Department of Computer Science
Series of Publications A
Report A-1997-3

Querying String Databases with Modal Logic

Matti Nykänen

University of Helsinki
Finland

Department of Computer Science
Series of Publications A
Report A-1997-3

Querying String Databases with Modal Logic

Matti Nykänen

To be presented, with the permission of the Faculty of Science of the University of Helsinki, for public criticism in the Auditorium, Teollisuuskatu 23, Helsinki, on November 28th, 1997, at 10 o'clock.

University of Helsinki
Finland

Contact information

Postal address:
Department of Computer Science
P.O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki
Finland

Email address: postmaster@cs.Helsinki.FI (Internet)

URL: http://www.cs.Helsinki.FI/

Telephone: +358 9 708 51

Telefax: +358 9 708 44441

ISSN 1238-8645
ISBN 951-45-7859-7
Computing Reviews (1991) Classification: F.1.1, F.4.3, H.2.3
Helsinki 1997
Helsinki University Printing House

Querying String Databases with Modal Logic

Matti Nykänen

Department of Computer Science
P.O. Box 26, FIN-00014 University of Helsinki, Finland
`matti.nykanen@cs.helsinki.fi`
`http://www.cs.helsinki.fi/~mnykanen`

PhD Thesis, Series of Publications A, Report A-1997-3
Helsinki, November 1997, 151 pages
ISSN 1238-8645, ISBN 951-45-7859-7

Abstract

New application areas for databases demand the possibility of managing not only the traditional atomic but also structured data types. One type of this kind is a finite sequence of characters drawn from a finite alphabet. These string databases are important for example in molecular biology data management, because they can for instance represent DNA sequences directly as strings from alphabet $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. Then the query language must be able to manipulate strings both as indivisible entities and as ordered sequences of distinct characters, perform pattern matching in strings, compare several strings with each other, and generate new strings not yet in the database. This work presents Alignment Calculus, a new modal logic extension of the relational calculus, which satisfies all these requirements with a new unified string manipulation formalism. This formalism is based on the concept of multiple alignment of several strings; computational molecular biology employs an analogous concept. In the language, alignments are described logically with string formulae, a new extension of linear-time temporal logic into multiple strings. The abstract expressive power of Alignment Calculus is shown to equal the Arithmetical Hierarchy. The work develops also a syntactically safe sublanguage, whose queries require only finite computational resources. This sublanguage is constructed by first identifying a decidable subcase of string formula safety analysis, even though the general problem is shown to be undecidable. This safe sublanguage is shown to equal the Polynomial-time Hierarchy in its expressive power, and therefore to capture all the string queries occurring in practical situations. The operational counterpart of Alignment Calculus, Alignment Algebra, is developed by replacing the selection operator of relational algebra with one employing multitape nondeterministic finite state automata corresponding to the aforementioned string formulae, and adding an explicit domain symbol. The aforementioned safe sublanguage has also a counterpart in this algebra: expressions, in which all the domain symbol occurrences are restricted by immediately enclosing automata. A finite evaluation strategy is developed for these expressions.

Computing Reviews (1991) Categories and Subject Descriptors:

F.1.1 [Computation by Abstract Devices] Models of Computation – *Automata*

F.4.3 [Mathematical Logic and Formal Languages] Formal Languages – *Classes Defined by Grammars or Automata*

H.2.3 [Database Management] Languages – *Query Languages*

General Terms:

Automata, Logic, Databases

Additional Key Words and Phrases:

multitape automata, modal logic, string databases, computational biology

Contents

Acknowledgements

I am most grateful to my advisors, Professor Esko Ukkonen and Associate Professor Gösta Grahne, who have patiently taught me how to do Science. Science is namely one of the few remaining vocations, where the apprentice is still instructed by the Masters.

Docent Heikki Lehväslaiho made me aware of the big picture in the whole bioinformatics field. Perhaps we can someday explore it together. Raul Hakli has endured many seemingly endless discussions on various subjects, even when they strayed far away from our common field of study. Going back in time, Timo Kovakoski has planted the seeds of this work by showing me languages as formal rule-based systems; by teaching me the Latin, he paradoxically invoked my interest in logic and computation. This interest was subsequently nurtured further by Professors Heikki Mannila and Pekka Orponen.

I am grateful to the Head of Department, Professor Martti Tienari, and to all the colleagues at the Department, with whom I have had the pleasure to work. I would like to thank Päivi Kuuppelomäki and Greger Lindén in particular, even though organizing the SCAI'97 conference together was strictly speaking not a part of *this* particular work. Although I am not the most easy-going person to deal with, at times acting as if I just swallowed a 6ft 6in long slide rule, I thank you all for your tolerance of my quirks.

My deepest gratitude is due to my family, however: my wife Erja, and my daughters Emma and Venla. My parents Leena and Kyösti have also unfailingly supported me – both financially and otherwise – throughout these years.

Financial support from the Helsinki Graduate School of Computer Science and Engineering (HeCSE), from the Academy of Finland (grant number 22856), and from the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) is gratefully acknowledged.

List of Figures

List of Algorithms

Chapter 1

Introduction

As we all know, there is a a kind of lazy pleasure in useless and out-of-the-way erudition.

Jorge Luis Borges [14]

In this work we focus on the problem of strings in databases. Strings are perhaps the most elementary example of a data type with both an external and an internal structure: in some data manipulation operations one wants to treat them as indivisible entities, while in other contexts access to individual characters and their relative order is required. The development and study of capabilities for manipulating these multifaceted items has been advocated as a central challenge in the database research of the current decade [83, 84].

Our primary source of motivation is the storage and qualitative processing of genetic information, represented as sequences of symbols. For instance, the theory of gene regulation explores the combinatorial or grammatical structure inherent in genetic sequences, as opposed to their statistical properties. This structure can be quite complex, since gene regulation involves non-context-free dependencies between different parts of a string [20]. These dependencies should be explicitly expressible, as they constitute knowledge about the family of strings that the current database represents. Such biological motivation suggests our work to have connections with the broad paradigm called *computational science*, which explores the new research possibilities in the natural sciences arising from advances in both theoretical and practical computational capabilities [95]. Indeed, the connection between computational and computer science is believed to be vital for the well-being of not only the former but also the latter [25, 69].

The starting point of this work is the well-known *relational data model* [2, Chapter 3][24, Part II][92, Section 2.3]. Intuitively, this model organizes data into rectangular tables, where the number of columns, or arity, is fixed.

In contrast, the number of rows varies, but remains finite. Each data item is represented as one table row, or tuple, and the values of its attributes are recorded as entries in the corresponding columns. An example of a simplified university telephone directory in this format is given below.

PREFIX	
Dept.	*Area*
CS	708
Pathology	191
⋮	

PERSON		
Name	*Dept.*	*Ext.*
Doe, John	Pathology	43210
Kylmä, Kalle	Pathology	43210
Nykänen, Matti	CS	44237
	⋮	

For example, dialing extension 43210 suffices when calling John Doe from within the campus, but his departmental prefix 191 must be supplied otherwise.

A basic objective of relational database design is to choose the table structure to be in *first normal form (1NF)* [2, page 265][24, page 269]; intuitively, all entries must be *atomic*, or without further internal structure. In other words, the organization of the data into tables must make all the inherent structure of the data being modelled explicitly available as intertabular dependencies, such as “a full telephone number consists of an area code and extension with equal departments”. We chose to separate the area codes from the extensions, in anticipation of queries like “list all the persons, whose area code is 708” involving parts of telephone numbers. On the other hand, we chose not to break the telephone numbers into smaller pieces still, because we do not anticipate queries like “list all the persons, whose extension has ‘2’ as its third digit”.

This anticipated query can be written in *the relational calculus*, the logical counterpart to the relational model [2, Section 5.3][24, Section 7.4][92, Section 3.8], which is based on *First-Order Logic (FO)* [94], as the formula

$$\exists y \exists z . \mathrm{PERSON}(x, y, z) \wedge \mathrm{PREFIX}(y, 708),$$

or “the names x of the persons, who work at some department y having prefix 708”.

However, some application areas may involve types of data with internal structure, which cannot be normalized to attain 1NF by the database designer. For example, DNA sequences are represented as strings drawn from the alphabet $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. The database designer cannot normalize this representation to ensure atomicity, because (s)he does not – and cannot – know beforehand, which properties of these sequences will eventually emerge as significant. (Indeed, the division of numbers – strings of digits – chosen in our telephone directory example would seem quite capricious to someone, who did not know that they denote telephone numbers.) What is needed, then, is an extension of the data model for dealing with these strings, and in particular an orderly, yet flexible way to access their internal structure when necessary. We attain this goal by extending the relational calculus with modal string-handling formulae.

We therefore extend the relational data model to include finite strings over some fixed finite alphabet Σ as primary objects of information. A relation of arity k is in our model a finite subset of the k-fold Cartesian product of Σ^*, the set of all finite strings in alphabet Σ. 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. We emphasize that this alphabet Σ is chosen once and for all when the database is designed; the alphabet does not gain new members as data is being added to the database. This is in contrast with *list* database languages [19, 38, 45, 65, 70, 81, 82, 89, 100], where the "alphabet" grows to accommodate new data items as they arrive. Hence, our proposal focuses on application areas like the aforementioned genetic informatics, where the alphabet can be decided beforehand.

Present-day database query languages offer little support for string relations. For example, the *Sequence Retrieval System (SRS)* [32] program, which has recently gained popularity among molecular biologists, allows the database administrator to draw links from one preformatted data file to another [33], but only on its non-sequence fields. That is, navigation in the database is possible, but only via its non-sequence fields. Not being able to exploit perhaps the most important field of a database entry is hardly a satisfactory long-term solution. This is an example of a more general problem: most present-day molecular biology databases are being built on top of commercial Relational Database Management Systems (RDBMS) with no built-in support for biological types [34, Chapter 4], while the CD-ROM libraries delivered to customers are essentially snapshots of their contents in flat file format. Consequently, their indexing facilities must be augmented with extraneous tools operating outside the database system in order to attain the desired sequence processing capabilities. On the other hand, our proposal does incorporate string comparisons according to user-defined criteria.

Biology database designers have been considering Object-Oriented Database Management Systems (OODBMS) [24, Part VI] as a remedy for the shortcomings of RDBMS [34, Chapter 4]. The main (but not only [76]) rationale seems to have been their extensibility with new data types like sequences and trees [89] required for biological data, rather than preferring object-oriented to relational database modeling and design methods [34, Chapter 4]. On the other hand, the same capabilities could also be achieved – and perhaps more elegantly – by adding these application-specific domains into the relational model [24, Chapter 25], as we have done here. Finally, 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 [9].

What kinds of string handling operations should a string database query language support? We suggest the following three types of queries as a yardstick for sizing up string query languages.

Pattern matching queries consist of a given pattern expressed in some notation, such as formal grammars [49, Chapter 9]. The desired result is to extract from the database all the strings that match the given pattern according to the rules of the pertinent grammar class.

An example of such a query would be "list all tuples of relation r where the second component is of the form $(\mathtt{gc} + \mathtt{a})^*$". Here the pattern notation is the regular expressions [49, Chapter 2.5]. Example 2.6 presents how this query can be expressed in our language.

All string database query languages support some form of pattern matching, although methods vary.

String comparison queries deal with two (or more) strings at the same time, and the aim is to test whether they satisfy a given comparison predicate.

The simplest comparison concept is equality; Example 2.2 presents it in our language.

Not all string database query languages are designed to handle multiple strings at a time. Furthermore, the available string comparison predicates can either be predefined in the query language or the user may be supplied with a language for defining new comparison predicates; this latter option offers more flexibility.

String restructuring queries specify a mapping f from (perhaps multiple) strings to (perhaps sets of multiple) strings, and ask for all the results $f(u)$ of this mapping for all strings u in the database.

One might for example want to translate a database of strings over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ by switching the letters to each other. Example 2.12 presents this mapping in our language.

> Note that the results $f(u)$ are not required to appear in the database; instead, they are to be *generated* by the query evaluation process. If a query language supports comparison, then it generally supports restructuring as well, although methods vary.

Our language proposal accommodates all these three types within a single unified string handling concept. We namely offer a language for defining new string comparison predicates. When restricted to pattern matching, or "comparison" of a single string, this class of definable predicates reduces essentially to the regular expressions. On the other hand, restructuring capabilities are attained within the same framework by allowing comparisons between already known and as yet unknown strings; then the query evaluator must find suitable suitable values for the missing strings so that the specified comparison succeeds.

Our string comparisons are based on the concept of *aligning multiple strings together*. Multiple string alignment is also a central concept of (computational) molecular biology [46, Chapter 14]. However, our alignment concept is *grammatical*: the sequence of character-to-character comparisons that make up the complete alignment must follow a grammatical (regular, to be more exact) pattern supplied by the user.

Example 1.1 Consider the alignments of three strings, where the first one is the concatenation of the second and third. The following picture presents a small alignment of this kind.

```
gcaat
gc---
--aat
```

In the mindset of our proposal, this class of alignments could be described as follows: "First check repeatedly that the next character of the top string equals the next character on the middle string. Then check repeatedly that the next character of the top string equals the next character on the bottom string. Finally succeed if and only if you have reached the end of all the strings." That is, we describe the series of movement and comparison operations to make, and restrict our attention to regular operation series only. Example 2.3 will show in detail how to express this class in our proposed language.

If we then want to concatenate two strings v and w, we ask for the strings u with the following property: when u is the top, v the middle, and w the bottom string, then there exists some operation series of the specified kind, which successfully aligns these three strings. ■

The remainder of this work is structured as follows. Section 1.1 discusses briefly the current data management situation in bioinformatics, our primary application area. Section 1.2 comments on some related language proposals found in the literature. Chapter 2 gives a detailed account of our proposed declarative language called the *Alignment Calculus*, while Chapter 3 develops a suitable procedural counterpart called the *Alignment Algebra*. Chapter 4 examines under what conditions can query evaluation be guaranteed to remain finite; note for instance that if we provide only string w in Example 1.1 and ask for u and v, there are infinitely many suitable choices. Then Chapter 5 provides a finite query evaluation strategy for a particular sublanguage, the *right-restricted productive formulae*, of our proposal; this sublanguage is our suggestion for practical use. The capabilities of our suggested languages are analyzed in Chapter 6; in particular, the right-restricted productive formulae are shown to capture exactly the string manipulation tasks that belong to the Polynomial-time Hierarchy [87], and therefore all the tasks expected to arise in practice. Chapter 7 concludes the presentation.

Most of the material in Chapters 2–4.2 and 6.1 has appeared previously [41, 42], albeit in a different form; so has most of the material in Chapters 4.3–5.1 and 5.3 as well [40].

1.1 On Grammatical Molecular Biology

This work has connections with the realm of computational science, as already mentioned on page 1. Computational science has traditionally been taken to mean large-scale numerical simulations, not database management. However, this view is being challenged by new information-rich sciences, in which a major research bottleneck is managing the sheer amount of incoming data. A leading example is molecular biology, in which the volume of DNA and protein data doubles every 22 months [62].

The top graph of Fig. 1.1 presents data that supports this notion of exponential growth for the EMBL data library, a popular public-domain nucleotide database [88]. Each point corresponds to an official release from the inception of the database up to release 49 (December 1996). The bottom graph in Fig. 1.1 shows, however, that the *average length* of the stored nucleotide sequences does not grow. For example, release 44 (September 1995) of the aforementioned EMBL database represented an annual increase of about 2.5 times the number of entries, but only about 1.7 times the number of bases (that is, the total number of characters in the sequences) [71].

This implies that the major bottleneck will be in managing the *relations* containing sequences, rather than the processing of individual sequences. Adding more expressive sequence processing capabilities to the query languages is therefore warranted, even if their evaluation is more costly on a per-sequence basis, because they can filter these huge database relations into more focused pieces, thereby alleviating the bottleneck.

An example of an entry in the EMBL database is provided by Fig. 1.2. Different fields are identified by the first two letters of the line; for example, the first line `ID` provides a unique running identifier, here `HSAIDSA1`, for each entry. The sequence itself is located at the end of the whole entry.

Note that the sequence is by no means the only datum with internal structure: in particular, the *feature table* `FT` lists what is known about different subsequences of the whole sequence. In this example, the feature table tells us that positions 6-10 of this sequence contain a target subsequence `GTAGT`, at which position the HIV-1 virus has inserted its own proviral DNA within the original human DNA. (That is, the viral RNA is reverse transcribed into proviral DNA, which is subsequently attached into the DNA of the cell for eventual transcription back into fresh copies of the original RNA [63, pages 843–844].)

Molecular biology sequence database searches fall into two distinct methodologies. The more established methodology is similarity search based on some *statistical* properties of biological sequences (DNA for instance), as exemplified by tools like FASTA [46, Chapter 15.5] and BLAST [46, Chapter 15.6]. Recently, however, a new methodology has emerged that is based on studying the *grammatical* properties of sequences instead [20, 21, 22, 23, 26, 57, 73, 77, 78, 79, 80]. The basic tenet of this methodology is that the information (genes for instance) in these sequences is encoded in a grammatical instead of a statistical manner, and so should be amenable to interpretation as a result of a discrete combinatorial system, a (formal) language [64, page 84]. (On the other hand, DNA does not seem to obey the same statistical laws as natural languages [91], and therefore the tenet should be regarded as only a simplifying hypothesis.)

For example, regulatory information encoded in a DNA strand has been modelled with formal grammars [20, 21, 22, 23, 73]; moreover, these grammars not only serve to detect this information, but also encode our knowledge about it. In the same vein, the three-dimensional structure of the protein encoded by a DNA strand is amenable to a linguistic analysis [26, 79]. Finally, the problem of aligning different nucleotide sequences with respect to each other allows automata-theoretic representations as well [57, 80].

Figure 1.1 The growth of the EMBL nucleotide sequence database.

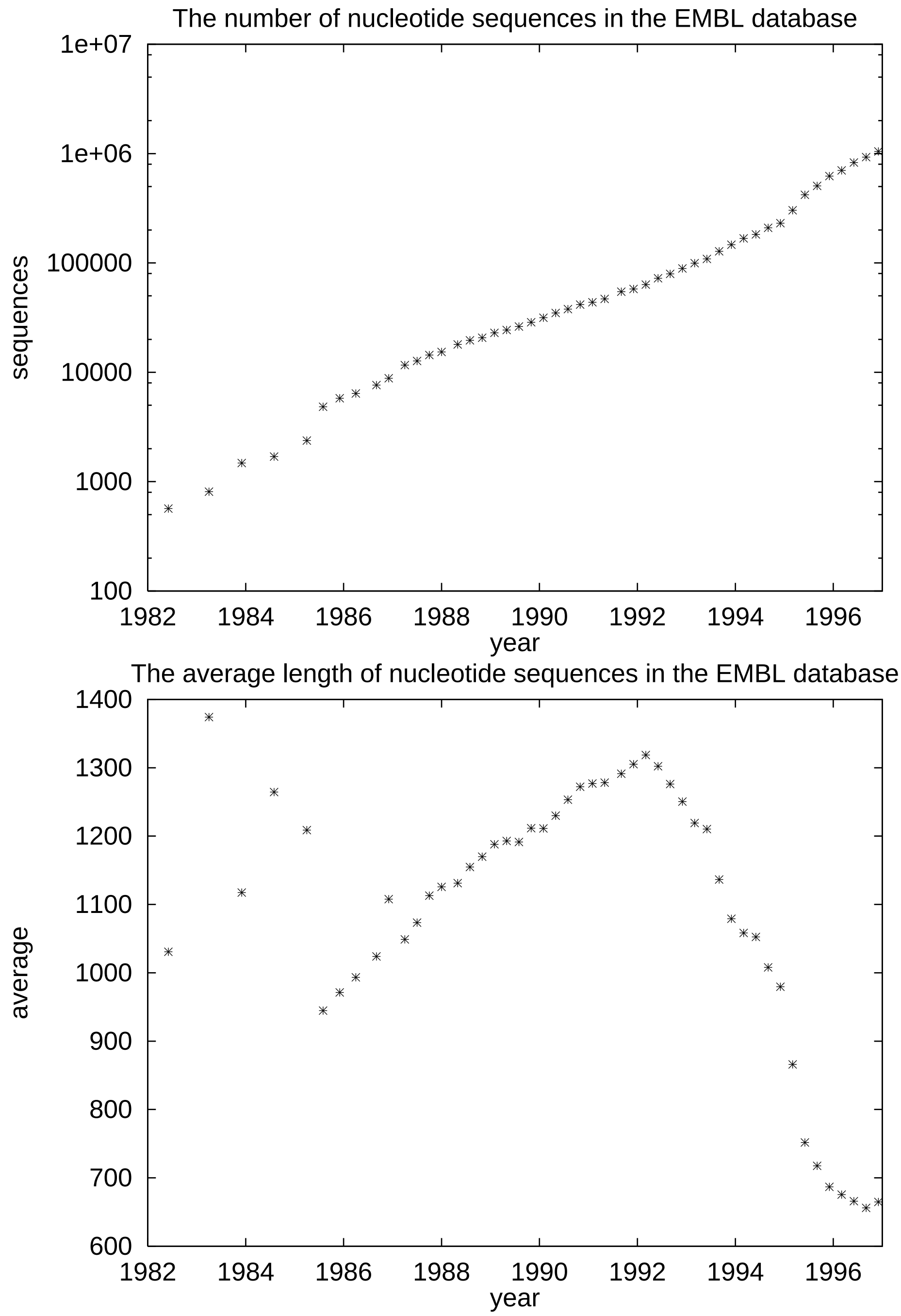


Figure 1.2 An entry in the EMBL nucleotide database.

```
ID   HSAIDSA1   standard; DNA; HUM; 45 BP.
XX
AC   M60134;
XX
NI   g178287
XX
DT   21-JUL-1991 (Rel. 28, Created)
DT   21-JUL-1991 (Rel. 28, Last updated, Version 1)
XX
DE   Human/HIV 5´ recombination junction.
XX
KW   long terminal repeat.
XX
OS   Homo sapiens (human)
OC   Eukaryotae; mitochondrial eukaryotes; Metazoa; Chordata;
OC   Vertebrata; Eutheria; Primates; Catarrhini; Hominidae; Homo.
XX
RN   [1]
RP   1-45
RX   MEDLINE; 91012810.
RA   Vink C., Groenink M., Elgersma Y., Fouchier R.A., Tersmette M.,
RA   Plasterk R.H.;
RT   "Analysis of the junctions between human immunodeficiency virus
RT   type 1 proviral DNA and human DNA";
RL   J. Virol. 64:5626-5627(1990).
XX
FH   Key             Location/Qualifiers
FH
FT   source          1. .45
FT                   /organism="Homo sapiens"
FT   repeat_unit     6. .10
FT                   /note="5´ insertion target sequence"
FT   misc_recomb     10^11
FT                   /note="Human DNA end/HIV DNA start"
XX
SQ   Sequence 45 BP; 17 A; 8 C; 10 G; 10 T; 0 other;
>P1;HSAIDSA1
    Length: 45
ACTTAGTAGT TGGAAGGGCT AATTCACTCC CAAAGAAGAC AAGAT
//
```

Fig. 1.2 serves to exemplify our three string query types in this context as follows.

- A *pattern matching* query would be "find all the sequences u from the database that contain this target GTAGT and are therefore susceptible to infection".

  That is, find from the database all the strings u that belong to the language denoted by the regular expression [49, Chapter 2.5] $(\mathtt{A}+\mathtt{C}+\mathtt{G}+\mathtt{T})^*\mathtt{GTAGT}(\mathtt{A}+\mathtt{C}+\mathtt{G}+\mathtt{T})^*$.

- A *string comparison* query would be "find all pairs $\langle u, v\rangle$ of sequences, where v might be an infected variant of u — that is, v can be obtained from u by inserting some 'infecting' sequence after an occurrence of the target".

  That is, find all pairs of the form $u = x\mathtt{GTAGT}z$ and $v = x\mathtt{GTAGT}yz$, where u and v are in the database.

- A *string restructuring* query would be "for GTAGT, infect the database with the tail $y =$ TGGAAGGGCT AATTCACTCC CAAAGAAGAC AAGAT from Fig. 1.2".

  That is, divide any string u from the database to $x\mathtt{GTAGT}z_1 \ldots \mathtt{GTAGT}z_n$ if you can, and output the string $v = x\mathtt{GTAGT}yz_1 \ldots \mathtt{GTAGT}yz_n$ for each possible division.

  Note that these new sequences v are not required to appear in the database yet. Instead, they can be interpreted as hypothetical forms that could arise were the string u subsequently infected.

1.2 Sequence Query Languages

Different languages can approach a given query type – pattern matching, string comparison, or string restructuring – from different directions: pattern matching can for instance be supported either by a separate explicit pattern notation [48, 79], or more implicitly via operations over the ordered index set $\{1, 2, 3, \ldots, n\} \subsetneq \mathbb{N}$ underlying a given n-character string $c_1c_2c_3 \ldots c_n$ [36, 65, 70, 81, 82]. Moreover, the index type $\mathbb{N}$ can either be a direct part of the language [65, 81, 82], or it can be hidden behind a small set of ordering operations [36, 70].

Consider for example the problem of defining palindromes u; that is, the strings that remain the same when read in reverse. An explicit pattern notation would be the corresponding Context-Free Grammar [49, Chapter 4]

$$S \longrightarrow \varepsilon \mid cSc \quad \text{for all } c \in \Sigma,$$

where ε denotes the empty string. A language with explicit index operations might state the same property with "the ith character from the beginning of the string must always match the ith character from the end". Finally, a language with implicit indexing might offer the statement "if you start walking right from the left end of the string one character at the time, and similarly from right to left, you will always encounter the same characters". This last step-by-step formulation is similar in spirit to Example 1.1.

Analogously, string restructuring can be supported either as a by-product of index manipulation [19, 36, 45, 65, 81, 82], or it can be explicitly introduced as a new distinct operation [13, 38, 59, 61, 100].

The same string query can therefore be expressed in various different forms, depending on the string handling primitives chosen for the language. This section presents some suggestions from the literature. Subsection 1.2.1 discusses two pattern specification languages. In turn, Subsection 1.2.2 discusses two logic-based and two algebraic languages with explicit string restructuring operations. Finally, Subsection 1.2.3 discusses string languages, which operate on the index type either explicitly or implicitly.

In comparison, Alignment Calculus manipulates strings and supports restructuring via implicit index operations.

1.2.1 Grammatical Pattern Matchers

Let us now discuss two grammatical pattern matching formalisms for bioinformatics, *PAttern Language for Molecular biology (PALM)* [48] and *String Variable Grammars (SVGs)* [79]. The former is a generalization of the patterns used in the PROSITE data bank [46, Chapter 15.8][48, page 173], while SVG parsers have been used for molecular structure studies [26, 78].

Consider for instance the problem of specifying a pattern for the language

$$\left\{ w^{2n+3} \colon w \in \Sigma^*, n \in \mathbb{N} \right\}, \tag{1.1}$$

or the strings formed by making an odd number of copies of a initial string w. The `typewritten` version on the left is PALM, while a (slightly simplified) SVG version is on the right.

	$S \to A_\alpha$
`odds(`V`):=*:`V`-*:`V`-*:`V	$A_\alpha \to \alpha\alpha\alpha$
`odds(`V`):=*:`V`-`V`-odds(`V`)`	$A_\alpha \to \alpha\alpha A_\alpha$

Both PALM and SVGs are essentially Context-Free Grammars, further enhanced with variables, here V and α, which receive as value a substring of the string being matched; once set, this value does not change. For example, the PALM subexpression '$\mathtt{*:-}V$' is read as follows: "Match any input substring u ('$*$'). If the variable V has already a value, then verify that this value is indeed u; otherwise V receives u as its subsequent value." This logical variable concept originates already in the common background of both languages: they are both built on top of *Prolog* [18], a logic programming [58] language originally developed for natural language processing.

In the versions above, this variable binds to the initial string w, and this binding then propagates to the recursive rule applications as well. Hence for instance the SVG above can be considered a finite description of an underlying *infinite* Context-Free Grammar, which has rules

$$\begin{aligned} S &\to A_u \\ A_u &\to uuu \\ A_u &\to uuA_u \end{aligned}$$

for all strings $u \in \Sigma^*$; that is, all the possible bindings of variable α are written out explicitly as the corresponding values u. It is straightforward to see from this expansion how for instance the strings $(\mathtt{ab})^{2n+3}$ are generated as

$$S \xrightarrow{S \longrightarrow A_{\mathtt{ab}}} A_{\mathtt{ab}} \xrightarrow[n\text{ times}]{A_{\mathtt{ab}} \longrightarrow \mathtt{abab}A_{\mathtt{ab}}} (\mathtt{ab})^{2n} A_{\mathtt{ab}} \xrightarrow{A_{\mathtt{ab}} \longrightarrow \mathtt{ababab}} (\mathtt{ab})^{2n+3}.$$

Example 2.13 on page 36 gives an Alignment Calculus query for Eq. (1.1). However, in contrast to PALM and SVGs, Alignment Calculus restricts attention to regular grammars [49, Chapter 9.1], in which the right-hand side of a rule can contain the left-hand side of some other rule only as the last element; all the example grammars in this subsection are of this type, but the addition of a variable concept extends their expressivity into nonregular languages, as witnessed by Eq. (1.1). Moreover, we restrict attention to a finite number of variables, whereas for instance the nonpropagating PALM variant

$$\begin{aligned} &\mathtt{odds2:=*:}V\mathtt{-*:}V\mathtt{-*:}V \\ &\mathtt{odds2:=*:}V\mathtt{-*:}V\mathtt{-odds2} \end{aligned}$$

stands for the patterns

$$\mathtt{*:}V_1\mathtt{-*:}V_1\mathtt{-*:}V_2\mathtt{-*:}V_2\mathtt{-}\cdots\mathtt{-*:}V_n\mathtt{-*:}V_n\mathtt{-*:}W\mathtt{-*:}W\mathtt{-*:}W,$$

which represent instead the strings $v_1 v_1 v_2 v_2 \dots v_n v_n w w w$ by using $n+1$ independent variables $V_1, \dots, V_n$ and W.

Both PALM and SVGs add also a possibility to perform additional computations on the contents of these string variables. In PALM, this is accomplished by escaping into the underlying implementation language; however, this powerful mechanism also means stepping out of the proposed language for performing certain tasks deemed to be important. In contrast, SVGs provide a less powerful but more declarative solution in *finite substitutions*. A finite substitution is a function, which maps each character c into a finite set of strings; for example, the function

$$g(c) = \begin{cases} \{\mathsf{a}, \varepsilon\} & \text{if } c = \mathsf{a} \\ \{c\} & \text{otherwise} \end{cases}$$

represents optionally deleting occurrences of the character a. These letter-wise substitutions f are generalized into strings with

$$\pm f(\varepsilon) = \{\varepsilon\}\,, \; +f(cw) = f(c).+f(w) \text{ and } -f(cw) = -f(w).f(c),$$

where

$$A.B = \{ab \colon a \in A, b \in B\} \tag{1.2}$$

denotes the concatenation of two sets of strings A and B. For example, $+g(w)$ denotes the set of strings v obtained from string w by optionally deleting occurrences of a, while $-g(w)$ denotes their reversals v^R. (The reversal of a string $c_1 c_2 c_3 \dots c_m$ is the string $c_m c_{m-1} c_{m-2} \dots c_1$, which lists its characters in reverse order.) Then $S \to \alpha^{+g} \alpha^{-g}$ is an SVG denoting the strings uv, for which there exists a common value w of α satisfying both $u \in +g(w)$ and $v \in -g(w)$. That is, the prefix u can be obtained from w by optionally deleting some occurrences of a, while the suffix v can be obtained by optionally deleting some occurrences of a from the reversal of w. In other words, this SVG represents concisely the palindromes ww^R after some occurrences of a may have been deleted. In this way, SVGs extend to approximate string matching as well; PALM attains the same capability by including an explicit operator for edit distances [46, Chapter 11.2]. Example 2.8 presents how Alignment Calculus deals with edit distaces.

In Alignment Calculus, both string matching conditions and computations on the extracted substrings are expressed using the same string processing concept. On the other hand, Alignment Calculus does include string comparisons absent from these grammatical pattern matchers. Lefebvre has suggested *Multi-Tape S-Attribute Grammars (MTSAGs)* as a grammatical method for multiple string comparison [57]. The variable concept in this extension of attribute grammars [7, Section 5.1] is no longer the general logic variable concept as above; it is instead similar to the Alignment Calculus variable concept, which models reading positions within strings. Furthermore, MTSAGs do not support string restructuring: all strings to be compared must be known beforehand in order to be parsed by a given MTSAG.

1.2.2 Restructuring via Transduction

This subsection presents two string database query languages, which are obtained by equipping a logic-based data model with (generalizations of) *transducers* [49, Chapter 2.7] for string processing. Intuitively, transducers are finite-state computational devices, which produce output from their given input, as opposed to automata, which merely accept or reject their input. This output capability is then used for providing explicit string restructuring support. In *sequence logic* [38, 100] the underlying logic is the relational calculus, while in *Transducer Datalog* [13, 59, 61] the underlying logic is *Datalog* [2, Chapter 12][92, Chapter 3.2], a sublanguage of Prolog for querying databases. (The transducer variant of sequence logic has been added to Datalog as well [100, Chapter 5.1].) These languages represent research contributions towards adding string handling, and especially restructuring, capabilities into declarative logic-based data models.

Let us consider sequence logic first. Its string manipulation construct of is the *sequence predicate*, a formula of the form $P \equiv y \in [\![A]\!]_n(x_1, \ldots, x_n)$, where A is a regular expression over the alphabet $\{\alpha_1, \ldots, \alpha_n\}$. This A expresses the program of the corresponding *generic* transducer $\mathcal{A}$ with n input and one output *channels*, where each instruction α_i is interpreted as "transfer the next item of input channel i into the output channel". These inputs are the sequences $u_1, \ldots, u_n$ currently bound to the sequence variables $x_1, \ldots, x_n$ and the sequence variable y binds to the outputs v of $\mathcal{A}$. Because sequence logic is in fact a language for databases containing *lists of atoms* from a conceptually infinite "alphabet" $\mathcal{U}$, the finite-state device $\mathcal{A}$ merely copies its inputs into the output in some order, rather than operating on them in any way.

The sequence predicate P is therefore true for the aforementioned assignment of the sequences $u_1, \ldots, u_n, v$ into the variables $x_1, \ldots, x_n, y$ if and only if the regular language represented by A contains a word β (an instruction trace of $\mathcal{A}$) fulfilling the following two requirements:

1. for all $1 \leq i \leq n$, β contains exactly $|u_i|$ occurrences of α_i – in other words, every input channel must be exactly exhausted at termination; and

2. for all $1 \leq i \leq n$, $1 \leq j \leq |u_i|$, and $1 \leq k \leq |v|$, the jth occurrence of α_i in β is in position k if and only if the atom $v[k]$ found at position k of sequence v is the same as the atom $u_i[j]$ found at position j of sequence u_i.

For example, $y \in [\![\alpha_1^* \alpha_2^*]\!]_2(x_1, x_2)$ is true if and only if v is the concatenation of u_1 and u_2, because the only words β of the regular language satisfying requirement 1 are of the form $\alpha_1^{|u_1|}\alpha_2^{|u_2|}$, and then requirement 2 leads to $v = u_1[1] \dots u_1[|u_1|]u_2[1] \dots u_2[|u_2|] = u_1 u_2$.

In this way, the sequence logic formula

$$\exists x_1 \exists x_2 . R_1(x_1, x_2) \wedge y \in [\![\alpha_1^* \alpha_2^*]\!]_2(x_1, x_2) \tag{1.3}$$

returns the concatenations $v = u_1 u_2$ of pairs $\langle u_1, u_2 \rangle$ from the database table R_1. Example 2.3 presents a related query in Alignment Calculus (assuming a finite alphabet Σ instead of $\mathcal{U}$).

However, consider the converse problem of Eq. (1.3): divide each string in the database table R_2 into a prefix and a suffix. This can also be stated in sequence logic with

$$\exists y . R_2(y) \wedge y \in [\![\alpha_1^* \alpha_2^*]\!]_2(x_1, x_2),$$

but this amounts to asking for the inputs x_1 and x_2 that could produce the known output y. Why does the string handling concept differentiate between input and output directions, when the logical language does not? As we shall see, Alignment Calculus does not differentiate between input and output variables. However, the idea of adding string handling to the relational calculus is common to both languages.

Let us then consider Transducer Datalog. As stated above, it is obtained by adding transducers into the Datalog model. However, these transducers are *generalized* in the following sense: they can call other transducers as subroutines during their own execution. That is, one possible program step is to transfer control to the start state of some other transducer, which receives as inputs the original inputs to the calling transducer and the output previously produced by the caller; if the called transducer subsequently accepts, the output it produced replaces the previously produced output of the caller, and then the caller continues from where it left off, oblivious to the change. (Again, inputs and outputs are considered to be channels, in which no looking back to reconsider the already passed characters is possible.) Generalized transducers cannot call itself, or recurse, in this manner (even indirectly), and therefore they can be regarded as acyclic networks of finite-state devices.

