth row, is a member of binary relation r_9. In symbolic notation this amounts to writing $\langle\sigma_A(2),\sigma_A(5)\rangle \in r_9$.

In our language there is a relation symbol R_i for each relation r_i in the database. We then evaluate the truth (under assignment θ) of an an *atomic relational formula* $R_i(x_1,\ldots,x_k)$ with respect to a pair consisting of an alignment and a database. Formally we have

10. $\langle A, db\rangle \models R_i(x_1,\ldots,x_k)\theta$ iff $\langle\sigma_A(\theta x_1),\ldots,\sigma_A(\theta x_k)\rangle \in r_i$.

To complete the definition of our language we take the set of all string formulas and atomic relational formulas and close it under $\wedge, \neg$, and $\exists$. The semantic definitions for $\wedge, \neg$, and $\exists$ are standard.

Note that truth definitions 1–9 do not hinge on a particular database. Therefore we can extend them to tuples $\langle A, db\rangle$ as such. (E.g. $2: \langle A, db\rangle \models (x=a)\theta$ iff $A(\theta x, 0) = a$. Note also that when the truth definition involves a transpose, the relational part remains intact, e.g. $6: \langle A, db\rangle \models [x_1,\ldots,x_k]_l\phi\theta$ iff $\langle[\theta x_1,\ldots,\theta x_k]_l A, db\rangle \models \phi\theta$.)

When formulating a query on a database, the programmer is to assume that all involved strings that he or she is reasoning about are in a starting position corresponding to an *initial alignment*, which is an alignment where the leftmost symbol in each string is placed one position to the right of the window, i.e. at column 1. Initial alignments are denoted A_0. We thus have for instance $A_0 \models (x=\bot)\theta$, for all assignments θ and initial alignments A_0.

We have now introduced all the apparatus necessary to define the meaning of queries in string databases. A query in our model is an expression $x_1,\ldots,x_k \mid \phi$ where ϕ is a formula whose set of free variables is $\{x_1,\ldots,x_k\}$. The answer to such a query when posed to a database *db* is

$$\bigcup_{A_0,\theta}\{\langle\sigma(\theta x_1),\ldots,\sigma(\theta x_k)\rangle : \langle A_0, db\rangle \models \phi\theta\}.$$

In other words, the query asks to list those tuples of strings, such that there is a way of initially aligning the strings (possibly together with other (quantified) strings), then substituting the strings for the free variables and having the formula become true w.r.t. the initial alignment and the database.

In the sequel we shall leave out the union over initial alignments A_0 and assignments θ, when denoting answer sets. The existential quantification will be tacitly assumed.

Examples. Let r_1 be a binary and r_2 a unary string relation. The following queries will serve as an illustration of our model. We freely omit parenthesis symbols and concatenation dots whenever we feel that the structure is clear without them. We shall also as a notational convenience abbreviate atomic window formulas of the form $x=\bot$ by $x_\bot$. The symbol $\top$ denotes a tautology, e.g. $x=x$.

- List the second component of all tuples in r_1 where the first component is *abc*:

$$\begin{aligned} x \quad | \quad \exists y \quad : \quad & R_1(y,x) \wedge \\ & ([y]_l y=a)([y]_l y=b) \\ & ([y]_l y=c)([y]_l y_\bot) \end{aligned}$$

- List all tuples in r_1 where the first component equals the second component:

$$x, y \quad | \quad R_1(x,y) \wedge ([x,y]_l x = y)^* \\ .(\ [x,y]_l x_\perp \wedge y_\perp\ )$$

- List all tuples of r_2 that are a concatenation of the two components in a tuple in relation r_1:

$$x \quad | \quad \exists y, z : R_1(y,z) \wedge R_2(x) \wedge \\ ([x,y]_l x = y)^* ([x,z]_l x = z)^* \\ .(\ [x,y,z]_l x_\perp \wedge y_\perp \wedge z_\perp\ )$$

- List all tuples of r_1 where the first component is a manifold[2] of the second:

$$x, y \mid R_1(x,y) \wedge \\ (\ ([x,y]_l x = y)^* ([y]_l y_\perp)([y]_r \neg y_\perp)^* ([y]_r y_\perp)\ )^* \\ .([x,y]_l x = y)^* (\ [x,y]_l x_\perp \wedge y_\perp\ )$$

The string formula repeatedly checks that y is indeed a prefix of the remaining part of x until x is exhausted.

