Safety, Translation and Evaluation of Alignment Calculus

Gösta Grahne Matti Nykänen

Department of Computer Science
P.O. Box 26, FIN–00014 University of Helsinki, Finland
{Gosta.Grahne,Matti.Nykanen}@cs.helsinki.fi

Abstract

In this paper we pursue the study of Alignment Calculus, a declarative string database query language that supports both string querying and restructuring. This language is aimed for applications such as molecular biology databases, where the sequence queries are combinatorial in nature. The declarative nature of this language does, however, require additional effort in its implementation. Here we solve this problem by first defining a domain independent syntactic subset of the full language and then developing a query evaluation mechanism for this sublanguage. This mechanism then handles the required restructuring operations in a finite manner.

1 Introduction

In this paper we continue our study of the problem of strings in databases [13]. Our primary source of motivation is the storage and qualitative processing of genetic information represented as sequences of symbols, and therefore amenable to methods from formal language theory [3, 4, 6, 19, 25, 26, 27]. Consequently, we have developed a declarative query language for combinatorial sequence queries. This language differs from such languages as SEQ [28, 29], where sequence elements and their underlying order type are distinct domains, in that the queries in our application area are more oriented toward parsing-type tasks than for example computing moving averages on temporally sequenced data [28, Example 2.1].

We therefore extend the relational model to include finite strings over some finite alphabet Σ (of at least 2 characters) as primary objects of information. A relation of arity k is in our model then a finite subset of the k-fold Cartesian product of Σ^*, the set of all finite strings in alphabet Σ, with itself. In other words, each position in a tuple of a relation contains a finite string of arbitrary length instead of just a single atomic value.

Present-day database query languages offer little support for string relations. For example, the *Sequence Retrieval System (SRS)* [8], which has recently gained popularity in molecular biology, does allow the database administrator to draw links from one preformatted data file to another [9], but only on its atomic non-sequence fields. Because the majority of current relational database management systems do not adequately support application-specific data types such as sequences, some molecular biology database designers have begun to move towards object-oriented database technology [10, Chapter 4]. Another solution, strongly advocated in for instance [5, Chapter 25], 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 [1].

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 $(\mathsf{gc}+\mathsf{a})^*$". However, in applications such as the aforementioned molecular biology databases, the language needs to have expressive power beyond regular sets [3]; hence, more expressive grammatic formalisms have been employed in computational biology [4, 6, 16, 25, 26]. In addition to data extraction features, the string language needs also *data restructuring* constructs [12, 20, 31]. 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; Example 1 below shows how to do it in our language.

Both of these goals can be simultaneously attained by a *declarative* language for expressing properties of strings. Richardson [23] suggested 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 [37], using only the modalities *next* and *until*, the language cannot express that a property holds in every even position of the string. 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 have therefore defined a logic called *Alignment Calculus* [13], 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*

To appear in the *First East-European Symposium on Advances in Databases and Information Systems (ADBIS'97)*, St.-Petersburg (Russia), September 2–5, 1997.

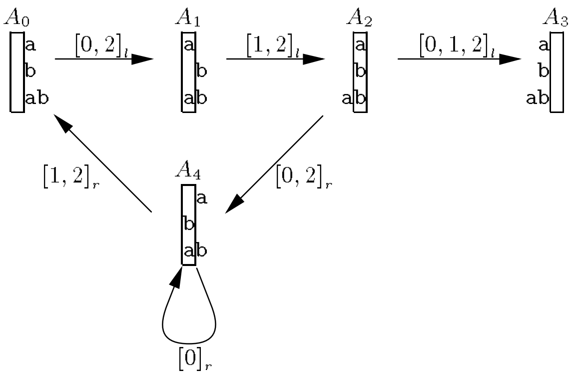

Figure 1: Alignments and their transposes

(as opposed to second or higher order) like for example 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. Hence, our work can be seen from three distinct, yet equivalent, viewpoints:

- as an extension of modal logic techniques to multiple sequence comparisons;
- as an introduction of the biologically relevant concept of multiple sequence alignment as a semantic primitive already at the database level; or
- as a study of finite-state computation capabilities as a sequence database query mechanism.

Thus, we believe our work to be applicable in a broad variety of string database applications.

The declarative nature of Alignment Calculus does not immediately suggest an implementation strategy, as pointed out by Mecca and Bonner [20]. The problem is to analyse a given declarative query to find out what kinds of basic sequence restructuring and database operations are implictly involved, and whether those operations can be carried out in a finite manner. In this paper we explain how this analysis can be done. Consequently, we arrive at an implementation method for queries expressed in a subset of Alignment Calculus.

We present a quick overview of Alignment Calculus in Section 1.1, with examples. In Section 1.2 we present the main focus of the current paper, namely the safety problem in Alignment Calculus. Then Sections 2 and 3 present our solution: first we suggest a solution for the string part of our language, and then extend it to the calculus part. Finally, Section 4 concludes our presentation.

1.1 Alignment Calculus

This section presents an overview of Alignment Calculus. By necessity, this overview must remain at an illustrative level; detailed exposition can be found elsewhere [14, Section 2][21, Section 3].

The basic semantical construction is the *alignment*. Figure 1 presents five different alignments $A_0, \ldots, A_4$ of the three strings `a`, `b`, and `ab`. A designated *window column* runs through each alignment. These alignments are connected together with modalities called *transposes*. Figure 1 shows as arrows six transposes between the alignments $A_0, \ldots, A_4$. Each arrow is labelled with $[\text{rows}]_{\text{direction}}$ to emphasize that the second alignment is obtained from the first by *sliding* the designated rows one character into the designated direction; for example, alignment A_2 is obtained from alignment A_1 by sliding rows 1 and 2 one character to the left. Intuitively, the string handling part of Alignment Calculus will consist of a language for specifying paths through such alignment-transpose structures, or equivalently a sequence of slides and window column tests. Note in particular that sliding is allowed to both directions.

Let us now introduce the logic. First, *assignments* θ map variable symbols $x_0, x_1, x_2, \ldots$ to alignment rows. Letting for example $\theta(x_i) = i$, the variable now x_0 refers to the row containing `a` in Figure 1. The language then consists of the following components.

Window formulae can be formed in a natural way to discuss the contents of the window column. For example, the window formula $\phi \equiv (x_0 = x_2)$ is true in alignment A_1 under assignment θ, customarily denoted as $A_1 \models \phi\theta$, while ϕ is not true in A_2 under θ.

A window formula can mention as constant terms not only all the elements of Σ but also the distinct special symbol '$\perp$', where $\phi' \equiv (x_1 = \perp)$ is interpreted as "the window column on the row (pointed to by) x_1 is empty". For example, ϕ' is true under θ in A_0, A_1 and A_3, and false in A_2 and A_4. In addition, all empty positions (no matter in which end of the row they are) are equal, so that ϕ is true in A_0 and A_3 under θ.

Atomic string formulae consist of a transpose with variables instead of rows and a window formula, as in $\psi \equiv [x_0, x_2]_l\phi$. Intuitively, an atomic string formula specifies what must be found in the window column of the alignment reached from the current one by the transpose; in other words, a "legal" move in an alignment-transpose structure. Now $A_0 \models \psi\theta$, because applying θ to $[x_0, x_2]_l$ yields the transpose $\tau = [0, 2]_l$, and applying this τ to the current alignment A_0 leads in turn into alignment A_1, where $A_1 \models \phi\theta$ was seen to hold.

A transpose can no longer move the row, if its window column is already past its corresponding end, as witnessed by trying to apply $[0]_r$ to A_4.

String formulae are regular expressions over the (conceptually infinite) alphabet of all atomic string formulae. For example, $\chi \equiv \psi.\psi^*$ is a simple string formula, which represents the sequences $\psi, \psi\psi, \psi\psi\psi, \ldots$ in a natural way. More involved examples of string formulae are given below. Intuitively, a string formula expresses a sequence of atomic string formulae, which should constitute a legal walk from the current alignment within the alignment-transpose structure. Then for example $A_0 \models \chi\theta$, because χ contains the single-step walk ψ such that $A_0 \models \psi\theta$.