Transducer Datalog emerges as the result of the following reasoning. Consider the problem of extracting from table R_2 the strings of Eq. (1.1). If we had at our disposal a string-oriented Datalog language with built-in interpreted infix function symbol '$\bullet$' for string concatenation, then this problem could be solved by the following program.

$$
\begin{array}{lll}
\mathtt{start}(X) & \leftarrow & R_2(X), \mathtt{odds}(Y, X). \\
\mathtt{odds}(Y, Y \bullet Y \bullet Y). & & \\
\mathtt{odds}(Y, Y \bullet Y \bullet Z) & \leftarrow & \mathtt{odds}(Y, Z).
\end{array}
$$

The rules for predicate `odds` in this program, and the corresponding PALM and SVG formulations on page 11 are similar, because all these languages employ logical variables and recursion.

However, indiscriminate use of string restrcturing with '$\bullet$' leads to programs, which create infinite (intermediate) results, and this is naturally undesirable. Therefore design also another language, whose *syntactic form* precludes writing those infinitary programs. This form is attained by restricting string processing into the aforementioned generalized transducers, which are subsequently translated into the aforementioned "core" string Datalog language (called Sequence Datalog). These transducers are represented as new interpreted function symbols, analogously to '$\bullet$' above. In this translation, a call of a subtransducer becomes simply a call of the corresponding translation.

Generalized transducers have a rather procedural feel, because their meaning is difficult to describe without recourse to computational terms; for instance, the subtransducer generalization stems directly from the subroutine call of the underlying language. Moreover, adding transducers into Datalog, a language which permits recursion, adds yet another dimension of computation into the semantics. In contrast, the declarative semantics of Alignment Calculus can be described without operational concepts, which enter the picture only in the query evaluation stage. Furthermore, Alignment Calculus confronts the problem of infinite results in a way, which attempts to take the *semantics* of string restructuring into account as well; our approach is discussed further in Chapter 4.

A slightly different, yet related, approach to list queries was taken in the *pomset algebra* of Grumbach and Milo [45]. As the name suggests, this is an algebraic instead of a logical language, and its operations are defined on pomsets, or partially ordered multisets. Linear lists form the subcase, where the underlying pomset, or index type, is a total order. List restructuring queries are specified by algebraic expressions that map pomsets into other pomsets. This mapping of relations into other relations gives the language a "second-order" feel. On the other hand, Alignment Calculus employs solely first-order concepts. List processing is specified by iterator operators reminiscent of the functional list processing language $\mathcal{L}_p$ of Colby et al [19]. In both languages, operations on lists are specified as functions, which employ recursion over the order of the list elements. These function definitions can be regarded as generalizing the aforementioned subtransducer concept even further by allowing recursion as well. Thus, these two languages no longer employ finite-state computational devices as their basic list or string processing constructs. For example, concatenation is performed in $\mathcal{L}_p$ [19, Section 3.3] by the well-known recursive algorithm, which corresponds to replacing the regular expression $\alpha_1^*\alpha_2^*$ in Eq. (1.3) with the equivalent regular Context-Free Grammar

$$S \to A \qquad A \to \alpha_1 A \mid B \qquad B \to \alpha_2 B \mid \varepsilon.$$

However, these languages are not restricted to merely copying the input channels into the output channel in some order, as was the case in sequence logic.

We shall encounter these languages again in Section 6.1.

1.2.3 Logics on Order

This chapter discusses a few database query languages, where string operations are specified either as explicit or implicit indexing operations. A major motivation for these languages is *temporal* data, where the underlying index type is time. The transition to string databases is to consider an n-character string $w = c_1c_2c_3\ldots c_n$ as events c_i for times $i = 1, 2, 3, \ldots, n$. Languages with explicit operations process sequences by manipulating these time stamps i; these include the language of Pistor and Traunmüller (PT for brevity) [65] and SEQ [81, 82] for general data management. However, SEQ does not have the concept of the *whole* sequence as a separate entity, it just manages ordered collections of data points. Languages with implicit operations use ordering terms like "before" and "after" instead; these include PROXIMAL [36] for molecular biology databases such as the EMBL database discussed in Section 1.1, and Richardson's logic (RL for brevity) [70] and AQUA [89] for list data management. (The index type can in general be exteded into any ordering relation, or directed acyclic graph, as in the pomset algebra [45] mentioned in Subsection 1.2.2; however, strings do demand a total ordering.)

Let us consider the problem of selecting from a database table T the strings w, where at some position i the corresponding character c_i differs from both its left and right neighbours c_{i-1} and c_{i+1}. In PT, this could be expressed as

```
SELECT W
FROM   W IN T
WHERE  (EXISTS I IN INDL(W) (1<I AND I<LEN(W) AND
                             W[I-1]<>W[I] AND W[I]<>W[I+1]))
```

Here LEN(W) returns the length of the string W, while INDL(W) returns the corresponding index range $\{1, 2, 3, \ldots, \texttt{LEN(W)}\}$. This query selects from T those w, for which there exists some position $1 < i < |w|$, where c_i differs from both c_{i-1} and c_{i+1}.

These explicit indexing languages like PT draw their inspiration from the *First-Order Logic of Linear Order (FOLLO)* [29, page 1022], which deals with position (or time stamp) ordering propositions $t_1 < t_2$. Practical languages add for instance arithmetic over positions, as above, complicating their expressivity analysis.

The language RL with implicit indexing approaches (the string part of) the same task with the logical formula

$$\Diamond \Lambda K_1 . \neg \bigcirc (\cdot = K_1 \vee \Lambda K_2 . \bigcirc \cdot = K_2). \tag{1.4}$$

Eq. (1.4) is read as follows. Eventually in the future ('$\Diamond$') – that is, at some position j to the right of the current position, which is intitially the beginning of the string – the following holds. Let K_1 denote the character c_j at this position j. (Technically, this character is denoted by the so-called *nonrigid* variable '$\cdot$'; that is, one whose value changes with respect to time, or position in this context. Then 'Λ' can be used to store the current value of '$\cdot$' into a given rigid variable K_l.) The character c_{j+1} at the next ('$\bigcirc$') position $j+1$ must finally differ from K_1 and from the character c_{j+2} at position $j+2$. In other words, Eq. (1.4) says that the contiguous character triplet at positions j, $j+1$ and $j+2$ witnesses the stated condition.

The language RL draws its inspiration from *Propositional Linear-time Temporal Logic (PLTL)* [29, Section 3]. As a pattern specification language for finite strings, PLTL is as expressive as the *star-free* regular expressions [29, Theorem 6.4]. These are in turn obtained by replacing the Kleene closure operator r^* with *complementation* $\overline{r}$ with respect to Σ^*. For example,

$$\overline{\emptyset} \underset{c \in \Sigma}{+} \left(\underset{e \in \Sigma \setminus \{c\}}{+} e \right) c \left(\underset{e \in \Sigma \setminus \{c\}}{+} e \right) \overline{\emptyset}$$

is a star-free expression equivalent to Eq. (1.4). Note how complementation is used for specifying repetition.

However, star-free regular expressions are not as expressive as regular expressions in general [29, pages 1023–1024], because repetition by complementation is not as general as Kleene closure. Therefore Wolper has proposed an extension of PLTL [103], which reintroduces the Kleene closure operator. (Another option to regain the expressivity of regular expression would have been to introduce a form of second-order quantification [29, Theorem 6.5].) Alignment Calculus incorporates a form of Wolper's proposal, and adds to it facilities for comparing multiple strings as well.

The pomset algebra and $\mathcal{L}_p$, the functional list processing languages encountered in Subsection 1.2.2, can also be regarded as having an index manipulation component: their recursive list function definitions can namely access the input list elements relative to their positions within the list. Furthermore, these positions can also be manipulated within these functions as well.

Let us conclude this chapter by comparing explicit and implicit indexing as follows. The expressive power of FOLLO and PLTL are known to coincide [29, Theorem 6.4] in the original pattern matching setting. However, their generalizations into the database setting become incomparable [1]. Hence, it is likely that different multiple string comparison languages emerge depending on whether explicit or implicit indexing is chosen. Alignment Calculus has chosen the latter in order to avoid having a two-sorted language.

Chapter 2

Alignment Calculus

> LOGIC, *n.* *The art of thinking and reasoning in strict accordance with the limitations and incapacities of the human misunderstanding.*
>
> Ambrose Bierce [11]

This chapter introduces formally *Alignment Calculus*, our logic for expressing string properties, and its interpretation as a string database query language.

Alignment Calculus has a *two-layered* structure. The bottom layer consists of formulae, which describe how several strings can be aligned to each other. This layer treats each string as a sequence of characters, and alignments are specified via series of character comparisons. This modal layer is a variant of Wolper's extended linear-time temporal logic, with support for multiple string comparisons. Section 2.1 describes this layer in detail.

The top layer provides then a relational calculus over the string relations specified as alignment conditions, where the strings are now considered as indivisible entities. The previous modal string handling layer is added into the standard relational calculus to obtain our data model. Accordingly, the *domain* of the upper relational level is the set of all strings from the given alphabet. Section 2.2 describes this layer in turn.

This two-layer structure is in contrast with *First-Order Linear Temporal Logic (FOLTL)* [29, Chapter 3.3], the first-order extension of PLTL, where modalities, connectives and quantifiers are allowed to mix freely.

Section 2.3 concludes this chapter with examples on how Alignment Calculus can be used to specify string database queries.

Figure 2.1 An alignment of three strings

```
 abc
abb
 cacd
```

2.1 Logical Descriptions of String Alignments

In the world of strings we distinguish as state an *alignment* of a given set of strings. Fig. 2.1 illustrates an alignment of the three strings abc, abb and cacd. Properties of an alignment are expressed with respect to the vertical window. For instance, in the alignment of Fig. 2.1 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 the *alphabet*, a set with at least two (2) elements called *characters*, assumed to be fixed beforehand. Formally an alignment is then a partial function $A : \mathbb{N} \times \mathbb{Z} \to \Sigma$, in which for each $i \in \mathbb{N}$ there exists a contiguous finite interval $K_i^A \subsetneq \mathbb{Z}$ for which

$$A(i, j) = \begin{cases} a & \text{for some } a \in \Sigma \text{ if } j \in K_i^A \\ \text{undefined} & \text{otherwise.} \end{cases} \tag{2.1}$$

Intuitively, the first argument i ranges vertically over the rows of the alignment, while the second argument j ranges horizontally over the columns within the current row i. The values $j \in K_i^A$ denote the positions containing a character from Σ. We further require that the *window column* 0 representing the vertical window at least touches the interval K_i^A, if possible; formally, $K_i^A \cap \{-1, 0, 1\} \neq \emptyset$ whenever $K_i^A \neq \emptyset$. Thus, if the alignment in Fig. 2.1 represents the three first rows of A, then we have for instance $A(2, -1) = \mathtt{c}$, $A(2, 0) = \mathtt{a}$, $A(2, 1) = \mathtt{c}$, $A(2, 2) = \mathtt{d}$, and $A(2, j)$ is undefined elsewhere.

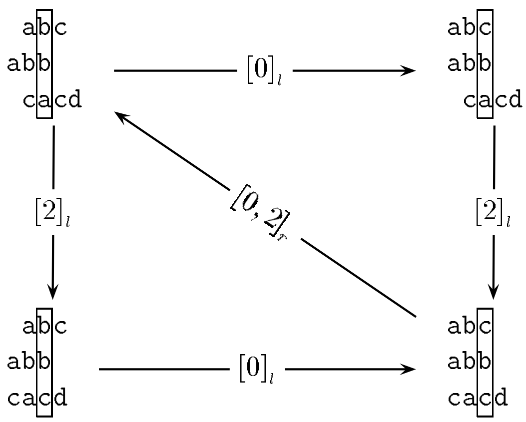
The notation $\sigma_A(i)$ denotes *the string represented by row i in the alignment A*. This is formally defined as

$$A(i, \min K_i^A) A(i, \min K_i^A + 1) A(i, \min K_i^A + 2) \ldots A(i, \max K_i^A) \in \Sigma^*.$$

Here Σ^* denotes the language of all the finite strings drawn from the alphabet Σ. For example, if A is the top left alignment in Fig. 2.2, then

$$\sigma_A(2) = A(2, -1)A(2, 0)A(2, 1)A(2, 2) = \mathtt{cacd}.$$

Figure 2.2 Transposing alignments



The *length* $|\sigma_A(i)|$ of string $\sigma_A(i)$ is defined to be the cardinality or size $\left|K_i^A\right|$ of the corresponding interval, or equivalently the number of character occurrences within the string, as usual. In particular, $\sigma_A(i)$ denotes the *empty* string ε whenever $K_i^A = \emptyset$. Accordingly, the window column is further required to be situated either within a nonempty string, or at its either end.

Alignments are connected to each other through state transitions called *transposes*. A transpose says that certain rows 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

$$[i_1, \ldots, i_k]_l A(i,j) = \begin{cases} A(i, j+1) & \text{if } i \in \{i_1, \ldots, i_k\} \text{ and } K_i^A \cap \{0,1\} \neq \emptyset \\ A(i,j) & \text{otherwise.} \end{cases}$$

That is, each row in $i_1, \ldots, i_k$ is shifted one character to the left with respect to the window column, unless the window column is already at the right end of the row, in which case the row does not move.

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). In Fig. 2.2 we show some transposes of the alignment in Fig. 2.1.

The propositions on the window contents of a particular alignment are expressed through *window formulae*, which are Boolean combinations of *atomic* window formulae of the form $x = \perp$, $x = a$, or $x = y$: here x and y

are members of a countably infinite set $V = \{x_0, x_1, x_2, \ldots\}$ of variables, the symbol $\perp$ 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 any injection $\theta : V \to \mathbb{N}$. Injectivity ensures that no two distinct variables can refer to the same row of the alignment at the same time, to preclude ambiguities as for whether or not to transpose a row. Given a particular alignment A and a window formula ϕ, we now define when *A satisfies ϕ by assignment θ*, in symbolic notation $A \models \phi\theta$.

1. $A \models (x = \perp)\theta$ if and only if $A(\theta x, 0)$ is undefined.

2. $A \models (x = a)\theta$ if and only if $A(\theta x, 0)$ is the character $a \in \Sigma$ of the alphabet.

3. $A \models (x = y)\theta$ if and only if $A(\theta x, 0)$ and $A(\theta y, 0)$ are either both undefined or the same character $a \in \Sigma$ of the alphabet.

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

5. $A \models (\neg\phi)\theta$ if and only if $A \models \phi\theta$ does not hold, often denoted $A \not\models \phi\theta$.

We use the shorthand '$\not\models$' for the complement of this truth relation '$\models$'. We also use the shorthands $x \neq y$ for $\neg(x = y)$, $\phi \vee \psi$ for $\neg(\neg\phi \wedge \neg\psi)$, and $x_1 = x_2 = x_3 = \ldots = x_m$ for $x_1 = x_2 \wedge x_2 = x_3 \wedge \ldots \wedge x_{m-1} = x_m$. Furthermore, we allow adding and removing redundant parenthesization to clarify the structure of window formulae.

For instance, if the three first rows of A are as in Fig. 2.1, and if θ maps x to 0, y to 1, and z to 2, then we have $A \models ((x = \mathtt{a}) \vee \neg(y = \mathtt{c}))\theta$, and $A \not\models (x = z)\theta$.

Transposes are expressed as indexed modalities in the language, and these modalities can be applied to window formulae. 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 called *atomic string formulae*. For example, $[x]_l(x = \mathtt{c} \wedge y = \mathtt{b})$, $[x]_l(x = \mathtt{c})$, and $[x, z]_r(z = \mathtt{a} \vee y = \mathtt{b})$ are all atomic string formulae. The definition of satisfaction for atomic string formulae in alignment A with assignment θ is as follows.

6. $A \models [x_1, \ldots, x_k]_l\phi\theta$ if and only if $[\theta x_1, \ldots, \theta x_k]_l A \models \phi\theta$.

7. $A \models [x_1, \ldots, x_k]_r\phi\theta$ if and only if $[\theta x_1, \ldots, \theta x_k]_r A \models \phi\theta$.

As an illustration, let A be the top left alignment, and A' the bottom right alignment in Fig. 2.2, and let θ map x to 0, y to 1, and z to 2.

Then we have $A \models ([x]_l(x = \mathtt{c} \wedge y = \mathtt{b}))\theta$, $A \not\models ([z]_l(x = \mathtt{c}))\theta$, and $A' \models ([x,z]_r(z = \mathtt{a} \vee y = \mathtt{b}))\theta$.

We can compose atomic string formulae by concatenating them. 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, $[x,z]_r(z = \mathtt{a} \vee y = \mathtt{b})[x]_l(x = \mathtt{c} \wedge y = \mathtt{b})$ and $[x,z]_r(z = \mathtt{a} \vee y = \mathtt{b})[z]_l(x = \mathtt{c})$ are formula words. The unique formula word containing no atomic string formulae is denoted λ; it is vacuously true in every alignment under every assignment.

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

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

8. $A \models (\tau_1\alpha_1\tau_2\alpha_2\ldots\tau_k\alpha_k)\theta$ if and only if both $\tau_1\theta A \models \alpha_1\theta$ and $\tau_1\theta A \models (\tau_2\alpha_2\ldots\tau_k\alpha_k)\theta$ hold.

Once again, let A' be the bottom right alignment in Fig. 2.2, and let θ map x to 0, y to 1, and z to 2. Then we have the following:

$$A' \models ([x,z]_r(z = \mathtt{a} \vee y = \mathtt{b})[x]_l(x = \mathtt{c} \wedge y = \mathtt{b}))\theta, \tag{2.2}$$

but

$$A' \not\models ([x,z]_r(z = \mathtt{a} \vee y = \mathtt{b})[z]_l(x = \mathtt{c}))\theta.$$

Sets of formula words can be represented by certain regular expressions as follows.

Definition 2.1 *The* string formulae ϕ *are expressions, which denote regular languages* $L(\phi)$ *of formula words as follows.*

Each atomic string formula $\tau\alpha$ *is a string formula, and denotes the singleton* $L(\tau\alpha) = \{\tau\alpha\}$.

If ϕ *is a string formula, so is also* $(\phi)^*$*, and it denotes the Kleene closure of* $L(\phi)$*, or the smallest set* K *(with respect to inclusion) satisfying the equation* $K = \{\lambda\} \cup K.L(\phi)$*, where the concatenation of two sets of strings was defined in Eq. (1.2).*

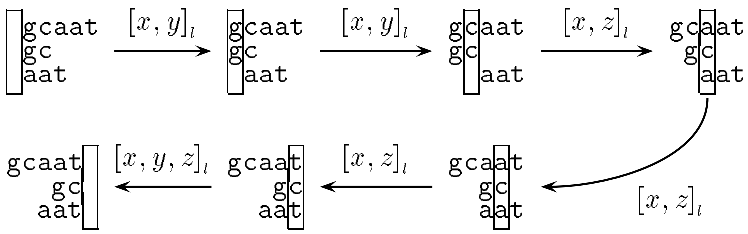
If ϕ *and* ψ *are string formulae, then so is* $(\phi.\psi)$*, and it denotes the concatenation* $L(\phi.\psi) = L(\phi).L(\psi)$.

If ϕ *and* ψ *are string formulae, then so is* $(\phi + \psi)$*, and it denotes the union* $L(\phi + \psi) = L(\phi) \cup L(\psi)$.

There are no other string formulae.

Figure 2.3 Performing a sequence of tests denoted by a formula word.

$$([x, y]_l x = y)^2([x, z]_l x = z)^3([x, y, z]_l x = y = z = \perp)$$



We also use the shorthand ϕ^+ for $\phi.\phi^*$, and ϕ^{n+1} for $\phi.\phi^n$, where $\phi^0 = \lambda$. Redundant parentheses can be removed by considering the operators in Definition 2.1 to be listed in decreasing precedence. Thus, $[x]_l x = \mathtt{a} + [x]_l x = \mathtt{a}.[x]_l x = \mathtt{a}^*$ means $([x]_l x = \mathtt{a} + ([x]_l x = \mathtt{a}.([x]_l x = \mathtt{a})^*))$, as is customary. Redundant parentheses can also be added at will.

For example, if ϕ is the string formula

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

then

$$\begin{aligned} L(\phi) = \{&[x, z]_r(z = \mathtt{a} \vee y = \mathtt{b})[x]_l(x = \mathtt{c} \wedge y = \mathtt{b}), \\ &[x, z]_r(z = \mathtt{a} \vee y = \mathtt{b})[z]_l(x = \mathtt{c})\} \end{aligned}$$

is the corresponding language of formula words.

We now define

9. $A \models \phi\theta$ for a string formula ϕ if and only if $A \models (\tau_1\alpha_1\tau_2\alpha_2 \ldots \tau_k\alpha_k)\theta$ holds for some formula word $\tau_1\alpha_1\tau_2\alpha_2 \ldots \tau_k\alpha_k \in L(\phi)$.

That is, ϕ specifies a series $\tau_1\alpha_1\tau_2\alpha_2 \ldots \tau_k\alpha_k$ of atomic string formulae, which can be successfully applied in the given sequence. A similar concept of regular languages of first-order formulae has been used to analyze fixpoint logics [4, Section 5]. Indeed, both concepts formalize the notion of iterating a logical test: one over sequences, the other over databases. Fig. 2.3 presents an example of what this step-by-step testing means in practice, reworking Example 1.1 in more detail. A similar approach to string handling was also taken in the sequence logic of Ginsburg and Wang [38, 100], except that the logical test to iterate was reduced to simply copying characters from inputs into output.

If ϕ is the formula word given in Eq. (2.2), A' is the bottom right alignment in Fig. 2.2, 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` positioned in the window, and θ mapped variable u to 3, then we would have $A' \models ([u]_l(u = \mathtt{b}).[u]_l(u = \mathtt{a}))^+\theta$, and $A' \not\models ([u]_l(u = \mathtt{a}).[u]_l(u = \mathtt{b}))^+\theta$.

Note that truth definition 9 covers all those logics, in which relations of sequences are described by formulae ϕ specifying languages $L(\phi)$ of formula words; Definition 2.1 is just the special case of regular expressions within this general framework. This idea was already implicit in Wolper's presentation of extended temporal logic [103]. For example, the formulae ϕ generating context-free languages $L(\phi)$ of two variables would provide a natural extension of the *cut grammars* [78] for modelling intermolecular connections between two strands of DNA. This theme has been pursued further by Lefvebre [57], but in a grammatical instead of a logical or a database setting.

2.2 A Calculus of String Relations

Alignment Calculus has a conceptually infinite collection $R_0, R_1, R_2, \ldots$ of *relation symbols* R_i, each equipped with a corresponding *arity* $a(R_i) \in \mathbb{N}$. A *database* db is any function mapping each relation symbol R_i with arity $a(R_i)$ into a finite subset of $(\Sigma^*)^{a(R_i)}$. In other words, *db interprets* the relation symbols R_i, and these interpretations are further constrained to be finite so that they can indeed be considered abstractions of relational tables. The collection of all databases is denoted DB. The interpretation $db(R_i)$ of a given R_i is customarily denoted as simply r_i instead. We will, however, later employ functions over DB, and hence retain the explicit notation.

We then define the truth of an *atomic relational formula* $R_i(x_{i_1}, \ldots, x_{i_k})$ under assignment θ with respect to an alignment A and a database db. Formally we continue the truth definition begun in Section 2.1 with

10. $\langle A, db\rangle \models R_i(x_{i_1}, \ldots, x_{i_k})\theta$ if and only if $\langle \sigma_A(\theta x_{i_1}), \ldots, \sigma_A(\theta x_{i_k})\rangle \in db(R_i)$, where $k = a(R_i)$.

To complete the definition of our language we take as atomic formulae the set of all string formulae and atomic relational formulae, and close it under $\wedge$, $\neg$, and $\exists$ with the following semantics.

11. $\langle A, db\rangle \models (\phi \wedge \psi)\theta$ if and only if $\langle A, db\rangle \models \phi\theta$ and $\langle A, db\rangle \models \psi\theta$.

12. $\langle A, db\rangle \models (\neg\phi)\theta$ if and only if $\langle A, db\rangle \not\models \phi\theta$.

13. $\langle A, db\rangle \models (\exists x\phi)\theta$ if and only if there exists an $i \in \mathbb{N}$ satisfying $\langle A, db\rangle \models \phi\theta[x = i]$. Here $\theta[x = i]$ denotes a *modified* assignment,

which agrees with θ everywhere except that x is now mapped to i instead.

We also use the shorthands $\forall x\phi$ for $\neg\exists x\neg\phi$, $\phi\vee\psi$ for $\neg(\neg\phi\wedge\neg\psi)$, and $\phi \rightarrow \psi$ for $(\neg\phi) \vee \psi$. Furthermore, we allow for adding and removing redundant parenthesization. To this end, we employ the convention that the scope of a 'dotted' quantification of the form $Qx.\phi$ extends to the largest possible syntactically correct subformula ϕ, while the 'dotless' variant $Qx\psi$ has the smallest possible syntactic scope.

Quantification introduces into Alignment Calculus the familiar concept of *free and bound variable occurrences*, defined in the customary way. We denote as free (ϕ) the set of variable symbols occurring free in the formula ϕ. We shall also consider the set V of variable symbols ordered by ascending index, and denote as

$$\mathrm{pos}(W, x_i) = |\{x_j \colon x_j \in W, j \leq i\}| \tag{2.3}$$

the number of elements in $W \subseteq V$ up to and including x_i. That is, $\mathrm{pos}(W, x_i)$ yields the position of x_i within the ordered subset W of variable symbols.

The truth definitions 1–9 in Section 2.1 do not depend on a particular database. Therefore they extend to tuples $\langle A, db\rangle$ as such; for example in definition 2, $\langle A, db\rangle \models (x = \mathtt{a})\theta$ if and only if $A(\theta x, 0) = \mathtt{a}$. In addition, the truth definitions 6 and 7 involving a transpose modify only the rows of the alignment A and not the other components db and θ of the truth relation; moreover, the modifications on A serve to slide the designated rows one step into the designated direction. Similarly, the truth definition 13 modifies only the assignment θ, thereby changing the designated variable symbol to point into some other row of the alignment A. This is in keeping with the philosophy behind modal logics: a modal operator (the transposes in our case) modifies the current state of the world (the alignment in our case) into a new world that is similar but not quite the same. In turn, quantification is used for discussing the elements of the current world (the rows of the current initial alignment in our case). Finally, the interpretation of the relation symbols remains fixed, because we restrict our attention to modelling queries against the current database rather than updating it.

When formulating a query on a database, the programmer is to assume that all the strings being reasoned about are in a starting position corresponding to an *initial* alignment: an alignment where the leftmost symbol in each string is placed one position to the right of the window, at column 1. In other words, $\min K_i^A = 1$ for each row i. Initial alignments are denoted A_0. We thus have for instance $A_0 \models (x = \perp)\theta$, for all assignments θ and initial alignments A_0. This also fixes left transposes as the "forward" and right transposes as the "reverse" direction of string processing. Accordingly we call a variable in a string formula *unidirectional* if it only appears in left transposes and *bidirectional* if it appears in right transposes as well.

We have now introduced all the apparatus necessary to define the meaning of queries in string databases. A query in our model is a formula ϕ with $\text{free}(\phi) = \{x_{i_1}, \ldots, x_{i_k}\}$ ordered by ascending index; that is, $i_1 < \ldots < i_k$. The *answer* to a query of this kind when posed to a database db is

$$[\![\phi]\!]_{db} = \bigcup_{A_0,\theta} \{\langle \sigma_{A_0}(\theta x_{i_1}), \ldots, \sigma_{A_0}(\theta x_{i_k})\rangle : \langle A_0, db\rangle \models \phi\theta\}. \tag{2.4}$$

In Eq. (2.4) the union is over all *full interpretations:* pairs consisting of an initial alignment A_0 and an assignment θ, where for every string $u \in \Sigma^*$ there exist infinitely many i outside the range of θ with $\sigma_{A_0}(i) = u$. This extra fullness condition ensures that the assignments in truth definition 13 can always be formed, and hence that the quantifiers range freely over Σ^*. The exact formulation of this fact is given below as the first claim of Lemma 2.2 (3).

A query ϕ then asks to list those tuples of strings, for which there is a way of initially aligning the strings, possibly together with other (quantified) strings freely selected from Σ^*, then substituting the strings for the free variables and having the formula become true with respect to the initial alignment and the database.

The formulae ϕ containing no atomic relational formulae constitute *pure* Alignment Calculus. As noted above, their truth does not depend on the current database, and hence we can denote the corresponding answers with $[\![\phi]\!]$ instead.

Given an initial alignment A_0 and a constant $\ell \in \mathbb{N}$, the *truncation* A_0^ℓ *of* A_0 *to length* ℓ is the initial alignment, where $A_0^\ell(i,j) = A_0(i,j)$ when $j \leq \ell$ and undefined otherwise. In other words, each $\sigma_{A_0^\ell}(i)$ is the string containing the ℓ first characters of string $\sigma_{A_0}(i)$. Similarly the *truncation* $[\![\phi]\!]_{db}^\ell$ *of* $[\![\phi]\!]_{db}$ *to length* ℓ is defined following Eq. (2.4) as

$$[\![\phi]\!]_{db}^\ell = \bigcup_{A_0,\theta} \left\{\left\langle \sigma_{A_0^\ell}(\theta x_{i_1}), \ldots, \sigma_{A_0^\ell}(\theta x_{i_k})\right\rangle : \left\langle A_0^\ell, db\right\rangle \models \phi\theta\right\}, \tag{2.5}$$

where the union is again over the full interpretations as in Eq. (2.4) above. Note in particular that when determining $[\![\phi]\!]_{db}^\ell$ the quantifiers range freely over $\Sigma^{\leq \ell}$, the language of all strings $w \in \Sigma^*$ of length $|w| \leq \ell$. This is shown below as the second claim of Lemma 2.2 (3).

Our interest in full interpretations comes from the following result, which is the Alignment Calculus counterpart to the classical result that the truth of a formula depends only on the values of its free variables. Without full interpretations, there might be some dependencies between variables and strings not visible at the logical level: for example, some string $w \in \Sigma^*$ might not be available as a tentative value for the variable x when determining the truth of a given Alignment Calculus formula $\exists x \phi$, either because w does not appear in the alignment at all, or all its appearances might already be assigned to other variables.

Lemma 2.1 *Let ϕ be an Alignment Calculus formula, and let A_0, θ and B_0, θ' be two full interpretations.*

If $\sigma_{A_0}(\theta x)$ and $\sigma_{B_0}(\theta' x)$ are equal strings for all $x \in \text{free}(\phi)$, then $\langle A_0, db \rangle \models \phi\theta$ and $\langle B_0, db \rangle \models \phi\theta'$ are equivalent.

Moreover, if $\sigma_{A_0^\ell}(\theta x)$ and $\sigma_{B_0^\ell}(\theta' x)$ are equal strings for all $x \in \text{free}(\phi)$, then $\left\langle A_0^\ell, db \right\rangle \models \phi\theta$ and $\left\langle B_0^\ell, db \right\rangle \models \phi\theta'$ are equivalent.

Proof: These two claims are proved by induction on the structure of ϕ as follows.

If ϕ is a relational atomic formula, then both claims follow directly from truth definition 10.

Let ϕ then be a string formula. For the first claim, it suffices to show that $\langle A_0, db \rangle \models \phi\theta$ implies $\langle B_0, db \rangle \models \phi\theta'$. By truth definition 9, $L(\phi)$ contains some formula word $\beta = \tau_1 \alpha_1 \ldots \tau_k \alpha_k$, where $\langle A_0, db \rangle \models \beta\theta$ as in truth definition 8. We now assert that the functions (alignment rows) $f_i^x(z) = \tau_i\theta \ldots \tau_1\theta A_0(\theta x, z)$ and $g_i^x(z) = \tau_i\theta' \ldots \tau_1\theta' B_0(\theta' x, z)$ are equal, for all $0 \leq i \leq k$ and $x \in \text{free}(\phi)$. Assume conversely that there exist some i and x satisfying $f_i^x \neq g_i^x$. Select the smallest such i and a corresponding x. Because both A_0 and B_0 are initial alignments, $f_0^x = g_0^x$ is equivalent to strings $\sigma_{A_0}(\theta x)$ and $\sigma_{B_0}(\theta' x)$ being equal. This in turn holds by assumption, and therefore $i > 0$. Then by the definition of transposition, $\tau_i\theta$ modifies f_{i-1}^x exactly the same way $\tau_i\theta'$ modifies g_{i-1}^x. This contradicts the minimality of i, and our assertion is verified. The first claim then follows by our assertion and truth definitions 1 to 5.

The same reasoning as above remains equally valid, if A_0 and B_0 are replaced with A_0^ℓ and B_0^ℓ, proving the second claim.

Both claims follow directly by induction and truth definitions 11 and 12, when $\phi \equiv \psi \wedge \chi$ or $\phi \equiv \neg\psi$.

Consider finally the case $\phi \equiv \exists x \psi$. For the first claim, it suffices to show that $\langle A_0, db \rangle \models \phi\theta$ implies $\langle B_0, db \rangle \models \phi\theta'$. By truth definition 13, there exists some $i \in \mathbb{N}$ satisfying $\langle A_0, db \rangle \models \psi\theta[x = i]$. Because B_0, θ' is a full interpretation, there exists some $i' \in \mathbb{N}$, for which $B_0, \theta'[x = i']$ remains a full interpretation, and $\sigma_{A_0}(i)$ and $\sigma_{B_0}(i')$ are the same string. (Note that in a non-full interpretation, there might be no rows i' containing the string

$\sigma_{A_0}(i)$ left.) Then $\langle B_0, db\rangle \models \psi\theta'[x = i']$, by the inductive assumption. Hence $\langle B_0, db\rangle \models \phi\theta'$, as required.

Similar reasoning suffices for the second claim: $\left\langle A_0^\ell, db\right\rangle \models \phi\theta$ implies the existence of some $i \in \mathbb{N}$, for which $\left\langle A_0^\ell, db\right\rangle \models \psi\theta[x = i]$ holds. Again, we can select some $i' \in \mathbb{N}$, for which $B_0, \theta'[x = i']$ remains a full interpretation, and $\sigma_{A_0^\ell}(i)$ and $\sigma_{B_0^\ell}(i')$ are the same string. Then $\left\langle B_0^\ell, db\right\rangle \models \psi\theta'[x = i']$, and therefore $\left\langle B_0^\ell, db\right\rangle \models \phi\theta'$. □

Let us close this chapter by remarking that the relational component of Alignment Calculus given by truth definitions 11 to 13 behaves as expected in queries.

Lemma 2.2 *The following properties hold for queries expressed in Alignment Calculus.*

1. *$\vec{u} \in [\![\neg\phi]\!]_{db}$ if and only if $\vec{u} \notin [\![\phi]\!]_{db}$.*

   Moreover, $\vec{u} \in [\![\neg\phi]\!]_{db}^\ell$ if and only if $\vec{u} \notin [\![\phi]\!]_{db}^\ell$ and $\vec{u}$ contains no string longer than ℓ.

2. *$\left\langle u_1, \ldots, u_{|\text{free}(\phi_1\wedge\phi_2)|}\right\rangle \in [\![\phi_1 \wedge \phi_2]\!]_{db}$ if and only if $\left\langle v_1^i, \ldots, v^i_{|\text{free}(\phi_i)|}\right\rangle \in [\![\phi_i]\!]_{db}$ for $i = 1, 2$, where $v^i_{\text{pos}(\text{free}(\phi_i),x)}$ and $u_{\text{pos}(\text{free}(\phi_1\wedge\phi_2),x)}$ are equal strings for each $x \in \text{free}(\phi_i)$.*

   Moreover, the same property holds for truncations $[\![\ldots]\!]_{db}^\ell$ as well.

3. *Let $x \in \text{free}(\phi)$. Then $\left\langle u_1, \ldots, u_{|\text{free}(\exists x\phi)|}\right\rangle \in [\![\exists x\phi]\!]_{db}$ if and only if there exists some $v \in \Sigma^*$, for which*

   $$\left\langle u_1, \ldots, u_{\text{pos}(\text{free}(\phi),x)-1}, v, u_{\text{pos}(\text{free}(\phi),x)+1}, \ldots, u_{|\text{free}(\phi)|}\right\rangle \in [\![\phi]\!]_{db}.$$

   Moreover, the same property holds for truncations $[\![\ldots]\!]_{db}^\ell$ as well, except that v is now chosen from among the strings no longer than ℓ.

Proof: Consider the first part of claim 1. By Eq. (2.4) and truth definition 12, $\vec{u} = \left\langle u_1, \ldots, u_{|\text{free}(\neg\phi)|}\right\rangle \in [\![\neg\phi]\!]_{db}$ if and only if there exists some full interpretation A_0, θ satisfying the following two conditions:

1. $\langle A_0, db\rangle \not\models \phi\theta$; and

2. for all $x \in \text{free}(\neg\phi) = \text{free}(\phi)$, the strings $\sigma_{A_0}(\theta x)$ and $u_{\text{pos}(\text{free}(\phi),x)}$ are the same.

By Lemma 2.1, this is equivalent to having condition 1 for *every* full interpretation A_0, θ satisfying condition 2. By Eq. (2.4), this is in turn equivalent to $\vec{u} \notin [\![\phi]\!]_{db}$, as promised by the first part of claim 1.

Note that $[\![\ldots]\!]^{\ell}_{db}$ consists only of strings with length at most ℓ, by Eq. (2.5). Therefore the second part of claim 1 follows by a similar reasoning as above.