- List all tuples in r_2 that are a shuffle[3] of the two components of a tuple in r_1:

$$x \quad | \quad \exists y, z : R_1(y,z) \wedge R_2(x) \wedge \\ (\ ([x,y]_l x = y) + ([x,z]_l x = z)\ )^* \\ (\ [x,y,z]_l x_\perp \wedge y_\perp \wedge z_\perp)$$

- List all tuples of r_1 where the second component is of the form $(gc + a)^*$:

$$x, y \mid R_1(x,y) \wedge \\ (\ ([y]_l y = g)([y]_l y = c) + ([y]_l y = a)\ )^* \\ .([y]_l y_\perp)$$

- List all tuples of r_1 where the first component occurs in the second:

$$x, y \quad | \quad R_1(x,y) \wedge \\ ([y]_l \top)^* ([x,y]_l x = y)^* ([x]_l x_\perp)$$

- List all tuples of r_1 where the edit distance[4] between the first and the second component is no larger than k:

$$x, y \mid R_1(x,y) \wedge ([x,y]_l x = y)^* \\ .(([x,y]_l \top + [x]_l \top + [y]_l \top)([x,y]_l x = y)^*\ )^k \\ .([x,y]_l x_\perp \wedge y_\perp)$$

The formula requires that x and y must match character by character, except that in at most k places the characters need not match. A replacement can be allowed by relaxing the window formula $x = y$ to $\top$. An insertion into x is taken into account by transposing only x, and a deletion from x by transposing only y.

- List all tuples of r_2 that are of the form $aXbXa$, where $X \in \Sigma^*$. Here we abbreviate by $x =_s y$ the formula that says that strings x and y are equal (see the second example).

$$x \quad | \quad \exists y \exists z: \quad (y =_s z) \wedge R_2(x) \wedge \\ ([x]_l x = a) \\ .([x,y]_l x = y)^* \\ .([x,y]_l x = b\ \wedge\ y_\perp) \\ .([x,z]_l x = z)^* \\ .([x,z]_l x = a\ \wedge\ z_\perp) \\ .([x]_l x_\perp)$$

The formula states the existence of a string y such that x is of the form $aybya$. Instead of explicitly transposing y to the right the formula states the existence of an identical "copy" z for verifying that the first and second occurrences of y are indeed equal. This shows how the logical "and" operation can be used to "reset" the strings into the initial alignment.

- List all tuples of r_2 that are in the language consisting of strings containing an equal number of a's and b's and only those symbols:

$$x \mid \exists y \exists z : R_2(x) \\ \wedge \\ (([x,y]_l x = a \wedge \neg y_\perp) + ([x,z]_l x = b \wedge \neg z_\perp))^* \\ .([x,y,z]_l x_\perp \wedge y_\perp \wedge z_\perp) \\ \wedge \\ ([y,z]_l \neg y_\perp \wedge \neg z_\perp)^* ([y,z]_l y_\perp \wedge z_\perp)$$

The formula says that each occurrence of the character a in x must match some position in

[2] String u is a manifold of string v if $u = vv \ldots v$.

[3] The shuffle, or interleaving, of strings u and v is the set of all strings of the form $u_1 v_1 u_2 v_2 \ldots u_k v_k$ where $u = u_1 u_2 \ldots u_k$, $v = v_1 v_2 \ldots v_k$, and each u_i and v_i can be of arbitrary length, including zero.

[4] The edit distance between strings u and v is the minimum number of steps required to transform u to v. Each step can consist of replacing one symbol by another, or of inserting or deleting a symbol, see e.g. [SaK83].

a string y. Likewise, each occurrence of the character b must match a position in a string z. Furthermore, strings y and z can be exhausted simultaneously, i.e. they are of equal length.

- List all tuples of r_2 that are in the language $\{a^n b^n c^n : n \in \mathbb{N}\}$:

$$
\begin{array}{lll}
x \mid \exists y : & R_2(x) \wedge \\
& ([x,y]_l x = a \ \wedge \ \neg y_\perp)^* \\
& .([y]_l y_\perp) \\
& .([x_l]_l \top . [y]_r x = b \ \wedge \ \neg y_\perp)^* \\
& .([y]_r y_\perp) \\
& .([x,y]_l x = c \ \wedge \ \neg y_\perp)^* \\
& .([x,y]_l x_\perp \wedge \ y_\perp)
\end{array}
$$

Here we require that x is of the form $a^*b^*c^*$ and that there is a "counter string" y, such that each of the three portions of x can be put into a one-to-one correspondence with y. The fourth line of the formula also shows how simultaneous left and right transpositions are expressible in string formulas.