String formulae are equivalent to nondeterministic multitape finite state automata (FSAs) [13, Theorems 3.1 and 3.2], which then provide an independent operational semantics. The string formula concept could naturally be extended above regular expressions as well [19].

The subclass of *right-restricted* string formulae is of particular interest. In these formulae, at most one variable is *bidirectional*; that is, mentioned also inside right transposes $[\ldots]_r$. All the other variables are *unidirectional* and appear only within left transposes $[\ldots]_l$.

Alignment Calculus is then domain relational calculus [32, Chapter 3.8] over string relations equipped with string formulae. The truth of such an Alignment Calculus formula Φ is examined with respect to an assignment θ and a pair A_0, db, and denoted customarily as $\langle A_0, db\rangle \models \phi\theta$. The first component A_0 is an *initial* alignment, where every row is just to the right of the window column, so that all the walks encoded by the string formulae in Φ start from such initial alignments and not at arbitrary ones. The alignment A_0 in Figure 1 is such an initial one. The second component db maps relational symbols $R_0, R_1, R_2, \ldots$ into finite string relations $db(R_i)$ (customarily denoted as r_i) of corresponding arities, thereby representing the currently stored database as an interpretation of the relation symbols; the collection of all such db is denoted DB.

We introduce the following notation for syntactic operations on formulae: $\mathrm{free}(\phi)$ denotes the set of variable symbols appearing free in the formula ϕ, $\phi^{x\mapsto y}$ denotes the formula obtained from ϕ by replacing every free occurrence of variable symbol x with the occurrence of a variable symbol y, and $\|\phi\|$ denotes the number of atomic string formula occurrences within the string formula ϕ.

Let us now present a few examples.

Example 1 The query

$$\begin{aligned}\Phi_1 &\equiv x_3 : \exists x_1, x_2 . R_1(x_1) \wedge R_2(x_2) \wedge \phi_1\\ \phi_1 &\equiv ([x_1, x_3]_l x_1 = x_3)^*.([x_2, x_3]_l x_2 = x_3)^*.\\ &\quad ([x_1, x_2, x_3]_l x_1 = x_2 = x_3 = \perp)\end{aligned}$$

denotes the concatenations uv of all the strings u from r_1 with all the strings v from r_2. This is because $\langle A_0, db\rangle \models \Phi_1\theta$ if and only if

- θ maps x_1 into a row of A_0 containing a string u from the relation r_1;
- x_2 is similarly mapped into some $v \in r_2$; and
- $A_0 \models \phi_1\theta$. This in turn holds if and only if x_3 is mapped into some row containing uv, because otherwise some move in the walk encoded by ϕ_1 fails.

  If for example $u = \mathtt{a}$ and $v = \mathtt{b}$, then walk $A_0, \ldots, A_4$ in Figure 1 is successful, provided that x_3 is mapped into the row containing $\mathtt{ab}$.

Let us henceforth denote such answer relations with the notation $[\![\Phi_1]\!]_{db}$. (We omit here some technicalities, which ensure that the alignments are sufficiently large for the quantifiers to indeed range freely over Σ^* as expected.)

This ϕ_1 is an example of a unidirectional string formula; that is, one without bidirectional variables.

Example 2 The query

$$\begin{aligned}\Phi_2 &\equiv x_2 : \exists x_1 . R_1(x_1) \wedge \phi_2\\ \phi_2 &\equiv ([x_1, x_2]_l x_1 = x_2 \neq \perp)^*.([x_1]_l x_1 = \perp).\\ &\quad (([x_1]_r x_1 \neq \perp).([x_2]_l x_1 = x_2))^*.\\ &\quad ([x_1]_r x_1 = \perp).([x_2]_l x_2 = \perp)\end{aligned}$$

denotes the relation $[\![\Phi_2]\!]_{db} = \left\{uu^{-1} : u \in r_1\right\}$, where u^{-1} denotes the reversal of string u. Here is an example of a query requiring the ability to transpose rows in both directions.

Accordingly, ϕ_2 is no longer unidirectional, but it still remains right-restricted, because only x_1 is ever transposed right. Then we call also the whole query Φ_2 right-restricted, because its every string formula is right-restricted (or unidirectional).

Example 3 The query

$$\begin{aligned}\Phi_3 &\equiv x_2 : R_2(x_2) \wedge \forall x_1(R_2(x_1) \rightarrow \phi_3)\\ \phi_3 &\equiv ([x_1, x_2]_l x_1 \neq \perp \wedge x_2 \neq \perp)^*.([x_1]_l x_1 = \perp)\end{aligned}$$

asks for the longest strings of relation r_2.

Example 4 Let A be an arbitrary recursively enumerable language. There exists a string formula ϕ_A with $\mathrm{free}(\phi_A) = \{x_1, x_2, x_3\}$ such that $[\![\exists x_2, x_3 . \phi_A]\!] = A$, where both quantified variables x_2 and x_3 are bidirectional [13, Theorem 5.2]. (This ϕ_A is therefore not even right-restricted.) From this, the full expressive power of Alignment Calculus is seen to equal the Arithmetical Hierarchy [24, Chapter 14]. It is even the case that ϕ_A can be replaced by the conjunction of two unidirectional string formulae ϕ'_A and ϕ''_A [21, Corollary 6.1][14, Corollary 5.1].

Note that this formula contains no relation symbols. We call such formulae *pure* and drop the subscript from their answers to indicate that they do not depend on the database.

The operational equivalent to the Alignment Calculus is the *Alignment Algebra* [13, Theorems 3.3 and 3.4] obtained from relational algebra with the following two modifications. First, selection conditions $\sigma_{\mathcal{A}}(\ldots)$ are expressed using FSAs $\mathcal{A}$. And second, a new explicit domain symbol Σ^* is added to represent all finite strings from the alphabet Σ, together with symbols $\Sigma^{\leq n}$, where $n \in \mathbb{N}$, for all strings $u \in \Sigma^*$ of length $|u| \leq n$.

1.2 The Safety Problem in Alignment Calculus

Relational calculus has the concept of a *domain independent* formula ψ [32, Chapter 3.8], whose answer stabilizes once the evaluation domain is "sufficiently large". What sufficiently large means depends in turn on the current database and the formula ψ itself, but for every database there must be some sufficiently large domain. The analogous concept in Alignment Calculus is that the answer of a domain independent query ϕ must stabilize after ϕ is evaluated on some finite subset $\Sigma^{\leq \ell}$ of the infinite conceptual domain Σ^*. This ℓ in turn depends on both the current database and the query ϕ.

Definition 1.1 *The Alignment Calculus formula ϕ is* domain independent *if and only if there exists a limit function $\mathcal{W}_\phi : DB \rightarrow \mathbb{N}$ such that $[\![\phi]\!]_{db} = [\![\phi]\!]^{\ell}_{db}$ for every database $db \in DB$ and constant $\ell \geq \mathcal{W}_\phi(db)$.*

Here $[\![\phi]\!]^{\ell}_{db}$ denotes the answer set of query ϕ evaluated over all truncated *initial alignments A_0^{ℓ}, in which all rows are truncated to length at most ℓ.*

For instance, Example 1 is domain independent, because it suffices to examine strings of length at most $\mathcal{W}_{\Phi_1}(db) = \max\{|u| + |v| : u \in db(R_1), v \in db(R_2)\}$ to compute the answer correctly.

It is essential to let $\mathcal{W}_\phi$ depend not only on ϕ but also on db; otherwise there would be no domain independent way to express for instance Example 1. The safety concept of Ginsburg and Wang [12] falls short in this respect. Another possible approach would have been to let the domain grow during query evaluation dynamically, but still in a controlled way, as in the Sequence Datalog of Mecca and Bonner [20].

As Definition 1.1 encompasses the undecidable domain independence of relational calculus [32, Exercise 3.32], practical query evaluation must restrict its attention into some syntactic subclass of all domain independent Alignment Calculus formulae. In what follows, the main theme is to identify, and provide an implementation for, such a subclass.

2 Limitation Functions for String Formulae

In [14, Definition 3.1][21, Definition 4.2] we introduced the *limitation problem* as follows.