Consider then the first part of claim 2. Its forward implication follows directly from Eq. (2.4) and truth definition 11. Conversely, $\left\langle v^i_1, \ldots, v^i_{|\mathrm{free}(\phi_i)|} \right\rangle \in [\![\phi_i]\!]_{db}$ implies by Eq. (2.4) the existence of some full interpretation $A_{0,i}, \theta_i$, for which $\langle A_{0,i}, db\rangle \models \phi_i\theta_i$, and $\sigma_{A_{0,i}}(\theta_i x)$ is the same string as $u_{\mathrm{pos}(\mathrm{free}(\phi_1 \wedge \phi_2),x)}$ for each $x \in \mathrm{free}\,(\phi_i)$. Lemma 2.1 allows us to choose these two $A_{0,i}, \theta_i$ to be the same full interpretation A_0, θ, because $\sigma_{A_{0,1}}(\theta_1 x)$ and $\sigma_{A_{0,2}}(\theta_2 x)$ are the same strings for all $x \in \mathrm{free}\,(\phi_1) \cap \mathrm{free}\,(\phi_2)$. This confirms the other implication, and completes the first part of claim 2.

The second part of claim 2 follows by the same reasoning over truncations $[\![\ldots]\!]^{\ell}_{db}$.

Consider finally the first part of claim 3. By Eq. (2.4) and truth definition 13 we have that $\left\langle u_1, \ldots, u_{|\mathrm{free}(\exists x\phi)|} \right\rangle \in [\![\exists x\phi]\!]_{db}$ is equivalent to the existence of some full interpretation A_0, θ and $i \in \mathbb{N}$ satisfying the following two conditions:

1. $\langle A_0, db\rangle \models \phi\theta[x = i]$; and
2. for all $y \in \mathrm{free}\,(\exists x\phi)$, the strings $\sigma_{A_0}(\theta[x = i]y)$ and $u_{\mathrm{pos}(\mathrm{free}(\exists x\phi),y)}$ are the same.

Now $A_0, \theta[x = i]$ is again a full interpretation, by definition. Moreover, $\theta[x = i]x = i$ and $\theta[x = i] = \theta$ otherwise, by truth definition 13. Choosing v to be the string $\sigma_{A_0}(i)$, conditions 1 and 2 are equal to

$$\left\langle u_1, \ldots, u_{\mathrm{pos}(\mathrm{free}(\phi),x)-1}, v, u_{\mathrm{pos}(\mathrm{free}(\phi),x)+1}, \ldots, u_{|\mathrm{free}(\phi)|} \right\rangle \in [\![\phi]\!]_{db}$$

by Eq. (2.4), thus proving the first part of claim 3.

Again, the second part of claim 3 follows by the same reasoning over truncations $[\![\ldots]\!]^{\ell}_{db}$. □

2.3 Examples of Alignment Calculus Queries

Let R_1 be a binary and R_2 a unary string relation symbol. 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. The symbol $\top$ denotes a tautological window formula, for example $x = x$.

Example 2.1 List the second component of all tuples in R_1 where the first component is abc:

$$\exists y.R_1(y,x) \wedge ([y]_l y = \mathtt{a}).([y]_l y = \mathtt{b}).([y]_l y = \mathtt{c}).([y]_l y = \perp).$$

■

Example 2.2 List all tuples in R_1 where the first component equals the second component:

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

The second conjunct is a string formula verifying that x and y contain the same string. In what follows this *string equality* will be denoted $x =_s y$. ■

Example 2.3 List all tuples from relation R_2 that are concatenations of the two components in some tuple of relation R_1:

$$\begin{aligned}&\exists y \exists z.R_1(y,z) \wedge R_2(x) \wedge\\ &\qquad ([x,y]_l x = y)^*.([x,z]_l x = z)^*.([x,y,z]_l x = y = z = \perp).\end{aligned}$$

An example of verifying the truth of this concatenation string formula was given by Fig. 2.3 in Section 2.1. Concatenation was discussed already in Example 1.1; note how this string formula is a straightforward transliteration of that description. ■

Example 2.4 List all tuples of R_1 where the first component is a manifold of the second (the manifolds of string v being the strings of the form $vvv\ldots v$):

$$\begin{aligned}&R_1(x,y) \wedge (([x,y]_l x = y)^*.([y]_l y = \perp).([y]_r y \neq \perp)^*.([y]_r y = \perp))^*.\\ &\qquad ([x,y]_l x = y)^*.([x,y]_l x = y = \perp).\end{aligned}$$

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

Example 2.5 List all tuples in R_2 that are a shuffle of the two components of a tuple in R_1.

$$\begin{aligned}&\exists y \exists z.R_1(y,z) \wedge R_2(x) \wedge\\ &\qquad (([x,y]_l x = y) + ([x,z]_l x = z))^*.([x,y,z]_l x = y = z = \perp).\end{aligned}$$

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. Think of u and v as two halves of the same playing card deck: these two halves can then be shuffled by bending and releasing them so that their cards merge. ■

Example 2.6 List all tuples of R_1 where the second component is of the form $(\mathsf{gc} + \mathsf{a})^*$.

$$R_1(x, y) \wedge (([y]_l y = \mathsf{g}).([y]_l y = \mathsf{c})) + ([y]_l y = \mathsf{a}))^*.([y]_l y = \bot).$$

■

Example 2.7 List all tuples of R_1 where the first component occurs in the second:

$$R_1(x, y) \wedge ([y]_l \top)^*.([x, y]_l x = y)^*.([x]_l x = \bot).$$

■

Example 2.8 List all tuples of R_1 where the edit distance between the first and the second component is not larger than k:

$$\begin{aligned} R_1(x, y) \wedge &([x, y]_l x = y)^*. \\ &(([x, y]_l \top + [x]_l \top + [y]_l \top).([x, y]_l x = y)^*)^k. \\ &([x, y]_l x = y = \bot). \end{aligned}$$

The *edit distance* of strings u and v is the minimum number of steps required to transform u into v. Each step can consist of replacing one symbol with another, or of inserting or deleting a symbol [46, Chapter 11.2].

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.

Edit distance is an example of *a measure of similarity* useful in sequence database queries [52, page 36][79, Section 3]. Compared with the very general framework studied by Jagadish et al [52], Alignment Calculus lacks essentially a means to express numerical *degrees* of similarity, as witnessed by the fact that k is a constant and not a parameter in the query above. Much of this functionality can, however, be captured already within the current semantics by using strings for counting; for example,

$$\begin{aligned} R_1(x, y) \wedge &([x, y]_l x = y)^*. \\ &(([x, y, z]_l (z = \mathsf{a}) + [x, z]_l (z = \mathsf{a}) + [y, z]_l (z = \mathsf{a})). \\ &\quad ([x, y]_l x = y)^*)^*. \\ &([x, y, z]_l x = y = z = \bot). \end{aligned}$$

lists all the tuples $\left\langle u, v, \mathtt{a}^k \right\rangle$, where $\langle u, v\rangle \in db(R_1)$, and the edit distance of u and v is at most $k \leq |u| + |v|$. Attaching these similarity measures to those string selations, which are expressible using string formulae would also yield biologically interesting and meaningful models [80]. ■

Example 2.9 List all tuples of R_2 that are of the form $\mathtt{a}X\mathtt{b}X\mathtt{a}$, where $X \in \Sigma^*$. (The string formula $x =_s y$ for string equality was defined in Example 2.2 above.)

$$\begin{aligned}
&\exists y \exists z.(y =_s z) \wedge R_2(x) \wedge \\
&\quad ([x]_l x = \mathtt{a}).([x,y]_l x = y)^*.([x,y]_l x = \mathtt{b} \wedge y = \bot). \\
&\quad ([x,z]_l x = z)^*.([x,z]_l x = \mathtt{a} \wedge z = \bot).([x]_l x = \bot).
\end{aligned}$$

The formula states the existence of a string y, for which string x is of the form $\mathtt{a}y\mathtt{b}y\mathtt{a}$. 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 their initial alignment. ■

Example 2.10 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:

$$\begin{aligned}
\exists y \exists z.R_2(x) \wedge &(([x,y]_l x = \mathtt{a} \wedge y \neq \bot) + ([x,z]_l x = \mathtt{b} \wedge z \neq \bot))^*. \\
&([x,y,z]_l x = y = z = \bot) \\
&\wedge([y,z]_l y \neq \bot \wedge z \neq \bot)^*.([y,z]_l y = z = \bot).
\end{aligned}$$

The formula says that each occurrence of the character a in x must match some position in a string y. Likewise, each occurrence of the character b must match a position in a string z. Furthermore, it says that strings y and z must be exhausted simultaneously, thus requiring them to be of equal length. ■

Example 2.11 List all tuples of R_2 in the language $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n : n \in \mathbb{N}\}$:

$$\begin{aligned}
&\exists y.R_2(x) \wedge \\
&\quad ([x,y]_l x = \mathtt{a} \wedge y \neq \bot)^*.([y]_l y = \bot). \\
&\quad ([x]_l \top.[y]_r x = \mathtt{b} \wedge y \neq \bot)^*.([y]_r y = \bot). \\
&\quad ([x,y]_l x = \mathtt{c} \wedge y \neq \bot)^*.([x,y]_l x = y = \bot).
\end{aligned}$$

Here we require that x is of the form $\mathtt{a}^*\mathtt{b}^*\mathtt{c}^*$ and that there is a "counter string" y having the property that each of the three portions of x can be put into a one-to-one correspondence with y. The third line of the formula also shows how "simultaneous" left and right transpositions are expressible in string formulae. ■

Example 2.12 List all strings of R_2 that are of the form $(\mathtt{a}+\mathtt{b})^*$, 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 translation problem is common in bioinformatics [20].

$$\begin{aligned}&\exists y \exists z. R_2(x) \wedge \\ &\qquad ([x,y]_l x = y)^*.([y]_l y = \perp).([x,z]_l x = z)^*.([z]_l z = \perp) \wedge \\ &\qquad ([y,z]_l (y = \mathtt{a} \wedge z = \mathtt{b}) \vee (y = \mathtt{b} \wedge z = \mathtt{a}))^*.([y,z]_l y = z = \perp).\end{aligned}$$

This formula first divides x into a prefix y and a suffix z, and then tests that z is a translation of y. ∎

Example 2.13 List all the strings v from R_2 of form w^{2n+3}, as specified in Eq. (1.1). This problem was first discussed in Subsection 1.2.1.

$$R_2(x) \wedge \exists y((\phi.\phi)^*.\phi.\phi.\phi.[x]_l x = \perp),$$

where

$$\phi \equiv ([x,y]_l x = y)^*.[y]_l y = \perp.([y]_r \top)^*.[y]_r y = \perp$$

verifies that y appears next starting from the current position of x, and then rewinds y for the next verification.

Hence this query specifies that y must contain w for a given v in x. ∎

Note that in all the examples above there is at most one bidirectional variable in any string formula. Formulae of this kind will become so useful later on that they warrant an explicit definition.

Definition 2.2 *A string formula is* right-restricted *if and only if all its right transposes (if any) specify one and the same variable symbol.*

An Alignment Calculus formula is right-restricted if and only if all its string formulae are.

In addition to counting the number of bidirectional variables in a formula ϕ, we shall also occasionally refer to its *length* $||\phi||$, which we take to mean the number of symbols needed to write ϕ down as text. In particular, the variable occurrences x_k are now considered to be subscripted by strings k instead of natural numbers. These subscript strings are drawn from a binary alphabet $\{\mathbf{0},\mathbf{1}\}$ distinct from all the other symbols occurring in string formulae. Thus, the atomic string formula $[x_{\mathbf{01}}]_l x_{\mathbf{01}} = \mathtt{a}$ is an example of a formula written down as text, and the length of this textual representation is 11 characters.

Chapter 3

Alignment Calculus and Computation

> *. . . [H]e construed the world as a computer construes it, by means of key features and schematic relationships. The scheme might be identified – in an 'identiti-kit' way – without the reality being grasped at all.*
>
> Oliver Sacks [75]

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

These devices are described in Section 3.1. However, before defining the algebra in Section 3.3, we discuss first in Section 3.2 how the concept of a "finite answer" should be interpreted in the context of Alignment Calculus, where string restructuring can generate infinitely many new strings.

3.1 Multitape Finite State Acceptors

Informally, a *k-tape finite state acceptor (k-FSA)* is a "nondeterministic k-tape two-way finite state automaton with endmarkers", a straightforward extension of the familiar two-way finite automaton with endmarkers [49, Chapter 2.6 and Exercises 2.18–20] to k independently moving tapes. Other variants of multitape automata in the literature, such as [16, 47, 67], differ in details but not in overall spirit.

Formally, a k-FSA $\mathcal{A} = \langle Q, s, F, T\rangle$ is a system 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

$$T \subseteq (Q \times (\Sigma \cup \{\cent, \$\})^k) \times (Q \times \{-1, 0, +1\}^k)$$

is *a transition relation.* The elements $t = \langle\langle p, c_1, \ldots, c_k\rangle, \langle q, d_1, \ldots, d_k\rangle\rangle \in T$, which are called *transitions*, have the additional restriction that if $c_i = \cent$ then $d_i \neq -1$ and if $c_i = \$$ then $d_i \neq +1$ for every *tape* $i = 1, \ldots, k$. This restriction means that the head cannot leave the tape area marked by the left and right *endmarkers* $\cent$ and $\$$. We employ a nondeterministic automaton model – that is, T is not required to be a function – because the truth definition of string formulae (case 9 on page 26), our logical counterpart, is inherently nondeterministic, stating only that some suitable formula word must exist. Nondeterminism is in turn known to differ from determinism in two-tape automata [16][67, Theorem 5.2].

We will often denote the transitions t typographically with

$$p \xrightarrow[d_1,\ldots,d_k]{c_1,\ldots,c_k} q.$$

That is, we draw an arrow from the source state p into the target state q, put the character combination $c_1, \ldots, c_k$ above the arrow, and the tape winding behaviour $d_1, \ldots, d_k$ below it.

The *configurations* of $\mathcal{A}$ on *input* $W = \langle w_1, \ldots, w_k\rangle$, where each w_i is a string from Σ^*, are tuples of the form $\langle p, n_1, \ldots, n_k\rangle$, in which $p \in Q$ and $0 \leq n_i \leq |w_i| + 1$ for every tape $i = 1, \ldots, k$. A configuration $\langle q, n_1 + d_1, \ldots, n_k + d_k\rangle$ is a *next* configuration of $\langle p, n_1, \ldots, n_k\rangle$ on W if and only if

$$p \xrightarrow[d_1,\ldots,d_k]{w_1[n_1],\ldots,w_k[n_k]} q \in T, \text{ where}$$

$$a_1 \ldots a_m[j] = \begin{cases} \cent & j = 0 \\ a_j & 1 \leq j \leq m \\ \$ & j = m+1 \end{cases}$$

denotes the character on square j on the tape, which contains the string $a_1 \ldots a_m \in \Sigma^*$. In addition, $w[j \ldots k]$ is a shorthand for $w[j]w[j+1]w[j+2] \ldots w[k]$.

A *computation of* $\mathcal{A}$ *on* W is any sequence $C_1C_2C_3\ldots$ of its configurations on W where C_{i+1} is a next configuration of C_i. A computation $\mathcal{C} = C_1C_2C_3\ldots C_m$ *accepts* W if and only if its first configuration C_1 is the *initial* configuration $\langle s, 0, \ldots, 0\rangle$, it is finite, the state of its last configuration C_m belongs to F, and C_m has no next configuration on W. The *language*

$L(\mathcal{A})$ *accepted by* $\mathcal{A}$ consists of the tuples W accepted by $\mathcal{A}$ – that is, those W for which a finite accepting computation exists.

We often view $\mathcal{A}$ as a graph with states Q as its vertices and transitions $p \xrightarrow[d_1,\ldots,d_k]{c_1,\ldots,c_k} q$ as its labelled directed edges. A computation can then be viewed as tracing a path in this graph.

In Fig. 3.1 we show an example of a string formula and a corresponding FSA drawn as a graph. The alphabet Σ is for simplicity assumed to be $\{\mathtt{a}, \mathtt{b}\}$. (Note, however, that the "algorithm" encoded in the FSA subtly refines the string formula specification: consider namely testing whether tape 2, which denotes variable y, has been consumed. The string formula performs this test at the end, while the FSA performs it between the two loops.)

Analogously to variables in string formulae, tape i of $\mathcal{A}$ is said to be *bidirectional* if and only if T contains a transition where $d_i = -1$; otherwise it is *unidirectional*. The operational intuition is that bidirectional tapes can be scanned back and forth, while unidirectional tape squares can only be entered once. A transition is *stationary* if and only if $d_1 = \ldots = d_k = 0$; that is, no tape movement occurs. These form the FSA counterpart of the ε-transitions in classical, one-tape automata theory.

We can compute the "bindings" making string formulae true in initial alignments by using the FSAs introduced above. Formally, we show that the string formulae of Section 2.1 and the FSAs of this chapter capture the same string relations in the same way.

Theorem 3.1 *For each string formula ϕ with* free $(\phi) = \{x_1, \ldots, x_k\}$*, there is a corresponding k-FSA $\mathcal{A}_\phi$ satisfying $[\![\phi]\!] = L(\mathcal{A}_\phi)$.*

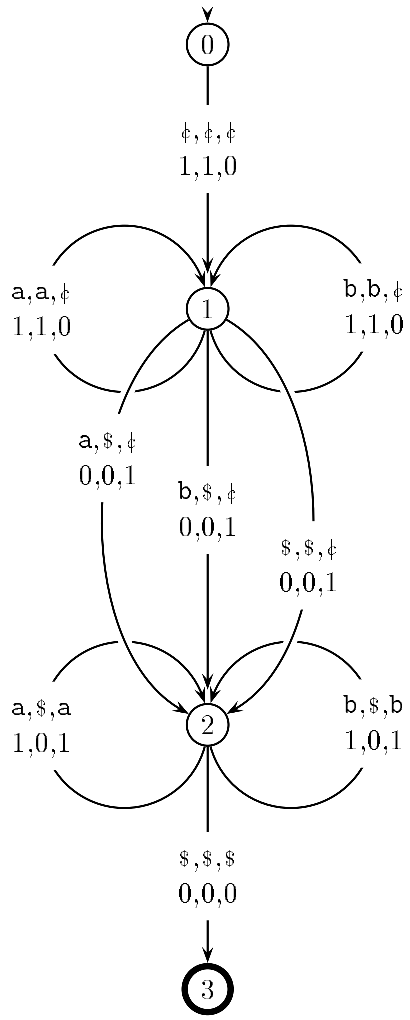
Moreover, if variable x_i is unidirectional in ϕ, the corresponding tape i is also unidirectional in $L(\mathcal{A}_\phi)$.

Proof: The intuition behind the theorem is the familiar equivalence of regular expressions and finite automata [49, Theorem 2.3], with string formulae as expressions and FSAs as automata. The fundamental building blocks of string formulae and FSAs, atomic string formulae and transitions, are closely related: both specify a one-step movement and a Boolean test. Our construction proceeds by induction on ϕ. First, however, we need some auxiliary definitions.

Because truth definition 9 does not alter variable assignments, it suffices to examine the truth of ϕ in finite alignments A with rows $1, \ldots, k$ under the fixed assignment θ, which maps each variable symbol x_i to row i. We omit mentioning this θ below for notational simplicity.

Let $P = \langle p, m_1, \ldots, m_k \rangle$ be a configuration of some k-FSA on input $\langle u_1, \ldots, u_k \rangle$. The alignment A_P corresponding to P is defined as follows.

Figure 3.1 A string formula and a corresponding 3-FSA



$$([x,y]_l\, x = y)^*.([x,z]_l\, x = z)^*.([x,y,z]_l\, x = y = z = \perp)$$

Row i contains the string u_i, for all $1 \leq i \leq k$. This string is positioned on the left of the window column, if $m_i = 0$; on the right, if $m_i = |u_i| + 1$; or on its m_ith character otherwise. That is, align the tapes of P so that their head positions coincide on the same column, call this column the window column in A_P, and delete the endmarkers. (This correspondence is not quite one-to-one, because P differentiates between the endmarkers of an empty input tape, while A_P does not.)

Let s_ϕ denote the start state of a k-FSA $\mathcal{A}_\phi$. Let f_ϕ similarly denote its final state; our construction ensures that there will be exactly one final state, and it will not be the start state. Moreover, s_ϕ will have no incoming transitions, and f_ϕ no outgoing transitions. We shall also ensure that all incoming transitions of f_ϕ are stationary.

The inductive claim is slightly complicated, because we need to keep track of several things at the same time, in order for the catenation case to succeed:

The Inductive Claim

$\mathcal{A}_\phi$ has a computation $\mathcal{C}$ on input $\langle u_1, \ldots, u_k \rangle$ starting from a (not necessarily initial) configuration $S = \langle s_\phi, m_1, \ldots, m_k \rangle$ and ending in a configuration $F = \langle f_\phi, n_1, \ldots, n_k \rangle$

if and only if

$L(\phi)$ contains some formula word $\Psi = \tau_1 \psi_1 \ldots \tau_r \psi_r$, for which $A_S \models \Psi$ and $A_F = \tau_r \ldots \tau_1 A_S$.

Intuitively, $\mathcal{C}$ and Ψ express the same sequence of operations.

If $\phi \equiv \tau\psi$, where τ is a left transpose and ψ a window formula, then $\mathcal{A}_\phi$ can be constructed as follows. Its states are s_ϕ, f_ϕ, and an intermediate state p. There is a transition $s_\phi \xrightarrow[d_1,\ldots,d_k]{c_1,\ldots,c_k} p$ exactly when the c_i and d_i can be chosen as follows. If x_i does not appear in τ, then $d_i = 0$ and $c_i \in \Sigma \cup \{¢, \$\}$. Otherwise either $d_i = 0$ and $c_i = \$$; or $d_i = +1$ and $c_i \in \Sigma \cup \{¢\}$. Then the configuration $S = \langle s_\phi, m_1, \ldots, m_k \rangle$ can be succeeded by the configuration $P = \langle p, m_1 + d_1, \ldots, m_k + d_k \rangle$ on input $\langle u_1, \ldots, u_k \rangle$ if and only if $A_P = \tau A_S$. That is, the transitions into p have exactly the same effect as τ. Add then a (stationary) transition $p \xrightarrow[0,\ldots,0]{c_1,\ldots,c_k} f_\phi$ exactly when the c_i are chosen so that the following interpretation $\mathcal{I}(\psi)$ is true: $\mathcal{I}(x_i = \perp)$ means that $c_i \in \{¢, \$\}$ holds; $\mathcal{I}(x_i = a)$ means that $c_i = a$ holds, where $a \in \Sigma$; $\mathcal{I}(\neg\chi)$ means that $\mathcal{I}(\chi)$ fails to hold; and $\mathcal{I}(\chi_1 \wedge \chi_2)$ means that both $\mathcal{I}(\chi_j)$ hold. Then the configuration P above can be succeeded by the final configuration $F = \langle f_\phi, m_1 + d_1, \ldots, m_k + d_k \rangle$ if and only if ψ is true in A_P. Moreover, $A_F = A_P$ by stationarity. Therefore the base case of the induction has been established.

If $\phi \equiv \psi_1 + \psi_2$, then $\mathcal{A}_\phi$ can be constructed as follows. Obtain inductively the $\mathcal{A}_{\psi_j}$ corresponding to ψ_j. Then merge s_{ψ_1} and s_{ψ_2} into s_ϕ, and merge similarly f_{ψ_1} and f_{ψ_2} into f_ϕ. This $\mathcal{A}_\phi$ has a computation on input $\vec{u} = \langle u_1, \ldots, u_k \rangle$ from $S = \langle s_\phi, m_1, \ldots, m_k \rangle$ to $F = \langle f_\phi, n_1, \ldots, n_k \rangle$ if and only if either $\mathcal{A}_j$ has a computation on $\vec{u}$ from $S_j = \left\langle s_{\psi_j}, m_1, \ldots, m_k \right\rangle$ to $F_j = \left\langle f_{\psi_j}, n_1, \ldots, n_k \right\rangle$. (Recall that the s_{ψ_j} have no incoming transitions.) This is equivalent by induction to the existence of some $\Psi \in L(\psi_1) \cup L(\psi_2) = L(\phi)$ true in $A_{S_j} = A_S$, whose verification ends in $A_{F_j} = A_F$. Therefore this inductive case has been established.

If $\phi \equiv \psi_1 . \psi_2$, then $\mathcal{A}_\phi$ can be constructed as follows. Obtain inductively the $\mathcal{A}_j$ corresponding to ψ_j. Then merge f_{ψ_1} and s_{ψ_2} into a new nonfinal state p, and let $s_\phi = s_{\psi_1}$ and $f_\phi = f_{\psi_2}$. This $\mathcal{A}_\phi$ has a computation on input $\vec{u} = \langle u_1, \ldots, u_k \rangle$ from $S = \langle s_\phi, m_1, \ldots, m_k \rangle$ to $F = \langle f_\phi, n_1, \ldots, n_k \rangle$ if and only if $\mathcal{A}_1$ has a computation on $\vec{u}$ from $S_1 = \langle s_{\psi_1}, m_1, \ldots, m_k \rangle$ to $F_1 = \langle f_{\psi_1}, o_1, \ldots, o_k \rangle$, and $\mathcal{A}_1$ has a computation on $\vec{u}$ from $S_2 = \langle s_{\psi_2}, o_1, \ldots, o_k \rangle$ to $F_2 = \langle f_{\psi_2}, n_1, \ldots, n_k \rangle$. (Recall that the transitions entering f_{ψ_1} are stationary.) This is equivalent by induction to the existence of some $\Psi_1 \in L(\psi_1)$ and $\Psi_2 \in L(\psi_2)$ having the following properties: Ψ_1 is true in A_{S_1} and its verification ends in A_{F_1}, while Ψ_2 is true in $A_{S_2} = A_{F_1}$ and its verification ends in A_{F_2}. This is in turn equivalent to the existence of $\Psi_1 \Psi_2 \in L(\phi)$ true in $A_S = A_{S_1}$, whose verification ends in $A_F = A_{F_2}$. Therefore this inductive case has been established.

If finally $\phi \equiv \psi^*$, then $\mathcal{A}_\phi$ can be constructed as follows. Obtain inductively the k-FSA $\mathcal{B}$ corresponding to ψ, and set $s_\phi = s_\psi$ and $f_\phi = f_\psi$. Add a stationary *bypassing* transition from s_ϕ to f_ϕ on every character combination from $(\Sigma \cup \{\cent, \$\})^k$; these implement choosing not to execute the loop at all. Consider then each transition pair

$$t_s = s_\phi \xrightarrow[d_1,\ldots,d_k]{c_1,\ldots,c_k} p \text{ and}$$
$$t_f = q \xrightarrow[0,\ldots,0]{c_1,\ldots,c_k} f_\phi,$$

where $p \neq f_\phi$ and $q \neq s_\phi$. Add a *looping* transition

$$t = q \xrightarrow[d_1,\ldots,d_k]{c_1,\ldots,c_k} p,$$

to implement choosing whether to execute the loop again or not. This is analogous to avoiding ε-transitions when constructing finite automata from regular expressions [39]. The inductive claim is established via a case-by-case analysis as follows. Consider first the computations of $\mathcal{A}_\phi$ that accept the input $\vec{u} = \langle u_1, \ldots, u_k \rangle$ by taking the appropriate bypassing transition, and therefore moving no tape. Its counterpart is the empty formula word $\lambda \in L(\phi)$, which is always true and transposes nothing. Consider then the computations $\mathcal{C}$ of $\mathcal{A}_\phi$, which do not take a bypassing transition. Let b be the number of times a computation $\mathcal{C}$ of this kind takes a looping transition. If $b = 0$, then $\mathcal{A}_\phi$ acts exactly like $\mathcal{B}$ during $\mathcal{C}$, and the claim follows directly from induction. Otherwise, there is a computation $\mathcal{C}$ of $\mathcal{A}_\phi$ on input $\vec{u} = \langle u_1, \ldots, u_k \rangle$ from $S = \langle s_\phi, m_1, \ldots, m_k \rangle$ to $F = \langle f_\phi, n_1, \ldots, n_k \rangle$ with $b > 0$ looping transition steps if and only if there is a computation $\mathcal{C}_1$ of $\mathcal{B}$ on $\vec{u}$ from $S_1 = \langle s_\psi, m_1, \ldots, m_k \rangle$ to $F_1 = \langle f_\psi, o_1, \ldots, o_k \rangle$ and a computation $\mathcal{C}_2$ of $\mathcal{A}_\phi$ on $\vec{u}$ from $S_2 = \langle s_\phi, o_1, \ldots, o_k \rangle$ to $F_2 = \langle f_\phi, n_1, \ldots, n_k \rangle$ with $b-1$ looping transition steps. Namely, let t be the first looping transition taken during $\mathcal{C}$, and let t_s, t_f be the transition pair that begot t. Then $\mathcal{C}_1$ is obtained by taking t_f instead of t, while $\mathcal{C}_2$ is obtained by starting with t_s and continuing as $\mathcal{C}$ did after taking t. Induction on the whole claim provides equivalently a formula word $\Psi_1 \in L(\psi)$ corresponding to $\mathcal{C}_1$, while induction on b provides a $\Psi_2 \in L(\phi)$ corresponding to $\mathcal{C}_2$. Then equivalently $\Psi_1\Psi_2 \in L(\phi)$ establishes this case of the induction on b by similar argumentation to the concatenation case above.

In particular, $\vec{u} \in L(\mathcal{A})$ if and only if there exists an accepting computation on $\vec{u}$ from the initial configuration $S = \langle s_\phi, 0, \ldots, 0 \rangle$ by definition, and this is equivalent by our inductive claim to the existence of some $\Psi \in L(\phi)$ true in the corresponding initial alignment A_S. This is in turn equivalent to ϕ being true in A_S, and therefore to $\vec{u} \in [\![\phi]\!]$, as required. □

The translation $\mathcal{A}_\phi$ of ϕ provided by Theorem 3.1 can be further normalized into the following form, which is more amenable to subsequent analysis.

Corollary 3.1 *The $\mathcal{A}_\phi$ constructed in Theorem 3.1 can be chosen to satisfy the following* neatness *conditions.*

1. *No transitions enter the start state s of $\mathcal{A}_\phi$, and all the transitions that leave s are on the character combination $¢, \ldots, ¢$.*

2. *There is a unique final state f, and f is distinct from s. No transitions leave f, and the transitions entering f are exactly the stationary transitions of $\mathcal{A}_\phi$.*

3. *Every state p of $\mathcal{A}_\phi$ other than s or f is on some path from s to f in the transition graph of $\mathcal{A}_\phi$.*

4. Every two consecutive transitions

$$p \xrightarrow[d_1,\ldots,d_k]{c_1,\ldots,c_k} q \xrightarrow[d'_1,\ldots,d'_k]{c'_1,\ldots,c'_k} r$$

satisfy the following local consistency conditions for all tapes $1 \leq i \leq k$*:*

- *if* $d_i = 0$*, then* $c'_i = c_i$*; and*
- *if* $d_i = +1$*, then* $c_i \neq ¢$*; and*
- *if* $d_i = -1$*, then* $c_i \neq \$$*.*

In other words, it is not a priori *impossible to continue with the second transition after the first one.*

Proof: Let us first construct the $\mathcal{A}_\phi$ provided by Theorem 3.1. The following construction refines it into an equivalent k-FSA $\mathcal{B}_\phi$, which satisfies neatness condition 4. The state space of $\mathcal{B}_\phi$ consists of tuples $P = \langle p, e_1, \ldots, e_k \rangle$, where p is a state of $\mathcal{A}_\phi$, and each $e_i \in \Sigma \cup \{¢, \$, \mathbf{L}, \mathbf{R}\}$. Here **L** and **R** are new symbols. The transition graph $\mathcal{G}$ of $\mathcal{B}_\phi$ is constructed by the following closure rule. Initially $\mathcal{G}$ consists only of the start state $S = \langle s, ¢, \ldots, ¢ \rangle$, where s is the start state of $\mathcal{A}_\phi$. Assume that $\mathcal{G}$ contains the state P as above, and $\mathcal{A}_\phi$ contains a transition $p \xrightarrow[d_1,\ldots,d_k]{c_1,\ldots,c_k} q$, where each $c_i = e_i$ when $e_i \in \Sigma \cup \{¢, \$\}$, $c_i \in \Sigma \cup \{\$\}$ when $e_i = \mathbf{L}$, and $c_i \in \Sigma \cup \{¢\}$ when $e_i = \mathbf{R}$. Add then to $\mathcal{G}$ the transitions

$$P \xrightarrow[d_1,\ldots,d_k]{c_1,\ldots,c_k} \underbrace{\langle q, g_1, \ldots, g_k \rangle}_{Q},$$

where each $g_i = c_i$ when $d_i = 0$, $g_i = \mathbf{L}$ when $d_i = +1$, and $g_i = \mathbf{R}$ when $d_i = +1$. In other words, $\mathcal{B}_\phi$ remembers in its finite control what the incoming transitions to state Q did on the tapes: whether the head was moved **L**eft or **R**ight or kept on a known character c_i. This enforces condition 1. State Q is final if and only if state q was final in $\mathcal{A}_\phi$.

Because the proof of Theorem 3.1 established the first part of condition 1 for $\mathcal{A}_\phi$, $\mathcal{B}_\phi$ now satisfies both parts. In addition, the proof of Theorem 3.1 established also the first three quarters of condition 2 for $\mathcal{A}_\phi$, and therefore $\mathcal{B}_\phi$ can be made to satisfy them as well by merging all its final states together as a single final state F.

The missing fourth quarter of condition 2 can be enforced by the following operation, which is analogous to ε-transition removal in ordinary finite automata [49, Theorem 2.2]. Let $\mathcal{H}$ be the (unlabelled) subgraph of $\mathcal{G}$ that consists of the transitions violating the fourth quarter; that is, transitions that are stationary but do not enter F. Compute the transitive closure $\mathcal{H}^*$ of $\mathcal{H}$. For every edge $P \longrightarrow Q$ in $\mathcal{H}^*$ with $P \neq Q$, and every transition $Q \xrightarrow[d_1,\ldots,d_k]{c_1,\ldots,c_k} R$ in $\mathcal{G}$ but not in $\mathcal{H}$, add into $\mathcal{G}$ the transition $P \xrightarrow[d_1,\ldots,d_k]{c_1,\ldots,c_k} R$. Then delete from $\mathcal{G}$ all transitions in $\mathcal{H}$.

Finally, condition 3 can be enforced by reachability analysis of $\mathcal{G}$; all transitions and states not on any path from the start state S into the final state F are deleted. □

The converse of Theorem 3.1 is also true, so the computational power of k-FSAs correspond exactly to string formulae over k variables.

Theorem 3.2 *For each k-FSA $\mathcal{A}$ there is a string formula $\phi_{\mathcal{A}}$ on variables $x_1, \ldots, x_k$ satisfying $[\![\phi_{\mathcal{A}}]\!] = L(\mathcal{A})$.*

Moreover, variable x_i is bidirectional if and only if tape i is.

Proof: The intuition behind this proof is again the correspondence of finite automata and regular eexpressions [49, Theorem 2.4]. We describe each transition $t = p \xrightarrow[d_1,\ldots,d_k]{c_1,\ldots,c_k} q$ of $\mathcal{A}$ with a string formula

$$\psi_t \equiv [\,]_l \left(\bigwedge_{i=1}^{k} x_i = c_i' \right) .\tau_l^t \top . \tau_r^t \top,$$

where $c_i' = c_i$ when $c_i \in \Sigma$, and '$\perp$' otherwise. Here $[\,]_l$ denotes the left transpose of no variables, which reduces to the identity function on transposes by definition. In addition x_i appears in τ_l^t if and only if $d_i = +1$ in t, and in $+tau_r^t$ if and only if $d_i = -1$ in t. The intention is that t can be taken in a configuration $C = \langle p, n_1, \ldots, n_k \rangle$ on input $\langle u_1, \ldots, u_k \rangle$ if and only if ψ_t is true in the corresponding alignments A. (Recall the correspondence between configurations and alignments from the proof of Theorem 3.1. We also omit mentioning the assignment θ explicitly here as well.) Moreover, $\tau_r^t \tau_l^t A$ corresponds to the resulting tape configuration $\langle q, n_1 + d_1, \ldots, n_k + d_k \rangle$. That is, ψ_t expresses exactly the operational semantics of t. This construction also implies the extra bidirectionality condition on $\phi_{\mathcal{A}}$, since τ_t performs exactly the transpositions corresponding to the tape movements in t.

We can safely assume that $\mathcal{A}$ has an unique final state f, which is also distinct from the start state and has no outgoing transitions. A slight complication arises since string formulae do not differentiate between the left and right ends of a string, while FSAs do. Namely, if any state p had transitions $t_1 = p \xrightarrow[\ldots]{\ldots,c_{i-1},\cent,c_{i+1},\ldots} q_1$ and $t_2 = p \xrightarrow[\ldots]{\ldots,c_{i-1},\$,c_{i+1},\ldots} q_2$, then it would be wrong to indiscriminately test for $x_i = \perp$ in both ψ_{t_1} and ψ_{t_2}. Fortunately $\mathcal{A}$ can be normalized to get rid of these conflicts by refining the state space as follows: the refined state space will consist of $p_{\langle b_1,\ldots,b_k\rangle}$ where $b_i \in \{¢, \star, \$\}$ denotes whether the state can be exited only on the condition that the head of tape i must be on ¢, between the endmarkers, or on \$. There will be a transition $p_{\langle b_1,\ldots,b_k\rangle} \xrightarrow[d_1,\ldots,d_k]{c_1,\ldots,c_k} q_{\langle e_1,\ldots,e_k\rangle}$ if and only if there originally was a transition $p \xrightarrow[d_1,\ldots,d_k]{c_1,\ldots,c_k} q$ and for each tape $1 \leq i \leq k$ one of the following cases holds:

- $d_i = 0$ and $b_i = e_i$; that is, the tape does not move.
- $d_i = +1$ and $e_i \in \{\star, \$\}$; that is, the tape moves left, and may end up between the endmarkers or on the right one (but certainly not on the left one).