- List all tuples of r_2 that are of the form $(a+b)^*$, and whose second half is a translation of the first half obtained by replacing each a by b, and each b by a. This type of problem occurs e.g. in [CoV91].

$$
\begin{array}{ll}
x \mid \exists y, z : & R_2(x) \wedge \\
& ([x,y]_l x = y)^* . ([y]_l y_\perp) \\
& .([x,z]_l x = z)^* . ([z]_l z_\perp) \\
& \wedge \\
& ([y,z]_l y = a \wedge z = b \vee y = b \wedge z = a)^* \\
& .([y.z]_l y_\perp \wedge z_\perp)
\end{array}
$$

The formula states that x is of the form yz where z is the translation of y.

3 Alignment Logic and Computation

Since the queries are expressed in alignment logic in a declarative fashion, we need a procedural interpretation of the formulas in order to be able to compute the answers. Our procedural language is an extension of relational algebra, with the main addition being a selection operator based on certain finite state devices. These finite state devices correspond to the string formulas in the query.

Intuitively a *k-tape finite state acceptor* is a "nondeterministic two-way finite state automaton," with k "input tapes." Formally, a k-FSA is a system $\mathcal{A} = \langle Q, s, F, T \rangle$ where Q is a finite set of states, $s \in Q$ is a distinguished start state, $F \subseteq Q$ is a set of final states, and T is a transition relation. The transition relation is a subset of $(Q \times (\Sigma \cup \{¢, \$\})^k) \times (Q \times \{-1, 0, +1\}^k)$. The symbols ¢ and \$ are used to mark the left and right ends of the input on each "tape." The markers are assumed not to be symbols in Σ. There is one "head" per tape, and given a state and the symbols under the heads on each tape, a transition consists of switching the control into a next state and of winding each tape one cell to the left (indicated by -1 in T), one cell to the right (indicated by $+1$ in T), or holding the tape stationary (indicated by 0 in T). Furthermore, tapes are never wound off the input area marked by ¢ and \$.

In the initial state each tape is positioned so that the symbol ¢ is on the cell under the head. The device *accepts* the tape contents if there is a sequence of transitions leading into a final state, such that there is no possible next state after that. If a k-FSA $\mathcal{A}$ accepts the tape contents with ¢w_i\$ on tape i, we say that $\langle w_1, \ldots, w_k \rangle \in L(\mathcal{A})$. Furthermore, such tuples $\langle w_1, \ldots, w_k \rangle$ are the only members of $L(\mathcal{A})$.

We now have a means of computing the "bindings" making string formulas true in initial alignments.

Theorem 3.1 *For each string formula ϕ on variables $x_1, \ldots, x_k$ there is a k-FSA $\mathcal{A}_\phi$ such that*

$$L(\mathcal{A}_\phi) = \{\langle \sigma_{A_0}(\theta x_1), \ldots, \sigma_{A_0}(\theta x_k) \rangle : A_0 \models \phi\theta\}.$$

The main idea in the proof is to transform ϕ into a normal form where each atomic string formula $\tau\alpha$ is such that the window formula α is a conjunction of atomic window formulas. Then we build an acceptor for each $\tau\alpha$, and proceed inductively on the $., +, *$-structure of normalized ϕ.

The computational power of k-FSA's correspond exactly to string formulas over k variables; i.e. the converse of Theorem 3.1 is also true.