Definition 2.1 *Let ϕ be a string formula with* $\mathrm{free}(\phi) = \{x_1, \ldots, x_{k+l}\}$. *The variables* $x_1, \ldots, x_k$ limit *the other variables* $x_{k+1}, \ldots, x_{k+l}$ *if and only if there exists a function* $\mathcal{W}_\phi \colon \mathbb{N}^k \to \mathbb{N}$ *such that if* $\langle u_1, \ldots, u_k, v_1, \ldots, v_l \rangle \in [\![\phi]\!]$, *then* $\max(|v_1|, \ldots, |v_l|) \leq \mathcal{W}_\phi(|u_1|, \ldots, |u_k|)$.

That is, the relation $[\![\phi]\!]$ represented by the string formula ϕ is such that the lengths of the strings appearing in columns $k+1, \ldots, k+l$ depend on the lengths of the string appearing in columns $1, \ldots, k$ by $\mathcal{W}_\phi$. This condition is denoted as $\Gamma = \{x_1, \ldots, x_k\} \rightsquigarrow \{x_{k+1}, \ldots, x_{k+l}\}$, as it is a *finiteness dependency* [22] of $[\![\phi]\!]$. The limitation problem is then to decide whether a given finiteness dependency Γ holds for the relation $[\![\phi]\!]$ given implicitly as the string formula ϕ.

Example 5 The string formula ϕ_1 from Example 1 satisfies the finiteness dependency $\{x_1, x_2\} \rightsquigarrow \{x_3\}$ with the associated function $\mathcal{W}^1_{\phi_1}(n_1, n_2) = n_1 + n_2$, and the converse dependency $\{x_3\} \rightsquigarrow \{x_1, x_2\}$ with the function $\mathcal{W}^2_{\phi_1}(n_3) = n_3$.

2.1 The Basis of Limitation Properties

We define the set $\mathrm{basis}(\phi)$ for a string formula ϕ to consist of all the finiteness dependencies $\Gamma = X \rightsquigarrow Y$ such that $X \cap Y = \emptyset$, $X \cup Y = \mathrm{free}(\phi)$, and Γ holds for ϕ in the sense described above. The limitation problem can then be stated as requesting the computation of $\mathrm{basis}(\phi)$ given ϕ as input.

If ϕ is instead an atomic relational formula, then it is natural to define $\mathrm{basis}(\phi) = \{\emptyset \rightsquigarrow \mathrm{free}(\phi)\}$, becuse it is certainly the case that the variables $\mathrm{free}(\phi)$ have a finite set of possible values, namely the ones in the relation. This also emphasizes the fact that every string must originate either from the database, or from restructurings of other strings with the same origin.

Note, however, that every $\Gamma \in \mathrm{basis}(\phi)$ must mention *all* the variables occurring in ϕ to avoid the following paradoxical situation. Consider the finite query

$$\begin{aligned}
\Psi &\equiv x_1, x_2, x_3 : R_2(x_2) \wedge \psi \wedge \phi_1 \\
\psi &\equiv ([x_1, x_2, x_3]_l x_1 \neq \bot \wedge x_2 \neq \bot \wedge x_3 \neq \bot)^*. \\
&\quad ([x_2, x_3]_l x_2 \neq \bot \wedge x_3 \neq \bot)^*.([x_1, x_2]_l x_1 = x_2 = \bot),
\end{aligned}$$

where ϕ_1 is the string formula in Example 1. This query denotes the triples $\langle u, v, uv \rangle$, where $v \in r_2$, and $|u| \leq |v|$. Now the string formula ψ satisfies the *partial* finiteness dependency $\Gamma_\psi = \{x_2\} \rightsquigarrow \{x_1\}$. The finiteness of $[\![\Psi]\!]_{db}$ can also be deduced by deriving $\emptyset \rightsquigarrow \{x_1, x_2, x_3\}$ from Γ_ψ, $\{x_1, x_2\} \rightsquigarrow \{x_3\} \in \mathrm{basis}(\phi_1)$, and $\emptyset \rightsquigarrow \{x_2\} \in \mathrm{basis}(R_2(x_2))$. The query Ψ is, however, difficult to evaluate because it involves an infinite intermediate result, no matter which evaluation order is attempted, either

$$\begin{aligned}
[\![R_2(x_2) \wedge \phi_1]\!]_{db} &= \{\langle u, v, uv \rangle : v \in r_2\} \text{ or} \\
[\![R_2(x_2) \wedge \psi]\!]_{db} &= \{\langle u, v, w \rangle : v \in r_2, |u| \leq |v| \leq |w|\}.
\end{aligned}$$

This is because Definition 1.1 took a *global* view of the whole query Ψ, while bottom-up evaluation relies on a *local* view instead. The evaluation of such queries Ψ can indeed proceed by first evaluating the corresponding limit function $\ell = \mathcal{W}_\Psi(db)$ from Definition 1.1 to provide the necessary global information, and restricting attention to strings no longer than ℓ during the subsequent computation. However, although this query evaluation method is declaratively correct and seductively expressive, it is also likely to be extremely expensive. Therefore we seek a more traditional, bottom-up method.

This particular query Ψ could be expressed equivalently as $R_2(x_2) \wedge \phi_3 \wedge \phi_1$, where the string formula ϕ_3 comes from Example 3. This reformulation can then be evaluated from left to right without infinite intermediate results. Such reformulations require in general the *decomposition* of a given string formula, here ψ, into others, here into ϕ_3 and $\phi_3^{x_2 \mapsto x_3, x_1 \mapsto x_2}$ (the latter of which is then redundant). Indeed, limitation-like problems do appear in the context of such decomposition problems [36]. It is, however, unclear whether such decomposition is feasible, or even possible, in view of Example 4, so we pose this *comprehensivity* restriction on the finiteness dependencies $\Gamma \in \mathrm{basis}(\phi)$ instead.

2.2 Estimates for Limitation Functions

While the limitation problem is undecidable in general [13, Theorem 4.1], it remains decidable for the right-restricted string formulae, which explains their importance. It is furthermore known that the corresponding functions are linear in the unidirectional and quadratic in the bidirectional case [14, Theorem 4.2][21, Theorem 5.2]. This can be generalized into the following result.

Theorem 2.1 *Let ϕ be a string formula such that* $\Gamma = \{x_1, \ldots, x_k\} \rightsquigarrow \{x_{k+1}, \ldots, x_{k+l}\} \in \mathrm{basis}(\phi)$, *where the variables* $x_1, \ldots, x_h$, $h \leq k$, *are bidirectional and the others unidirectional. The function*

$$\begin{aligned}
\mathcal{W}_\phi(n_1, \ldots, n_k) &= a_\phi \left(\prod_{i=1}^{h} (n_i + 2) \right) \left(k + 1 + \sum_{j=1}^{k} n_j \right) - 1 \\
a_\phi &= \|\phi\| \left(|\Sigma| + 1 \right)^l
\end{aligned}$$

is a limitation function for ϕ and Γ.

That is, when ϕ has h bidirectional variables, none of which appears in the right side $\{x_{k+1}, \ldots, x_{k+l}\}$ of Γ, then the limitation function is a polynomial of degree $h+1$.

The proof of Theorem 2.1 is by a straightforward enumeration argument, intuitively counting the number of possible nonrepeating configurations in verifying the truth of ϕ.

Example 6 Consider the unidirectional string formula ϕ_1 from Example 1. Theorem 2.1 yields

$$\begin{aligned}
\mathcal{W}^1_{\phi_1}(n_1, n_2) &= 3(|\Sigma| + 1)(n_1 + n_2 + 3) - 1 \text{ and} \\
\mathcal{W}^2_{\phi_1}(n_3) &= 3(|\Sigma| + 1)^2 (n_3 + 2) - 1
\end{aligned}$$

for $\{x_1, x_2\} \rightsquigarrow \{x_3\}$ and $\{x_3\} \rightsquigarrow \{x_1, x_2\}$.

Turning now to the case when the right side has one bidirectional variable but the left side has none, we have the following result.