  Note that the transitions leaving $q_{\langle e_1,\ldots,e_k\rangle}$ verify that the index is indeed respected.
- Symmetrically $d_i = -1$ and $e_i \in \{¢, \star\}$.

The new start state will be $s_{\langle ¢,\ldots,¢\rangle}$, where s is the original start state, because in the beginning all heads are known to be on ¢. The states $f_{\langle\ldots\rangle}$ will be the new final states. By the assumption, no transitions exit from them, and so they can even be merged into one new unique final state. Every state and transition not on any path from the new start state to the new final state can be ignored.

Let now the states of the normalized $\mathcal{A}$ be numbered $1,\ldots,n$ so that the start state is assigned number 1 and the final one n (in addition, $n \geq 2$ by assumption). Following [85, Theorem 3.17] we define inductively the formulae E_{ijk} to express the paths from state i to state j not going through any of the states $k+1,\ldots,n$: initially $E_{ij0} = \psi_{t_1} + \ldots + \psi_{t_a}$ where $t_1,\ldots,t_a$ are the transitions from i to j. If there are none, $E_{ij0} = [\,]_l \neg\top$ instead to signal that no valid computation can proceed in one step from state i to state j. We then define

$$E_{ijk} = \begin{cases} E^{+}_{ij(k-1)} & i = j = k \\ E_{ij(k-1)} . E^{*}_{jk(k-1)} & i \neq j = k \\ E^{*}_{ik(k-1)} . E_{kj(k-1)} & i = k \neq j \\ E_{ij(k-1)} + E_{ik(k-1)} . E^{*}_{kk(k-1)} . E_{kj(k-1)} & i \neq j \neq k \end{cases}$$

by induction on k.

The invariant of the inductive construction above is as follows: for all (and not just initial) alignments A and formula words $\Psi = \tau_1\psi_1 \ldots \tau_m\psi_m \in$

$L(E_{ijk})$, $A \models \Psi$ if and only if when $\mathcal{A}$ is started in state i with tapes corresponding to A, the computation can proceed into state j with tapes corresponding to $\tau_m \ldots \tau_1 A$ without visiting any state $k+1, \ldots, n$. Finally choosing $\phi_{\mathcal{A}} = E_{1nn}$ yields the claim. □

3.2 Domain Independence with Restructuring

The relational calculus has the concept of a *domain independent* formula ψ [92, 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 the current database and the query ϕ.

Intuitively, domain independence tries to capture the ideas that (1) a database query should always produce only a finite answer, and (2) its evaluation should consider only finitely many elements of the domain in the process, but (3) this finitely computed finite value must still be the same as if the computation had indeed considered every possible data value. Idea (1) is often referred to as *(semantic) safety*, while the elements in idea (2) are called the *active* domain of the query.

For instance, the answer to the query $([x]_l x = \mathsf{a})^*.([x]_l x = \bot)$ is the infinite set $\{\varepsilon, \mathsf{a}, \mathsf{aa}, \ldots\}$. Since our alphabet Σ is finite, a necessary and sufficient condition for finite answers (1) would be that every component in a tuple in the answer is of bounded length. This safety condition corresponds to the full semantics of Section 2.2, where quantification ranges over the entire Σ^* as expected. The analogous counterpart in the standard relational calculus is the *output-restricted unlimited interpretation* [51], which is obtained by interpreting quantifications to range over all possible elements of the universe, whether they are known to the database or not, and subsequently restricting the answers to the known elements.

However, it does not suffice to limit the length of the free variables in a query; the bound variables should also be constrained somehow to ensure the finiteness of query evaluation. For example, the formula

$$\forall x \exists y.([x, y]_l x = y)^*.[x, y]_l x = \bot \wedge y \neq \bot \tag{3.1}$$

states that there are arbitrarily long strings by explicitly constructing as y an arbitrary string that has x as its proper prefix. Intuitively, Eq. (3.1) does not seem amenable to any finite evaluation scheme other than somehow proving it true without generating the strings it specifies. Hence, we restrict ourselves to formulae whose evaluation generates only a finite amount of new information, although we might not *a priori* know how much. In other words, we adopt theoretically the *countable invention interpretation* and seek in practice to limit it further into the *finite* case [51]. In the standard relational calculus, these inventive interpretations are obtained from the unlimited case mentioned above by restricting the interpretations of quantifiers to range over a countable or a finite universe of previously unknown elements.

Definition 3.1 *The Alignment Calculus formula ϕ is* domain independent *if and only if there exists a* limitation function $\mathcal{W}_\phi \colon DB \to \mathbb{N}$*, for which* $[\![\phi]\!]_{db} = [\![\phi]\!]_{db}^{\ell}$ *is satisfied for every database db and constant* $\ell \geq \mathcal{W}_\phi(db)$*.*

The intuition is that the finite domain $\Sigma^{\leq \mathcal{W}_\phi(db)}$ is now a sufficient active domain for the correct evaluation of $[\![\phi]\!]_{db}$. In fact, Eq. (3.3) on page 55 will demonstrate that the limitation function $\mathcal{W}_\phi$ can actually be chosen so that its value depends only on the *lengths* of the strings in the database, and not on the actual strings themselves.

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 example the safe query asking for the concatenations of the strings in the database [61, page 24] (recall Example 2.3 on page 33 on how to attain this in Alignment Calculus). For example, the safety concept of Ginsburg and Wang [38, page 299][100, Chapter 4.5] requires that instead of a limitation function dependent on the current database there is a *constant* $\mathcal{W}_\phi \in \mathbb{N}$ common to all databases, making the concatenation query unsafe, although it can intuitively be evaluated with finite resources. On the other hand, the safety concept of Mecca and Bonner [61] allows the active domain to grow in a controlled way, thus accommodating for instance safe concatenation. This controlled growth is, however, intimately tied to the fixpoint semantics of the underlying Datalog query language [58], while our domain independence is a purely declarative set-theoretic concept.

The relational calculus does not differentiate between different interpretations of quantification (unlimited, countable or finite invention) [51, Theorem 3.1], but the situation is less clear for query languages with predefined operations [10], such as Alignment Calculus. For example, Eq. (3.1) demonstrates that at least finite invention is insufficient.

Our main tool in reasoning about domain independence is the *finiteness dependency* [68] used previously for example in analyzing the safety of Datalog queries [56]. In our setting, this concept can be defined as follows.

Definition 3.2 *Let ϕ be an Alignment Calculus formula and X, Y subsets of* $\mathrm{free}(\phi) = \{x_1, \ldots, x_n\}$*. Then ϕ* satisfies the finiteness dependency $X \rightsquigarrow Y$ *if and only if for all databases db and finite $A \subseteq \Sigma^*$ the set*

$$\{v_j \colon \langle v_1, \ldots, v_n \rangle \in [\![\phi]\!]_{db},\ x_j \in Y,\ \text{and } v_i \in A \text{ for all } x_i \in X\}$$

remains finite.

The intuition is that whenever the values for the variables in X are drawn from a finite set A, the acceptable values for the variables Y are also constrained to be finite.

If a formula ϕ is domain independent, then it satisfies the finiteness dependency $\emptyset \rightsquigarrow \mathrm{free}(\phi)$, because all strings appearing in $[\![\phi]\!]_{db}$ are drawn from $\Sigma^{\leq \mathcal{W}_\phi(db)}$. The converse does not hold in general, however: Eq. (3.1) satisfies the finiteness dependency $\emptyset \rightsquigarrow \emptyset$, but is not domain independent, because its bound variables were not constrained in any way. On the other hand, the converse does hold when ϕ is quantifier-free.

As an example of a safe query in our formalism consider

$$\exists y, z.R_1(y, z) \wedge \phi \tag{3.2}$$

where

$$\phi \equiv ([x, y]_l x = y)^*.([x, z]_l x = z)^*.([x, y, z]_l x = y = z = \perp),$$

which is Example 2.3 from Section 2.3, except that now the result x of the concatenation is no longer retrieved from a database relation, but is instead generated during query evaluation. The answer set is finite by the following reasoning. There are only finitely many $\langle v, w \rangle \in db(R_1)$ to provide values for variables y and z. Hence, the possible range of the variable x is also finite, namely $\{vw \colon \langle v, w \rangle \in db(R_1)\}$. This query is therefore domain independent, and its limitation function is $\mathcal{W}(db) = \max\{|v| + |w| : \langle v, w \rangle \in db(R_1)\}$, or any majorant thereof.

The reasoning above used implicitly the fact that the string formula ϕ satisfied a limitation function of its own, namely that $\langle u, v, w \rangle \in [\![\phi]\!]$ only if $|u| = |v| + |w|$. We capture these "atomic" limitation functions for string formulae, or equivalently FSAs, by the following definition.

Definition 3.3 *Let $\mathcal{A}$ be a $(k + l)$-FSA. Its tapes $1, \ldots, k$* limit *the other tapes $k+1, \ldots, k+l$ if and only if there is a limitation function $\mathcal{W}_{\mathcal{A}} : \mathbb{N}^k \to \mathbb{N}$, for which $\langle u_1, \ldots, u_k, v_1, \ldots, v_l \rangle \in L(\mathcal{A})$ implies* $\max\{|v_i| : 1 \leq i \leq l\} \leq \mathcal{W}_{\mathcal{A}}(|u_1|, \ldots, |u_k|)$.

Definition 3.3 specifies a finiteness dependency

$$\Gamma = \{1, \ldots, k\} \rightsquigarrow \{k+1, \ldots, k+l\}$$

that spans *all the tapes of* $\mathcal{A}$, as in Definition 3.2; in other words, it must be a *full* dependency. Moreover, it associates with Γ a limitation function $\mathcal{W}_\mathcal{A}$ reminiscent of the limitation functions associated with domain independence in Definition 3.1. Hence, Definition 3.3 provides the building bricks for a bridge between domain independence and finiteness dependencies.

The *limitation problem* is to determine the existence (and maybe even a method of computing the values) of $\mathcal{W}_\mathcal{A}$ given $\mathcal{A}$ and Γ. Intuitively the limitation problem is, "Consider $\mathcal{A}$ no longer an acceptor but a transducer with input tapes $1, \ldots, k$, output tapes $k+1, \ldots, k+l$, nondeterministic control and no requirement to consume input while producing output. Does $\mathcal{A}$ have any input $\langle u_1, \ldots, u_k \rangle$ for which infinitely many different outputs $\langle v_1, \ldots, v_l \rangle$ are possible?" This transducer reformalization will be discussed further in Section 5.2.

This problem of actually constructing $\mathcal{W}_\mathcal{A}$ is similar to the problem of estimating how a given logic program [58] alters the sizes of its arguments during evaluation [86, 96]. Our situation is easier, however, because the iterative behaviour of FSAs $\mathcal{A}$ is generally simpler than the recursive behaviour of logic programs. We shall encounter such a construction in Subsection 5.2.3.

In Eq. (3.2), we used the finiteness dependency $\{y, z\} \rightsquigarrow \{x\}$ satisfied by ϕ, or "given two strings, concatenating them yields only finitely many answers." Other conditions satisfied by ϕ are for example $\{x\} \rightsquigarrow \{y, z\}$ ("a given string can be divided finitely into a prefix and a suffix"), and $\{x, y\} \rightsquigarrow \{z\}$ ("given a string and its prefix, the suffix can be finitely extracted"). The condition $\{y\} \rightsquigarrow \{x, z\}$ ("a string can be extended in only finitely many ways") is an example of a condition not satisfied by ϕ.

As an example of the limitation functions from Definition 3.3, let us consider the the case, in which none of the tapes to be limited is bidirectional. Then the functions are polynomials of degree $p+1$, where p is the number of bidirectional limiting tapes.

Theorem 3.3 *Let $\mathcal{A}$ be a $(p+q+r)$-FSA with tapes $1, \ldots, p$ bidirectional and others undirectional satisfying the neatness conditions of Corollary 3.1. If $\mathcal{A}$ satisfies the finiteness dependency*

$$\Gamma = \{1, \ldots, p+q\} \rightsquigarrow \{p+q+1, \ldots, p+q+r\},$$

then

$$\mathcal{W}_{\mathcal{A}}^{\Gamma}(m_1, \ldots, m_p, n_1, \ldots, n_q) = \\ (a-2)\left(\prod_{i=1}^{p}(m_i+2)\right)\left(\max(q,1)+\sum_{j=1}^{q} n_j\right) - 1,$$

where a is the number of states in $\mathcal{A}$, is a corresponding limitation function.

Proof: Assume to the contrary that $\mathcal{A}$ has an accepting computation $\mathcal{C} = C_1 \ldots C_f$ on some input $\langle u_1, \ldots, u_p, v_1, \ldots, v_q, w_1, \ldots, w_r \rangle$, where some $|w_k| > b = \mathcal{W}_{\mathcal{A}}^{\Gamma}(|u_1|, \ldots, |u_p|, |v_1|, \ldots, |v_q|)$.

If tape head $p+q+k$ does not visit \$ during $\mathcal{C}$, then Γ fails, because $\mathcal{C}$ is an accepting computation for the inputs

$$\mathbf{u} = \langle u_1, \ldots, u_p, v_1, \ldots, v_q, w_1, \ldots, w_{k-1}, w_k\Sigma^*, w_{k+1}, \ldots, w_r \rangle$$

as well. Therefore tape head $p+q+k$ visits every square, and therefore moves (to direction $+1$) at least $b+1$ times, during $\mathcal{C}$.

Let us post a *fence* $C_m | C_{m+1}$ between two consecutive configurations C_m and C_{m+1} of $\mathcal{C}$ whenever tape head $p+q+k$ moved on the transition t_m taken in C_m to reach C_{m+1}. By the reasoning above and condition 2 of Corollary 3.1, at least $b+1$ of these fences are posted.

On the other hand, *color* the configurations in $\mathcal{C}$ as follows. Two configurations C_m and C_n receive the same color if and only if they share the same state and the same head positions for tapes $1, \ldots, p+q$. This takes no more than $b+1$ colors: the first multiplicative factor $(a-2)$ in the expression for $\mathcal{W}_{\mathcal{A}}^{\Gamma}$ is an upper bound for the number of states that can repeat during a computation; conditions 1 and 2 of Corollary 3.1 show that neither the start nor the final state of $\mathcal{A}$ can repeat. In turn, the second multiplicative factor is an upper bound on the number of different head positions for the bidirectional input tapes $i = 1, \ldots, p$ that can occur during a computation. Finally, the third and last multiplicative factor is an upper bound on the number of different head positions for the unidirectional input tapes $j = p+1, \ldots, p+q$ that can occur during a computation.

$\mathcal{C}$ must therefore contain two configurations C_m and C_n with $m < n$, which share the same color, but are separated by a fence. Let $T_1 = t_1 \ldots t_{m-1}$, $T_m = t_m \ldots t_{n-1}$, and $T_n = t_n \ldots t_{f-1}$ be the division of the sequence $t_1 \ldots t_{f-1}$ of transitions taken in $\mathcal{C}$ into the part T_1 entering C_m from the initial configuration C_1, the part T_m entering C_n from C_m, and the part T_n entering the final configuration C_f from C_n. Also let each w_l be divided into $w_{(l,1)}w_{(l,m)}w_{(l,n)}$, where $w_{(l,o)}$ consists of the characters from Σ

that tape head $p+q+l$ lands onto during the sequence T_o of transitions. Then $\mathcal{A}$ accepts also each input

$$\mathbf{u}^g = \left\langle u_1, \ldots, u_p, v_1, \ldots, v_q, w_{(1,1)}(w_{(1,m)})^g w_{(1,n)}, \ldots, w_{(r,1)} w^g_{(r,m)} w_{(r,n)} \right\rangle$$

with $g \in \mathbb{N}$, because $\mathbf{u}^g$ is accepted by the computation $\mathcal{C}'$, which takes the sequence $T_1(T_m)^g T_n$ of transitions. The fact that $\mathcal{C}'$ is indeed a valid computation follows from condition 4 of Corollary 3.1 and the unidirectionality of tapes $p+1, \ldots, p+q+r$. This in turn contradicts Γ, because at least $w_{(k,m)} \neq \varepsilon$ since there was a fence between C_m and C_n in $\mathcal{C}$. □

The limitation problem extends naturally to any partition of tapes, or equivalently variables, to inputs and outputs. Hence, we try to determine the ways in which a given string acceptance device can be used safely as a string production device in this particular query, rather than forcing the database programmer to work with explicit string production devices as in some other languages [13, 38, 61, 100]. Related transducer problems have been studied by Weber [101, 102]. We defer, however, further discussion of the limitation problem and inferring domain independence to Chapter 4.

3.3 Alignment Algebra

Let us now define *Alignment Algebra*, our string-oriented counterpart to standard relational algebra [2, Chapter 5.1][24, Chapter 6][92, Chapter 2.4].

relation symbols: If R is a relation symbol of arity k, it is also an expression in Alignment Algebra of the same arity. Its *value* when applied to particular database db is $db(R)$, the relation assigned to symbol R by db.

domain symbols: Σ^* is an expression of arity 1 whose value is Σ^*, the set of all finite strings from the alphabet Σ.

In addition $\Sigma^{\leq l}$, where $l \in \mathbb{N}$ is a constant, is an expression of arity 1 with $db(\Sigma^{\leq l}) = \{u \in \Sigma^* : |u| \leq l\}$ as its value.

selection: If E is an expression and $\mathcal{A}$ an $a(E)$-FSA, then $\sigma_{\mathcal{A}} E$ is an expression of arity $a(E)$ with $db(\sigma_{\mathcal{A}} E) = db(E) \cap L(\mathcal{A})$. In other words, selection is performed by retaining only the tuples of E that $\mathcal{A}$ accepts. That is, (only) selections do have access to the *internal structure* of the strings they compare.

set difference: If E and F are expressions with equal arities, then $E \setminus F$ is an expression of this common arity with $db(E \setminus F) = db(E) \setminus db(F)$.

union: If E and F are expressions with equal arities, then $E \cup F$ is an expression of this common arity with $db(E \cup F) = db(E) \cup db(F)$.

Cartesian product: If E and F are expressions, then $E \times F$ is an expression of arity $a(E) + a(F)$, for which

$$\left\langle u_1, \ldots, u_{a(E)}, v_1, \ldots, v_{a(F)} \right\rangle \in db(E \times F)$$

holds if and only if both $\left\langle u_1, \ldots, u_{a(E)} \right\rangle \in db(E)$ and $\left\langle v_1, \ldots, v_{a(F)} \right\rangle \in db(F)$ do hold.

projection: If E is an expression and $i_1, \ldots, i_u$ a list of distinct numbers between 1 and $a(E)$, then $\pi_{i_1,\ldots,i_u} E$ is an expression of arity u with

$$db(\pi_{i_1,\ldots,i_u} E) = \left\{ \langle v_{i_1}, \ldots, v_{i_u} \rangle : \left\langle v_1, \ldots, v_{a(E)} \right\rangle \in db(E) \right\}.$$

It is also possible to have $u = 0$ in which case $db(\pi E)$ is a relation of arity 0; it is $\emptyset$ if $db(E) = \emptyset$ and the full relation $\{\langle\,\rangle\}$ otherwise. Note that $\{\langle\,\rangle\}$ can be represented algebraically with for example $\pi\Sigma^{\leq 0}$.

There are no other expressions in Alignment Algebra. However, we do introduce the shorthand $E \cap F$ for $E \setminus (E \setminus F)$.

We define a sublanguage of full Alignment Algebra that intends to capture those expressions, in which string restructuring is controlled to be finite by an immediately enclosing selection.

Definition 3.4 *Let E be an Alignment Algebra expression, and let $\mathcal{A}$ be an $(a(E) + n)$-FSA, which satisfies the limitation condition*

$$\{1, \ldots, a(E)\} \rightsquigarrow \{a(E) + 1, \ldots, a(E) + n\}.$$

Then the selection expression $\sigma_{\mathcal{A}}(E \times (\Sigma^*)^n)$ is called a generator *.*

An Alignment Algebra expression is called finitely evaluable *if and only if Σ^* occurs only within generators.*

The intuition behind Definition 3.4 is that the infinitary operator Σ^* is constrained to appear only in a position whose possible values are constrained to be finite by Definition 3.3. Thus, a finitely evaluable expression does indeed admit a finite evaluation strategy, provided that its generators can be evaluated finitely to produce their finite results.

Another view of Definition 3.4 is that $\mathcal{A}$ is now considered an extended *transducer* [49, Chapter 2.7], which maps an input tuple from $db(E)$ into a set of output tuples from $(\Sigma^*)^n$. The evaluation strategy then consists of computing these mappings, and Definition 3.3 ensures that $\mathcal{A}$ maps every input into a finite set. We will consider the problem of evaluating generators in Section 5.2, after we have seen what kinds of generators, or transducers, do arise during query translation in Section 5.1. Subsection 1.2.2 discussed related languages, where transducers were added *explicitly* into the language.

Example 3.1 Unions are strictly speaking unnecessary in Alignment Algebra, because the same result can be achieved with generators. This contrasts with the standard relational algebra [2, Exercise 5.5].

Consider namely the expression $E \cup F$ with $a(E) = a(F) = n$. Let us study the string formula

$$\begin{aligned}\phi \equiv &(([x_1, \ldots, x_n, x_{2n+1}, \ldots, x_{3n}]_l x_1 = x_{2n+1} \wedge \ldots \wedge x_n = x_{3n})^* . \\ &\quad [x_1, \ldots, x_n, x_{2n+1}, \ldots, x_{3n}]_l x_1 = \ldots = x_n = x_{2n} = \ldots = x_{3n} = \bot) \\ &+ \\ &(([x_{n+1}, \ldots, x_{3n}]_l x_{n+1} = x_{2n+1} \wedge \ldots \wedge x_{2n} = x_{3n})^* . \\ &\quad [x_{n+1}, \ldots, x_{3n}]_l x_{n+1} = \ldots = x_{3n} = \bot).\end{aligned}$$

Now $\langle u_1, \ldots, u_n, v_1, \ldots, v_n, w_1, \ldots, w_n \rangle \in [\![\phi]\!]$ if and only if either $u_i = w_i$ for all $i = 1, \ldots, n$ or $v_i = w_i$ for all $i = 1, \ldots, n$. Let $\mathcal{A}$ be the $3n$-FSA corresponding to ϕ by Theorem 3.1. Then $\mathcal{A}$ satisfies the limitation property $\Gamma = \{1, \ldots, 2n\} \rightsquigarrow \{2n+1, \ldots, 3n\}$ by construction. Hence $G = \sigma_{\mathcal{A}}(E \times F \times (\Sigma^*)^n)$ is a generator, and the expression $\pi_{2n+1,\ldots,3n} G$ is equivalent to $E \cup F$. ■

Let us now prove that Alignment Calculus and Algebra are equivalent in their expressive power. First we prove that Alignment Algebra is no more expressive than Alignment Calculus, and further that finitely evaluable expressions are no more expressive than domain independent queries.

Theorem 3.4 *Let E be an Alignment Algebra expression. There exists an Alignment Calculus formula ϕ_E, for which $db(E) = [\![\phi_E]\!]_{db}$ holds for every database $db \in DB$.*

Moreover, if E is finitely evaluable, then ϕ_E is domain independent.

Proof: We prove these claims by induction on the structure of E. Let furthermore free $(\phi_E) = \left\{x_1, \ldots, x_{a(E)}\right\}$ for specificity.

If E is a relation symbol R, then $\phi_E \equiv R(x_1, \ldots, x_{a(R)})$. The correctness of this choice follows directly from Eq. (2.4) and truth definition 10. E is finitely evaluable, and ϕ_E is domain independent with the function

$\mathcal{W}_{\phi_E}(db) = \max(R, db)$, where the function

$$\max(R, db) = \max\left\{|u_i| : \left\langle u_1, \ldots, u_{a(R)}\right\rangle \in db(R), 1 \leq i \leq a(R)\right\} \tag{3.3}$$

computes the maximum length of any string assigned into R by db.

If $E = \Sigma^*$, then $\phi_E \equiv \phi_1$ suffices, where

$$\phi_i \equiv [\,]_l x_i = \bot, \tag{3.4}$$

is a tautology with variable symbol x_i; that is, $\langle A_0, db\rangle \models \phi_i\theta$ is true for all db, A_0 and θ, while free$(\phi_i) = \{x_i\}$. E is not finitely evaluable outside generators.

If $E = \Sigma^{\leq l}$ for $l \in \mathbb{N}$, then $\phi_E \equiv ([x_1]_l \top)^l.[x_1]_l x_1 = \bot$, because $\langle A_0, db\rangle \models \phi_E\theta$ if and only if $|\sigma_{A_0}(\theta x_1)| \leq l$. E is finitely evaluable, and ϕ_E is domain independent with the constant function $\mathcal{W}_{\phi_E}(db) = l$.

Consider then $E = F \setminus G$. The first case of Lemma 2.2 (2) yields

$$[\![\phi \wedge \psi]\!]_{db} = [\![\phi]\!]_{db} \cap [\![\psi]\!]_{db} \text{ when } \operatorname{free}(\phi) = \operatorname{free}(\psi)\,. \tag{3.5}$$

By definition, the inductively obtained formulae ϕ_F and ϕ_G indeed do have free$(\phi_F) =$ free(ϕ_G). Moreover, the first case of Lemma 2.2 (1) yields that $[\![\neg\psi]\!]_{db}$ is the complement of $[\![\psi]\!]_{db}$. Therefore by Eq. (3.5)

$$[\![\underbrace{\phi_F \wedge \neg\phi_G}_{\phi_E}]\!]_{db} = [\![\phi_F]\!]_{db} \setminus [\![\phi_G]\!]_{db}, \tag{3.6}$$

as required. If E is finitely evaluable, so are F and G. Hence, ϕ_F and ϕ_G are domain independent with functions $\mathcal{W}_{\phi_F}$ and $\mathcal{W}_{\phi_G}$. Then ϕ_E is then also domain independent with function $\mathcal{W}_{\phi_E}(db) = \max(\mathcal{W}_{\phi_F}(db), \mathcal{W}_{\phi_G}(db))$: the derivation of Eq. (3.6) above remains equally valid for truncations $[\![\ldots]\!]^{\ell}_{db}$, and therefore $[\![\phi_E]\!]^{\ell}_{db} = [\![\phi_F]\!]^{\ell}_{db} \setminus [\![\phi_G]\!]^{\ell}_{db} = [\![\phi_F]\!]_{db} \setminus [\![\phi_G]\!]_{db} = [\![\phi_E]\!]_{db}$, when $\ell \geq \mathcal{W}_{\phi_E}(db)$, as required.

Note that choosing $\mathcal{W}_{\phi_E} = \mathcal{W}_{\phi_F}$ is not sufficient, because the answer truncations $[\![\phi_G]\!]^{\ell}_{db}$ do not necessarily grow monotonically with respect to increasing the truncation length ℓ. To see this, consider the domain independent formula

$$\psi \equiv R(x) \wedge \neg\exists y.R(y) \wedge ([x, y]_l x \neq \bot)^*.([x, y]_l x = \bot \wedge y \neq \bot),$$

which extracts the longest strings from $db(R)$. On the other hand, every truncation $[\![\psi]\!]^{\ell}_{db}$ examines only those strings from $db(R)$, whose lengths are at most ℓ, and selects the longest ones from among them. Thus, $[\![\psi]\!]^{\ell-1}_{db} \cap [\![\psi]\!]^{\ell}_{db} = \emptyset$ whenever $db(R)$ contains strings of length ℓ.

Consider then $E = F \cup G$. The definition of $\phi \vee \psi$ as a shorthand for $\neg((\neg\phi) \wedge (\neg\psi))$, the first part of Lemma 2.2 (1), Eq. (3.5), and DeMorgan's laws yield

$$[\![\phi \vee \psi]\!]_{db} = [\![\phi]\!]_{db} \cup [\![\psi]\!]_{db} \text{ when } \operatorname{free}(\phi) = \operatorname{free}(\psi). \tag{3.7}$$

Since the inductively obtained formulae ϕ_F and ϕ_G do have $\operatorname{free}(\phi_F) = \operatorname{free}(\phi_G)$ by definition, Eq. 3.7 leads us to choose $\phi_E \equiv \phi_F \vee \phi_G$. The second claim is verified as in the previous case.

Consider then $E = F \times G$. The first part of Lemma 2.2 (2) leads to $[\![\phi \wedge \psi]\!]_{db} = [\![\phi]\!]_{db} \times [\![\psi]\!]_{db}$ when $\operatorname{pos}(\operatorname{free}(\phi \wedge \psi), x) < \operatorname{pos}(\operatorname{free}(\phi \wedge \psi), y)$ for all $x \in \operatorname{free}(\phi)$ and $y \in \operatorname{free}(\psi)$. Hence we proceed as follows. Obtain inductively ϕ_F and ϕ_G. By suitably renaming the variables of ϕ_G, we can obtain a ϕ'_G satisfying $[\![\phi'_G]\!]_{db} = [\![\phi_G]\!]_{db}$, but with the free variables

$$\operatorname{free}(\phi'_G) = \left\{x_{|\operatorname{free}(\phi_F)|+1}, \ldots, x_{|\operatorname{free}(\phi_F)|+|\operatorname{free}(\phi_G)|}\right\}.$$

Then it remains to choose $\phi_E \equiv \phi_F \wedge \phi'_G$. The second claim is verified as in the previous two cases.

Consider then $E = \pi_{i_1,\ldots,i_u} F$. Obtain inductively ϕ_F. Let p be any permutation of $\{1, \ldots, a(F)\}$, where $p(i_j) = j$ for $1 \leq j \leq u$. As in the previous case, suitable renaming of ϕ_F provides a ϕ'_F, for which $\left\langle v_1, \ldots, v_{a(F)}\right\rangle \in [\![\phi_F]\!]_{db}$ holds if and only if $\left\langle v_{p(1)}, \ldots, v_{p(a(F))}\right\rangle \in [\![\phi'_F]\!]_{db}$. The first part of Lemma 2.2 (3) then leads us to choose $\phi_E \equiv \exists x_{u+1} \ldots \exists x_{a(F)} \phi'_F$. If F is finitely evaluable, then ϕ_F is domain independent, and therefore so is ϕ'_F. Hence, ϕ_E is also domain independent, and with the same function $\mathcal{W}_{\phi_E} = \mathcal{W}_{\phi'_F} = \mathcal{W}_{\phi_F}$, because $[\![\phi]\!]_{db} = [\![\phi]\!]^{\ell}_{db}$ implies $[\![\exists x \phi]\!]_{db} = [\![\exists x \phi]\!]^{\ell}_{db}$ by Lemma 2.2 (3).

Consider finally $E = \sigma_{\mathcal{A}} F$. Obtain inductively ϕ_F and the string formula ψ corresponding to $\mathcal{A}$ from Theorem 3.2. Because by definition $\operatorname{free}(\psi) = \operatorname{free}(\phi_F)$, Eq. (3.5) leads us to choose $\phi_E \equiv \psi \wedge \phi_F$. If E is finitely evaluable, then $F = G \times (\Sigma^*)^n$ for some finitely evaluable G of arity $m = a(F) - n$, and $\mathcal{A}$ satisfies the limitation condition $\Gamma = \{1, \ldots, m\} \rightsquigarrow \{m+1, \ldots, m+n\}$. This construction has furthermore produced $\phi_E = \psi \wedge \phi_G \wedge \bigwedge_{i=1}^{n} \phi_{m+i}$, where ϕ_G is domain independent, and the ϕ_{m+i} are as in Eq. (3.4). We claim that this ϕ_E is domain independent with function

$$\mathcal{W}_{\phi_E}(db) = \max\left\{\mathcal{W}_{\mathcal{A}}(n_1, \ldots, n_m) : n_1, \ldots, n_m \leq \mathcal{W}_{\phi_G}(db)\right\},$$

where $\mathcal{W}_{\mathcal{A}} \colon \mathbb{N}^m \to \mathbb{N}$ is a limitation function associated with $\mathcal{A}$ and Γ in Definition 3.3. We can safely assume this $\mathcal{W}_{\mathcal{A}}$ to be chosen so that it satisfies the monotonicity condition

$$\mathcal{W}_{\mathcal{A}}(n_1, \ldots, n_m) \geq \max\{n_j \colon 1 \leq j \leq m\}. \tag{3.8}$$

We have by Lemma 2.2 (2) that $\langle u_1, \ldots, u_m, v_1, \ldots, v_n \rangle \in [\![\phi_E]\!]_{db}$ if and only if $\langle u_1, \ldots, u_m \rangle \in [\![\phi_G]\!]_{db}$ and $\langle u_1, \ldots, u_m, v_1, \ldots, v_n \rangle \in L(\mathcal{A}) = [\![\psi]\!]_{db}$. Γ implies $|v_j| \leq \mathcal{W}_{\mathcal{A}}(|u_1|, \ldots, |u_m|)$ for $1 \leq j \leq n$. Because ϕ_G is domain independent, $[\![\phi_G]\!]_{db} = [\![\phi_G]\!]^{\ell}_{db}$ for $\ell \geq \mathcal{W}_{\phi_G}(db)$; in particular, $|u_i| \leq \mathcal{W}_{\phi_G}(db)$ for $1 \leq i \leq m$. Hence we have $[\![\phi_E]\!]_{db} = [\![\phi_E]\!]^{\ell'}_{db}$, when $\ell' \geq \mathcal{W}_{\phi_E}(db)$, as required. □

Now we proceed to prove the converse direction that all of Alignment Calculus can also be translated to Alignment Algebra. At the same time we demonstrate the role Σ^* plays in the result: it is the universe from which the strings $\sigma_{A_0}(\theta x)$ are drawn when verifying the truth of ϕ, as anticipated in Lemma 2.2 (3). In order to state this property precisely, we introduce the notion of *restricting an Alignment Algebra expression into a given length*, as follows. If E is an Alignment Algebra expression and $k \in \mathbb{N}$ a constant, then $E \upharpoonright k$ is the Alignment Algebra expression obtained by replacing each occurrence of Σ^* with an occurrence of $\Sigma^{\leq k}$ in E.

However, first we show how to build Alignment Algebra expressions that are analogous to *natural joins* [2, pages 56–58][24, pages 152–153][92, pages 59–60] in relational algebra.

Lemma 3.1 *Let E be an Alignment Algebra expression, and $\mathcal{P}$ an ordered partition of the set $\{1, \ldots, a(E)\}$ into nonempty subsets $P_1, \ldots, P_r$. Then there exists an Alignment Algebra expression $E^{\mathcal{P}}$ of arity r, for which the following property holds: $\langle u_1, \ldots, u_r \rangle \in db(E^{\mathcal{P}})$ if and only if there exists some $\left\langle v_1, \ldots, v_{a(E)} \right\rangle \in db(E)$, for which $j \in P_i$ implies $v_j = u_i$ for all $1 \leq j \leq a(E)$.*

Moreover, $E^{\mathcal{P}}$ is finitely evaluable if and only if E is.

Proof: Construct first an $a(E)$-FSA $\mathcal{A}$ to check the following condition: if $j_1, j_2 \in P_i$, then tapes j_1 and j_2 contain exactly the same string. This construction proceeds as follows. Consider the set

$$Q = \{x_j = x_{\min P_i} \colon j \in P_i \setminus \{\min P_i\}, 1 \leq i \leq r\},$$

of atomic string formulae, and let τ be the left transpose of all the variables appearing in Q. Form then the string formula

$$\phi \equiv \left(\tau \bigwedge_{e \in Q} e\right)^* . \tau x_1 = \ldots = x_{a(E)} = \bot.$$

Now $\mathcal{A}$ is the translation of this ϕ via Theorem 3.1, because ϕ iterates the characterwise comparisons in Q until the compared strings are exhausted. The comparisons in Q specify in turn the desired condition.

Given $\mathcal{A}$, it suffices to construct $E^{\mathcal{P}} = \pi_{\min P_1,\ldots,\min P_r}\sigma_{\mathcal{A}}E$ as the desired expression. $\square$

Theorem 3.5 *Let ϕ be a formula in Alignment Calculus. There exists an Alignment Algebra expression E_ϕ, for which $[\![\phi]\!]_{db} = db(E_\phi)$ holds for every database $db \in DB$.*

Moreover, $[\![\phi]\!]^{\ell}_{db} = db(E_\phi \upharpoonright \ell)$ for every $\ell \geq \max(\phi, db)$, where the function

$$\max(\phi, db) = \max\{\max(R, db) \colon \textit{relation symbol } R \textit{ occurs in } \phi\}$$

computes the maximum length of any string that db stores into a relation mentioned in ϕ. (The function $\max(R, db)$ *was defined in Eq. (3.3).)*

Proof: We construct E_ϕ by induction on the structure of ϕ.