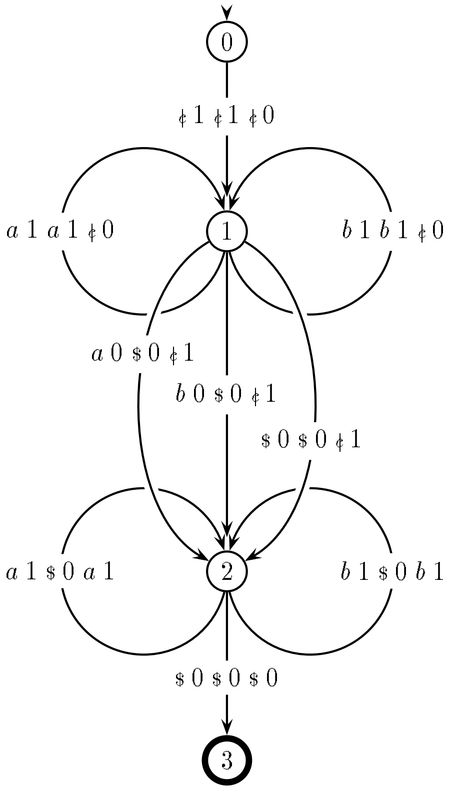
Theorem 3.2 *For each k-FSA $\mathcal{A}$ there is a string formula $\phi_\mathcal{A}$ on variables $x_1, \ldots, x_k$ such that*

$$\{\langle \sigma_{A_0}(\theta x_1), \ldots, \sigma_{A_0}(\theta x_k) \rangle : A_0 \models \phi_\mathcal{A}\theta\} = L(\mathcal{A}).$$

For this theorem we construct an atomic string formula for each transition in $\mathcal{A}$, and then we build $\phi_\mathcal{A}$ in the same manner as regular expressions are built from automata.

In Figure 3 we show an example of a string formula and the corresponding finite state acceptor. The convention in the figure is that a tuple $\langle p, a, b, ¢, q, -1, 0, +1 \rangle$ in the transition relation T

is drawn as an arc labelled "a -1 b 0 ¢ 1" going from node p to node q. The alphabet Σ is for simplicity assumed to be $\{a, b\}$.



$([x, y]_\iota x = y)^*([x, z]_\iota x = z)^*$
$.(\ [x, y, z]_\iota x_\perp \wedge y_\perp \wedge z_\perp\ )$

Figure 3: A string formula and the corresponding 3-FSA.

If we now turn our attention to the problem of evaluating queries formulated in the full language, the first issue we have to tackle is the possible infiniteness of the answer set. For instance, the answer to the query $x \mid ([x]_\iota x = a)^*([x]_\iota x_\perp)$ is the infinite set $\{a, aa, aaa, aaaa, \ldots\}$. Since our domain (i.e. Σ) is finite, a necessary and sufficient condition for finite answers is that every component in a tuple in the answer is of bounded length. If a formula ϕ has this boundedness property we shall say that ϕ is (semantically) *safe*. As an example of a safe query consider $x \mid R(x) \wedge ([x]_\iota x = a)^*([x]_\iota x_\perp)$. The answer set is finite because the candidate strings have to appear in database relation r. An answer set can of course also be finite as a consequence of the finiteness of the language specified by a string formula. (More involved reasons for finiteness are studied in the next section.) In either case there is an explicit bound depending on the fomula and/or the database.

It now turns out that if we take the relational algebra operators projection, union, difference, Cartesian product, the constant unary relations $\Sigma^{\leq l}$ (all strings of at most l characters) and Σ^* (all finite strings), and an extended selection operator, we end up with a language having the same expressive power as alignment logic. The extended selection operator is defined as

$$\delta_{i_1,\ldots,i_k,\mathcal{A}}(r) = \{t \in r : \pi_{i_1,\ldots,i_k}(t) \in L(\mathcal{A})\},$$

where $\mathcal{A}$ is a k-FSA. In other words, we project a relation on some k components (the arity of r has to be $\geq k$) and filter the resulting relation through the acceptor $\mathcal{A}$. We then retain the original version of the accepted tuples.

Let us call this extended algebra *alignment algebra*. An *expression* in the algebra is built up from the basic operators and relation symbols as usual. The *value* of such an expression when applied to a particular database is obtained by replacing each relation symbol with the corresponding actual relation from the database and then applying the operators.

Theorem 3.3 *For each alignment formula there is an expression in alignment algebra such that the answer set for the formula on any database is the same as the value of the expression on that database. The converse is also true.*

In the algebra Σ^* is an infinitary operator needed for representing the underlying infinite domain. In practice we will be interested in evaluating safe queries only. For safe queries finite domains suffice. It turns out that algebraic expressions without Σ^* and safe formulas are closely related.