Theorem 2.2 *Let ϕ be a right-restricted string formula, whose bidirectional variable is x_{k+1}, and $\Gamma = \{x_1, \ldots, x_k\} \rightsquigarrow \{x_{k+1}, \ldots, x_{k+l}\} \in \mathrm{basis}(\phi)$. If $l = 1$, then the function*

$$\mathcal{W}'_\phi(n_1, \ldots, n_k) = (2e_\phi)^{2e_\phi} \left(1 + \sum_{i=1}^{k}(n_i + 1)\right) - 1$$
$$e_\phi = \|\phi\| \left(|\Sigma| + 2\right)^{k+l} + 2$$

is a limitation function for ϕ and Γ. If $l > 1$, then the limitation function is

$$\mathcal{W}_\phi(n_1, \ldots, n_k) = \mathcal{W}''_\phi(n_1, \ldots, n_k, \mathcal{W}'_\phi(n_1, \ldots, n_k)),$$

where the exterior function $\mathcal{W}''_\phi : \mathbb{N}^{k+1} \to \mathbb{N}$ is obtained as follows. Transfer first x_{k+1} from the right side to the left side in Γ to yield $\Gamma =' \{x_1, \ldots, x_{k+1}\} \rightsquigarrow \{x_{k+2}, \ldots, x_{k+l}\}$, and then apply Theorem 2.1 to ϕ and Γ'.

Theorem 2.2 therefore yields again a quadratic limitation function for a formula with one bidirectional variable, just like Theorem 2.1, but with a multiplicative constant e_ϕ that depends now exponentially instead of linearly on $\|\phi\|$, as a_ϕ did. This reflects the fact that being able to both first produce and subsequently *rescan* x_{k+1} can lead to very elaborate reasons for finiteness, where no single scan is finite in itself but their joint effect is. Indeed, if x_{k+2} is also bidirectional, these reasons become so elaborate that the limitation problem becomes undecidable [14, Theorem 4.1][21, Theorem 5.1]. (This can also be seen from Example 4.)

The proof of Theorem 2.2 is by estimating as $\mathcal{W}'_\phi : \mathbb{N}^k \to \mathbb{N}$ the size of the graph built during the decision algorithm for the limitation properties of right-restricted string formulae [14, Theorem 4.2][21, Theorem 5.2]. This is in turn exponential, because the algorithm utilizes a generalization of the crossing sequence construction for converting two-way finite automata into classical ones [17, Chapter 2.6 and Exercises 2.18 to 2.20]. (Note that we are *not* eliminating bidirectionality from a right-restricted string formula; indeed, Example 2 already shows this to be impossible. We are merely annotating the construction so that the result tells us enough about the limitation properties of the original formula.)

Because Theorems 2.1 and 2.2 use only minimal amount of information about ϕ, namely just $\|\phi\|$, they tend to overestimate grossly the actual limitation functions, as witnessed by Examples 5 and 6. More detailed constructions based on the structural analysis of ϕ are possible, using analogous techniques to argument size analysis of logic programs [34]. For example, tight limitation functions can be obtained for unidirectional string formulae, solving the mismatch between Examples 5 and 6. These constructions are, however, difficult to achieve without sophisticated analysis of ϕ, as the next example reveals. The unidirectional string formula

$$\phi \equiv ([x_2]_l x_2 = \mathtt{a})^*.([x_1]_l x_1 = x_2 = \mathtt{b}).$$
$$([x_1, x_2]_l x_1 = x_2 = \bot)$$

does not appear to satisfy the finiteness dependency $\Gamma = \{x_1\} \rightsquigarrow \{x_2\}$, because ϕ contains a loop where x_2 is advanced, while x_1 is not. It is, however, the case that this loop cannot be taken, because the loop condition $x_2 = \mathtt{a}$ conflicts with the condition $x_2 = \mathtt{b}$ after the loop. Hence, ϕ is actually unsatisfiable and thus trivially satisfies Γ with $\mathcal{W}_\phi(n_1) = 0$. Therefore, improving limitation function estimates imply improving the semantic analysis of string formulae, which is in turn nontrivial. However, the number of variables in one string formula is likely to be small, which helps the semantic analysis.

3 Limitation Functions for Calculus Formulae

Now that Section 2 has provided us with finiteness dependencies and limitation functions for the atomic formulae, here we show how this information can be extended into the formulae for the whole calculus. First, Section 3.1 defines a new normal form for Alignment Calculus, which is an extension of the safe DRC proposed by Ullman [32, Chapter 3.8]. Then in Section 3.2 we analyze these normalized formulae for limitedness. In Section 3.3 we discuss what kinds of combinatorial queries can be expressed using the formulae that survive this scrutiny. Finally in Sections 3.4 and 3.5 we discuss the translation and subsequent evaluation of these formulae.

3.1 Quantified Disjunctive Normal Form

We say that an Alignment Calculus formula ϕ is in *Quantified Disjunctive Normal Form (QDNF)* if its structure adheres to the following context-free grammar [17, Chapter 4].

$$\begin{array}{ll} S \longrightarrow D \mid S \vee D & D \longrightarrow C \mid QD \\ Q \longrightarrow \exists \text{ variable symbol} & C \longrightarrow B \mid C \wedge B \\ B \longrightarrow A \mid \neg A \mid \neg QD & A \longrightarrow \text{atomic formula} \end{array}$$

That is, ϕ is a disjunction S of conjunctions C, where each C consists of an optional quantifier prefix Q^*, possibly negated atomic formulae, and negated subconjunctions QD, now with mandatory quantifier prefixes Q. Without loss of generality we further assume that a formula in QDNF has no redundant quantifiers (that is, $x \in \mathrm{free}(\phi)$ for every subformula $\exists x\phi$), and that its bound variables are standardized apart (that is, if the subformula $\exists x\phi$ occurrs within some larger formula ψ, then $x \notin \mathrm{free}(\psi)$). For example, the QDNF of Φ_3 from Example 3 is

(1) $$R_2(x_2) \wedge \neg\exists x_1(R_2(x_1) \wedge \neg\phi_3).$$

Every Alignment Calculus formula can be transformed into QDNF by repeated application of the following rewrite rules. (Logical shorthands like implication and the universal quantifier are expanded first so that the formula contains only the connectives '$\wedge$', '$\vee$', '$\neg$', and '$\exists$'.)

$$\begin{aligned} \neg\neg\phi &\longrightarrow \phi \\ \neg\left(\phi \overset{\wedge}{\vee} \psi\right) &\longrightarrow (\neg\phi) \overset{\vee}{\wedge} (\neg\psi) \\ \exists x(\phi \vee \psi) &\longrightarrow (\exists x\phi) \vee (\exists x\psi) \\ \exists x\phi &\longrightarrow \phi, \text{ where } x \notin \mathrm{free}(\phi) \end{aligned}$$

(2) $$\phi \wedge (\psi \vee \chi) \longrightarrow (\phi \wedge \psi) \vee (\phi \wedge \chi)$$

(3) $$\phi \wedge \exists x\psi \longrightarrow \exists y(\phi \wedge \psi^{x \mapsto y}), \text{ where } y \notin \mathrm{free}(\phi) \cup (\mathrm{free}(\psi) \setminus \{x\})$$

Rules (2) and (3) have additional symmetric variants with respect to the commutativity of '$\wedge$'. In addition, we consider '$\wedge$' and '$\vee$' implictly polyadic.

Strictly speaking, the rewrite rules above do not alone define a proper normal form, because different rule application orders lead to different final forms for the same formula to be normalized. Therefore we further define QDNF to be the form obtained by applying these rules in the order in which they are written. Intuitively, we always try to move negations down as far as they can go without crossing quantifiers.

3.2 Productive Formulae

Viewing the QDNF definition as a grammar, we attach two attributes to QDNF subformulae ϕ as follows. First, the inherited attribute required(ϕ) contains a finiteness dependency that the subformula ϕ is expected to fulfill. And second, the synthesized attribute productive(ϕ) is a Boolean value telling this required dependency indeed holds for ϕ. The attribute evaluation rules are then as follows.