If ϕ is a relational atom $R(\nu_1, \ldots, \nu_{a(R)})$, then it is translated as follows. Form for each $x \in \text{free}(\phi)$ the set $P_x = \{i \colon x \text{ occurs as } \nu_i\}$, and order these sets into an ordered partition $\mathcal{P}$ of $\{1, \ldots, a(R)\}$. Then form $E_\phi = R^{\mathcal{P}}$ by Lemma 3.1. The first claim follows by the construction of $\mathcal{P}$. The second claim follows from the first, because $E_\phi \upharpoonright \ell = E_\phi$, and $\max(\phi, db) = \max(R, db)$.

If ϕ is a string formula, then $[\![\phi]\!]_{db} = L(\mathcal{A})$, where $\mathcal{A}$ is the translation of ϕ by Theorem 3.1. $L(\mathcal{A})$ can in turn be expressed in Alignment Algebra with $E_\phi = \sigma_{\mathcal{A}}\left((\Sigma^*)^{|\text{free}(\phi)|}\right)$. The second claim also holds for all $\ell \geq \max(\phi, db) = 0$, because $[\![\phi]\!]^{\ell}_{db}$ is the restriction of $L(\mathcal{A})$ into strings of length at most ℓ, and so is $E_\phi \upharpoonright \ell$.

If $\phi \equiv \exists x \psi$, then construct first inductively the translation E_ψ of the subformula ψ. If $x \notin \text{free}(\psi)$, then $[\![\phi]\!]_{db} = [\![\psi]\!]_{db}$, and so $E_\phi = E_\psi$ suffices unmodified. Otherwise we reason as follows. Let $m = \text{pos}(\text{free}(\psi), x)$. By the first part of Lemma 2.2 (3), $\left\langle u_1, \ldots, u_{|\text{free}(\phi)|}\right\rangle \in [\![\phi]\!]_{db}$ if and only if there exists some $v \in \Sigma^*$, for which $\left\langle u_1, \ldots, u_{m-1}, v, u_{m+1}, \ldots, u_{|\text{free}(\phi)|}\right\rangle \in [\![\psi]\!]_{db}$ holds. This is in turn equivalent to

$$\left\langle u_1, \ldots, u_{|\text{free}(\phi)|}\right\rangle \in db(\underbrace{\pi_{1,\ldots,m-1,m+1,\ldots,|\text{free}(\psi)|}E_\psi}_{E_\phi}),$$

because $[\![\psi]\!]_{db} = db(E_\psi)$ by induction, fulfilling the first claim. The second claim follows by similar reasoning, because inductively $[\![\psi]\!]^{\ell}_{db} = db(E_\psi \upharpoonright \ell)$, when $\ell \geq \max(\phi, db) = \max(\psi, db)$.

If $\phi \equiv \neg\psi$, then the first part of Lemma 2.2 (1) yields for the first claim the choice $E_\phi = (\Sigma^*)^{|\mathrm{free}(\psi)|} \setminus E_\psi$, where E_ψ was provided by induction. This choice satisfies the second claim as well, by induction and the second part of Lemma 2.2 (1).

The only remaining case $\phi \equiv \psi_1 \wedge \psi_2$ is translated as follows. For each $x \in \mathrm{free}(\phi)$, form the set P_x to contain $\mathrm{pos}(\mathrm{free}(\psi_1), x)$ if $x \in \mathrm{free}(\psi_1)$, and $\mathrm{pos}(\mathrm{free}(\psi_2), x) + |\mathrm{free}(\psi_1)|$ if $x \in \mathrm{free}(\psi_2)$. Then form the ordered partition $\mathcal{P}$ of $\{1, \ldots, \mathrm{free}(\psi_1) + \mathrm{free}(\psi_2)\}$ into nonempty parts by ordering these P_x, and choose $E_\phi = (E_{\psi_1} \times E_{\psi_2})^{\mathcal{P}}$, where each E_{ψ_i} is obtained inductively. The first claim then follows by induction, the construction of $\mathcal{P}$, and the first part of Lemma 2.2 (2). The second claim follows similarly via the second part of Lemma 2.2 (2). $\square$

The second claim of Theorem 3.5 shows that given a value $\ell \in \mathbb{N}$ we can evaluate $[\![\phi]\!]^{\ell}_{db}$ finitely by considering each Σ^* in E_ϕ a generic symbol standing for $\Sigma^{\leq \ell}$. On the other hand, consider for any domain independent formula ϕ and database db any value $\ell \geq \mathcal{W}_\phi(db)$, where $\mathcal{W}_\phi \colon DB \to \mathbb{N}$ is a limitation function for ϕ; then $[\![\phi]\!]_{db} = [\![\phi]\!]^{\ell}_{db}$. Hence, we conclude that domain independent formulae of Alignment Calculus are finitely evaluable using Alignment Algebra with

$$[\![\phi]\!]_{db} = db(E_\phi \restriction \mathcal{W}_\phi(db)). \tag{3.9}$$

That is, given any database db we first determine $\mathcal{W}_\phi(db)$ and then restrict attention to strings of at most that length when evaluating $db(E_\phi)$, making it a finitary operation. Chapter 4 examines in more detail the problem of inferring whether $\mathcal{W}_\phi$ exists for a given query ϕ.

Example 3.2 Consider the domain independent query $\phi \equiv \exists y, z.R_1(y) \wedge R_3(z) \wedge \psi$, where ψ is the formula in Fig. 3.1 on page 40. (See also Example 2.3 on page 33). The query asks for the strings that are a concatenation of a string from $db(R_1)$ and another string from $db(R_3)$, as was mentioned before. Its translation to Alignment Algebra via Theorem 3.5 is as follows.

$$\begin{aligned}
E_{R_1(y)} &= R_1^{\langle\{1\}\rangle} \\
&= R_1, \\
E_{R_3(z)} &= R_3^{\langle\{1\}\rangle} \\
&= R_3, \\
E_\psi &= \sigma_{\mathcal{A}}(\Sigma^* \times \Sigma^* \times \Sigma^*),
\end{aligned}$$

where $\mathcal{A}$ is the 3-FSA corresponding to ψ from Theorem 3.1 (with variable x on tape 1, y on 2, and z on 3.)

$$\begin{aligned}
E_{R_1(y)\wedge R_3(z)} &= (R_1 \times R_3)^{\langle\{1\},\{2\}\rangle} \\
&= R_1 \times R_3, \\
E_{R_1(y)\wedge R_3(z)\wedge\psi} &= (E_{R_1(y)\wedge R_3(z)} \times E_\psi)^{\mathcal{P}} \\
&= \pi_{3,1,2}\sigma_{\mathcal{B}}(R_1 \times R_3 \times \sigma_{\mathcal{A}}(\Sigma^* \times \Sigma^* \times \Sigma^*)),
\end{aligned}$$

where $\mathcal{P} = \langle\{3\}, \{1,4\}, \{2,5\}\rangle$, and $\mathcal{B}$ is the 5-FSA corresponding to $\mathcal{P}$ built in Lemma 3.1.

$$\begin{aligned}
E_{\exists z.R_1(y)\wedge R_3(z)\wedge\psi} &= \pi_{1,2}E_{R_1(y)\wedge R_3(z)\wedge\psi} \\
&= \pi_{3,1}\sigma_{\mathcal{B}}(R_1 \times R_3 \times \sigma_{\mathcal{A}}(\Sigma^* \times \Sigma^* \times \Sigma^*)), \\
E_\phi &= \pi_1 E_{\exists z.R_1(y)\wedge R_3(z)\wedge\psi} \\
&= \pi_3\sigma_{\mathcal{B}}(R_1 \times R_3 \times \sigma_{\mathcal{A}}(\Sigma^* \times \Sigma^* \times \Sigma^*)) \\
&= \pi_1\sigma_{\mathcal{A}}(\Sigma^* \times R_1 \times R_3).
\end{aligned}$$

Here thie last simplification follows from the meaning of the 5-FSA $\mathcal{B}$ built during the construction. Note how Σ^* is needed in E_ϕ to implement the generation of the new concatenation strings. Function $\mathcal{W}(db) = \max(R_1, db) + \max(R_3, db)$, where 'max' was defined in Eq. (3.3), is evidently a limitation function for this query. Therefore given any database db it is sufficient to evaluate

$$db(E_\phi \restriction \mathcal{W}(db)) = db\left(\pi_1\sigma_{\mathcal{A}}\left(\Sigma^{\leq \max(R_1,db)+\max(R_3,db)} \times R_1 \times R_3\right)\right)$$

to obtain $[\![\phi]\!]_{db}$, and this is indeed a finitary operation.

However, straightforward evaluation of this expression is exteremely inefficient, because it generates first all the exponentially many possible strings that might be concatenations of strings from $db(R_1)$ and $db(R_3)$, and subsequently filters this intermediate value into the final result. Therefore Chapters 4 and 5 present an alternative method, which purports to be more focused. ■

Chapter 4

Safety Analysis

You can be fast or you can be safe. We offer the option. Users will always choose fast.

Michael Stonebraker

We have up to now assumed that our Alignment Calculus queries were domain independent in the sense of Definition 3.1. An important question is whether the domain independence of a given query can be determined *syntactically*. It is well known that this is undecidable for purely relational formulae [92, Exercise 3.32]. Thus there is no hope that it would be decidable for Alignment Calculus formulae. We shall, however, first look into the possibility of determining safety for string formulae alone, continuing the thread begun in Section 3.2.

It turns out that there is a source of undecidability already at the level of the string formulae. Namely, Section 4.1 will show that the limitation problem of Definition 3.3 is in general undecidable.

Far from everything is lost, however. This undecidability result holds only for those formulae that have at least two bidirectional variables. Indeed, Section 4.2 provides a decision method for the right-restricted string formulae introduced by Definition 2.2. Thus, right-restricted string formulae can be used as a basis for a syntactically safe sublanguage of Alignment Calculus. Our proposal, the (right-restricted) *productive* formulae, will be developed in Section 4.4.

We saw in Section 2.3 that many interesting queries are expressible by using right-restricted string formulae only. Moreover, Theorem 6.5 in Section 6.2 will show that our proposed sublanguage can express exactly the string queries in the Polynomial-time Hierarchy **PH** [87], lending further support to its usefulness.

However, the decision method for the right-restricted string formulae given in Section 4.2 is quite involved. Accordingly, Section 4.3 provides also a more straightforward heuristic for the limitation problem of Definition 3.3.

4.1 Limitation Is Undecidable in General

Let us now prove the undecidability result stated above. The proof consists of encoding Turing machine computations as string formulae. The same method will find further applications also in Chapter 6.

Theorem 4.1 *The limitation problem is undecidable for those string formulae that contain two bidirectional variables.*

Proof: Consider the string formulae ϕ, which have one unidirectional variable x_1 and two bidirectional variables x_2 and x_3. We consider all the different finiteness dependencies that can be formed with these three variables, and show that for each of these dependencies there is a family of string formulae ϕ, whose testing contains an undecidable problem.

First we show that given a ϕ of this kind it is undecidable to determine if the limitation property $\Gamma_1 = \{x_1\} \rightsquigarrow \{x_2, x_3\}$ holds. We reduce the totality problem of Turing machines to this problem. Let M then be a Turing machine with input alphabet Γ, tape alphabet Δ containing the blank symbol '$\sqcup$', the set S of states, and the start state $s_M \in S$. Without loss of generality we assume M to have a two-way infinite tape, and to execute at least one computation step on each input. Our aim is to write a formula ϕ_M of the desired form on an extended finite alphabet Σ' consisting of the distinct sets Γ, Δ, S, and a new separator $\sharp$, which is true if and only if x_1 contains an input string $w \in \Delta^*$, while x_2 and x_3 together represent a finite sequence of computation steps performed by M on input w. Then M is total – that is, M halts on every input – if and only if for all x_1 the set of suitable x_2 and x_3 remains finite; that is, if and only if ϕ_M satisfies the requested limitation property.

One configuration of M can be represented as the string $D = \sharp\alpha p\beta\sharp$, where $\alpha, \beta \in (\Gamma \cup \Delta)^*$, and $p \in S$. This denotes the configuration where M is in state p, the tape contains $\ldots\sqcup\sqcup\sqcup\alpha\beta\sqcup\sqcup\sqcup\ldots$, and the tape head is scanning the first character after α. We construct $[\![\phi_M]\!]$ to consist of the string triples

$$\langle \beta_0, \sharp\alpha_0 s_0 \beta_0 \sharp \ldots \sharp\alpha_{n-1} s_{n-1} \beta_{n-1}\sharp, \sharp\alpha_1 s_1 \beta_1 \sharp \ldots \sharp\alpha_n s_n \beta_n \sharp\rangle\,,$$

where $\alpha_0 = \varepsilon$, $s_0 = s_M$, $\beta_0 \in \Gamma^*$, and for all $1 \leq i \leq n$ the pair $D_{i-1} = \sharp\alpha_{i-1} s_{i-1} \beta_{i-1}\sharp$ and $D_i = \sharp\alpha_i s_i \beta_i \sharp$ of adjacent configurations satisfies the additional constraints that M moves in one computation step from D_{i-1} to D_i. The string formula

$$\begin{aligned}
\phi_{\text{guess}} &\equiv [x_2]_l x_2 = \sharp.[x_2]_l x_2 = s_M.\psi(\Gamma, x_1, x_2)^*.[x_1]_l x_1 = \bot. \\
&\quad [x_2]_l x_2 = \sharp.\gamma(x_2, x_3)^*.\gamma(x_3, x_3).\psi(\{\bot\}, x_2, x_3), \text{ where} \\
\psi(A, x, y) &\equiv \mathop{+}_{c \in A} [x, y]_l x = y = c, \text{ and} \\
\gamma(x, y) &\equiv \psi(\{\sharp\}, x, y).\psi(\Gamma \cup \Delta, x, y)^*.\psi(S, x, y). \\
&\quad \psi(\Gamma \cup \Delta, x, y)^*.\psi(\{\sharp\}, x, y)
\end{aligned}$$

expresses the constraints of ϕ_M other than the one between D_{i-1} and D_i. In order to express that one as well, it now suffices to verify that for each $1 \leq j \leq n$, the jth occurrence of a symbol from S in x_2, namely s_{j-1}, and the jth occurrence of a symbol from S in x_3, namely s_j, satisfy a certain local constraint δ_M. Then it suffices to choose

$$\begin{aligned}
\phi_M &\equiv \phi_{\text{guess}}.\phi_{\text{rewind}}.\phi_{\text{check}}, \text{ where} \\
\phi_{\text{rewind}} &\equiv ([x_2, x_3]_r x_2 \neq \bot \vee x_3 \neq \bot)^*.[x_2, x_3]_r x_2 = x_3 = \bot, \text{ and} \\
\phi_{\text{check}} &\equiv (\psi(\Gamma \cup \Delta \cup \{\sharp\}, x_2, x_3)^*.\delta_M)^*. \\
&\quad \psi(\Gamma \cup \Delta \cup \{\sharp\}, x_2, x_3)^*.\psi(\{\bot\}, x_2, x_3).
\end{aligned}$$

This local constraint δ_M expresses the transition table T of M as follows. Consider a transition $t \in T$ from state $p \in S$ on character $c \in \Gamma \cup \Delta$ that writes the character $d \in \Gamma \cup \Delta$ over c and enters state $q \in S$.

- If t moves the head one step to the right, and $c \neq \sqcup$, then the local constraint is that x_2 must contain pc, while x_3 must contain dq. This can be expressed as $\delta_t \equiv \delta(pc, dq)$, where

$$\begin{aligned}
\delta(u_1 \ldots u_a, v_1 \ldots v_b) \equiv\ & [x_2]_l x_2 = u_1. \cdots .[x_2]_l x_2 = u_a. \\
& [x_3]_l x_3 = v_1. \cdots .[x_3]_l x_3 = v_b.
\end{aligned}$$

- If t moves the head right, and $c = \sqcup$, then $\delta_t \equiv \delta(p\sqcup, dq) + \delta(p\sharp, dq\sharp)$. (That is, the rightmost $\sharp$ in a configuration D is implicitly expanded into $\sqcup\sharp$ to account for visiting previously unvisited tape squares.)

- If t moves the head left, then $\delta_t \equiv \delta(\sharp pc, \sharp q \sqcup d) + \mathop{+}_{e \in \Gamma \cup \Delta} \delta(epc, qed)$.

Choosing $\delta_M \equiv \mathop{+}_{t \in T} \delta_t$ completes the construction.

The extended alphabet Σ' must naturally be encoded into the original fixed alphabet Σ, and in a manner easily decodable within string formulae.

We select the following straightforward block encoding f: each $c \in \Sigma'$ is encoded as some distinct string $f(c) \in \Sigma^e$, where $e = \left\lceil \log_{|\Sigma|} |\Sigma'| \right\rceil$. This encoding is particularly easy to decode, if the atomic string formula $\tau\phi$ in question is of a form, where the window formula ϕ is a conjunction of atomic window formulae, and the transpose τ contains all the variables mentioned in ϕ. For example, $[x, y]_l x = y = c$ is of this form, and its decoding version is $[x, y]_l x = y = (f(c)[1]). \cdots .[x, y]_l x = y = (f(c)[e])$. Hence, all the formulae involving characters from Σ' that were generated by the construction above were of this easily decodable form. The corresponding decoding versions were not presented in the interest of clarity.

The unidirectional variable x_1 can also be eliminated altogether from ϕ_M by considering instead the undecidable problem of determining whether a given Turing machine M halts on the empty input: eliminate all subformulae mentioning x_1 from ϕ_{guess} to reach $\beta_0 = \varepsilon$, and call the resulting string formula ϕ_M^ε. Then M halts on an empty input if and only if ϕ_M^ε satisfies $\Gamma_1' = \emptyset \rightsquigarrow \{x_2, x_3\}$.

Consider then the converse problem of determining if $\Gamma_2 = \{x_2, x_3\} \rightsquigarrow \{x_1\}$ holds for a given ϕ. The halting problem on empty input mentioned above can be reduced to the complement of this problem as follows. (Recall that the complement of a decidable problem is also decidable [49, Theorem 8.1].) Note first that if we assume without loss of generality that M halts by entering a designated state f, then the construction above can produce a formula ϕ_M, which is satisfied only on encodings of halting computations: replace S in subformula $\gamma(x_3, x_3)$ of ϕ_{guess} with $\{f\}$. Build then the string formula $\phi_M' \equiv \phi_M^\varepsilon.[\,]_l x_1 = \bot$. Now $[\![\phi_M']\!]$ is empty when M does not halt on the empty input, and otherwise it consists of the tuples $\langle u, v, w\rangle$, where v and w encode the halting computation and $u \in \Sigma^*$. In other words, M halts on the empty input exactly when ϕ_M' does not satisfy Γ_2.

The reduction for Γ_2 above works also for $\Gamma_2' = \{x_3\} \rightsquigarrow \{x_1, x_2\}$, because there is at most one acceptable value for the variable pair x_2 and x_3, or the computation encoding strings v and w. In this case, dropping the requirement that x_2 must end after configuration D_{n-1} obviates the need for x_1, which can then be eliminated as above.

The remaining finiteness dependencies $\{x_2\} \rightsquigarrow \{x_1, x_3\}$ and $\{x_2\} \rightsquigarrow \{x_3\}$ can be handled by switching the roles of x_2 and x_3 in the reductions for Γ_2 and Γ_2'. $\square$

4.2 Right-Restricted Limitation Is Decidable

Let us then prove the decidability result stated above. Our key technical tool is as follows. Given a right-restricted m-FSA $\mathcal{A}$, we can build a finite structure that still makes explicit enough of its behaviour to admit safety analysis. The first step of the construction is to encode the computation steps of $\mathcal{A}$ in a different order, namely by listing them ordered by the bidirectional tape consumption.

Definition 4.1 *Let $\mathcal{A}$ be a right-restricted m-FSA, whose transitions are $T = \{t_1, \ldots, t_n\}$, and let $\mathcal{C}$ be a computation of $\mathcal{A}$. The* transition word $\mathrm{tw}(\mathcal{C}) \in (T^+\sharp)^+$ *(where $\sharp$ is a new separator) corresponding to $\mathcal{C}$ is the word*

$$t_0^1 t_0^2 t_0^3 \ldots \sharp \underbrace{t_1^1 t_1^2 t_1^3 \ldots \sharp}_{block\ 1} t_2^1 t_2^2 t_2^3 \ldots \sharp \ldots$$

satisfying the following property for all i and j: $t_i^j \in T$ is the transition applied during $\mathcal{C}$ at the moment when the bidirectional cell i is being scanned for the jth time.

In other words, $\mathrm{tw}(\mathcal{C})$ lists the actions performed by $\mathcal{C}$ with the bidirectional head first on '¢', then on the first input character, then on the second, and so forth. That is, each block of $\mathrm{tw}(\mathcal{C})$ corresponds to one *crossing sequence* [43, Figs. 1–3][49, Section 2.6][74, Section 2] during the computation C.

Example 4.1 Consider the following right-restricted 2-FSA $\mathcal{A}_{\text{reverse}}$, for which $L(\mathcal{A}_{\text{reverse}}) = \left\{\left\langle w, w^R \right\rangle : w \in \Sigma^*\right\}$. Its transitions are as follows, with start state I and final state IV.

$$t_1 = \mathrm{I} \xrightarrow[1,0]{¢,¢} \mathrm{II} \qquad t_3 = \mathrm{II} \xrightarrow[-1,1]{\$,¢} \mathrm{III} \qquad t_5 = \mathrm{III} \xrightarrow[0,0]{¢,\$} \mathrm{IV}$$

$$t_2^x = \mathrm{II} \xrightarrow[1,0]{x,¢} \mathrm{II} \qquad t_4^x = \mathrm{III} \xrightarrow[-1,1]{x,x} \mathrm{III} \qquad \text{for each } x \in \Sigma$$

Then the transition words $\mathrm{tw}(\mathcal{C})$ corresponding to the accepting computations $\mathcal{C}$ of $\mathcal{A}_{\text{reverse}}$ are given by the regular expression

$$t_1 t_5 \sharp \left(\underset{x \in \Sigma}{+} t_2^x t_4^x \sharp \right)^* t_3 \sharp. \tag{4.1}$$

For each string $\alpha = a_1 \ldots a_m \in \Sigma^*$, Eq. (4.1) associates a transition word

$$A = t_1 t_5 \sharp t_2^{a_1} t_4^{a_1} \sharp \ldots t_2^{a_m} t_4^{a_m} \sharp t_3 \sharp,$$

which indeed encodes the actions of $\mathcal{A}_{\text{reverse}}$, when its tape 1 contains α as input. The contents of tape 2 associated with A are in turn α^R. ■

The second step of the construction is to identify repetitive parts of the computation we can do without.

Definition 4.2 *Let $\mathcal{A}$ be a right-restricted m-FSA with tape 1 bidirectional, and let $\mathcal{C}$ be a computation of $\mathcal{A}$ on input $\langle w_1, \ldots, w_m \rangle$.*

1. *$\mathcal{C}$ is k-*mobile* if and only if $\mathcal{C}$ contains k distinct pairs of consecutive configurations $\langle p_i, n-d, \ldots \rangle, \langle q, n, n_i^2, \ldots, n_i^m \rangle$, where $1 \le i \le k$, $d = \pm 1$, and $w_j[n_i^j] = w_j[n_1^j]$ for all $2 \le i \le k$ and $2 \le j \le m$.*

   In other words, $\mathcal{C}$ crosses at least k times the boundary from square $n-d$ to square n on tape 1 while satisfying the following constraint: the same state q is entered on each crossing, and for each unidirectional tape head $j = 2, \ldots, m$, the character (but not necessarily the square) head j lands onto on each crossing remains the same.

2. *$\mathcal{C}$ is k-*stable* if and only if $\mathcal{C}$ contains a contiguous subsequence of configurations $\langle p_i, n, n_i^2, \ldots, n_i^m \rangle$, where the state q appears $k+1$ distinct times as p_i, and $w_j[n_i^j] = w_j[n_1^j]$ for all $i > 1$ and $2 \le j \le m$.*

   In other words, $\mathcal{C}$ stays on square n of tape 1 long enough for a configuration with an identical state q and identical characters under the unidirectional tapes $2, \ldots, m$ to repeat at least k times, without leaving square n in the meantime.

*$\mathcal{C}$ is (a, b)-*direct* if and only if it is neither $(a+2)$-mobile nor $(b+2)$-stable. In other words, $\mathcal{C}$ is at most $(a+1)$-mobile or $(b+1)$-stable.*

The repetitive behaviour captured in Definition 4.2 leads immediately to the concept of *cutting off some repetition* from a computation. This in turn will be useful later in demonstrating that a certain behaviour manifests itself already in "short" computations.

More precisely, let $\mathcal{C} = \mathcal{P} \langle q, n, n_2, \ldots, n_m \rangle \mathcal{Q} \langle q, n, n_2', \ldots, n_m' \rangle \mathcal{R}$ be a computation of a right-restricted m-FSA $\mathcal{A}$ with tape 1 bidirectional on input $\vec{u} = \langle w_1, \ldots, w_m \rangle$, where $w_j[n_j] = w_j[n_j']$ for all $2 \le j \le m$. Then, $\mathcal{C}$ is 1-mobile if both $\mathcal{P}$ and $\mathcal{Q}$ end by moving tape head 1 onto square n from the same neighbouring square $(n-d)$. $\mathcal{C}$ is in turn 1-stable if $\mathcal{Q}$ never moves tape head 1 off square n. Now $\mathcal{C}$ can be cut into $\mathcal{C}' = \mathcal{P} \langle q, n, n_2, \ldots, n_m \rangle \mathcal{R}'$, where each configuration $\langle p, n'', n_2'', \ldots, n_m'' \rangle$ in $\mathcal{R}$ is replaced in $\mathcal{R}'$ with the configuration

$$\langle p, n'', n_2'' - (n_2' + n_2), \ldots, n_m'' - (n_m' + n_m) \rangle .$$

That is, the intermediate part $\mathcal{S} = \mathcal{Q}\langle q, n, n'_2, \ldots, n'_m \rangle$ of $\mathcal{C}$ is cut away to form $\mathcal{C}'$. Now $\mathcal{C}'$ is also a computation on $\mathcal{A}$, now with the input

$$\vec{u}' = \langle w_1, w_2[1 \ldots n_2]w_2[n'_2 + 1 \ldots], \ldots, w_m[1 \ldots n_m]w_m[n'_m + 1 \ldots] \rangle ,$$

which in turn cuts off from $\vec{u}$ the unidirectional input substrings scanned during $\mathcal{S}$, namely $w_j[n_j + 1 \ldots n'_j]$ for $2 \leq j \leq m$.

The third and final part of the construction is to show that once the computations have been rearranged as in Definition 4.1, and cut as in Definition 4.2, what remains can be finitely encoded into a structure suitable for the decision method. This encoding simply performs the well-known *crossing sequence construction* [49, Theorem 2.5 and Exercises 2.18–2.20] for converting two-way finite automata to one-way, except that the result now accepts sequences of matching sequences instead of input strings. That is, we form the finite automata corresponding to regular expressions like Eq. (4.1).

Lemma 4.1 *Let $\mathcal{A}$ be a right-restricted m-FSA, and $a, b \in \mathbb{N}$. Then we can effectively construct a finite automaton* $\mathrm{twa}(\mathcal{A}, a, b)$ *to accept exactly the transition words* $\mathrm{tw}(\mathcal{C})$*, where $\mathcal{C}$ is an (a, b)-direct accepting computation of $\mathcal{A}$.*

Proof: For notational simplicity, let the bidirectional tape of $\mathcal{A}$ be tape 1. Assume also for simplicity that $\mathcal{A}$ satisfies condition 4 of Corollary 3.1, so that we can safely construct those paths through the transition graph of $\mathcal{A}$ that behave consistently on tape 1, and still get corresponding accepting computations. Intuitively, this relieves us from keeping track of the indices x in Eq. (4.1) during the construction.

We construct $\mathrm{twa}(\mathcal{A}, a, b)$ in three stages. The first stage consists of constructing the following directed labelled multigraph G_1 from the transition graph of $\mathcal{A}$.

- The states of $\mathcal{A}$ become the nodes of G_1.

  Add also the new node $\imath$ to G_1 to simplify the handling of boundary cases. Intuitively, state $\imath$ winds tape 1 past '\$' upon acceptance.

- For each transition $t = p \xrightarrow[\pm 1, \ldots]{c, \ldots} q$ moving tape 1, add to G_1 the edge $p \xrightarrow[c]{\pm t} q$.

  For the new node $\imath$ add the edges $\imath \xrightarrow[c]{+\varepsilon} \imath$ and $f \xrightarrow[c]{+\varepsilon} \imath$ for all $c \in \Sigma \cup \{¢, \$\}$, where f is the final state of $\mathcal{A}$.

- Close G_1 with respect to the following edge-addition rule. Let $\mathcal{A}$ contain a transition $t = p \xrightarrow[0,\ldots]{c,\ldots} q$ not moving tape 1, and let G_1 contain an edge $q \xrightarrow[c]{\pm\alpha} r$, where the word $\alpha \in T^*$ contains at most b transitions, which leave from state p. Then add into G_1 the edge $p \xrightarrow[c]{\pm t\alpha} r$.

Then G_1 will contain an edge $q \xrightarrow[c]{\pm\alpha} r$ if and only if $\mathcal{A}$ admits the sequence α of transitions specifying character c on tape 1, whose last transition either moves tape 1 to direction ± 1 or leads to f, and the other transitions of α keep tape 1 still. Moreover, α was chosen not to violate $(b+1)$-stability; that is, if a computation $\mathcal{C}$ of $\mathcal{A}$ performs the contiguous sequence α of transitions, then $\mathcal{C}$ does not violate $(b+1)$-stablility during α.

The second stage performs the well-known crossing sequence construction on G_1. More precisely, the following directed labelled multigraph G_2 is constructed from G_1.

- The nodes of G_2 are sequences of $\pm p$, where p is a node of G_1, the signs alternate (that is, $\pm$ is followed by $\mp$), and no pair appears more than $a + 1$ times.
- Initially, the only edge in G_2 is $\varepsilon \xrightarrow[c]{\natural} \varepsilon$.
- Close G_2 with respect to the following edge-addition rules.

  Let G_1 contain an edge $p \xrightarrow[c]{+\alpha} q$ and G_2 an edge $R \xrightarrow[c]{\beta} S$, where neither R nor S begins with with the sign '+'. Add to G_2 the edges $+pR \xrightarrow[c]{\alpha\beta} +qS$ and $R \xrightarrow[c]{\alpha\beta} -p+qS$. (If some endpoint, say $+pR$, falls outside the nodes of G_2, then omit adding that edge.)

  Symmetrically, let G_1 contain an edge $p \xrightarrow[c]{-\alpha} q$ and G_2 an edge $R \xrightarrow[c]{\beta} S$, where neither R nor S begins with the sign '$-$'. Add to G_2 the edges $-qR \xrightarrow[c]{\alpha\beta} -pS$ and $+p-qR \xrightarrow[c]{\alpha\beta} S$.

Then G_2 will contain an edge $+r_1-r_2+r_3\ldots+r_i \xrightarrow[c]{\beta\natural} +s_1-s_2+s_3\ldots+s_j$ if and only if the following four conditions hold.

- All transitions of $\mathcal{A}$ in β are on some square n of tape 1 containing the character c.
- At the ith time square n is entered from square $(n-1)$, the state of $\mathcal{A}$ entered is r_{2i-1}.

At the ith time square n is entered from square $(n+1)$, the state of $\mathcal{A}$ entered is s_{2i}.

At the ith time square $(n-1)$ is entered from square n, the state of $\mathcal{A}$ entered is r_{2i}.

At the ith time square $(n+1)$ is entered from square n, the state of $\mathcal{A}$ entered is s_{2i-1}.

- The transitions β are consistent with the entry conditions above.
- The transitions β, including the entry coditions, are at most $(a+1)$-mobile and $(b+1)$-stable.

In the third and final phase, the graph G_2 is interpreted as the desired automaton twa$\,(\mathcal{A}, a, b)$ as follows.

- The initial state will be the node $+s$, where s is the initial state of $\mathcal{A}$.
- The final state will be the node $+\imath$.
- Each edge $+pR \xrightarrow[c]{t_1 t_2 t_3 \ldots t_m \sharp} +qS$ induces the transition of control from state $+pR$ to state $+qS$ by reading the block $t_1 t_2 t_3 \ldots t_m \sharp$ via the path

  $$+pR \xrightarrow{t_1} g_1 \xrightarrow{t_2} g_2 \xrightarrow{t_3} \cdots \xrightarrow{t_{m-1}} g_{m-1} \xrightarrow{t_m} g_m \xrightarrow{\sharp} +qS,$$

  where the g_i are new nonfinal states.

  Omit, however, from consideration all the edges $+\imath \xrightarrow[c]{\sharp} +\imath$.

Then twa$\,(\mathcal{A}, a, b)$ accepts a word $\beta_0\sharp \ldots \beta_n\sharp$, where each $\beta_i \in T^+$, if and only if G_2 contains some path

$$\Xi \equiv +s = R_0 \xrightarrow[c_0 = \not{c}]{\beta_0\sharp} R_1 \xrightarrow[c_1]{\beta_1\sharp} \cdots \xrightarrow[c_{n-1}]{\beta_{n-1}\sharp} R_n \xrightarrow[c_n]{\beta_n\sharp} R_{n+1} = +\imath.$$

Equivalently, we can decode Ξ into the sequence of transitions taken in some (a, b)-direct accepting computation $\mathcal{C}$ with $c_0 \ldots c_n \ldots$ on tape 1, by starting at the first transition of β_0, and respecting the movements of tape 1 represented by the R_i. □

Now we are finally ready to prove the decidability result. The proof amounts to translating effectively the question "is ϕ limited?" into a series of decidable questions about finite automata twa$\,(\mathcal{A}_\phi, \ldots)$, where $\mathcal{A}_\phi$ is an FSA corresponding to ϕ.

Theorem 4.2 *Let* ϕ *be a right-restricted string formula with* $\text{free}(\phi) = \{x_1, \ldots, x_{k+l}\}$*. Then it is decidable to determine whether* ϕ *satisfies the limitation property* $\{x_1, \ldots, x_k\} \rightsquigarrow \{x_{k+1}, \ldots, x_{k+l}\}$*.*

Moreover, the corresponding limitation function is at most quadratic.

Proof: By Theorem 3.1, we may solve the problem for the right-restricted $(k+l)$-FSA $\mathcal{A}$ corresponding to ϕ. In addition, we assume $\mathcal{A}$ to satisfy Corollary 3.1 as well.

The proof falls into two *cases (1)* and *(2)*, depending on whether the bidirectional tape of $\mathcal{A}$ is on the right or the left side in the limitation property $\Gamma = \{1, \ldots, k\} \rightsquigarrow \{k+1, \ldots, k+l\}$ under study. (If $\mathcal{A}$ is unidirectional, then choose arbitrarily tape 1 as bidirectional, execpt that it is never rewound into direction -1.) The general line of reasoning is similar in both cases, and runs as follows.

The first *subcase (a)* determines whether $\mathcal{A}$ admits an accepting computation $\mathcal{C}$ on some input $\langle v_1, \ldots, v_k, w_1, \ldots, w_k \rangle$, where some tape $k+i$, $1 \leq i \leq l$, does not visit its rightmost endmarker '\$' at all. If such a $\mathcal{C}$ does exist, then Γ fails, because $\mathcal{C}$ is an accepting computation for all inputs of the form $\langle v_1, \ldots, v_k, w_1, \ldots, w_{i-1}, w_i\Sigma^*, w_{i+1}, \ldots, w_k \rangle$. If such a $\mathcal{C}$ does exist, there exists even an $(0,0)$-direct $\mathcal{C}$ of the same kind by a repeated application of the cutting operations presented after Definition 4.2, because these operations cannot introduce additional visits on tape squares. Hence, it suffices to determine whether $L(\text{twa}(\mathcal{B}, 0, 0)) \neq \emptyset$, where $\mathcal{B}$ is a variant of $\mathcal{A}$ that is suitably modified not to visit '\$' on tape $k+i$. Therefore subcase (a) is decidable [49, Theorem 3.7 (1)].

Otherwise, the second *subcase (b)* determines whether there exists some $\vec{v} = \langle v_1, \ldots, v_k \rangle$ with the following property: there exist infinitely many $\vec{w} = \langle w_1, \ldots, w_l \rangle$, for which the input $\vec{v}\vec{w} = \langle v_1, \ldots, v_k, w_1, \ldots, w_k \rangle$ is accepted by $\mathcal{A}$. Denote the corresponding accepting computations as $\mathcal{C}_{\vec{v}\vec{w}}$. (Note that Γ holds if and only if there are no such $\vec{v}$.) Because subcase (a) was seen not to hold, $\mathcal{C}_{\vec{v}\vec{w}}$ must scan all of $\vec{w}$, so for arbitrarily large $\vec{w}$ the corresponding $\mathcal{C}_{\vec{v}\vec{w}}$ must have arbitrarily long intervals during which $\vec{v}$ is not scanned. Subcase (b) then leads to an inspection of some suitably chosen $\text{twa}(\mathcal{A}, a, b)$, once we show that this pumpable behaviour of $\mathcal{A}$ manifests itself already when $\mathcal{C}_{\vec{v}\vec{w}}$ is (a,b)-direct.