Theorem 3.4 *For each expression f in alignment algebra not using the Σ^*-operator there is a safe alignment formula ϕ_f such that the answer set for ϕ_f on any database db is the same as the result of evaluating f on db.*

In the other direction, for each safe alignment formula ϕ there exists an alignment algebra expression f_ϕ with the following property: corresponding to each database db there is a constant l_{db} such that when every occurrence of Σ^* in f_ϕ is substituted with $\Sigma^{\leq l_{db}}$, the result of evaluating the substituted expression on db is the same as the answer set for ϕ on db.

The theorem means that in order to determine the answer set for a safe alignment formula on a database db it is sufficient to consider some finite subset $\Sigma^{\leq l_{db}}$ of the domain Σ^*. This finite subset depends on the database (actually on the length of its longest string) as well as on the formula. We could thus in the algebraic expressions corresponding to safe formulas regard occurrences of the operator Σ^* as a *generic symbol* to be substituted with some particular operator $\Sigma^{\leq l_{db}}$ when actually evaluating the algebraic expression, just as relation symbols in the expression are substituted with the actual relations.

The above wrinkle is due to the fact that alignment logic queries have the ability to *generate* new (and longer) strings not appearing in the database. An example is the safe query

$$x \quad | \quad \exists y, z : R_1(y) \wedge R_2(z) \wedge \psi(x, y, z),$$

where ψ is the formula in Figure 3 (cf. also the third example in Section 3). The query asks for the strings that are a concatenation of a string from r_1 and another string from r_2. In this case l_{db} can be chosen to be k, where k equals twice the length of the longest string occurring in r_1 or r_2. An algebraic exression equivalent to the query is

$$\pi_3(\delta_{1,2,3,\mathcal{A}_\psi}(R_1 \times R_2 \times \Sigma^*)),$$

where $\mathcal{A}_\psi$ is the 3-FSA in Figure 3. For our particular relations r_1 and r_2 the answer set for the query can be obtained by evaluating

$$\pi_3(\delta_{1,2,3,\mathcal{A}_\psi}(r_1 \times r_2 \times \Sigma^k)).$$

4 Safety Analysis

In the previous section we assumed that the alignment formulas were semantically safe. An important question is whether semantic safety can be syntactically determined. It is well known that safety is undecidable for purely relational formulas [Ull88]. Thus there is no hope that safety would be decidable for alignment formulas. We shall however first look into the possibility of determining safety for string formulas alone. It turns out that there is a source of undecidability in the string formulas too.

We are interested in more than just determining whether the language defined by a string formula is finite. Consider the following two queries, where we abbreviate by $x \in_s y^*$ the formula that says that x is a manifold of y (see the fourth sample query).

$$y \quad | \quad \exists x : R(x) \wedge y \in_s x^*$$

$$y \quad | \quad \exists x : R(x) \wedge x \in_s y^*$$

The first query is unsafe, whereas the second one is safe. In neither case does a boundedness of y come from y having to appear in a database relation. However, in the second query x has to be a manifold of y, and thus y is at most as long as x. Furthermore, x has to appear in a database relation. The interesting observation is that "x limits y." Note that in the first query it is not true that "x limits y." As an additional example, in the query of the previous section asking for compositions of strings in relations r_1 and r_2 it is true that "x and y together limit z."

Let ϕ be a string formula and $\{\{x_1, \ldots, x_k\}, \{y_1, \ldots, y_m\}\}$ be a partition of its variable set. Then we say that $\{x_1, \ldots, x_k\}$ *limits* $\{y_1, \ldots, y_m\}$ *in* ϕ if for every sequence $w_1, \ldots, w_k$ of not necessarily distinct elements of Σ^*, the set

$$\{w_1\} \times \ldots \times \{w_k\} \times \underbrace{\Sigma^* \times \ldots \times \Sigma^*}_{m\ times}$$

$$\bigcap$$

$$\{\langle \sigma(\theta x_1), .., \sigma(\theta x_k), \sigma(\theta y_1), .., \sigma(\theta y_m)\rangle : A_0 \models \phi\theta\}$$

is finite. We are thus interested in inferring the finiteness constraints $\phi : \{x_1, \ldots, x_k\} \rightsquigarrow \{y_1, \ldots, y_m\}$ proposed by Ramakrishnan et al [RBS87].

Theorem 4.1 *The limitation problem is undecidable when $k \geq 1$ and $m \geq 3$.*

The proof is based on the fact that we can encode the behaviour of an arbitrary 2-counter machine as a string formula on variables x_1, y_1, y_2, y_3. The undecidability of the limitation problem then follows from the undecidability of the totality problem for 2-counter machines.