- For the whole formula ϕ we define required$(\phi) = \emptyset \rightsquigarrow$ free(ϕ). The intuition is that ϕ must produce in a finite manner the values for all the answer variables free(ϕ). If productive$(\phi) = \mathsf{true}$, then ϕ is called *productive.* In addition, we will call productive any Alignment Calculus formula whose QDNF form is productive under this definition.

- If $\phi = \psi \vee \chi$, then we define

$$\begin{aligned}\text{required}(\psi) &= \text{required}(\chi) = \text{required}(\phi) \text{ and}\\ \text{productive}(\phi) &= \text{productive}(\psi) \wedge \text{productive}(\chi)\,.\end{aligned}$$

  That is, both disjuncts must meet the same requirement. Furthermore, by the definition of QDNF, this common requirement is the one posed in the previous case.

- If $\phi = \exists z \psi$ and required$(\phi) = X \rightsquigarrow Y$, then we define required$(\psi) = X \rightsquigarrow Y \cup \{z\}$, and productive$(\phi) =$ productive(ψ). That is, ψ must now produce z as well. (Note that $z \notin X \cup Y$, because variables were standardized apart.)

- If ϕ is finally a conjunction C of the form

$$\underbrace{\bigwedge_{i=1}^{p} a_i}_{\alpha_\phi} \wedge \bigwedge_{j=1}^{q} \neg b_j \wedge \bigwedge_{k=1}^{r} \neg D_r \tag{4}$$

  where the a_i and b_j are atomic formulae and the D_k are not, and required$(\phi) = X \rightsquigarrow Y$. Now we define productive(ϕ) to mean

$$\mathcal{B}_\phi \vdash \text{required}(\phi) \wedge \bigwedge_{k=1}^{r} \text{productive}(D_k)\,,$$

  where $\mathcal{B}_\phi = \bigcup_{i=1}^{p} \text{basis}(a_i)$, required$(D_k) = (X \cup Y) \cap$ free$(D_k) \rightsquigarrow \emptyset$, and '$\vdash$' denotes "follows by Armstrong's axioms" [32, Chapter 7.3].

  This complex rule is justified as follows. By definition, the burden of producing the variables X is on the closest similar conjunction ϕ' containing ϕ, or $X = \emptyset$ if such a ϕ' does not exist. The burden of producing the variables Y given X is on ϕ, and the available tools are α_ϕ; whether they suffice can be determined by Armstrong's axioms. If they do, each D_k is subsequently required to handle its own bound variables (which do exist, by the definition of QDNF) given $X \cup Y$.

  Note that in our case forming $\mathcal{B}_\phi$ explicitly is costly. Instead, the closure algorithm for functional dependecies [32, Algorithm 7.1] should be replaced with a version which uses heuristics for selecting the next a_i from α_ϕ to try.

For example, Formula (1) is productive by this definition. On the other hand, if the universal quantifier of Example 3 is replaced with an existential one, then the resulting formula is no longer productive. Its QDNF form would namely be

$$\exists x_1(R_2(x_2) \wedge \neg R_2(x_1)) \vee \exists x_1(R_2(x_2) \wedge \phi_3),$$

and there the left disjunct is unable to produce x_1.

Note that the conjunction α_ϕ of positive atoms in Equation (4) can satisfy additional limitation conditions not deducible from $\mathcal{B}_\phi$ by Armstrong's axioms. For example, if A in Example 4 is recursive, then ϕ_A satisfies the limitation property $\Gamma = \{x_1\} \rightsquigarrow \{x_2, x_3\}$ with $\mathcal{W}_{\phi_A}$ proportional to the time-space complexity of the algorithm used for deciding A. It is, however, not necessarily the case that $\Gamma \in \text{basis}(\phi'_A) \cup \text{basis}(\phi''_A)$. Moreover, Example 4 also shows that finding these additional limitation constraints of α_ϕ is undecidable in general. Even in the cases when this could be done, it causes similar evaluation order problems as the ones leading to the comprehensivity restriction in Section 2.1, because the constituent string formulae ϕ'_A and ϕ''_A should somehow be evaluated at the same time. Hence, we are content with $\mathcal{B}_\phi$, and call this the *conjunctivity*restriction. Note that it deals with limitation properties that emerge from the compositions of string formulae, while the comprehensivity restriction in Section 2.1 dealt dually with decompositions.

The productive formulae thus identified do indeed constitute a syntactic subclass of the domain independent Alignment Calculus formulae. The proof of this result is included as Appendix A.

Theorem 3.1 *Productive Alignment Calculus formulae are domain independent.*

3.3 Expressive Power of Productive Formulae

Here we consider what kinds of string database selection conditions can be expressed using productive formulae, and as such subsequently evaluated finitely. It turns out that these are closely related to the *polynomial-time hierarchy* **PH** [30].

First, the following result shows that any query in **PH** has a productive formulation as a right-restricted Alignment Calculus formula; that is, one containing only right-restricted string formulae.

Theorem 3.2 *For every language A in* **PH** *there exists a right-restricted pure Alignment Calculus formula ϕ_A such that $[\![\phi_A]\!] = A$, and the formula $\psi_A = R_1(x_1) \wedge \phi_A$ is productive, where* free$(\phi_A) = \{x_1\}$.

The proof of Theorem 3.2 rests on the earlier expressivity result that ϕ_A can be chosen from a syntactic subset of pure Alignment Calculus called the *quantifier limited* formulae [14, Theorem 5.5][21, Lemma 5.1]. As the name implies, this subset is such that every quantifier occurrence

$$\overset{\exists}{\forall}\, y_1, \ldots, y_n \left(\psi \overset{\wedge}{\to} \phi\right)$$

is explicitly paired with a right-restricted string formula ψ, whose finiteness property free$(\psi) \rightsquigarrow \{y_1, \ldots, y_n\}$ serves to bound the relevant domain of the quantifiers. These quantifier limiters ψ then serve to propagate downwards the limitedness spawned by the introduction of $\emptyset \rightsquigarrow \{x_1\}$ by $R_1(x_1)$ at the top level of ϕ_A.

A converse of Theorem 3.2 also holds.

Theorem 3.3 *If the formula $R_1(x_1) \wedge \phi$ is productive, where ϕ is a right-restricted pure Alignment Calculus formula with* $\mathrm{free}(\phi) = \{x_1\}$, *then $[\![\phi]\!]$ is a language in* **PH**.

This is because Theorems 2.1 and 2.2 provide polynomials as the starting point for the construction in Appendix A, which subsequently combines them without escaping polynomiality.

An interesting question for further study is whether some other syntactic subclass of Alignment Calculus could be developed to capture more queries than Theorems 3.2 and 3.3 indicate. One possibility would be to relax somehow the conjunctivity restriction posed in Section 3.2. Another, related possibility would be to solve some limitation problems not covered by Theorem 2.1 or 2.2 with superpolynomial limit functions. A third possibility would be to add a fixpoint construct to the language, for example developing a Datalog with string formulae [20] [35, Chapter 5.1], although the interplay of sequence production and fixpoint computation would have to be carefully designed, perhaps using a sophisticated safety analysis [18]. Finally, a fourth possibility would be to extend string formulae with stronger computational capabilities [19, 20], although in this case it might be problematic to maintain separate logical and operational semantics.

An alternative to our productive formulae would have been to employ the *embedded allowed (em-allowed)* formulae [7]. (In particular, the expressive power of Theorems 3.2 and 3.3 would not have been affected.) We did not, however, adopt them, because our situation is different in the following three remarks.

1. First, the em-allowed construction assumes that sets $\mathrm{basis}(\phi)$ for the atomic formulae ϕ are "freely" available for closure computations. In Alignment Calculus, however, testing whether $X \rightsquigarrow Y \in \mathrm{basis}(\phi)$ can be an expensive operation, so we try to minimize the number of such questions, as already noted in our discussion of Formula (4).