Here then are the different details of each case:

Case (1): the bidirectional tape is $(k+1)$. This case reduces to case (2) below with $\Gamma' = \{1, \ldots, k+1\} \rightsquigarrow \{k+2, \ldots, k+l\}$, if we first determine whether tape $(k+1)$ is limited by tapes $1, \ldots, k$. (Naturally, if $l = 1$, then we are already done.) This reduction is done as follows.

Subcase (1a): Does $\mathcal{A}$ admit an accepting computation which does not visit '\$' on tape $k+1$? Let $\mathcal{B}$ to be $\mathcal{A}$ without any transitions, which specify '\$' on tape $k+1$. Then apply the line of reasoning explained above.

Subcase (1b): Assume that subcase (1a) does not hold. Does there exist some $\vec{v} = \langle v_1, \ldots, v_k \rangle$ having infinitely many w_1, for which $\mathcal{A}$ accepts some $\langle v_1, \ldots, v_k, w_1, \ldots \rangle$? The following claim reduces this problem to one about $\mathrm{twa}(\mathcal{A}, 1, 0)$.

Let us say that a block $t_1 t_2 t_3 \ldots \sharp$ in a transition word is *consuming* if and only if contains a transition t_i of $\mathcal{A}$ that reads some tape $1, \ldots, k$.

> *Claim (i):* there exists such a $\vec{v}$ *if and only if* $\mathrm{twa}(\mathcal{A}, 1, 0)$ contains a loop consisting of nonconsuming blocks only.

The existence of a loop of this kind is in turn decidable as follows. Let $t = p \xrightarrow{c} q$ be a transition in $\mathrm{twa}(\mathcal{A}, 1, 0)$. If c is a transition of $\mathcal{A}$ that reads some tape $1, \ldots, k$, then set weight (t) to be 1, and 0 otherwise. Then compute the all-pairs shortest paths of at least one edge in this weighted transition graph of $\mathrm{twa}(\mathcal{A}, 1, 0)$. A nonconsuming loop exists if and only if there exists a node with a path back to itself of total weight 0. Claim (i) is proved as follows.

Assume the first part. Then there exists a family $\mathfrak{A} = \{\mathcal{C}_n \mathcal{D}_n : n \in \mathbb{N}\}$ satisfying the property that each $\mathcal{C}_n \mathcal{D}_n \in \mathfrak{A}$ is an accepting computation of $\mathcal{A}$ on some $\langle v_1, \ldots, v_k, w_1^n, \ldots \rangle$ having $|w_1^n| \geq n$, in which $\mathcal{C}_n$ is the partial computation up to the first time tape $(k+1)$ visits its rightmost endmarker '\$', and $\mathcal{D}_n$ is the partial computation after that. Let $\mathcal{C}'_n$ ($\mathcal{D}'_n$, respectively) be any $(0,0)$-direct partial computation obtained by deletions from $\mathcal{C}_n$ ($\mathcal{D}_n$, respectively). Then $\mathcal{C}'_n \mathcal{D}'_n$ is a $(1,0)$-direct accepting computation of $\mathcal{A}$ on some input of the form $\langle v_1^n, \ldots, v_k^n, w_1^n, \ldots \rangle$, where each v_i^n is a (not necessarily contiguous) subsequence of v_i, and the '\$' trailing w_1^n is still visited by tape $(k+1)$. Therefore no $\mathrm{tw}(\mathcal{C}'_n \mathcal{D}'_n)$ can contain more than

$$s = \sum_{i=1}^{k} (|v_i| + 1) \tag{4.2}$$

consuming blocks, and since $\mathrm{tw}(\mathcal{C}'_n \mathcal{D}'_n)$ contains at least $n+2$ blocks, the automaton $\mathrm{twa}(\mathcal{A}, 1, 0)$ must contain a loop of nonconsuming blocks for sufficiently large n. Thus the forward direction of claim (i) holds. The converse direction follows from straightforward pumping of this nonconsuming loop.

Claim (i) provides also a partial result towards the second part of the theorem: if tape $(k+1)$ is indeed limited by tapes $1, \ldots, k$, then for all $\langle v_1, \ldots, v_k, w_1, \ldots \rangle \in L(\mathcal{A})$ we must have

$$|w_1| \leq (\text{the number of states in } \mathrm{twa}(\mathcal{A}, 1, 0))(s+1), \tag{4.3}$$

where s is given by Eq. (4.2). That is, $|w_1|$ is linear in $|v_1|, \ldots, |v_k|$. (However, the multiplicative factor can be exponential in $||\phi||$.) The rest of the second part then follows from this partial result and Theorem 3.3 on page 50.

Case (2): the bidirectional tape is 1. Then all the tapes $k+1, \ldots, k+l$ to be limited are unidirectional.

Subcase (2a): Does $\mathcal{A}$ admit an accepting computation $\mathcal{C}$ which does not visit '\$' on some tape $k+1, \ldots, k+l$? Let $\mathcal{B}$ be obtained from $\mathcal{A}$ by removing every transition into the final state that specifies some other character than '\$' on any tape $k+1, \ldots, k+l$, and apply line the of reasoning above.

Subcase (2b): Assume that subcase (2a) does not hold. Does there exist some $\vec{v} = \langle v_1, \ldots, v_k \rangle$, for which $\mathcal{A}$ accepts infinitely many different inputs $\langle v_1, \ldots, v_k, w_1, \ldots, w_k \rangle$? The following claim reduces this problem to one about twa $(\mathcal{A}, 1, 1)$.

Let us say that a computation $\mathcal{C}$ of $\mathcal{A}$ on input $\langle v_1, \ldots, v_k, w_1, \ldots, w_k \rangle$ is *jamming* if and only if it is of the form

$$\cdots \underbrace{\langle p, n_1, \ldots, n_k, m_1, \ldots, m_l \rangle \cdots \langle p, n_1, \ldots, n_k, m'_1, \ldots, m'_l \rangle}_{\text{loop } L} \cdots$$

where $w_i[m_i] = w_i[m'_i]$ for all $1 \leq i \leq l$, and $m_j < m'_j$ for some $1 \leq j \leq l$. That is, $\mathcal{C}$ contains a pumpable loop L on which the unidirectional tapes $2, \ldots, k$ do not move, the bidirectional tape 1 may move but returns to square n_1, and some tape j is advanced. Call square n_1 *the spine* of L, and any transition that advances some tape j within L *a rib* of L. The claim is as follows.

> *Claim (ii):* there exists a $\vec{v}$ of this kind *if and only if* $\mathcal{A}$ has a jamming $(1,1)$-direct accepting computation.

Then it only remains to show that it is decidable whether $L(\text{twa}(\mathcal{A}, 1, 1))$ contains a transition word twa $(\mathcal{C})$ of a jamming computation $\mathcal{C}$. Claim (ii) is proved as follows.

Assume the first part. Let $\mathcal{C}$ be an accepting computation of $\mathcal{A}$ on input $\vec{u} = \langle v_1, \ldots, v_k, w_1, \ldots, w_k \rangle$ satisfying

$$\max\{|w_j| : 1 \leq j \leq l\} > e(|v_1| + 2)\left(1 + \sum_{i=2}^{k}(|v_i| + 1)\right), \text{ where}$$
$$e = (\text{number of states in } \mathcal{A})(|\Sigma| + 2)^l.$$

Because $\mathcal{C}$ must reach '\$' on every, and hence especially the longest, tape $k+1,\ldots,k+l$, $\mathcal{C}$ is jamming. Partition $\mathcal{C}$ into

$$\mathcal{P}\underbrace{\langle p,n_1,\ldots,n_k,m_1,\ldots,m_l\rangle}_{p}\mathcal{Q}\underbrace{\langle p,n_1,\ldots,n_k,m'_1,\ldots,m'_l\rangle}_{q}\mathcal{R},$$

where $p\mathcal{Q}q$ is a minimal partial computation witnessing the jamming. Perform then suitable cutting operations to $\mathcal{C}$ to yield $\mathcal{C}'=\mathcal{P}'\mathcal{Q}'\mathcal{R}'$, where $\mathcal{P}'$, $\mathcal{Q}'$, and $\mathcal{R}'$ are $(0,0)$-direct partial computations obtained from $\mathcal{P}p$, $\mathcal{Q}$, and $q\mathcal{R}$, respectively. By the minimality of $\mathcal{Q}$, $\mathcal{Q}'$ retains the ribs of the jamming loop. Then $\mathcal{C}'$ is an accepting $(1,1)$-direct accepting jamming computation, proving the forward direction of claim (ii). The converse direction follows from straightforward pumping of the jamming loop.

Let us now show that it suffices to restrict attention to a finite subset of $L(\mathrm{twa}\,(\mathcal{A},1,1))$ in our search for a jamming computation $\mathcal{C}$. Consider a $W\in L(\mathrm{twa}\,(\mathcal{A},1,1))$ satisfying

$$|W|\geq 5(\text{number of states in }\ \mathrm{twa}\,(\mathcal{A},1,1)). \tag{4.4}$$

Then some state a of $\mathrm{twa}\,(\mathcal{A},1,1)$ must repeat at least five times when accepting W. Hence, W can be partitioned into $\alpha\beta_1\beta_2\beta_3\beta_4\gamma$ so that each $W_i=\alpha\ldots\beta_{i-1}\beta_{i+1}\ldots\gamma$ omitting the nonempty subword β_i is still accepted by $\mathrm{twa}\,(\mathcal{A},1,1)$, now with fewer repeats of state a than before. If in addition W was the transition word of a jamming computation, then so is at least one of the W_i: the spine of the jamming loop can span at most two consecutive βs, because the block length is finite, and at most one additional β is required to incorporate also some rib(s) of the spine. Hence, some redundant β always remains. It suffices therefore to restrict attention to the W not satisfying Eq. (4.4), and this is indeed a finite search space, yielding decidability. $\square$

Example 4.2 Let us continue Example 4.1 by considering the limitation properties $\Gamma_1=\{1\}\rightsquigarrow\{2\}$ and $\Gamma_2=\{2\}\rightsquigarrow\{1\}$ satisfied by $\mathcal{A}_{\text{reverse}}$. Note first that because $\mathcal{A}_{\text{reverse}}$ moves its bidirectional tape head 1 in an orderly manner – one sweep to its rightmost endmarker \$, and another sweep back to its leftmost endmarker ¢ – all its computations encoded in Eq. (4.1) are (0,0)-direct. Hence, it suffices to study Eq. (4.1).

The finite automaton $\mathcal{A}'=\mathrm{twa}\,(\mathcal{A}_{\text{reverse}},0,0)$ accepting the language of Eq. (4.1) consists of the following components:

1. the beginning phase $\mathrm{i}\xrightarrow{t_1}\mathrm{ii}\xrightarrow{t_5}\mathrm{iii}\xrightarrow{\sharp}\mathrm{iv}$ from the start state i;
2. the loops $\mathrm{iv}\xrightarrow{t_2^x}\mathrm{v}_x\xrightarrow{t_4^x}\mathrm{vi}_x\xrightarrow{\sharp}\mathrm{iv}$ for all $x\in\Sigma$; and

3. the ending phase iv $\xrightarrow{t_3}$ vii $\xrightarrow{\sharp}$ viii into the final state viii.

Inferring Γ_1 represents case (2) in the proof of Theorem 4.2. Subcase (2a) boils down to checking whether $\mathcal{A}_{\text{reverse}}$ can accept without tape 2 reaching \$. This is impossible, because $\mathcal{A}'$ requires t_5 to always appear. Subcase (2b) boils in turn down to checking that the finite sublanguage

$$t_1 t_5 \sharp \left(\mathop{+}_{x\in\Sigma} t_2^x t_4^x \sharp \right)^{\leq 40} t_3 \sharp$$

of Eq. (4.1) encodes no jamming computation.

Conversely, inferring Γ_2 represents case (1). It is reduced to the trivial limitation problem $\{1,2\} \rightsquigarrow \emptyset$ of case (2) as follows. Subcase (1a) boils down to checking whether $\mathcal{A}_{\text{reverse}}$ can accept without tape 1 visiting \$. This is impossible, because $\mathcal{A}'$ requires t_3 to always appear. In turn, subcase (1b) boils down to checking that the loops in $\mathcal{A}'$ are consuming due to the presence of transitions t_4^x, which consume input from tape 2. ∎

4.3 An Efficient Heuristic for Safety Analysis

Although the decidability result in Section 4.2 does indeed provide us with a syntactic method for determining whether a given right-restricted string formula satisfies a given limitation property, the algorithmic construction behind the proof of Theorem 4.2 is unwieldy in practice. Therefore we seek here an alternative simpler construction.

This construction is based on the idea that an $(k+l)$-FSA $\mathcal{A}$ must surely satisfy the limitation dependency $\{1,\ldots,k\} \rightsquigarrow \{k+1,\ldots,k+l\}$, if "writing" characters onto the output tapes $k+1,\ldots,k+l$ is never free, but instead must always be paid for by "reading" characters from the input tapes $1,\ldots,k$. We define this idea formally as follows.

Definition 4.3 *Let $\mathcal{A}$ be a k-FSA. A path*

$$P = p_0 \xrightarrow[d_1^1,\ldots,d_k^1]{c_1^1,\ldots,c_k^1} p_1 \xrightarrow[d_1^2,\ldots,d_k^2]{c_1^2,\ldots,c_k^2} p_2 \cdots p_{n-1} \xrightarrow[d_1^n,\ldots,d_k^n]{c_1^n,\ldots,c_k^n} p_n$$

in the transition graph of $\mathcal{A}$ is strictly advancing (respetively, strictly retreating) on tape i *if and only if some edge in P moves tape i to the forward direction $+1$, (respectively, to direction -1) while no edge of P moves it to the opposite backward direction -1 (respectively, to direction $+1$).*

This path P is terminal on tape i *if and only if $c_i^h = \$$ for some $1 \leq h \leq n$.*

$\mathcal{A}$ is clocked by tapes $X \subseteq \{1, \ldots, k\}$ *if and only if the following two conditions hold:*

1. *Let P be a loop (that is, $p_0 = p_n$), where $d_j^h = +1$ for some tape $j \notin X$ and $1 \leq h \leq n$. Then P is strictly advancing or retreating on some tape $i_P \in X$.*

2. *Every path through the transition graph of $\mathcal{A}$ (that is, path P with p_0 the initial and p_n a final state of $\mathcal{A}$) is terminal on all tapes $\{1, \ldots, k\} \setminus X$.*

The construction in this section will make the simplifying, but not always valid, assumption that whenever a given FSA $\mathcal{A}$ is not clocked by the given subset X of its tapes, then there is also an accepting computation of $\mathcal{A}$, which traverses the path of transitions that violated clockedness. In contrast, the decision method developed in Section 4.2 applied the crossing sequence construction to further verify that suspicious-looking paths do indeed admit also an accepting computation that traverses them. In the same vein, the undecidability result of Section 4.1 can be interpreted as showing this verification to be impossible in general.

Algorithm 1 provides a computational counterpart to Definition 4.3, as shown by Theorem 4.3 below. (Algorithms will be presented using the Guarded Command Language introduced by Dijkstra [44, Part II].)

Subroutines Cond1 and Cond2 correspond to Conditions 1 and 2 of Definition 4.3, respectively. In particular, the idea behind subroutine Cond1 is to repeatedly delete transitions that cannot be a part of any loop that would violate Condition 1. This is in turn accomplished as follows.

1. First the transition graph G of the k-FSA $\mathcal{A}$ being studied is divided into maximal strongly connected components C_i. The transitions between components can now be eliminated (step (†) of the algorithm), because they are not a part of any remaining loop at all.

2. Then each of these components C_i is further broken apart, if a suitable strictly advancing or retreating tape can be found to remove some of the loops in C_i from further consideration (step (‡) of the algorithm).

3. Finally recurse to examine these broken components further.

Strongly connected component analysis has been used in similar problems about logic programs [56, 96]. Here, however, the interactions between different tapes are easier to determine than the interactions between variables in a general logic program. In fact, condition 4 of Corollary 3.1 can be seen as making these interactions explicit.

Algorithm 1 A heuristic safety algorithm.

```
proc Algorithm1( value   𝒜 : a k-FSA to test;
                 value   X : a subset of its tapes;
                 result  b : is 𝒜 clocked by X?);
     call Cond1(the transition graph of 𝒜, X, b₁);
     call Cond2(the transition graph of 𝒜, X, b₂);
     b := b₁ ∧ b₂
end  Algorithm1;

proc Cond1( value   G : a transition subgraph of a k-FSA 𝒜 to test;
            value   X : a subset of its tapes;
            result  b : does Condition (1) of Definition 4.3 hold?);
     C₁,...,Cₘ := the maximal strongly connected components of G;
     Delete from G all transitions between distinct components (†);
     b, i := true, 1;
     do  b ∧ (i ≤ m) ⟶
              if  there is some j ∈ X, for which some transition of Cᵢ
                  moves j into direction ±1, but none into the
                  opposite direction ∓1 ⟶
                       Delete from Cᵢ all transitions that moved j (‡);
                       call Cond1(Cᵢ, X, b)
               ▯  there is no such j (¶) ⟶
                       b := Cᵢ does not move any tape in
                       {1,...,k} \ X into direction +1
              fi;
              i := i + 1
     od
end  Cond1;

proc Cond2( value   𝒜 : a k-FSA to test;
            value   X : a subset of its tapes;
            result  b : does Condition (2) of Definition 4.3 hold?);
     b, Y := true, {1,...,k} \ X;
     do  b ∧ (Y ≠ ∅) ⟶
              i := pick and delete a random element from Y;
              ℬ := 𝒜 except the transitions specifying '$' for tape i;
              b := no final state of ℬ is reachable from its start state
     od
end  Cond2;
```

Theorem 4.3 *The k-FSA $\mathcal{A}$ is clocked by X if and only if calling Algorithm 1 on $\mathcal{A}$ and X yields b to be true.*

Proof: Let us first assume that Condition 1 of Definition 4.3 fails. Then the transition graph $\mathcal{G}$ of $\mathcal{A}$ contains some loop P that falsifies it. This P persists untouched through subroutine Cond1 of Algorithm 1. Consider namely which of the two steps (†) and (‡) would be the first to delete a transition belonging to P. It cannot be step (†), because P is a strongly connected component of $\mathcal{G}$ in the beginning. On the other hand, let C_i be any strongly connected component of $\mathcal{G}$ containing all the transitions in P, and let tape $j \in X$ enable the application of step (‡) to C_i. Then P must keep tape j stationary by its definition, and hence step (‡) can only delete from C_i transitions outside P. Therefore P persists untouched, and consequently subroutine Cond1 eventually returns `false`.

Assume for the other direction that subroutine Cond1 returns `false`. Then $\mathcal{G}$ contains some strongly connected component C_i, in which some tape in $\{1, \ldots, k\} \setminus X$ moves into direction $+1$, while all the tapes in X either remain stationary or move into both directions ± 1. This C_i falsifies Condition 1.

We have thus shown that subroutine Cond1 enforces exactly Condition 1. It can be seen directly from Algorithm 1 that the other subroutine Cond2 does the same for the other Condition 2. □

Let us now show that Definition 4.3, and hence also Algorithm 1, indeed accepts only those limitation conditions that actually do hold. At the same time we can show that the corresponding limitation functions from Definition 3.3 remain polynomial. (Theorem 4.1 demonstrated a string formula ϕ, where this limitation function can in general be superpolynomial.) Indeed, the functions inferred via Definition 4.3 are the same as in Theorem 3.3, because Definition 4.3 ignored the effects of rewinding a bidirectional tape to be limited.

Lemma 4.2 *Let $\mathcal{A}$ be a $(p+q+r)$-FSA satisfying the following assumptions:*

- *Tapes $1, \ldots, p$ of $\mathcal{A}$ are bidirectional.*
- *Tapes $p+1, \ldots, p+q$ of $\mathcal{A}$ are unidirectional.*
- *$\mathcal{A}$ is clocked by tapes $1, \ldots, p+q$.*
- *$\mathcal{A}$ satisfies Corollary 3.1.*

Let in addition $\mathcal{C}$ be any computation of $\mathcal{A}$ on some input

$$\langle u_1, \ldots, u_p, v_1, \ldots, v_q, w_1, \ldots, w_r \rangle .$$

Then $\mathcal{C}$ does not move any of the tapes $p+q+1,\ldots,p+q+r$ to direction $+1$ more than $f(|u_1|,\ldots,|u_p|,|v_1|,\ldots,|v_q|)$ times, where the function $f\colon \mathbb{N}^{p+q} \to \mathbb{N}$ is defined as follows:

$$f(m_1,\ldots,m_p,n_1,\ldots,n_q) = \\ (a-2)\left(\prod_{i=1}^{p}(m_i+2)\right)\left(\max(q,1)+\sum_{j=1}^{q} n_j\right),$$

where a is the number of states in $\mathcal{A}$.

Proof: We proceed as in the proof of Theorem 3.3, but derive our contradiction differently.

Assume that some tape $p+q+k$ is moved to direction $+1$ more than $b = f(|u_1|,\ldots,|u_p|,|v_1|,\ldots,|v_q|)$ times during $\mathcal{C}$. Post a fence $C_m|C_{m+1}$ between two adjacent configurations in $\mathcal{C}$ if and only if the chosen guilty output tape $p+q+k$ moves to direction $+1$ in the computation step (transition taken) from C_m to C_{m+1}. By assumptions, at least b of these fences are posted. Consider on the other hand two configurations C_m and C_n in $\mathcal{C}$ to have the same color if and only if they share the same state of $\mathcal{A}$ and the same head positions for the input tapes $1,\ldots,p+q$. This cannot use more than at most b different colors. Therefore $\mathcal{C}$ must contain two configurations C_m and C_n, which have the same color but are separated by an intervening fence.

Consider the path P of transitions traversed by the subcomputation $C_m \ldots C_n$ of $\mathcal{C}$. P contains a transition that moves the guilty output tape $p+q+k$ to direction $+1$, namely the one causing the fence that separated C_m and C_n to be posted. P is also a loop, because C_m and C_n share the same state. Let then $1 \leq h \leq p+q$ be any input tape. Because C_m and C_n shared the same color, the head position on tape h is the same just before and just after traversing P, and therefore P can be neither strictly advancing nor strictly retreating on tape h. Hence, P violates Condition 1 of Definition 4.3, leading to the desired contradiction. $\square$

Theorem 4.4 *Let $\mathcal{A}$ be a $(p+q+r)$-FSA satisfying the following assumptions:*

- *Tapes $1,\ldots,p$ of $\mathcal{A}$ are bidirectional.*
- *Tapes $p+1,\ldots,p+q$ of $\mathcal{A}$ are unidirectional.*
- *$\mathcal{A}$ is clocked by tapes $1,\ldots,p+q$.*
- *$\mathcal{A}$ satisfies Corollary 3.1.*

Then $L(\mathcal{A})$ satisfies the finiteness dependency

$$\Gamma = \{1, \ldots, p+q\} \rightsquigarrow \{p+q+1, \ldots, p+q+r\}$$

with the limitation function

$$\mathcal{W}_{\mathcal{A}}^{\Gamma}(m_1, \ldots, m_p, n_1, \ldots, n_q) = f(m_1, \ldots, m_p, n_1, \ldots, n_q) - 1.$$

Proof: Let $Z = \langle u_1, \ldots, u_p, v_1, \ldots, v_q, w_1, \ldots, w_r \rangle \in L(\mathcal{A})$, and let $\mathcal{C}$ be any computation of $\mathcal{A}$ accepting Z. This $\mathcal{C}$ follows some path through the transition graph of $\mathcal{A}$ into a final state, and by Condition 2 of Definition 4.3, \$ is visited on every tape $p+q+k$ during $\mathcal{C}$. Hence, no w_k can be longer than $d_k - 1$ characters, where d_k is the number of times tape $p+q+k$ is advanced to direction $+1$ during $\mathcal{C}$. In turn, $\max(d_1, \ldots, d_r) \leq f(m_1, \ldots, m_p, n_1, \ldots, n_q)$ by Lemma 4.2. □

Let us then turn to assessing the applicability of this heuristic. We can show that the heuristic Algorithm 1 solves the case of unidirectional FSAs completely. In this case, Algorithm 1 reduces to testing the following two simple conditions: (1) that if all transitions moving some tape in X are removed from the unidirectional automaton $\mathcal{A}$, no loops remain; and (2) that all the incoming transitions into the final state f of $\mathcal{A}$ specify '\$' for all the tapes not in X.

Theorem 4.5 *Let $\mathcal{A}$ be a unidirectional $(k+l)$-FSA that satisfies both the neatness conditions given in Corollary 3.1 and the limitation property $\{1, \ldots, k\} \rightsquigarrow \{k+1, \ldots, k+l\}$. Then $\mathcal{A}$ is clocked by $\{1, \ldots, k\}$.*

Proof: Every accepting computation of $\mathcal{A}$ traverses some path P through the transition graph G of $\mathcal{A}$. Conversely, conditions 2 and 4 of Corollary 3.1 and the unidirectionality of $\mathcal{A}$ together imply that for every path P through G there indeed exists an accepting computation traversing exactly P.

Now G cannot contain a loop L failing Condition 1 of Definition 4.3: condition 4 assures that there would then be a path of the form $P_1 L P_2$ through G, and unidirectionality assures that the input tapes X remain stationary in L. Then the reasoning above provides accepting computations for each path $P_1 L^m P_2$, where $m \in \mathbb{N}$, with the same strings on the input tapes X. This contradicts the limitation property, because L does advance some tape.

Similarly, G cannot fail Condition 2, because that would admit an accepting computation, which does not reach \$ on some output tape $\{1, \ldots, k\} \setminus X$. Then the contents of this tape can be suffixed with an arbitrary string, and still get the same accepting computation for this new input. This contradicts the limitation property, proving the claim. □

Our heuristic can even solve successfully some problems with multiple bidirectional tapes, even though the general case was seen to be undecidable by Theorem 4.1. On the other hand, our heuristic cannot handle all the right-restricted cases Theorem 4.2 could, as witnessed by the following example.

Example 4.3 Consider the right-restricted 2-FSA $\mathcal{A}_{\mathrm{reverse}}$ given in Example 4.1, and its limitation properties Γ_1 and Γ_2 established in Example 4.2 via Theorem 4.2.

Γ_1 can be determined to hold also by Algorithm 1 as follows. Eliminating transition t_5 disconnects I and IV, and therefore Cond2 returns $b_2 = \mathtt{true}$. Every state of $\mathcal{A}_{\mathrm{reverse}}$ forms its own strongly connected component. Transitions t_1, t_3, and t_5 are removed by (†). The component consisting of state II and transitions t_2^x can be handled by selecting $j = 1$ in (‡), leading into a transition-free component. Finally, the component of III and t_4^x is handled similarly. Hence, Cond1 also returns $b_1 = \mathtt{true}$.

On the other hand, Γ_2 cannot be so determined. The reason is that tape head 1 moves forward but tape head 2 does not in the component of II and t_2^x. ■

Hence the heuristic constructed here and the uniform decidability method developed in Theorem 4.2 have *incommensurable* discerning power. (Their powers do, however, coincide for the unidirectional string formulae, by Theorem 4.5 above.) In contrast, Theorem 6.5 and Corollary 6.2 will later demonstrate that either will serve equally well as the basis for a syntactically safe sublanguage of Alignment Calculus to be developed in Section 4.4 below. In other words, when the rest of the Alignment Calculus machinery – quantification and logical connectives – are added, this incommensurability disappears. We can therefore prefer this more straightforward Algorithm 1 to Theorem 4.2 with impunity.

4.4 A Safe Sublanguage of Alignment Calculus

In this chapter we propose an Alignment Calculus sublanguage, which is syntactically safe. That is, this sublanguage has the following two characteristics.

First, any query expressed in the sublanguage can be answered by performing some finite computation. Our computational concept is the finitely evaluable expressions from Definition 3.4. Intuitively, these correspond to a straightforward evaluation strategy, where generators lead to automata simulations that are finite by the corresponding limitation conditions. Chapter 5 provides more details of this evaluation process, in particular these finite simulations.

And second, it is computationally testable whether a query written by the user falls within the confines of this class, and is therefore allowed to proceed into the evaluation stage without fear of a nonterminating computation.

Note that our proposal is not specific to Alignment Calculus. Instead, it uses the methods developed in Theorems 4.2 and 4.4 as "black boxes" to provide initial safety information about the string formulae. This proposal can therefore be used for other languages as well by merely supplying different black boxes for their non-relational atoms.

4.4.1 Safe Conjunctions

Let us begin our journey towards a syntactic subset of the domain independent queries by considering first *conjunctions of atomic formulae*. Here we demonstrate how finiteness dependencies, our tool from Definition 3.2, can be applied in this basic case. There are three aspects to consider: (1) the syntactic inference of new finiteness dependencies from known ones; (2) the extraction of implicit finiteness dependencies from a given atomic formula; and (3) the effect of conjunctions on these extracted dependencies.

Finiteness dependencies are one kind of *functional dependencies* [68, Theorem 1][92, Chapter 7.2]. In particular, sound and complete inference about them can be carried out by the following *Armstrong's Axioms*. Let F be a set of functional dependencies, and U the set of variable symbols appearing in F. Then the concept "F implies the functional dependency Γ" is denoted by $F \vdash \Gamma$, and defined inductively as follows.

Basis: If $\Gamma \in F$ holds, then also $F \vdash \Gamma$ holds.

Reflexivity: If $Y \subseteq X \subseteq U$ holds, then $F \vdash X \rightsquigarrow Y$ also holds.

These dependencies are called *trivial*, because they do not depend on F.

Augmentation: If both $Z \subseteq U$ and $F \vdash X \rightsquigarrow Y$ hold, then also $F \vdash (X \cup Z) \rightsquigarrow (Y \cup Z)$ holds.

Transitivity: If both $F \vdash X \rightsquigarrow Y$ and $F \vdash Y \rightsquigarrow Z$ hold, then also $F \vdash X \rightsquigarrow Z$ holds.

Lemma 4.3 gives a form of this inference that suits our intended application. Thus, aspect (1) has been dealt with.

Let us then deal with aspect (2) by defining the set $F = \mathrm{basis}(\phi)$ of "given" finiteness dependencies for an atomic formula ϕ of Alignment Calculus as the starting point of our inference. If ϕ is an atomic relational formula, then this task is simple: $\mathrm{basis}(\phi) = \{X \rightsquigarrow (\mathrm{free}(\phi) \setminus X) : X \subseteq \mathrm{free}(\phi)\}$, because each relation symbol always denotes a finite set by definition, and therefore ϕ satisfies every finiteness dependency that can be formed by using the variables in $\mathrm{free}(\phi)$. (A given set of dependencies is usually represented with a minimal subset, from which the complete set can be inferred by Armstrong's Axioms. However, we choose to represent the sets $\mathrm{basis}(\ldots)$ explicitly to simplify the presentation of Algorithm 2.)

On the other hand, if ϕ is a string formula, then $\mathrm{basis}(\phi)$ should be the set of finiteness dependencies $\Gamma = X \rightsquigarrow (\mathrm{free}(\phi) \setminus X)$, where the variables in $X \subseteq \mathrm{free}(\phi)$ limit the other variables $\mathrm{free}(\phi) \setminus X$ in ϕ, as in Definition 3.3. Sections 4.1–4.3 discussed computing $\mathrm{basis}(\phi)$ in this case, ans we will later restrict attention to the computable subcases identified there. However, the construction in this section merely assumes the sets $\mathrm{basis}(\ldots)$ to be correct in the sense that if $\Gamma \in \mathrm{basis}(\psi)$, then ψ also satisfies the limitation property Γ.

Recall from Definition 3.3 that we required these finiteness dependencies to be full. Consider the following two string formulae to see why.

$$\phi \equiv ([x, y, z]_l x = y = z)^*.([y, z]_l y = z)^*.[x, y]_l x = y = \bot$$
$$\psi \equiv ([x, z]_l x \neq \bot.[z]_l z \neq \bot)^*.[x, z]_l x = z = \bot$$

The formula ϕ states that "x is a prefix of y, which is in turn a prefix of z", while ψ states that "the length of z is twice the length of x". Note that ϕ satisfies the non-full finiteness dependency $\Gamma = \{y\} \rightsquigarrow \{x\}$. Now the query

$$R(y) \wedge \phi \wedge \psi \tag{4.5}$$

is domain independent, because it suffices to consider only strings that are at most twice the length of the longest string in R. However, *evaluating* this query finitely poses a problem: in which order should ϕ and ψ be evaluated? If we try the order in which the query is written, evaluating $R(y) \wedge \phi$ does not yet constrain z; on the other hand, in the reverse order $R(y) \wedge \psi \wedge \phi$ evaluating $R(y) \wedge \psi$ constrains neither x nor z. In either case, we end up with an infinite intermediate result.

This line of reasoning explains why Theorems 3.4 and 3.5 could not be made exactly dual to each other: while finite evaluability implies domain independence, the converse direction fails to hold in general. These problems could perhaps be resolved by *factoring* the bothersome string formulae into pieces: for example, ϕ is indeed naturally factorizable into $\phi'(x, y) \wedge \phi'(y, z)$, where $\phi'(a, b)$ states that "a is a prefix of b". Then Eq. (4.5) could be rewritten into $R(y) \wedge \phi'(x, y) \wedge \psi \wedge \phi'(y, z)$, and evaluated finitely from left to right. While these factoring problems have indeed been studied in automata theory [102], it is still unclear whether this factoring approach can be made to work in all cases, or whether the resulting algorithms are feasible in practice. Hence, we decide to adopt this fullness restriction.

Let us finally deal with aspect (3) by studying what finiteness dependencies a conjunction $\chi \equiv \phi \wedge \psi$ of two atoms ϕ and ψ satisfies. It is straightforward to see that χ satisfies at least the dependencies that can be inferred from $B_\chi = \text{basis}(\phi) \cup \text{basis}(\psi)$ via Armstrong's Axioms. The converse does not hold, however: if ϕ states that "x is of even length", and ψ states that "x is of odd length", then $\text{basis}(\phi) = \text{basis}(\psi) = \emptyset$, but χ satisfies the nontrivial dependency $\emptyset \rightsquigarrow \{x\}$, because there are no suitable values for x at all. Even so, we pose the *conjunctivity* restriction that we restrict our attention to B_χ when reasoning about the finiteness dependencies satisfied by χ; later, Corollary 6.1 justifies our choice by exhibiting an undecidable problem hidden under this seemingly innocuous surface.

Given the fullness and conjunctivity restrictions, and the syntactic inference tool provided by Armstrong's Axioms, we can define our algorithmic criterion for acceptable conjunctions of atomic formulae, as follows. At the same time we reorder the conjuncts with respect to the order, in which their limitation properties "fire" [30, Lemma 7.10][31, Lemma 7.14].

Lemma 4.3 *Let $\phi \equiv a_1 \wedge \ldots \wedge a_p$ be a conjunction of unnegated atomic formulae a_i. Then $\bigcup_{i=1}^{p} \text{basis}(a_i) \vdash X \rightsquigarrow \text{free}(\phi)$ holds if and only if ϕ can be permuted into $\phi' \equiv a'_1 \wedge \ldots \wedge a'_p$ so that $\bigcup_{j=1}^{q} \text{basis}\left(a'_j\right) \vdash X \rightsquigarrow \text{free}\left(\phi'_q\right)$ holds for every prefix $\phi'_q \equiv a'_1 \wedge \ldots \wedge a'_q$ of ϕ'.*

Proof: Consider Algorithm 2, which is a variant of the standard closure algorithm [24, Fig. 9.2][92, Algorithm 7.1] for determining if a functional dependency follows from a set of known dependencies by Armstrong's Axioms. That is, it computes the closure X^+ of X with respect to the dependencies $F = \bigcup_{i=1}^{p} \text{basis}(a_i)$. At the same time, it lists the order in which each a_i is processed.

The fullness restriction assures that on each round of the loop the search for a suitable atom a from the set A can indeed consider all finiteness dependencies in $\bigcup_{a \in A} \text{basis}(a)$, and that once a suitable a is found, moving it from A to the end of ϕ' does not eliminate from consideration any finiteness dependency that might have been useful in expanding the closure Y of X further. (In fact, similar inference on nonfull functional dependencies is known to be difficult [12].) With this in mind, the correctness and completeness of

Algorithm 2 Reordering a conjunction of atoms with respect to a finiteness dependency.

```
proc Algorithm2( value   X, φ : as in Lemma 4.3;
                 result  φ'   : as in Lemma 4.3;
                 result  b    : does φ' contain all of φ?);
     Y, A, φ' := X, {a_1, ..., a_p}, an initially empty conjunction;
     do  there is some a ∈ A with either free(a) ⊆ Y or
         (free(a) ∩ Y ⇝ free(a) \ Y) ∈ basis(a)  ⟶
             Y, A, φ' := (Y ∪ free(a)), (A \ {a}), φ' ∧ a
     od;
     b := (A = ∅)
end  Algorithm2;
```

the variant follow by standard argumentation [92, Theorem 7.2].

The claim about the prefixes ϕ'_q of ϕ' follows in turn from the fact that the chosen a is always inserted at the end of the current ϕ'. $\square$

4.4.2 Quantified Disjunctive Normal Form

In Lemma 4.3 we proposed a syntactic test for detecting when a conjunction of atomic formulae satisfies a finiteness dependency. It remains to extend this test to more complex formulae. Our stategy is to propose a *normal form* for Alignment Calculus formulae, where conjunctions are maximally large. Then Lemma 4.3 can be applied at its fullest to provide us with as much finiteness dependency information as possible.