Far from everything is lost however. The simulation of counter machines makes use of the fact that rows of an alignment can be transposed both left and right. We believe that for practical queries this is not needed: it is enough to have the ability to transpose only one row both to the right and to the left in a string formula. All other rows are to be transposed to the left only. Note that all our sample queries fall into such a *right-restricted* class. We shall also see in the next section that certain right-restricted formulas characterize the polynomial hierarchy.

Theorem 4.2 *The limitation problem is decidable for right-restricted string formulas.*

This proof is based in the fact that "x does not limit y" in ϕ if and only if there is a string s for which we can find strings u, w, and v, such that $\langle s, uw^i v\rangle \in L(\mathcal{A}_\phi)$ for all $i \in \mathbb{N}$. Furthermore, in right-restricted formulas the length of w can be bounded. We can thus determine the existence of such strings w by inspecting the transition graph of $\mathcal{A}_\phi$. Technically, strings s, u, w, and v are found using a crossing-sequence construction.

The solution to the limitation problem represents a first step towards a syntactic characterization of safety and an implementation of alignment logic. However, in addition to be able to determine whether a query in the logic is safe, we also need to determine the upper bound of the length of y as a function of the length of x, when $\{x\} \rightsquigarrow \{y\}$. Only then we can constructively determine the constant l_{db} of Theorem 3.4 for each database db. Note also that not all string formulas of a safe alignment logic formula need to be right-restricted, provided that only right-restricted string formulas are used for generating new strings while the others are used only for verifying them.

5 Expressive power

In this section we study the definitional power of alignment logic. For simplicity we focus on formulas without relation symbols. Furthermore, we assume that there is only one free variable and w.l.o.g. that all quantifiers are at front. Given such an "unrelational" formula ϕ we say that ϕ *defines* the set

$$\{\sigma_{A_0}(\theta x) : A_0 \models \phi\theta\}.$$

By the (data) complexity [Var82] of a set defined by a formula ϕ we mean the complexity of the decision problem "$w \in \{\sigma_{A_0}(\theta x) : A_0 \models \phi\theta\}$?," measured as a function of the length of the string w.

First we observe that if there are no quantifiers in the formulas, then the corresponding acceptors are 1-FSA's, and conversely. As a consequence of Theorems 3.1 and 3.2 we thus obtain the following result.

Theorem 5.1 *Every quantifier-free formula defines a regular set, and every regular set is definable by a quantifier-free formula.*

Using the fact that counter machines can be specified by alignment formulas, and that the 2-counter machines characterize the recursively enumerable sets we obtain the next theorem.

Theorem 5.2 *Every formula with two existential quantifers and no negation defines an r.e. set, and every r.e. set is definable by such a formula.*

The following result follows from the fact that extended temporal logic has the expressive power of regular expressions.

Theorem 5.3 *Every set definable in extended temporal logic is definable in alignment logic. There are sets definable in alignment logic that are not definable in extended temporal logic.*

For those programmers who are comfortable reasoning with temporal modalities, our logic offers the possibility of regarding a transpose as going into the future (left) or the past (right) in those linear time structres (rows) that are mentioned in the transpose. The following conventions can then be used.

next along $x_1, \ldots, x_k\ \phi \quad =_{df.} \quad [x_1, \ldots, x_k]_l \phi$

ϕ *along* $x_1, \ldots, x_k$ *until* ψ
$=_{df.} \quad ([x_1, \ldots, x_k]_l \phi)^*([x_1, \ldots, x_k]_l \psi)$

eventually along $x_1, \ldots, x_k\ \phi$
$=_{df.} \quad ([x_1, \ldots, x_k]_l \top)^*([x_1, \ldots, x_k]_l \phi)$

henceforth along $x_1, \ldots, x_k\ \phi$
$=_{df.} ([x_1, \ldots, x_k]_l \phi)^*([x_1, \ldots, x_k]_l x_{1\perp} \wedge \ldots \wedge x_{k\perp})$.

For instance, the two-place predicate "*x occurs in y*" (see the seventh sample query) could be formulated as

eventually along y ($x = y$ *along* x, y *until* $x_\perp$).

The modalities for past tense are obtained by using right transposes instead of the left ones in the list above, e.g.