2. Second, the em-allowed construction loses finiteness information, because it is designed to minimize the amount of rewriting required to convert the original query into a bottom-up evaluable one. This namely leads to smaller algebraic expressions. As a result, the safe syntactic subclass is not as large as it could have been. For example, the query

$$x_1, x_2, x_3 : ((R_1(x_1) \wedge R_2(x_2)) \vee R_3(x_3)) \wedge \phi_1,$$

where ϕ_1 is from Example 1, is not em-allowed, because the finiteness information for the disjunction is essentially taken to be $\{\emptyset \rightsquigarrow \{x_1, x_2\}\} \cap \{\emptyset \rightsquigarrow \{x_3\}\} = \emptyset$. Our method, however, considers this query safe, because it can be rewritten first as

$$x_1, x_2, x_3 : (R_1(x_1) \wedge R_2(x_2) \wedge \phi_1) \vee (R_3(x_3) \wedge \phi_1),$$

which yields $\{\emptyset \rightsquigarrow \{x_1, x_2, x_3\}\}$ from both disjuncts.

3. Third, and most important, minimizing rewriting as in Remark 2 above can lead to unnecessary restructuring. Em-alloweness was originally developed in the context of adding calls to the host programming language into relational calculus queries, and there the cost of eventually calling these functions is unavoidable. In our case, however, the cost of evaluating an expression depends dramatically on the amount of restructuring performed, as the following example illustrates.

Example 7 Consider the query

$$x_1, x_2, x_3 : (R_1(x_3) \wedge \phi_1 \vee R_2(x_2, x_3) \wedge \phi_3)) \wedge R_3(x_1, x_2),$$

whose string formulae ϕ_i come from Examples i. This formula is em-allowed, and can indeed be evaluated in a bottom-up fashion "as is". The first disjunct satisfies $\Gamma = \emptyset \rightsquigarrow \{x_1, x_2, x_3\}$ by restructuring x_1 and x_2 from the x_3 in R_1. In other words, every string in r_3 is divided into a prefix and a suffix in all possible ways. The second disjunct also satisfies Γ, this time by restructuring x_1 from the x_2 in the first column of R_2. In other words, x_1 receives as its values all the strings that are at most as long as the value of x_2.

However, applying (the commutative variant of) Rule (2) once more leads into a QDNF form, where no restructuring at all is required. In particular, the *exponential*restructuring in the second disjunct is eliminated.

3.4 Translation of Productive Formulae

Our aim is to translate productive Alignment Calculus formulae into Alignment Algebra for evaluation, but general Alignment Algebra expressions are not easy to evaluate due to the presence of the domain symbol Σ^*. Hence, the target of the translation is the subset of *finitely evaluable* expressions, defined inductively as follows. An expression E of Alignment Algebra is finitely evaluable if and only if every occurrence of Σ^* in E is within a selection expression $\sigma_{\mathcal{A}}(F \times (\Sigma^*)^l)$, where the expression F of arity k is again finitely evaluable, and the $(k+l)$-tape FSA $\mathcal{A}$ satisfies the limitation property $\{1, \ldots, k\} \rightsquigarrow \{k+1, \ldots, k+l\}$. (The meaning of limitations properties for FSAs carries naturally over from string formulae by their equivalence [13, Theorems 3.1 and 3.2].) Note that these finitely evaluable expressions are indeed compatible with the bottom-up evaluation method advocated in Section 2.1 above.

The translation of productive Alignment Calculus formulae in QDNF into equivalent finitely evaluable Alignment Algebra expressions is straightforward, once the following auxiliary result is established (see [7, Lemma 7.10]).

Lemma 3.1 *Let $\alpha = a_1 \wedge \ldots \wedge a_p$ be a conjunction of atomic formulae. Then the condition $\bigcup_{i=1}^{p} \mathrm{basis}(a_i) \vdash \Gamma = X \rightsquigarrow \mathrm{free}(\alpha)$ holds if and only if α can be reordered into $a'_1 \wedge \ldots \wedge a'_p$ such that $\bigcup_{j=1}^{q} \mathrm{basis}(a'_j) \vdash X \rightsquigarrow \mathrm{free}(a'_1 \wedge \ldots \wedge a'_q)$ holds for all $1 \leq q \leq p$.*

Note that Lemma 3.1 fails if the comprehensivity restriction discussed in Section 2 is dropped. Note also that this ordering is implicitly generated during the heuristic closure computation in Formula (4).

Applying Lemma 3.1 to Formula (4), we see that Formula (4) can indeed be translated into an equivalent finitely evaluable Alignment Algebra expression, if a variant of the *Sideways Information Passing Strategy (SIPS)* [33, Chapter 12.6] is employed. In fact, Appendix A provides an example of our strategy in its construction of explicit limitation functions for productive formulae. The translation of productive formulae to finitely evaluable expressions is very similar, and also outlined in Appendix A.

For example, Formula (1) is translated into the finitely evaluable Alignment Algebra expression

$$r_2 \setminus \pi_{\$2}\left((r_2 \times r_2) \setminus \sigma_{\mathcal{A}_3}(r_2 \times r_2)\right),$$

where $\mathcal{A}_3$ is the (unidirectional) 2-tape FSA corresponding to the string formula ϕ_3. The SIPS variant therefore implicitly rewrote Formula (1) into

$$(5) \qquad R_2(x_2) \wedge \neg\exists x_1(\underline{R_2(x_2)} \wedge R_2(x_1) \wedge \neg\phi_3)$$

before translation, where the underlined part was *copied* (and not moved) from outside to inside the quantifier range. These implicit copyings are similar to the explicit transformations preceding the translation of an em-allowed formula into an algebraic expression [7, Transformation T15].

Choosing the reordering in Lemma 3.1 wisely is of paramount importance in Alignment Calculus, because intermediate results can have exponential size, as noted in Example 7. In fact, no reorderings are saved in Example 7 at all, unless the resulting disjuncts are reoredered to apply the string formulae ϕ_i last. This problem is similar to optimizing left-deep joins with user-defined predicates [2], except that our predicates not only test existing relation columns but also create new ones, and their cost is heavily (even exponentially) influenced by the number of created columns. In other words, we are adding *user-defined set-valued functions*, whose costs depend on the expected sizes of their values.

Finally we remark that some algebraic optimizations [33, Chapter 11.6] involving selections are no longer feasible in Alignment Algebra, because subclasses of string formulae are not necessarily closed under the required logical operation. In particular, cascading unidirectional selections $\sigma_{\mathcal{A}}(\sigma_{\mathcal{A}'}(E \times \Sigma^*)) = \sigma_{\mathcal{A}''}(E \times \Sigma^*)$ may result in a condition $\mathcal{A}''$ that is not right-restricted [21, Corollary 5.3], and hence might be difficult to evaluate. These questions are again related to the comprehensivity and conjunctivity restrictions of Sections 2.1 and 3.2.

3.5 Finite Evaluation of Expressions

Now that the translation process explained above has produced a finitely evaluable Alignment Algebra expression, how can the actual finite evaluation be done? This problem reduces to actually evaluating a finitely evaluable expression $E = \sigma_{\mathcal{A}}\left(F \times (\Sigma^*)^l\right)$, where the expression F of arity k has already been evaluated to yield a finite value f. Moreover, we would like to evaluate E without resorting to the global domain independence function of the whole query from Definition 1.1, as already mentioned in Section 2.1.

This goal is attained as follows. Intuitively, $\mathcal{A}$ can now be considered a kind of an extended automaton $\mathcal{A}'$ with k input and l *output* buffers; a kind of an extended Mealy machine, as it were [17, Chapter 2.7], for performing the requested sequence restructuring. Alignment Calculus does not, however, force the database programmer to think in these operational terms while writing queries; instead, the query translator assigns these roles to variables. Note also that one of these buffers can be read in both directions, in contrast to other restructuring automaton models in the literature, where they are regarded as read-only channels [12, 20]. Then this $\mathcal{A}'$ is simulated on every input $\vec{u} = \langle u_1, \ldots, u_k\rangle \in f$, and the output(s) $\vec{v} = \langle v_1, \ldots, v_l\rangle$ recorded as entries $\langle u_1, \ldots, u_k, v_1, \ldots, v_l\rangle$ into the result relation e of the expression E.