Conjunctions would be maximized in *Disjunctive Normal Form (DNF)*

$$\overset{\exists}{\forall} x_1 \cdots \overset{\exists}{\forall} x_p . \bigvee_{i=1}^{q} \bigwedge_{j=1}^{r_i} \phi_{ij},$$

where each ϕ_{ij} is a possibly negated atomic formula. However, this choice suffers not only from efficiency problems but also from problems related to finiteness dependency inference. Consider namely the formula $\phi \equiv R(x) \wedge \forall y \psi$. As our aim is to determine whether the query ϕ can be correctly evaluated using only finitely many different values for its variables x and y, we see that the scope ψ of the quantification must satisfy the limitation property that "for every x there exists only finitely many y, for which ψ may *fail*". In other words, its negation $\neg\psi$ must satisfy the finiteness dependency $\Gamma = \{x\} \rightsquigarrow \{y\}$. On the other hand, converting ϕ into DNF leads to

Figure 4.1 The grammar for QDNF

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

$\phi' \equiv \forall y.R(x) \wedge \psi$. However, Γ now fails for the negated scope $(\neg R(x)) \vee (\neg\psi)$ of the quantification. What went wrong here was that $R(x)$ entered the scope of the implicit *negations* $\forall y\psi \equiv \neg\exists y \neg\psi$ during the conversion, and Γ was lost in the process.

We therefore seek an alternative normal form, in which the formulae are as close to DNF as possible, except that the order of negations and quantifications is not altered. Thus we say that an Alignment Calculus formula ϕ is in *Quantified Disjunctive Normal Form (QDNF)* if its structure adheres to the Context-Free Grammar in Fig. 4.1. 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 Q^+C, now with mandatory quantifier prefixes Q^+. Without loss of generality we further assume that a formula in QDNF has no redundant quantifiers (that is, $x \in \text{free}(\phi)$ for every subformula $\exists x\phi$), and that its bound variables are standardized apart (that is, if the subformula $\exists x\phi$ occurs within some larger formula ψ, then $x \notin \text{free}(\psi)$). QDNF is syntactically closest to the *Safe Domain Relational Calculus (Safe DRC)* [92, Chapter 3.8] obtained from safe nonrecursive Datalog.

Rewriting a query formula into a safely evaluable form originated in the context of the classical relational calculus [50, 97]. The approach was later extended to incorporate *conditional* safety requirements of the form "*if* variable x is guaranteed to have a finite range, *then* so is variable y" [30, 31]. These were modelled by finiteness dependencies, as we have done here. However, the overall spirit of these approaches was to *avoid expensive transformations* of the formula [50]. This namely leads to smaller algebraic expressions as well. We on the other hand seek to *gather as much limitation information* as possible. The following example indicates why.

Example 4.4 Consider the query $(R_1(x, y) \vee (R_2(x) \wedge \phi)) \wedge R_2(y)$, where ϕ is a string formula satisfying $\{x\} \rightsquigarrow \{y\} \in \text{basis}(\phi)$. This query needs no rewriting, because the disjunction can be expressed as $R_1 \cup \sigma_{\mathcal{A}}(R_2 \times \Sigma^*)$, where $\mathcal{A}$ is a 2-FSA corresponding to ϕ from Theorem 3.1. On the other

Figure 4.2 Rewrite rules into QDNF.

$$
\begin{aligned}
\neg\neg\phi &\longrightarrow \phi \\
\neg(\phi \wedge \psi) &\longrightarrow (\neg\phi) \vee (\neg\psi) \\
\neg(\phi \vee \psi) &\longrightarrow (\neg\phi) \wedge (\neg\psi) \\
\exists x(\phi \vee \psi) &\longrightarrow (\exists x\phi) \vee (\exists x\psi) \\
\exists x\phi &\longrightarrow \phi, \text{ where } x \notin \text{free}\,(\phi) \\
\phi \wedge (\psi \vee \chi) \text{ or } (\psi \vee \chi) \wedge \phi &\longrightarrow (\phi \wedge \psi) \vee (\phi \wedge \chi) \\
\phi \wedge (\exists x\psi) \text{ or } (\exists x\psi) \wedge \phi &\longrightarrow \exists y(\phi \wedge \psi^{x\mapsto y})
\end{aligned}
$$

hand, rewriting this query as

$$(R_1(x,y) \wedge R_2(y)) \vee (R_2(x) \wedge \phi \wedge R_2(y)) \tag{4.6}$$

lets us replace the generator with the selection $\sigma_{\mathcal{A}}(R_2 \times R_2)$, which is likely to be much more efficient to evaluate. For example, if ϕ states that y is no longer than x, then there is an *exponential* difference in the intermediate result sizes of these two expressions. ■

Example 4.4 shows that because string restructuring is potentially very expensive, it should be used as sparingly as possible. Avoiding unnecessary restructuring requires in turn more limitation information than merely avoiding unnecessary rewriting.

Every Alignment Calculus formula can be transformed into QDNF by repeated application of the rewrite rules in Fig. 4.2. Logical shorthands like implication and the universal quantifier are expanded first so that the formula contains only the connectives '$\wedge$', '$\vee$', '$\neg$', and '$\exists$'. In the last rewrite rule, $y \notin \text{free}\,(\phi) \cup (\text{free}\,(\psi) \setminus \{x\})$, and $\psi^{x\mapsto y}$ denotes the formula obtained by replacing in formula ψ each free occurrence of the variable symbol x by the variable symbol y. Henceforth we consider implicitly '$\wedge$' and '$\vee$' polyadic, so that both $\alpha \wedge (\beta \wedge \gamma)$ and $(\alpha \wedge \beta) \wedge \gamma$ are "flattened" simply into $\alpha \wedge \beta \wedge \gamma$.

Strictly speaking, the rewrite rules in Fig. 4.2 do not yet define a proper normal form, because different rule application orders lead to different final forms for the same initial formula. Therefore we further specify that these rules are to be tried in the order listed. That is, we always try to push negations as deep down in the formula as possible without crossing any quantifiers, with quantifiers moving down as well to make way for the enclosing negations. The intuition for this choice is that a negated subformula is difficult to utilize as a source of limitation properties for the enclosing formula, and therefore negated formulae are to be minimized.

Example 4.5 Consider the query

$$R(x) \wedge (\forall y.\psi(y,x) \rightarrow R(y)) \wedge (\forall z.\psi(x,z) \rightarrow \neg R(z)),$$

where the string formula

$$\psi(x,y) \equiv ([x,y]_l x = y)^*.[x,y]_l x = \bot \wedge y \neq \bot$$

denotes the relation "x is a proper prefix of y". This query extracts from R all the strings x, whose every proper prefix y is also in R. Moreover, these y, which would also satisfy the criterion, are excluded from the answer, so that only the maximal "prefix-closed" x are reported. In QDNF, this formula becomes

$$\mathrm{R}(x) \wedge \neg(\exists y.\psi(y,x) \wedge \neg R(y)) \wedge \neg(\exists z.\psi(x,z) \wedge R(z)). \qquad (4.7)$$

This formula is then just a conjunction: the implicit topmost disjunction '$\vee$' is unary, and the optional quantifier prefix Q^* is missing. Fig. 4.3 presents the syntax tree for Eq. (4.7). ■

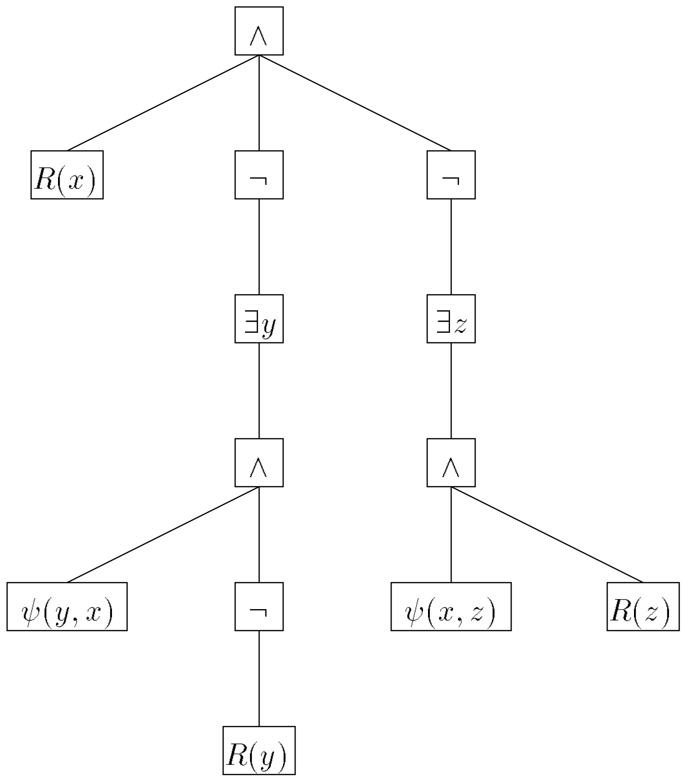
Recalling that $\neg\exists x(\phi \wedge \psi)$ is logically equivalent to $\forall x(\phi \rightarrow \neg\psi)$, we classify QDNF formulae into a type hierarchy with respect to their (implicit) quantifier alternation structure, as follows.

Definition 4.4 *Let ϕ be a nondisjunctive QDNF formula. We define its* type *as follows:*

- *If ϕ is an atomic formula, or a negation of one, then its type is Π_0.*
- *If $\phi \equiv \exists z\psi$, then its type is Σ_k, where the type of the range ψ is either Π_{k-1} or Σ_k.*
- *If $\phi \equiv \neg\psi$ and ψ is not atomic, then its type is Π_k, where the type of ψ is Σ_k.*
- *If $\phi \equiv \psi \wedge \chi$, then its type is $\Pi_{\max(m,n)}$, where the type of ψ is Π_m and the type of χ is Π_n.*

Note that we departed slightly from the usual convention that unquantified formulae are simultaneously of type Π_0 and Σ_0 in Definition 4.4. This departure was made possible by the restricted syntactic form of QDNF formulae.

Figure 4.3 The structure of a formula in QDNF.



For example, we classify Eq. (4.7) in Example 4.5 as having the expected type Π_1: In the tree of Fig. 4.3, the subtrees below the existential nodes '$\exists y$' and '$\exists z$' receive type Π_0. Hence, these existential nodes receive type Σ_1, making their enclosing negations to receive type Π_1. The whole formula is then a conjunction of leaf $R(x)$ with type Π_0, and two subformulae of type Π_1.

4.4.3 Safe QDNF

Let us now formulate the promised syntactic test that aims to delineate those QDNF formulae that are guaranteed to be domain independent. Later, we will show that this test is indeed a sound one.

Definition 4.5 *Let ϕ be a formula in QDNF. It is called* productive *if it* satisfies the finiteness dependency required $(\phi) = \emptyset \rightsquigarrow$ free (ϕ) *in the follow-*

ing sense.

The disjunction $\phi \equiv \psi \vee \chi$ *satisfies* required (ϕ) *if and only if* ψ *satisfies* required $(\psi) =$ required (ϕ) *and* χ *satisfies* required $(\chi) =$ required (ϕ).

The quantification $\phi \equiv \exists z \psi$ *satisfies* required $(\phi) = X \rightsquigarrow Y$ *if and only if its range* ψ *satisfies* required $(\psi) = X \rightsquigarrow (Y \cup \{z\})$.

Consider finally a maximal conjunction C *of the form*

$$\phi \equiv \underbrace{a_1 \wedge \ldots \wedge a_p}_{\alpha_\phi} \wedge \underbrace{\neg b_1 \wedge \ldots \wedge \neg b_q}_{\beta_\phi} \wedge \underbrace{\neg D_1 \wedge \ldots \wedge \neg D_r}_{\Delta_\phi}, \qquad (4.8)$$

where the a_i *and* b_j *are atomic formulae and the* D_k *are not. This conjunction satisfies* required $(\phi) = X \rightsquigarrow Y$ *if and only if*

1. $\bigcup_{i=1}^{p}$ basis $(a_i) \vdash$ required (ϕ) *holds; and*
2. *every subformula* D_k *satisfies in turn*

$$\text{required}\,(D_k) = (\text{free}\,(D_k) \cap (X \cup Y)) \rightsquigarrow \emptyset.$$

In addition, we call an Alignment Calculus formula not in QDNF productive if and only if its QDNF form given by the rewrite rules in Fig. 4.2 is productive.

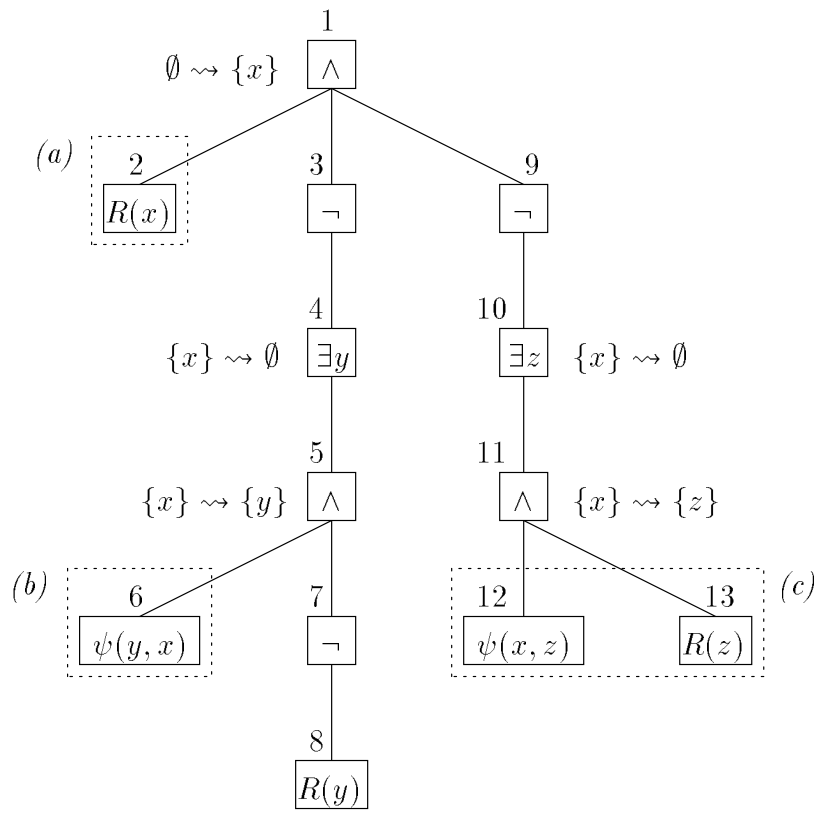
Recall that we are treating here the finiteness dependencies required (...) syntactically and not semantically; in particular, subcase (2) in the conjunctive case of Definition 4.5, is such that the right-hand side $\emptyset$ of required (D_k) will receive the outermost bound variable of D_k at the next stage of the inductive definition.

Definition 4.5 approaches the detection of domain independent QDNF formulae as an *attribute evaluation* [7, Chapter 5] problem. The finiteness dependencies required (ϕ) are *inherited top-down* in the syntax tree (such as the one in Fig. 4.3), while their truth is *synthesized bottom-up* from the results of verifying the corresponding Conditions (1) by applying Lemma 4.3.

Example 4.6 Fig. 4.4 adorns Fig. 4.3 with the inherited dependencies, and shows as dotted boxes *(a)*, *(b)* and *(c)* the atoms used in verifying condition (1) for the nearest enclosing dependency. For instance, node 11 inherits the dependency $\{x\} \rightsquigarrow \{z\}$, and nodes 12 and 13 that comprise box *(c)* are the ones used in verifying it.

Let us now denote by Γ_i the dependency inherited by node i, and by U_d the union of all basis (χ), where χ belongs to box *(d)*. Then $U_a \vdash \Gamma_1$, because $\Gamma_1 \in$ basis (node 2); $U_b \vdash \Gamma_5$, because $\Gamma_5 \in$ basis (node 6) – any given

Figure 4.4 Labelling a QDNF formula with finiteness dependencies.



string x has only finitely many proper prefixes y (but not conversely); and finally, $U_c \vdash \Gamma_{11}$, because $\emptyset \rightsquigarrow \{z\} \in$ basis (node 13). Therefore Eq. (4.7) in Example 4.5 is productive. Note, however, that Algorithm 2 reverses the order of nodes 12 and 13 in box *(c)* of Fig. 4.4. ■

Definition 4.5 relies on the ability to perform membership tests on the sets basis (a_i). This chapter has identified three different classes of these tests: (1) Section 4.1 showed them to be undecidable in general. (2) Conversely, Section 4.2 showed that they remain decidable for the right-restricted string formulae a_i of Definition 2.2. (3) And finally, Section 4.3 presented a heuristic for computing some tests correctly. Accordingly, we identify three different variants of productivity for a QDNF formula ϕ:

1. ϕ is productive *in the abstract sense*, if ϕ satisfies Definition 4.5 under the assumption that all the membership tests to the basis (a_i) are answered correctly in all cases, disregarding whether this is computable or not.

   Note that this is already a strictly weaker notion than the domain independence of Definition 3.1, simply because of the fullness and conjunctivity restrictions.

2. ϕ is *right-restricted* productive, if ϕ could be seen to satisfy Definition 4.5, and all the membership tests that had to performed on the sets basis (a_i) were of a form, where a_i was a relational atom or a right-restricted string formula.

   In other words, we take basis $(a_i) = \emptyset$, whenever a_i is a string formula, which is not right-restricted.

   Then by Theorem 4.2 the right-restricted productive Alignment Calculus formulae form the syntactically safe sublanguage we have been searching for.

3. ϕ is *heuristically* productive, if ϕ satisfied Definition 4.5 as in right-restricted productivity above, except that now all membership tests performed on the basis (a_i) must either be performed on a relational a_i atom or the test must be solved by the heuristic developed in Section 4.3.

   In other words, we take basis (a_i) to contain exactly those finiteness dependencies that Algorithm 1 could verify to hold.

   This provides another notion of a syntactically safe sublanguage.

Variants 2 and 3 are special cases of variant 1, and they share a common subset by Theorem 4.5, but remain distinct by Example 4.3.

In Example 4.6 the string formula $\psi(y, x)$, which was used for generating y given x, is unidirectional, and therefore Eq. (4.7) is seen to be productive under all these three variants of the notion.

It remains to prove that productive formulae are indeed domain independent. Let us first prove an auxiliary result about how Definition 4.5 handles variable symbols.

Lemma 4.4 *Let ϕ be a subformula of a formula Φ in QDNF, for which the corresponding finiteness dependency* required $(\phi) = X \rightsquigarrow Y$ *is both defined and satisfied. Then X and Y partition* free (ϕ).

Proof: Consider the following five claims:

1. $X \cap Y = \emptyset$.
2. $X \subseteq \text{free}(\phi)$.
3. $\text{free}(\phi) \subseteq X \cup Y$.
4. If $z \in X \cup Y$, then $z \in \text{free}(\Phi)$ or ϕ appears within the range of some quantification $\exists z(\ldots)$ in Φ.
5. $Y \subseteq \text{free}(\phi)$.

Claims (1) to (3) and (5) together imply the claim. First, we prove Claims (1) to (4) by induction on the top-down order, in which the finiteness dependencies $\text{required}(\phi)$ are defined. In doing so, we do not yet assume these dependencies to be satisfied.

For the whole formula Φ, $\text{required}(\Phi) = \emptyset \rightsquigarrow \text{free}(\Phi)$ by Definition 4.5. This propagates through disjunctions $\phi \equiv \psi \vee \chi$, because $\text{required}(\psi) = \text{required}(\chi) = \text{required}(\phi) = \text{required}(\Phi)$. Claims (1) and (2) hold, because the left side of the finiteness dependencies is $X = \emptyset$. In turn, Claims (3) and (4) hold, because $X = \emptyset$ and $Y = \text{free}(\Phi)$.

Consider then a quantification $\phi \equiv \exists z \psi$. By induction, $\text{required}(\phi) = X \rightsquigarrow Y$ satisfies all the claims above, and we have to verify them for $\text{required}(\psi) = X \rightsquigarrow (Y \cup \{z\})$. We have $z \notin X \cup Y$ by the inductive assumption Claim (4), and the assumption that all the bound variables are standardized apart in Φ. Therefore Claim (1) continues to hold. Claims (2) and (3) continue to hold by their respective inductive assumptions, because $\text{free}(\psi) = \text{free}(\phi) \cup \{z\}$. Finally, Claim (4) continues to hold by the inductive assumption Claim (4) and the form of ϕ.

Consider finally a maximal conjunction ϕ as in Eq. (4.8). By induction, $\text{required}(\phi) = X \rightsquigarrow Y$ satisfies all the claims above, and we have to verify them for each $\text{required}(D_k) = (\text{free}(D_k) \cap (X \cup Y)) \rightsquigarrow \emptyset$. Claim (1) contines to hold, because the right side of $\text{required}(D_k)$ is $\emptyset$. Claim (2) continues to hold by the definition of the left side of $\text{required}(\phi)$. Claim (3) continues to hold, because $\text{free}(D_k) \subseteq \text{free}(\phi) \subseteq X \cup Y$ by the inductive assumption Claim (3). Finally, Claim (4) continues to hold by the inductive assumption Claim (4), the definition of the left side of $\text{required}(\phi)$, and the emptiness of its right side.

We finally prove the remaining Claim (5) by induction on the bottom-up order, in which the finiteness dependencies $\text{required}(\phi)$ are satisfied. Here we do assume them to be satisfied; we can apply Claims (1) to (4), because this assumption was not used above in proving them.

Consider first a maximal conjunction ϕ of the form given in Eq. (4.8). Because $\text{required}(\phi) = X \rightsquigarrow Y$ is satisfied, $\bigcup_{i=1}^{p} \text{basis}(a_i) \vdash X \rightsquigarrow Y$ holds.

Now $X \cap Y = \emptyset$ by Claim (1), so Y must be a subset of the variable symbols appearing in $\bigcup_{i=1}^{p} \text{basis}(a_i)$, by the soundness of Armstrong's axioms [92, Lemma 7.1]. Because the variable symbols appearing in any basis (a_i) are the ones in free (a_i), we have that $Y \subseteq \text{free}(\alpha_\phi) \subseteq \text{free}(\phi)$, satisfying Claim (5).

For a quantification $\phi \equiv \exists z \psi$ with required $(\phi) = X \rightsquigarrow Y$, induction provides that $Y \cup \{z\} \subseteq \text{free}(\psi) = \text{free}(\phi) \cup \{z\}$. We have $z \notin Y$, by Claim (4) and the assumption that quantified variables are standardized apart in Φ. Hence, $Y \subseteq \text{free}(\phi)$, as required for Claim (5).

For a disjunction $\phi \equiv \psi \vee \chi$ we have by Definition 4.5 that required $(\phi) =$ required $(\psi) =$ required $(\chi) =$ required $(\Phi) = \emptyset \rightsquigarrow \text{free}(\Phi)$. Induction provides that free $(\Phi) \subseteq$ free (ψ) and free $(\Phi) \subseteq$ free (χ). Hence, free $(\Phi) =$ free $(\psi) =$ free $(\chi) =$ free (ϕ), and Claim (5) holds. □

We are now in a position to present the central result that connects the syntactic condition given in Definition 4.5 into the semantic condition of a formula satisfying a finiteness dependency given in Definition 3.2. Formally, the result says that if the subformula ϕ satisfies its limitation condition required $(\phi) = X \rightsquigarrow Y$ in the manner described by Definition 4.5, then ϕ is "relatively domain independent" in the following sense: if ψ is a domain independent formula with free $(\psi) = X$, then $\phi \wedge \psi$ is also a domain independent formula with free $(\phi \wedge \psi) = X \cup Y$. That is, ϕ can produce finitely the values for Y, if given the values for X by ψ. Hence, this construction can be viewed as an extension of Definition 3.3 from string to Alignment Calculus formulae. And indeed, the proof of the result presents exactly such an extension.

Lemma 4.5 *Let Φ be a formula in QDNF, and ϕ its nondisjunctive subformula, for which the finiteness dependency* required $(\phi) = X \rightsquigarrow Y$ *is both defined and satisfied by Definition 4.5. Then there exists a function $\mathcal{W}_\phi \colon DB \times \mathbb{N} \to \mathbb{N}$ satisfying the following property: for all databases db, initial alignments A_0, assignments θ, and constants*

$$\ell \geq \mathcal{W}_\phi(db, \max\{|\sigma_{A_0}(\theta x)| : x \in X\}),$$

the statements $\langle A_0, db\rangle \models \phi\theta$ and $\left\langle A_0^\ell, db\right\rangle \models \phi\theta$ are equivalent.

Proof: The proof is by induction on the structure of the subformula ϕ. Note that necessarily $\mathcal{W}_\phi(db, m) \geq m$.

Consider first a quantification of the form $\phi \equiv \exists z \psi$. By Definition 4.5, required $(\psi) = X \rightsquigarrow (Y \cup \{z\})$ is satisfied. Then inductively the truth values $\langle A_0, db\rangle \models \psi\theta[z = i]$ and $\left\langle A_0^\ell, db\right\rangle \models \psi\theta[z = i]$ are equivalent for all db, A_0, θ, $i \in \mathbb{N}$, and $\ell \geq \mathcal{W}_\psi(db, \max\{|\sigma_{A_0}(\theta x)| : x \in X\})$. Then by the truth definition of quantification (case 13) $\langle A_0, db\rangle \models \phi\theta$ and $\left\langle A_0^\ell, db\right\rangle \models \phi\theta$ are also equivalent for all db, A_0, θ, and $\ell \geq \mathcal{W}_\psi(db, \max\{|\sigma_{A_0}(\theta x)| : x \in X\})$. Hence, the claim holds by choosing $\mathcal{W}_\phi = \mathcal{W}_\psi$.

Consider then a maximal conjunction ϕ, as in Eq. (4.8). By Definition 4.5 we have both $\bigcup_{i=1}^p \operatorname{basis}(a_i) \vdash \operatorname{required}(\phi)$, and every D_k satisfies $\operatorname{required}(D_k) = (\operatorname{free}(D_k) \cap (X \cup Y)) \rightsquigarrow \emptyset$. Furthermore, we assume α_ϕ to be ordered as in Lemma 4.3; it applies, because $\operatorname{free}(\alpha_\phi) \subseteq \operatorname{free}(\phi) = X \cup Y$, by Lemma 4.4. Our first task is to construct by induction on i functions $\mathcal{W}_i \colon DB \times \mathbb{N} \to \mathbb{N}$ for the prefixes $\alpha_\phi^i = a_1 \wedge \ldots \wedge a_i$ of this ordered α_ϕ having the following property: for all db, A_0, and θ, the truth of $\langle A_0, db\rangle \models \alpha_\phi^i\theta$ implies

$$\max\left\{|\sigma_{A_0}(\theta z)| : z \in X \cup \operatorname{free}\left(\alpha_\phi^i\right)\right\} \leq \mathcal{W}_i(db, \max\{|\sigma_{A_0}(\theta x)| : x \in X\}).$$

That is, $\mathcal{W}_i$ bounds from above the lengths of the strings that can be assigned to the free variables of α_ϕ^i to make it true. Note that this requirement is exactly that $\mathcal{W}_i$ is the function sought in this lemma for the subconjunction α_ϕ^i; this easier formulation is possible, because α_ϕ^i is quantifier-free, while the more involved one in the statement of the lemma is mandated by the need to account for quantifiers as well as free variables. This is in analogy to Section 3.2, where we noted the difference between finiteness dependencies and domain independence on quantified formulae.

The base case of the induction is α_ϕ^0, the empty prefix, which is variable-free and identically true. It suffices to choose $\mathcal{W}_0(db, m) = m$.

Assume then inductively that $\mathcal{W}_{i-1}$ has been defined for the prefix α_ϕ^{i-1}, and consider how to extend it to cover the next atom a_i as well. If $\operatorname{free}(a_i) \subseteq X \cup \operatorname{free}\left(\alpha_\phi^{i-1}\right)$, it suffices to choose $\mathcal{W}_i = \mathcal{W}_{i-1}$. Otherwise there are two cases to consider.

The first case to consider is when a_i is a relational atom $R(\ldots)$. Then $\mathcal{W}_{i-1}$ must be enlarged so that none of the strings in $db(R)$ are omitted. This is achieved by choosing $\mathcal{W}_i(db, m) = \max(\mathcal{W}_{i-1}(db, m), \max(R, db))$, where $\max(R, db)$ extracts the length of the longest string assigned to R by db, as in Eq. (3.3).

The remaining case to consider is when a_i is a string formula. Then a_i can generate from its "input" variables $I = \operatorname{free}(a_i) \cap \left(X \cup \operatorname{free}\left(\alpha_\phi^{i-1}\right)\right)$ new values for its "output" variables $O = \operatorname{free}(a_i) \setminus I$. Then $\mathcal{W}_{i-1}$ bounds the lengths of the strings assigned to the variables in I, and the problem is to enlarge it so that none of the strings assignable to O to make a_i true are omitted. Now $I \rightsquigarrow O \in \operatorname{basis}(a_i)$, because otherwise a_i could not have appeared in position i of the ordered α_ϕ. Therefore there exists a corresponding limitation function $\mathcal{W}_{a_i} \colon \mathbb{N}^{|I|} \to \mathbb{N}$, as in Definition 3.3; this function gives the required upper bound on the lengths of the outputs O given the lengths of the inputs I. We can again assume without loss of generality that $\mathcal{W}_{a_i}$ satisfies the monotonicity condition of Eq. (3.8). Then it remains to combine $\mathcal{W}_{a_i}$ and $\mathcal{W}_{i-1}$ into $\mathcal{W}_i$ by choosing

$$\mathcal{W}_i(db, m) = \max\left\{\mathcal{W}_{a_i}(m_1, \ldots, m_{|I|}) \colon m_1, \ldots, m_{|I|} \leq \mathcal{W}_{i-1}(db, m)\right\}.$$

Because free $(\phi) = X \cup Y$ by Lemma 4.4, the function $\mathcal{W}_p$ produced by the subinduction above suffices unmodified as the function $\mathcal{W}_{\alpha_\phi \wedge \beta_\phi}$ for the quantifier-free conjunction $\alpha_\phi \wedge \beta_\phi$. If Δ_ϕ is missing, the whole induction of the lemma is complete.

Otherwise we have to expand $\mathcal{W}_{\alpha_\phi \wedge \beta_\phi}$ further to account for the subformulae $\neg D_k$ in Δ_ϕ. Induction provides functions $\mathcal{W}_{D_k} \colon DB \times \mathbb{N} \to \mathbb{N}$, for which $\langle A_0, db\rangle \models D_k\theta$ and $\left\langle A_0^\ell, db\right\rangle \models D_k\theta$ are equivalent for all db, A_0, θ, and $\ell \geq \mathcal{W}_{D_k}(db, \max\{|\sigma_{A_0}(\theta x)| : x \in \text{free}\,(D_k)\})$. (Recall that free $(D_k) \subseteq$ free $(\phi) = X \cup Y$.) In turn, $\mathcal{W}_{\alpha_\phi \wedge \beta_\phi}$ provides an upper bound for the strings bound to the variables in free (D_k). Therefore choosing

$$\mathcal{W}_\phi(db, m) = \max\left\{\mathcal{W}_{D_k}(db, \mathcal{W}_{\alpha_\phi \wedge \beta_\phi}(db, m)) \colon k = 1, \ldots, r\right\}$$

suffices to get a function that satisfies the claim of this lemma. □

Note in particular that by Lemma 4.5, if ϕ satisfies required (ϕ) by Definition 4.5, then ϕ also satisfies required (ϕ) by Definition 3.2 as well. That is, the syntactic criterion does indeed imply the semantic one, as claimed.

Another view of Lemma 4.5 is that it applies repeatedly the following implicit rewriting rule, starting at the top of the subformula ϕ, and working downwards. Let $\phi' \equiv \chi \wedge \exists z\psi$ be a subformula, where previous rewritings have endowed this χ known inductively to satisfy the limitation condition $\emptyset \rightsquigarrow$ free (χ). In addition, ψ is known to satisfy both free $(\psi) \subseteq$ free $(\chi) \cup \{z\}$ and the limitation condition free $(\psi) \rightsquigarrow \{z\}$. Then rewriting ϕ' as $\phi'' \equiv \chi \wedge \exists z(\chi \wedge \psi)$, and proceeding inductively on $\chi \wedge \psi$, puts eventually the query into a form, where every subformula is easily translated into finitely evaluable expressions. This is similar in spirit to rewriting a formula, which has already been determined to be safe by rewritings, further into a form with a straightforward algebraic translation [30, Section 7.2][31, Section 7.3][97, Section 9].

Example 4.7 Let us apply Lemma 4.5 to Eq. (4.7), whose structure is depicted in Fig. 4.4, and see what kind of functions the construction generates.

Here $\mathcal{W}_i$ denotes the function corresponding to node i or box *(i)*.

$$\begin{aligned}
\mathcal{W}_1(db, m) &= \max(\mathcal{W}_4(db, \mathcal{W}_a(db, m)), \\
&\qquad\qquad \mathcal{W}_{10}(db, \mathcal{W}_a(db, m))). && (4.9)\\
\mathcal{W}_a(db, m) &= \mathcal{W}_2(db, m) \\
&= \max(m, \max(R, db)). \\
\mathcal{W}_4(db, m) &= \mathcal{W}_5(db, m) \\
&= \mathcal{W}_b(db, m) \\
&= \max\left(m, \max\left\{\mathcal{W}_6(m_1) : m_1 \leq m\right\}\right). \\
\mathcal{W}_6(db, m) &= m. && (4.10)\\
\mathcal{W}_{10}(db, m) &= \mathcal{W}_{11}(db, m) \\
&= \mathcal{W}_c(db, m) \\
&= \mathcal{W}_{13}(db, m) \\
&= \max(m, \max(R, db)).
\end{aligned}$$

All of the above simplifies into

$$\mathcal{W}_1(db, m) = \max(m, \max(R, db)). \tag{4.11}$$

Eq. (4.10) is obtained by applying Theorem 3.3 to the 2-FSA $\mathcal{A}_{\text{prefix}}$ corresponding to $\psi(x, y)$. This $\mathcal{A}_{\text{prefix}}$ has the initial state s, the final state f, and one intermediate state p, connected with the following transitions.

$$t_1 = s \xrightarrow[1,1]{¢,¢} p \qquad t_2^c = p \xrightarrow[1,1]{c,c} p \qquad t_3^c = p \xrightarrow[0,0]{\$,c} f \qquad \text{for all } c \in \Sigma$$

Eq. (4.11) finally shows that we should not omit any strings from R from consideration when evaluating the query in Eq. (4.7). ∎

As witnessed by Example 4.7 above, Lemma 4.5 can be viewed as the semantic counterpart to the bottom-up synthesized part of the syntactic attribute evaluation scheme given in Definition 4.5. We are now synthesizing the relative limitation function $\mathcal{W}(db, m)$, where the second argument m acts as a placeholder for the top-down inherited part. For example, Eq. (4.9) synthesized $\mathcal{W}_1$ from $\mathcal{W}_a$, $\mathcal{W}_4$, and $\mathcal{W}_{10}$, by using $\mathcal{W}_a$ to provide the value of m for $\mathcal{W}_4$ and $\mathcal{W}_{10}$. It then remains to bind the initial value of m at the root (node 1 in our example), or to initiate the inheritance of the top-down attribute.

Theorem 4.6 *If an Alignment Calculus formula is productive in the abstract sense, then it is domain independent.*

Proof: Let Φ be a QDNF formula that is productive in the abstract sense. Then by Definition 4.5 Φ is of the form $\Phi \equiv D_1 \vee \ldots \vee D_s$, where the D_i are no longer disjunctions, and each D_i satisfies required $(D_i) =$ required $(\Phi) = \emptyset \rightsquigarrow$ free (Φ).

Lemma 4.5 applies to each D_i, and produces a function $\mathcal{W}_{D_i} : DB \times \mathbb{N} \to \mathbb{N}$ having the following property: for all db, A_0, θ, and $\ell \geq \mathcal{W}_{D_i}(db, \max\emptyset)$, the statements $\langle A_0, db\rangle \models D_i\theta$ and $\left\langle A_0^\ell, db\right\rangle \models D_i\theta$ are equivalent. Then $[\![D_i]\!]_{db} = [\![D_i]\!]^\ell_{db}$ for all db and $\ell \geq \mathcal{W}'_{D_i}(db) = \mathcal{W}_{D_i}(db, 0)$, by Eq. (2.4). Therefore each D_i is domain independent with $\mathcal{W}'_{D_i} : DB \to \mathbb{N}$ as the corresponding function, by Definition 3.1.

On the other hand, Lemma 4.4 leads to free $(D_1) = \ldots =$ free $(D_s) =$ free (Φ). Then $[\![\Phi]\!]_{db} = [\![D_1]\!]_{db} \cup \ldots \cup [\![D_s]\!]_{db}$, by Eq. (3.7). Therefore $[\![\Phi]\!]_{db} = [\![\Phi]\!]^\ell_{db}$ for all db and $\ell \geq \mathcal{W}_\Phi(db) = \max(\mathcal{W}'_{D_1}(db), \ldots, \mathcal{W}'_{D_s}(db))$. Hence, the whole Φ is also domain independent with $\mathcal{W}_\Phi$ as the corresponding function, by Definition 3.1. □

The converse of Theorem 4.6 does not hold; for example, Eq. (4.5) is in QDNF and domain independent, but not productive.