ϕ *along* $x_1, \ldots, x_k$ *since* ψ
$=_{df.} \quad ([x_1, \ldots, x_k]_r \phi)^*([x_1, \ldots, x_k]_r \psi)$.

As far as the *sequence logic* of Ginsburg and Wang [GiW92] is concerned, it turns out that it is included in alignment logic.

Theorem 5.4 *Every set definable in sequence logic is definable in alignment logic.*

We note that the programming paradigms underlying sequence logic and alignment logic are different. The basic stringological construct in sequence logic is the transducer mapping. Since regular sets are closed under transducer mappings [GiS65], computational power is achieved by resorting to "extra-stringological" features, such as first order logic. In alignment logic, on the other hand, full Turing computability is available already when using unquantified string formulas on three variables and a projection operator (cf. Theorem 5.2). As pointed out by Ginsburg and Wang [GiW92], it is of course always possible to increase the expressive power of sequence logic by strengthening the language used to specify the transducers. However, in alignment logic it not necessary to go beyond regular expressions in the syntax of the string formulas.

We have also been able to obtain a correspondence between alignment formulas and the polynomial hierarchy [Sto77]. For this correspondence we need the class of *quantifier limited* formulas. A formula ϕ with variable set $\{x, y_1, \ldots, y_m\}$, where x is free and the y:s are quantified, is said to be quantifier limited if $\{x\}$ limits $\{y_1, \ldots, y_m\}$.

The reader is asked to recall from the previous section that right-restriction in formulas means that at most one variable is transposed in both directions.

Theorem 5.5 *Each right-restricted and quantifier limited formula with a leading existential (universal) quantifier and with k quantifier alternations defines a set in* Σ_k^p *(in* Π_k^p*). Furthermore, for each* $k \in \mathbb{N}$ *there is a right-restricted quantifier limited formula with a leading existential (universal) quantifier and with k quantifier alternations that defines a* Σ_k^p*-hard (resp.* Π_k^p*-hard) set.*

The upper bound follows from the fact that if $\{x\}$ limits $\{y\}$, then the limit is a polynomial of the length of x. Thus quantifier limitation implies that the quantification is polynomially bounded. Also, model checking for string formulas can be done in time polynomial in the length of the strings. For the lower bound we show that for each $k \in \mathbb{N}$ there is a formula ϕ_k of the form stated in the theorem, such that for any quantified boolean formula ψ that has k quantifier alternations and leading quantifier existential, the following holds: there is a log-space reduction from the formula ψ to a string s_ψ such that ψ is true if and only if $s_\psi \in \{\sigma_{A_0}(x\theta) \colon A_0 \models \phi_k\theta\}$. Similar formulas ϕ_k can also be exhibited when the leading quantifier is universal.

References

[CoV91] 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* (1991), 321–326.

[GaS83] Z. Galil and J. Seiferas. Time-space optimal string matching. *JCSS 26* (1983), 280–294.

[GaJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman Press 1979.

[GiS65] S. Ginsburg and H. Spanier. Mappings of languages by two-tape devices. *J. ACM 12* (1965), 423–434.

[GiW92] S. Ginsburg and X. Wang. Pattern matching by rs-operations: towards a unified approach to querying sequenced data. *PODS '92*, pp. 293–300

[HeS93] C. Helgesen and P. R. Sibbald. PALM - A pattern language for molecular biology. In: *Proc. First Int. Conf. on Intelligent Systems for Molecular Biology (ISMB-93)*, AAAI Press 1993, pp. 172–180.

[PiT86] P. Pistor and R. Traunmueller. A database language for sets, lists and tables. *Information Systems 11* (1986), 323–336.

[RBS87] R. Ramakrishnan, F. Bancilhon and A. Silberschatz. Safety of recursive Horn clauses with infinite relations. *PODS '87*, pp. 328–339.

[Ric92] J. Richardson. Supporting lists in a data model (a timely approach). IBM research report RJ 8853, June 1992.

[SaK83] D. Sankoff and J. B. Kruskal (eds.) *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wesley 1983.

[Sto77] L. J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science 3* (1977), 1–22.

[Ull88] J. D. Ullman. *Principles of Database and Knowledge-base Systems.* Computer Science Press 1988.

[Var82] M. Vardi. The complexity of relational query languages. *STOC '82*, pp. 137–145.

[Wol83] P. Wolper. Temporal logic can be more expressive. *Information and Control 56* (1983), 72–93.