One such simulation is a finite operation, unless $\mathcal{A}'$ starts looping while producing more and more output. By the limitation property $\Gamma = \{1, \ldots, k\} \rightsquigarrow \{k+1, \ldots, k+l\}$ of $\mathcal{A}$, which was required for the finite evaluability of expression E, it is known that once such voluminous looping has produced "too much" output, $\mathcal{A}'$ will eventually reject. The question is, however, how much is too much? The answer is supplied by the limitation function $\mathcal{W}_{\mathcal{A}} : \mathbb{N}^k \to \mathbb{N}$ associated with $\mathcal{A}$ and Γ: as soon as the current simulation of $\mathcal{A}'$ on input $\vec{u}$ outputs more than $\mathcal{W}_{\mathcal{A}}(|u_1|, \ldots, |u_k|)$ characters into some output buffer $k+1 \leq j \leq k+l$, this simulation can be terminated as ultimately unsuccessful. Hence, a finite simulation strategy does indeed exist. Note also that this strategy used local instead of global limitation information.

Moreover, when none of the output buffers $k+1, \ldots, k+l$ is bidirectional, the simulation can be carried out without any explicit limitation functions at all. Namely, a straightforward algorithm can produce from $\vec{u}$ and $\mathcal{A}$ an l-FSA $\mathcal{B}$ accepting exactly those $\vec{v}$ the simulation would produce. Moreover, Γ ensures that this $\mathcal{B}$ can be chosen not to have any loops. Therefore these $\vec{v}$ can also be extracted from $\mathcal{B}$ in a topological order.

On the other hand, the evaluation of E seems to become much more complicated, when one of the output buffers is bidirectional. A 2-FSA $\mathcal{A}$ with one unidirectional input buffer 1 and one bidirectional output buffer 2 can namely be considered a *nondeterministic checking stack automaton* $\mathcal{C}$ [15, Definition 1.4], which on input u first guesses a stack v to check but not modify, and then verifies that $\mathcal{A}$ would accept u and v. Then known complexity results [11, Problem AL5] lead directly to the following complexity results.

1. If both $\mathcal{A}$ and u are given as input (that is, an *ad hoc* Alignment Calculus query is being evaluated), then it is **PSPACE**-complete to tell whether acceptable outputs v exist.

   (An ad hoc query will also have incurred the additional normalization, domain independence analysis, and translation costs.)

2. If $\mathcal{A}$ is fixed beforehand (that is, a *precompiled* query is being evaluated), then determining the existence of acceptable v can be **NP**-complete in the worst case.

Although it remains to be seen whether the extra limitation condition $\{1\} \rightsquigarrow \{2\}$ on $\mathcal{A}$ ensured by finite evaluability can ameliorate the situation, it does seem prudent to avoid bidirectional output if at all possible. This provides one guiding principle for selecting good reorderings in Section 3.4.

4 Conclusion

We have discussed the implementation aspects of Alignment Calculus, a declarative string database query language based on modal logic. In particular, we defined the productive formulae, a syntactic subclass of the domain independent Alignment Calculus formulae, and analysed the expressive power of these formulae. We also indicated how these productive formulae can be translated into a subset of Alignment Algebra, the finitely evaluable expressions, which in turn provided us with computational tools for performing the string restructuring operations specified by the formulae in a bottom-up manner. Therefore, the construction provides an explicit implementation guideline for our declarative string database query language.

References

[1] N. Balkir, E. Sukan, G. Ozsoyoglu, and Z. Ozsoyoglu. VISUAL – a graphical icon-based query language. In *IEEE International Conference on Data Engineering*, pages 524–533, 1996.

[2] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. In *Very Large Data Bases Conference*, pages 87–98, 1996.

[3] 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.

[4] J. Collado-Vides. Grammatical model of the regulation of gene expression. *Proceedings of the National Academy of Sciences of the USA*, 89:9405–9409, 1992.

[5] C. Date. *An Introduction to Database Systems.* Addison-Wesley, 6th edition, 1994.

[6] S. Dong and D. Searls. Gene structure prediction by linguistic methods. *Genomics*, 23:540–551, 1994.

[7] M. Escobar-Molano, R. Hull, and D. Jacobs. Safety and translation of calculus queries with scalar functions (extended abstract). In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems*, pages 253–264, 1993.

[8] 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.

[9] 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.

[10] European Molecular Biology Network (EMBNET). *Strategies in Bioinformatics: a European Perspective*, 1994.

[11] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, 1979.

[12] S. Ginsburg and X. Wang. Pattern matching by rs-operations: Towards a unified approach to querying sequenced data (extended abstract). In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems*, pages 293–300, 1992.

[13] G. Grahne, M. Nykänen, and E. Ukkonen. Reasoning about strings in databases. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems*, pages 303–312, 1994.

[14] G. Grahne, M. Nykänen, and E. Ukkonen. Reasoning about strings in databases. *Journal of Computer and System Sciences*, 1997. To appear. Abstracted in [13].

[15] S. Greibach. Checking automata and one-way stack languages. *Journal of Computer and System Sciences*, 3:196–217, 1969.

[16] C. Helgesen and P. Sibbald. PALM - a pattern language for molecular biology. In *International Conference on Intelligent Systems in Molecular Biology*, pages 172–180, 1993.

[17] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[18] R. Krishnamurthy, R. Ramakrishnan, and O. Shmueli. A framework for testing safety and effective computability. *Journal of Computer and System Sciences*, 52:100–124, 1996.

[19] F. Lefebvre. A grammar-based unification of several alignment and folding algorithms. In *International Conference on Intelligent Systems in Molecular Biology*, pages 143–155, 1996.

[20] G. Mecca and A. Bonner. Sequences, Datalog and transducers. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems*, pages 23–35, 1995.

[21] M. Nykänen. Alignment Calculus: a modal logic for querying string databases. Technical Report C–1996–97, Department of Computer Science, University of Helsinki, Finland, 1996.

[22] R. Ramakhrisnan, F. Bancilhon, and A. Silberschatz. Safety of recursive Horn clauses with infinite relations (extended abstract). In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems*, pages 328–339, 1987.

[23] J. Richardson. Supporting lists in a data model (a timely approach). In *Very Large Data Bases Conference*, pages 127–138, 1992.

[24] H. Rogers. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, 1967.

[25] D. Rosenblueth, D. Thieffry, A. Huerta, H. Salgado, and J. Collado-Vides. Syntactic recognition of regulatory regions in *Escherichia Coli*. *Computer Applications in the Biosciences*, 12(5):415–422, 1996.

[26] D. Searls. String Variable Grammar: A logic grammar formalism for the biological language of DNA. *Journal of Logic Programming*, pages 73–102, 1995.

[27] D. Searls and K. Murphy. Automata-theoretic models of mutation and alignment. In *International Conference on Intelligent Systems in Molecular Biology*, pages 341–349, 1995.

[28] P. Seshadri, M. Livny, and R. Ramakrishnan. SEQ: A model for sequence databases. In *IEEE International Conference on Data Engineering*, pages 232–239, 1995.

[29] P. Seshadri, M. Livny, and R. Ramakrishnan. The design and implementation of a sequence database system. In *Very Large Data Bases Conference*, pages 99–110, 1996.

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

[31] B. Subramanian, T. Leung, S. Vandenberg, and S. Zdonik. The AQUA approach to querying lists and trees in object-oriented databases. In *IEEE International Conference on Data Engineering*, pages 80–89, 1995.

[32] J. Ullman. *Principles of Database and Knowledge-Base Systems*, volume I. Computer Science Press, 1988.

[33] J. Ullman. *Principles of Database and Knowledge-Base Systems*, volume II. Computer Science Press, 1989.

[34] A. van Gelder. Deriving constraints among argument sizes in logic programs. Technical Report UCSC-CRL-89-41, University of Califormia, Santa Cruz, 1989.

[35] X. Wang. *Pattern Matching by Rs-Operations: Towards a Unified Approach to Querying Sequenced Data.* PhD thesis, University of Southern Carolina, 1992.

[36] A. Weber. On the lengths of values in a finite transducer. *Acta Informatica*, 29:663–687, 1992.

[37] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56:72–99, 1983.

A Proof of Theorem 3.1

Here we give a proof for Theorem 3.1. It consists of an inductive way to combine the limitation functions of the atomic formulae into limitation functions for progressively larger and larger nonatomic formulae. The gist of the construction is the following technical lemma, where $\sigma_A(\theta x)$ denotes the string on the row θx of the alignment A.

Lemma A.1 *If ϕ satisfies* required$(\phi) = \{x_1, \ldots, x_m\} \rightsquigarrow Y$ *then there exists a limit function $\mathcal{W}_\phi\colon DB \times \mathbb{N} \to \mathbb{N}$ with the following two properties:*

1. *If $\langle A_0, db\rangle \models \phi\theta$ then*

$$|\sigma_{A_0}(\theta z)| \leq \mathcal{W}_\phi(db, \max\{|\sigma_{A_0}(\theta x_i)| : 1 \leq i \leq m\})$$

for all $z \in \{x_1, \ldots, x_m\} \cup \mathrm{free}(\phi)$. In other words, the lengths of the strings assigned to the free variables of ϕ are bounded by the lengths of the strings assigned to $x_1, \ldots, x_m$.

2. *If $\ell \geq \mathcal{W}_\phi(db, \max\{|\sigma_{A_0}(\theta x_i)| : 1 \leq i \leq m\})$ then the statements $\langle A_0, db\rangle \models \phi\theta$ and $\left\langle A_0^\ell, db\right\rangle \models \phi\theta$ are equivalent. In other words, the lengths of the strings assigned to the bound variables of ϕ are also bounded by the lengths of the strings assigned to $x_1, \ldots, x_m$.*

Proof: If $\phi = \exists z\psi$, then required$(\psi) = \{x_1, \ldots, x_m\} \rightsquigarrow Y \cup \{z\}$ is satisfied; hence induction provides a limit function $\mathcal{W}_\psi\colon DB \times \mathbb{N} \to \mathbb{N}$. An examination of the truth definition [14, Section 2][21, Section 3] shows that $\mathcal{W}_\phi = \mathcal{W}_\psi$ suffices.

Let ϕ then be as in Formula (4). It follows from the definition of productivity that free$(\alpha_\phi) \setminus \{x_1, \ldots, x_m\} = Y$, so we can assume α_ϕ ordered with respect to $\{x_1, \ldots, x_m\}$ by Lemma 3.1. Then we produce limit functions $\mathcal{W}_{\alpha_s}\colon DB \times \mathbb{N} \to \mathbb{N}$ for its prefixes $\alpha_q = a_1 \wedge \ldots \wedge a_q$ by induction on q as follows.

We can consider the empty prefix α_0 to consist of an implicit occurrence of the identically true empty string formula λ with free$(\lambda) = \emptyset$. Choosing $\mathcal{W}_{\alpha_0}(db, \ell) = \ell$ suffices.

Perform the partition

$$U = \mathrm{free}(\alpha_{q+1}) \cap (\mathrm{free}(\alpha_s) \cup \{x_1, \ldots, x_m\}) \text{ and}$$
$$V = \mathrm{free}(\alpha_{q+1}) \setminus U.$$

If $V = \emptyset$, then choosing $\mathcal{W}_{\alpha_{q+1}} = \mathcal{W}_{\alpha_q}$ suffices, because a_{q+1} introduced no new variables. Otherwise there are two possibilities to introduce them.

If $a_{q+1} = R(\ldots)$ for a relation symbol R, the new variables V introduced here receive their values from r; choosing

$$\mathcal{W}_{\alpha_{q+1}}(db, \ell) = \max(\mathcal{W}_{\alpha_q}(db, \ell), \max(r, db)), \text{ where}$$
$$\max(r, db) = \max\{|u| : \langle \ldots, u, \ldots\rangle \in r\})$$

accommodates this.

Otherwise a_{q+1} is a string formula such that $U \rightsquigarrow V \in$ basis(a_{q+1}) and the variables V are now introduced here by using the already introduced variables U. This implies the existence of a limit function $\mathcal{W}_{a_{s+1}}\colon \mathbb{N}^{|U|} \to \mathbb{N}$ limiting the lengths of the strings assigned to the output variables V given the lengths of the strings assigned to the input variables U, as defined in Definition 2.1. In particular, these functions can be provided by Theorems 2.1 and 2.2, depending on what kind of a string formula a_{q+1} is. Without loss of generality we can choose this $\mathcal{W}_{a_{q+1}}$ to satisfy $\mathcal{W}_{a_{q+1}}(\ell_1, \ldots, \ell_{|U|}) \geq \max(\ell_1, \ldots, \ell_{|U|})$. Function $\mathcal{W}_{\alpha_q}$ in turn limits the lengths of the strings assigned to V given the lengths of the strings assigned to $x_1, \ldots, x_m$; therefore choosing

$$\mathcal{W}_{\alpha_{q+1}}(db, \ell) = \mathcal{W}_{a_{q+1}}(\mathcal{W}_{\alpha_q}(db, \ell), \ldots, \mathcal{W}_{\alpha_q}(db, \ell))$$

suffices.

free$(\beta_\phi) \subseteq \{x_1, \ldots, x_m\} \cup Y$ follows from the definition of productivity. Hence, the $\mathcal{W}_{\alpha_\phi}$ constructed above suffices unmodified also as $\mathcal{W}_{\alpha_\phi \wedge \neg b_1 \wedge \ldots \wedge \neg b_q}$. If $r = 0$, the base case of the induction is complete.

Otherwise denote the left-hand side of required(D_k) for a subformula D_k as $W_k = (\{x_1, \ldots, x_m\} \cup Y) \cap \mathrm{free}(D_k)$. By induction the function $\mathcal{W}_{D_k}\colon DB \times \mathbb{N} \to \mathbb{N}$ limits the lengths of values assigned to the bound variables of D_k (which do exist, by definition of QDNF) given the lengths of the values for W_k, which in turn are limited by the $\mathcal{W}_{\alpha_\phi}$ constructed above given the lengths of the values assigned to $x_1, \ldots, x_m$. Therefore choosing

$$\mathcal{W}_\phi(db, \ell) = \max\left\{\mathcal{W}_{D_k}(db, \mathcal{W}_{\alpha_\phi}(db, \ell)) : 1 \leq k \leq, r\right\}$$

suffices. ■

Given Lemma A.1, the proof of Theorem 3.1 is as follows.

Proof: Let $\phi = D_1 \vee \ldots \vee D_s$ be productive. By definition, each D_l satisfies required$(D_l) = \emptyset \rightsquigarrow$ free(ϕ). Hence, Lemma A.1 yields a function $\mathcal{W}_{D_l}\colon DB \to \mathbb{N}$, which limits the lengths of all the answer variables free(ϕ) given the current database db. That is, D_l is domain independent by Definition 1.1. Then the disjunction ϕ of domain independent formulae D_l with identical free variables free(ϕ) is also domain independent itself with

$$\mathcal{W}_\phi(db) = \max\{\mathcal{W}_{D_k}(db) : 1 \leq l \leq r\}$$

as its corresponding limit function. ■

The translation of a productive QDNF formula into a finitely evaluable expression can be performed by a recursive algorithm, whose behaviour follows the inductive construction in the proof of Theorem 3.1 above. However, in contrast to the usual translation methods of relational calculus into the algebra [7, Section 7.3][32, Theorem 3.11], this algorithm takes as input not only the subformula ϕ to translate, but also a finitely evaluable expression E. The role of this E is to provide the values for the variables X appearing in the left side of the corresponding inherited finiteness dependency required$(\phi) = X \rightsquigarrow Y$. Hence, this E passes variable binding information sideways [33, Chapter 12.6] into the recursion.

In particular, a string formula a_{q+1} is translated only after the current expression E can produce values for the whole right side U of some finiteness dependency $U \rightsquigarrow V \in$ basis(a_{q+1}). Then its translation can be chosen finitely evaluable as well.

For example, the quantified subformula in Formula (5) received the translation $E = r_2$ instead of the actual underlined copy, which then provided the values for the variable $X = \{x_2\}$.