Example 4.8 Theorem 4.6 shows that a domain independence function $\mathcal{W}_{\text{Eq. (4.7)}}(db)$ of the formula Eq. (4.7) in Example 4.5 can be obtained from Eq. (4.11) by substituting $m = 0$. Hence, $\mathcal{W}_{\text{Eq. (4.7)}}(db) = \max(R, db)$, and the truth of the formula in Eq. (4.7) can indeed be correctly determined by restricting attention to all strings no longer than the ones currently appearing in R. ■

Chapter 5

Evaluating Alignment Calculus Queries

This chapter presents a concrete evaluation stategy for the two syntactically safe sublanguages of Alignment Calculus identified in Subsection 4.4.3, the right-restricted and heuristically productive formulae. The strategy has two components: a query compilation algorithm, and an execution algorithm for the compiled queries.

The first component is explained in Section 5.1. It gives a compilation algorithm that converts productive queries into finitely evaluable algebraic expressions.

The second component then evaluates a given finitely evaluable expression on the current database. We omit discussing the algebraic operations that do not access the internal structure of strings (these being the relation symbols, set differece, union, Cartesian product, and union); instead, Section 5.2 concentrates on selections, and especially generators, which combine selections with domain symbols to perform string restructuring. The aim is to show how generators can be evaluated without fear of a nonterminating computation.

Finally, Section 5.3 discusses briefly the computational costs inherent in the second component presented in Section 5.2.

5.1 Translation to Alignment Algebra

This chapter demonstrates an algorithm, which receives as input a productive Alignment Calculus formula Φ, and produces as output its *finitely evaluable translation* F; that is, F is a finitely evaluable Alignment Algebra expression with $db(F) = [\![\Phi]\!]_{db}$ for all $db \in DB$. Theorem 3.5 presented a related transformation from *general* Alignment Calculus into *general* Alignment Algebra; our aim here is to show how the extra condition of productivity can be used to translate Φ into a finitely evaluable F. This translation is depicted as Algorithm 3.

Algorithm 3 implements (a variant of) Theorem 4.6, where finitely evaluable expressions are constructed instead of limitation functions. This construction proceeds in turn by walking the adorned syntax tree of the productive QDNF formula, like the one depicted in Fig. 4.4, exactly as the proof of Theorem 4.6 did by induction.

The parameters of subroutine Lemma4.5 in Algorithm 3 are sufficiently involved to warrant a separate exposition.

- The first input parameter ϕ is a non-disjunctive QDNF subformula, which satisfies required $(\phi) = X \rightsquigarrow Y$.

- The second input parameter E is a finitely evaluable expression of arity $a(E) = |X|$.

  Denote as $\mathrm{src}(X, E)$ an Alignment Calculus formula ψ, for which free $(\psi) = X$ and $[\![\psi]\!]_{db} = db(E)$. That is, $\mathrm{src}(X, E)$ stands for an already translated formula, whose free variables are X and whose translation is already known to be E.

- The output parameter F is a finitely evaluable expression of arity $a(F) = |X| + |Y|$.

  This F will be a translation of $\mathrm{src}(X, E) \wedge \phi$.

Hence, E acts as a source of candidate values for the "input" variables X. F in turn uses these candidate values to generate the corresponding "output" values Y, and in the process eliminates those X for which no suitable Y are found. This subroutine therefore implements Lemma 4.5, as its name implies. E acts as the attribute that is inherited top-down in the construction, while F acts as the bottom-up synthesized attribute.

Another view of parameter E to subroutine Lemma4.5 is that it implements the *Sideways Information Passing Strategy (SIPS)* used in optimizing Datalog evaluation [93, Chapter 12.6]. Consider for example the adorned syntax tree in Fig. 4.4. The subtree rooted at node 3 receives as E the translation of node 2, its left sibling. Similarly, the subtree rooted at node 9 receives as its E the translation of both its left siblings, and these consist of nodes 2–8. In general, when the translation has reached node n in a syntax tree, the strategy has collected the joint effects of all subtrees that are to the left of the path from the root of the whole tree into node n.

Definition 4.5 presented the satisfaction of required $(\phi) = X \rightsquigarrow Y$ syntactically, while Lemma 4.5 provided a semantic interpretation. Now this

Algorithm 3 The query translation algorithm.

```
proc Algorithm3( value    Φ : a productive formula in QDNF;
                 result   F : a translation of Φ);
     D_1,...,D_s := the disjuncts of Φ;
     Perform call Lemma4.5(D_i, πΣ^{≤0}, F_i) for 1 ≤ i ≤ s;
     F := F_1 ∪ ... ∪ F_s
end  Algorithm3;

proc Lemma4.5( value    φ, E : as in the text;
               result   F    : as in the text);
     if   φ is of the form ∃zψ ⟶
               call Lemma4.5(ψ, E, G);
               F := translate by Theorem 3.5 ∃z src(free (ψ), G)
       ▯  φ is a conjunction as in Eq. (4.8) ⟶
               Z_0, F_0, i, j, k := left side X of required (φ), E, 1, 1, 1;
               do   (i ≤ p) ∧ (a_i is a string formula) ⟶
                         call Algorithm4(Z_{i−1}, F_{i−1}, a_i, F_i);
                         Z_i, i := Z_{i−1} ∪ free (a_i), i + 1
                 ▯  (i ≤ p) ∧ (a_i is a relational atom) ⟶
                         Z_i, i, F_i := Z_{i−1} ∪ free (a_i), i + 1, translate
                         by Theorem 3.5 src(Z_{i−1}, F_{i−1}) ∧ a_i
               od;
               do   (j ≤ q) ∧ (b_j is a string formula) ⟶
                         call Algorithm4(Z_p, F_p, b_j, G);
                         j, F_p := j + 1, F_p \ σ_A G
                 ▯  (j ≤ q) ∧ (b_j is a relational atom) ⟶
                         G := translate by Theorem 3.5 src(Z_p, F_p) ∧ b_j;
                         j, F_p := j + 1, F_p \ G
               od;
               do   k ≤ r ⟶
                         G := translate by Theorem 3.5 ∃z⃗ src(Z_p, F_p),
                         where z⃗ consists of the variables in X \ free (D_k);
                         call Lemma4.5(D_k, G, H);
                         k, F_p := k + 1, translate by Theorem 3.5
                         src(Z_p, F_p) ∧ src(free (D_k), G \ H)
               od;
               F := F_p
     fi
end  Lemma4.5;
```

Algorithm 4 Translating string formulae to generators.

proc Algorithm4(`value` X : set of variable symbols;
`value` E : finitely evaluable expression;
`value` ϕ : string formula;
`result` F : translation of $\mathrm{src}(X, E) \wedge \phi)$;
Define a bijection $f\colon A \to \{1, \ldots, |A|\}$ for $A = \mathrm{free}\,(\phi) \cup X$ as

$$f(z) = \begin{cases} \mathrm{pos}(X, z) & z \in X \\ |X| + \mathrm{pos}(\mathrm{free}\,(\phi) \setminus X, z) & z \notin X \end{cases}$$

ψ := rename $[\,]_l \bigwedge_{x \in X} x = \bot.\phi$ with $z \mapsto x_{f(z)}$ for all $z \in A$;
$\mathcal{A}$:= the $|A|$-FSA corresponding to ψ by Corollary 3.1;
F := $\pi_{g(1),\ldots,g(|A|)}\sigma_{\mathcal{A}}\left(E \times (\Sigma^*)^{|\mathrm{free}(\phi)\setminus X|}\right)$,
where $g(n) = f^{-1}(n\text{th element of } A)$.
end Algorithm4;

subroutine provides an algorithmic interpretation as a guide for constructing translations, where the only string restructuring mechanism is the generator introduced in Definition 3.4. The construction of these generators is in turn possible by the aforementioned sideways information passing, which guarantees that a nonnegated string formula a_i is translated only after the strategy has collected enough information for E to provide values for all the variables I on the left side of some limitation property $I \rightsquigarrow O \in \mathrm{basis}\,(a_i)$.

In turn, Algprithm 4 constructs the necessary generators in a straightforward manner. The required reorderings of the FSA tapes are presented explicitly for the sake of completeness.

Example 5.1 Let us for comparison see how Algorithm 3 treats the query ϕ from Example 3.2 on page 59. This query is productive, because ϕ is already in QDNF, and $\{y, z\} \rightsquigarrow \{x\} \in \mathrm{basis}\,(\psi)$. (This latter property can furthermore be decided, because ψ is unidirectional.)

The computation proceeds as follows. The initial `call` Algorithm3(ϕ, F) reduces to `call` Lemma4.5(ϕ, $\pi\Sigma^{\leq 0}$, F), because ϕ is not disjunctive. The quantifier prefix "$\exists y, z$" of ϕ is first eliminated by having $F = \pi_1\pi_{1,2}F'$, where F' is obtained recursively from `call` Lemma4.5($R_1(y) \wedge R_3(z) \wedge \psi$, $\pi\Sigma^{\leq 0}$, F'). This call in turn reduces to an execution of the first loop in the latter case of the subroutine.

Initially $Z_0 = \emptyset$ and $F_0 = \pi\Sigma^{\leq 0}$. Because $a_1 = R_1(y)$ is a relational atom, $Z_1 = \{y\}$ and and $F_1 = F_0 \times R_1$; similarly $Z_2 = \{y, z\}$ and $F_2 =$

$F_1 \times R_3$. Because $a_3 = \psi$ is a string formula, $Z_3 = \{x, y, z\}$, and $F' = F_3 = \pi_{3,1,2}\sigma_{\mathcal{A}'}(F_2 \times \Sigma^*)$ is obtained from Algorithm 4 with inputs Z_2, F_2 and ψ. Here $\mathcal{A}'$ is a 3-FSA corresponding to ψ, except that input tapes 1 and 2 correspond to the already known variables Z_2, while output tape 3 corresponds to the remaining variable x.

In summary,

$$\begin{aligned} F &= \pi_1\pi_{1,2}\pi_{3,1,2}\sigma_{\mathcal{A}'}((\pi\Sigma^{\leq 0}) \times R_1 \times R_3 \times \Sigma^*) \\ &= \pi_3\sigma_{\mathcal{A}'}(R_1 \times R_3 \times \Sigma^*), \end{aligned}$$

which is the same expression as the translation obtained in Example 3.2, except for the ordering of the columns. However, Algorithm 3 produced it via less involved simplifications than in Example 3.2. ■

Example 5.2 As another example, let us determine the translation of the query in Example 4.5 on page 87. A similar computation to Example 5.1 above leads to the expression

$$\begin{aligned} F_1 &= F_3 \setminus \pi_1\sigma_{\mathcal{A}_{12}}(F_3 \times R)F_{10}, \text{ where} \\ F_3 &= R \setminus \pi_1(F_6 \setminus \pi_{1,2}\sigma_{\mathcal{B}}(F_6 \times R)), \\ F_6 &= \sigma_{\mathcal{A}_6}(R \times \Sigma^*), \text{ and} \\ L(\mathcal{B}) &= \{\langle u, v, v\rangle : u, v \in \Sigma^*\}. \end{aligned}$$

The indices here refer to the adorned syntax tree of this query given in Fig. 4.4 in the following way: each $\mathcal{A}_i$ refer to 2-FSA corresponding to the string formula in node i by Theorem 3.1, while each F_j represents the result of subroutine Lemma4.5 in Algorithm 3 when called for the subtree rooted at node j.

The meanings of these (sub)expressions are as follows. F_6 generates finitely the values for the variables x and y, where x comes from R and y is a proper prefix of x. The innermost subtraction of F_3 eliminates from F_6 those x, y for which y is in R as well; hence, the whole F_3 yields those x from R, whose proper prefixes are all in R. Finally, the translation F_1 of the whole query eliminates from F_3 those x that are a proper prefix of some z in R. ■

The form of the finitely evaluable expression produced by Algorithm 3 depends crucially on the ordering of the positive atoms $\alpha_\phi \equiv a_1 \wedge \ldots \wedge a_p$ from Lemma 4.3. Because subroutine Lemma4.5 builds a natural join [93, Chapter 12.6] of the input expression E and α_ϕ, as witnessed by Example 5.1, finding good reorderings of α_ϕ is analogous to optimizing joins containing user-defined predicates [17]. However, the evaluation costs of our "user-defined predicates", namely generators, do not depend only on the sizes of their inputs, but also on the number of their outputs, and this in turn depends on their chosen position within the join.

Consider for instance the rightmost disjunct $\psi \equiv R_2(x) \wedge \phi \wedge R_2(y)$ in Eq. (4.6) from Example 4.4 on page 86, where ϕ is a string formula satisfying $\{x\} \rightsquigarrow \{y\}$. This ψ does not have to be reordered by Lemma 4.3, but then the optimization suggested by Example 4.4 would not be performed, and the cost of evaluating the generator corresponding to $R_2(x) \wedge \phi$ could be substantial, because the values for y are first generated and only then compared against the relation $db(R_2)$. On the other hand, reordering ψ to $R_2(x) \wedge R_2(y) \wedge \phi$ reduces the generator into just a selection over the Cartesian product of R_2 with itself, as was seen in Example 4.4.

It seems therefore prudent to require that Algorithm 2 may not select the next conjunct a to become a generator (that is, a string formula a satisfying $\text{free}(a) \nsubseteq Y$) unless no other choice is available. Section 5.3 will in turn identify some guidelines for choosing between different generators when necessary.

Furthermore, Algorithm 3 could equivalently choose to insert a negated subformula $\neg\psi$ of the conjunction ϕ in the middle of the positive atoms α_ϕ as soon as $\text{free}(\psi) \subseteq Z_{i-1}$. This would then cause the corresponding subtraction to be evaluated earlier, and that would cut down the size of the intermediate join result before subsequent (perhaps costly) restructuring. In a similar vein, an enclosing quantifier could be translated as soon as all the occurrences of the corresponding variable have been translated.

5.2 Evaluation of Generators

Let us now turn to evaluating the generators from Definition 3.4 that Algorithm 3 uses to implement the string restructurings in a given productive query. We view the problem of evaluating the expression $F = \sigma_{\mathcal{A}}(E \times (\Sigma^*)^l)$ as follows, where the $(k+l)$-FSA $\mathcal{A}$ satisfies the finiteness dependency $\Gamma = \{1, \ldots, k\} \rightsquigarrow \{k+1, \ldots, k+l\}$. We reconsider $\mathcal{A}$ no longer an acceptor but a computational device *with output* instead; $\mathcal{A}$ reads as its input the tuples $\langle u_1, \ldots, u_k \rangle \in db(E)$ from the already computed relation E, and inserts into $db(F)$ all the tuples $\langle u_1, \ldots, u_k, v_1, \ldots, v_l \rangle \in L(\mathcal{A})$, where the strings $v_1, \ldots, v_l$ are generated as outputs.

A given $(k+l)$-FSA $\mathcal{A}$ can be considered a *extended Mealy machine [49, Chapter 2.7] with k inputs and l outputs (a (k,l)-EMM)* by reinterpreting its operational semantics from Section 3.1 as follows.

The configurations of $\mathcal{A}$ on input $\vec{u} = \langle u_1, \ldots, u_k \rangle$ are tuples

$$P = \langle p, m_1, \ldots, m_k, w_1, \ldots, w_l, n_1, \ldots, n_l \rangle ,$$

where:

- p is a state of $\mathcal{A}$;
- each $0 \leq m_i \leq |u_i| + 1$ is the current position of the input tape head i on u_i;
- each w_j is a string from the regular language $\cent\Sigma^*(\$ + \varepsilon)$; and
- each $1 \leq n_j \leq |w_j| + 1$ is the current position of the output tape head j on w_j.

In other words, the w_j added to the previous configuration concept on page 38 represent the already written prefixes of the output tapes $k + 1, \ldots, k + l$.

The configuration

$$Q = \langle q, m_1 + d_1, \ldots, m_k + d_k, w_1 o_1, \ldots, w_l o_l, n_1 + e_1, \ldots, n_l + e_l \rangle$$

is a next configuration of P if and only if $\mathcal{A}$ contains a transition

$$t = p \xrightarrow[d_1,\ldots,d_k,e_1,\ldots,e_l]{u_1[m_1],\ldots,u_k[m_k],c_1,\ldots,c_l} q$$

sstisfying the following conditions for all $1 \leq j \leq l$:

- $w_j[n_j] = c_j$ and $o_j = \varepsilon$, when $n_j \leq |w_j|$; and
- $o_j = c_j$, when $n_j = |w_j| + 1$.

In other words, t applies if and only if its character combination matches the current configuration P for those tapes, whose heads are on known characters; whereas for those output tapes j, whose heads have passed over the right end of w_j, the effect of taking t is to *write* the missing character c_j.

A computation of $\mathcal{A}$ on input $\vec{u}$ is a sequence $\mathcal{C} = C_1 C_2 C_3 \ldots$ of its configurations on $\vec{u}$, in which C_{r+1} is always a next configuration of C_r. $\mathcal{A}$ prints the output $\vec{v} = \langle v_1, \ldots, v_l \rangle$ on input $\vec{u}$ if and only if $\mathcal{A}$ has a computation on $\vec{u}$ from the initial configuration

$$S = \langle s, 0, \ldots, 0, \cent, \ldots, \cent, 1, \ldots, 1 \rangle ,$$

where s is a start state of $\mathcal{A}$, into some final configuration

$$F = \langle f, m_1, \ldots, m_k, w_1, \ldots, w_l, n_1, \ldots, n_l \rangle ,$$

where f is a final state of $\mathcal{A}$, F has no next configuration, and each v_j is obtained from w_j by deleting the leading $¢$ and the (optional) trailing $\$$.

The *evaluation problem* of determining which outputs $\vec{v}$ a given EMM $\mathcal{A}$ prints on input $\vec{u}$ can be considered as searching for the final configurations F from the graph of all possible configurations of $\mathcal{A}$ on $\vec{u}$ with edges denoting computation steps. A depth-first solution is presented as Algorithm 5. We assume throughout this chapter that $\mathcal{A}$ is normalized as in Corollary 3.1; this assumption is natural, since the same normal form was employed in the methods for determining Γ given in Theorems 4.2 and 4.4.

The procedure in Algorithm 5 need obviously not terminate on all $\mathcal{A}$ and $\vec{u}$. However, we are only interested in the case, where $\mathcal{A}$ is known to satisfy the limitation condition Γ. The limitation function $f\colon \mathbb{N}^k \to \mathbb{N}$ associated with $\mathcal{A}$ and Γ from Definition 3.3 has namely the property that once $|w_j| > g = f(|u_1|, \ldots, |u_k|) + 1$ for some w_j in the current configuration P of $\mathcal{A}$ on $\vec{u}$, we know that the current computation cannot possibly accept, because it would then print something longer than the limitation function f allows. Hence it is sufficient to consider only a finite number of configurations for any given input $\vec{u}$, and this information can be used to ensure the termination of the procedure.

For instance, the following strategy suffices. Before entering the loop in Algorithm 5, compute the bound g for the current input $\vec{u}$. Then verify the following two further requirements on the current transition t_r before it can be applied to the current configuration P:

1. the application of t_r must not cause any $|w_j|$ to exceed g; and

2. the application of t_r must not lead into a configuration Q already present in stack T. (Although the configurations in T are represented only implicitly, they can be reconstructed as in the backtracking branch $(*)$ of Algorithm 5 while browsing T.)

The rest of this chapter explores how this strategy could be implemented efficiently. Namely, knowing *how* $\mathcal{A}$ was seen to satisfy Γ lets us optimize the verification of requirements 1 and 2 of the strategy above.

The evaluation problem can naturally be approached with other strategies besides depth-first search of the configuration graph for the EMM $\mathcal{A}$ on input $\vec{u}$. Indeed, a breadth-first strategy has been implemented in the BiO_2 system based on Alignment Calculus [55, Chapter 4.8]. A similar

Algorithm 5 A generic evaluator for EMMs.

```
proc Algorithm5( value   𝒜          : a (k,l)-EMM;
                 value   u_1,...,u_k : strings from Σ*;
                 result  𝒱          : a subset of (Σ*)^l );
     Let m_i = 0 for all i = 1,...,k;
     Let w_j = ¢ and n_j = 1 for all j = 1,...,l;
     Let the transitions of 𝒜 be numbered t_1,...,t_a;
     𝒱,r,T := ∅,1,the empty stack;
     do  stack T is nonempty or r ≤ a  ⟶
           See if transition t_r = g --(a_1,...,a_k,c_1,...,c_l / d_1,...,d_k,e_1,...,e_l)--> h applies to configu-
           ration P = ⟨p, m_1,...,m_k, w_1,...,w_l, n_1,...,n_l⟩, where
               p = { s   T is empty
                   { q   top of T is ⟨r',...⟩ and target state of t_r' is q.
           if   t_r applies to P  ⟶
                   Push to stack T the entry ⟨r, b_1,...,b_l⟩,
                   where each b_j tells if n_j = |w_j| + 1 holds;
                   w_j := w_j c_j for all 1 ≤ j ≤ l with b_j true;
                   m_i := m_i + d_i for all 1 ≤ i ≤ k;
                   n_j := n_j + e_j for all 1 ≤ i ≤ l
            []  t_r does not apply to P, but r < a  ⟶
                   r := r + 1
            []  t_r does not apply to P, and r = a (*)  ⟶
                   if   p is a final state of 𝒜  ⟶
                          𝒱 := 𝒱 ∪ ⟨v'_1,...,v'_l⟩,
                          where each v'_j is w_j without endmarkers
                    []  p is not a final state of 𝒜  ⟶
                          skip
                   fi;
                   Pop the topmost entry ⟨r', b'_1,...,b'_l⟩ from
                   stack T, where t_r' = g' --(a'_1,...,a'_k,c'_1,...,c'_l / d'_1,...,d'_k,e'_1,...,e'_l)--> h';
                   Delete the last character of each w_j for which b'_j;
                   m_i := m_i − d'_i for all 1 ≤ i ≤ k;
                   n_j := n_j − e'_j for all 1 ≤ i ≤ l;
                   r := r' + 1
           fi
     od
end  Algorithm5;
```

strategy has also been implemented for Transducer Datalog [13, Theorem 7]. However, both implementations proved to be slow [60], warranting this study of optimizing depth-first strategies instead.

5.2.1 When the Safety Heuristic Applies

The first case to consider is when the limitation condition Γ could be detected by the heuristic Algorithm 1 on page 76. Lemma 4.2 shows that in every computation $\mathcal{C}$ of $\mathcal{A}$ – whether accepting or rejecting – the lengths of the output buffers w_j are bounded by the lengths of the inputs $\vec{u}$. In other words, requirement 1 of the evaluation strategy outlined in Section 5.2 above is guaranteed *automatically*, and therefore Algorithm 5 does not need refinements in this respect.

Consider then requirement 2. Our aim is to reduce browsing the stack T when looking for a duplicate of the next configuration Q to be added into T. This can be achieved by considering how Algorithm 1 came to its positive conclusion about $\mathcal{A}$ and Γ.

The successful execution of Algorithm 1 on inputs $\mathcal{A}$ and Γ partitions the transitions t of $\mathcal{A}$ into two categories:

1. the transitions that were *deleted* in steps (†) or (‡); and

2. the transitions that were *retained* all through the recursion to step (¶).

Let each t be labelled with the category it was assigned to. We claim the following:

> Let $\mathcal{C} = C_1 \ldots C_n$ be any computation of $\mathcal{A}$, where C_n and C_m are the same configuration for some $1 \leq m < n$. Then the path L of transitions leading from C_m to C_n consists of transitions from category 2.

L is namely a loop in the transition graph of $\mathcal{A}$, because C_m and C_n share the same state. Moreover, since all input tape heads $1 \leq i \leq k$ are in the same position in both C_m and C_n, there is no input tape i that moves in only one direction during L. Therefore none of the transitions in L can be deleted in steps (†) or (‡) of Algorithm 1, verifying our claim.

Our claim implies that *testing requirement 2 can safely stop browsing the stack T as soon as the first transition not belonging to category 2 is met.*

This is already a potent optimisation. If for example $\mathcal{A}$ is unidirectional, then Algorithm 1 applies by Theorem 4.5, and all the transitions of $\mathcal{A}$ belong to category 1 by condition 2 of Corollary 3.1 and condition 1 of Definition 4.3. Hence, Algorithm 5 suffices as is in this case. The class of EMMs with an empty category 2 looks therefore especially amenable to fast evaluation; Example 4.3 shows that this class is not restricted into unidirectional EMMs alone.

We can do even more, however. Because each loop L identified in our claim must meet condition 1 of Definition 4.3, we know that no output tape $1 \leq j \leq l$ moves to direction $+1$ in L. Therefore we also know that if some output tape j moves to direction -1 in some transition t of L, then the configuration C_m cannot repeat below this t in stack T, because at least the output head j cannot return to its former position. Hence category 2 can be strengthened further to consist only of transitions t that take part in a loop L', which moves no output tape and is retained untouched by Algorithm 1. This strengthened category can also be detected by Algorithm 1, if its successful nonrecursive case (¶) deletes from the current C_i all transitions that move some output tape j, if any, and recurses once to delete further transitions.

Indeed, this analysis of $\mathcal{A}$ is useful even when $\mathcal{A}$ is used as a selection condition on already known strings, and not as a generator for string restructuring ($l = 0$), because this labelling of $\mathcal{A}$ identifies when the costly loop detection can be omitted during the subsequent simulation of the FSA $\mathcal{A}$.

5.2.2 When an Input Is Bidirectional

Let us now consider the case, where $\mathcal{A}$ is right-restricted, its bidirectional tape is the first input tape 1, and the limitation property Γ had to be deduced via Theorem 4.2 instead of Theorem 4.4. We build our terminating refinement of Algorithm 5 on Theorem 3.3, which could provide the bounds g if required. However, we can do without explicit bounds g as in Subsection 5.2.1 above.

Recall the proof of Theorem 3.3. Two configurations C_m and C_n of the $(k+l)$-FSA $\mathcal{A}$ on the same input were said to have the same *color* if and only if they shared the same state and the head positions of the input tapes $1, \ldots, k$. Because colors do not depend on the output tapes $k+1, \ldots, k+l$, they carry over to the (k, l)-EMM $\mathcal{A}$ unmodified. Now we claim the following:

> Let $\mathcal{A}$ print the output $\vec{v}$ on input $\vec{u}$. Then $\vec{v}$ can be printed by $\mathcal{A}$ on $\vec{u}$ by a computation $\mathcal{C}$, where no color appears twice.

Consider namely a computation $\mathcal{C}'$ of $\mathcal{A}$ on $\vec{u}$, which prints $\vec{v}$, and contains two configurations C_m and C_n with $m < n$ having the same color. Then the sequence $T_m = t_m \ldots t_{n-1}$ of transitions taken from C_m to reach C_n cannot move any output tape $k+1, \ldots, k+l$, because this would violate Γ

by the same reasoning as in the proof of Theorem 3.3. On the other hand, the subcomputation from C_m to C_n can then be cut from $\mathcal{C}'$ as in Section 4.2 to yield a computation $\mathcal{C}''$ of $\mathcal{A}$ on $\vec{u}$, which still prints $\vec{v}$, but with fewer repetitions of colors, verifying our claim.

Our claim leads to the following refinement of Algorithm 5: *do not push (the transition leading into) a configuration C into stack T, if T already contains a configuration having the same color as C.* Because there are only a finite number of different colors for each input $\vec{u}$, this refinement ensures termination, thereby obviating both requirements 1 and 2 of the strategy in Chapter 5.2.

It only remains to see how far stack T must be browsed in order to find a configuration of the same color, if one exists. This depth can be reduced similarly to Subsection 5.2.1 as follows. Consider a variant of Algorithm 1, which always returns `true` from its nonrecursive case (¶). Applying this variant to $\mathcal{A}$ and Γ partitions the transitions of $\mathcal{A}$ into two categories: the ones that were deleted (category 1), and the ones that were retained (category 2). Again, *browsing stack T can safely stop as soon as the first transition not belonging to category 2 is met*, because the proof of the claim in Subsection 5.2.1 remains valid even when C_m and C_n are only required to have the same color.

5.2.3 When an Output Is Bidirectional

Let us finally consider the remaining case, where $\mathcal{A}$ is right-restricted, its bidirectional tape is the first output tape $k+1$, and the limitation property Γ had to be deduced via Theorem 4.2. Here we do resort to constructing an explicit function to bound the maximum length of the bidirectional output given the lengths of the unidirectional inputs. This function does not, however, bound the remaining unidirectional outputs.

More precisely, we construct a function $\mathcal{W}'_{\mathcal{A}} \colon \mathbb{N}^k \to \mathbb{N} \cup \{-\infty\}$ with the following property: if the (k,l)-EMM $\mathcal{A}$ prints on the input $\vec{u} = \langle u_1, \ldots, u_k \rangle$ the output $\vec{v} = \langle v_1, \ldots, v_l \rangle$, then $|v_1| \leq \mathcal{W}'_{\mathcal{A}}(|u_1|, \ldots, |u_l|)$. This construction is based on Theorem 4.2.

Let us define the *price* of a transition sequence

$$T = p_0 \xrightarrow[d_1^b,\ldots,d_k^b,e_1^b,\ldots,e_l^b]{a_1^1,\ldots,a_k^1,c_1^1,\ldots,c_l^1} p_1 \cdots p_{b-1} \xrightarrow[d_1^b,\ldots,d_k^b,e_1^b,\ldots,e_l^b]{a_1^b,\ldots,a_k^b,c_1^b,\ldots,c_l^b} p_b$$

to be $\mathrm{price}(T) = \langle D_1, \ldots, D_k \rangle \in \mathbb{N}^k$, where each $D_i = \sum_{j=1}^{b} d_i^j$. In other words, $\mathrm{price}(T)$ tells the input consumption of T as the number D_i of times the (unidirectional) input tape head i is moved (to direction $+1$) during T.

These prices can be partially ordered with $\langle D_1,\ldots,D_k\rangle \preccurlyeq \langle D'_1,\ldots,D'_k\rangle \equiv \bigwedge_{i=1}^{k} D_i \leq D'_i$.

Define further price($\mathcal{C}$) for a computation $\mathcal{C}$ to be price($T_{\mathcal{C}}$) for the sequence $T_{\mathcal{C}}$ of transitions traversed in $\mathcal{C}$. We can rephrase $\mathcal{W}'_{\mathcal{A}}(m_1,\ldots,m_k)$ as the maximum length of output $\mathcal{A}$ prints onto the bidirectional tape during any computation $\mathcal{C}$ with price($\mathcal{C}$) $\preccurlyeq \langle m_1+1,\ldots,m_k+1\rangle$. Let us now show that this maximum is reached by some $(0,0)$-direct computation of $\mathcal{A}$; Definition 4.2 carries easily over from FSAs to EMMs. Let the computation $\mathcal{C}'$ namely reach this maximum without being $(0,0)$-direct. Then we can cut a portion $\mathcal{P}$ from $\mathcal{C}'$ to get another accepting computation $\mathcal{C}''$, as in Section 4.2. Because $T_{\mathcal{C}''}$ is the result of deleting a contiguous subsequence of $T_{\mathcal{C}'}$, price($\mathcal{C}''$) $\preccurlyeq$ price($\mathcal{C}'$). $\mathcal{C}''$ does still print the same output on the bidirectional tape as $\mathcal{C}'$, because otherwise the portion $\mathcal{P}$ contained all the visits $\mathcal{C}'$ made onto the \$ of the bidirectional output tape, and in this case the existence of $\mathcal{C}''$ would contradict the limitation property Γ by Subcase (1a) in the proof of Theorem 4.2. Repeating this cutting argument we finally find a $(0,0)$-direct computation reaching this maximum, as promised.

This reasoning leads to $\mathcal{W}'_{\mathcal{A}}(m_1,\ldots,m_k) = \mathcal{T}(m_1+1,\ldots,m_k+1)-2$, where $\mathcal{T}(n_1,\ldots,n_k)$ yields the maximum number of blocks (that is, block terminators $\sharp$) in any transition word $\mathcal{E}$ accepted by twa $(\mathcal{A},0,0)$ satisfying price($\mathcal{E}$) $\preccurlyeq \langle n_1,\ldots,n_k\rangle$. (The price of a transition word is naturally the price of the transition sequence it represents, while twa $(\mathcal{A},0,0)$ is provided by Lemma 4.1.) This $\mathcal{T}\colon \mathbb{N}^k \to \mathbb{N}\cup\{-\infty\}$ can in turn be obtained via the following labelling process.

- If f is a final state of twa $(\mathcal{A},0,0)$, then f has no outgoing transitions, because there are no blocks after the one corresponding to \$. Therefore it suffices to define $\mathcal{T}_f(n_1,\ldots,n_k)=0$.

- If p is a nonfinal state of twa $(\mathcal{A},0,0)$, then define $\mathcal{T}_p(n_1,\ldots,n_k)$ to be the maximum of all $\mathcal{T}_e(n_1,\ldots,n_k)$, where e is any transition leaving p.

- If e is a transition $p \xrightarrow{\sharp} q$ of twa $(\mathcal{A},0,0)$, then define $\mathcal{T}_e(n_1,\ldots,n_k) = \mathcal{T}_q(n_1,\ldots,n_k)+1$.

- If e is a transition $p \xrightarrow{t} q$ of twa $(\mathcal{A},0,0)$, then define

$$\mathcal{T}_e(n_1,\ldots,n_k) = \begin{cases} \mathcal{T}_q(n_1-D_1,\ldots,n_k-D_k) & \bigwedge_{i=1}^{k}(n_i \geq D_i) \\ -\infty & \text{otherwise,} \end{cases}$$

  where price(t) $= \langle D_1,\ldots,D_k\rangle$.

- Finally, $\mathcal{T}(n_1, \ldots, n_k) = \mathcal{T}_s(n_1, \ldots, n_k)$, where s is the start state of twa $(\mathcal{A}, 0, 0)$.

This labelling does indeed yield a function: if some value $\mathcal{T}_p(n_1, \ldots, n_k)$ referred back to itself, this would imply a loop $\mathcal{L}$ in the twa $(\mathcal{A}, 0, 0)$ that reads a crossing subword $\mathcal{E}'$ with price$(\mathcal{E}') = \langle 0, \ldots, 0\rangle$. This $\mathcal{E}'$ must contain $\sharp$ by 1-stability, and hence the existence of $\mathcal{L}$ violates the limitation property Γ by Subcase (1b) in the proof of Theorem 4.2.

Example 5.3 Consider the 2-FSA $\mathcal{A}_{\text{reverse}}$ from Example 4.1. Example 4.2 showed that $\mathcal{A}_{\text{reverse}}$ satisfies the limitation condition $\Gamma_2 = \{2\} \rightsquigarrow \{1\}$, while Example 4.3 showed that determining this is beyond the heuristic Algorithm 1.

Let us therefore apply the construction above. Labelling the finite automaton $\mathcal{A}'$ given in Example 4.2 yields the following recurrence.

$$\begin{aligned}
\mathcal{W}'_{\mathcal{A}_{\text{reverse}}}(m) &= \mathcal{T}_{\text{i}}(m+1) - 2. \\
\mathcal{T}_{\text{i}}(n) &= \mathcal{T}_{\text{iv}}(n) + 1. \\
\mathcal{T}_{\text{iv}}(n) &= \max(\mathcal{T}_{\text{v}_x}(n), \mathcal{T}_{t_3}(n)). \\
\mathcal{T}_{t_3}(n) &= \begin{cases} 1 & n \geq 1 \\ -\infty & n = 0. \end{cases} \\
\mathcal{T}_{\text{v}_x}(n) &= \begin{cases} \mathcal{T}_{\text{iv}}(n-1) + 1 & n \geq 1 \\ -\infty & n = 0. \end{cases}
\end{aligned}$$

The solution to this recurrence is

$$\mathcal{W}'_{\mathcal{A}_{\text{reverse}}}(m) = m,$$

and indeed, the reversal of a string is no longer than the string itself. ■

The function $\mathcal{W}'_{\mathcal{A}}$ yields a terminating refinement of Algorithm 5 as follows. Determine first how $\mathcal{A}$ satisfies the derived finiteness dependency $\Gamma' = \{1, \ldots, k+1\} \rightsquigarrow \{k+2, \ldots, k+l\}$, and choose either the refinements of Subsection 5.2.1 or 5.2.3 as the strategy to limit the unidirectional outputs not bounded by $\mathcal{W}'_{\mathcal{A}}$. Note that this choice needs to be made only once, during the translation of the query. Compute on input $\vec{u}$ first the bound $b = \mathcal{W}_{\mathcal{A}}(|u_1|, \ldots, |u_k|)$, and perform the simulation of $\mathcal{A}$ on $\vec{u}$ as dictated by the compile-time choice above, with the additional condition that a configuration is not added to stack T if its bidirectional output tape has more than b characters from Σ.

Note that it suffices to compute the bound $b = \mathcal{W}'_{\mathcal{A}}(M_1, \ldots, M_k)$, where each M_i is the maximum length of any string in column i of the current input relation $db(E)$, and use this same b for all $\vec{u} \in db(E)$. In addition, a similar labelling as above could also produce a unidirectional $(k+1)$-FSA $\mathcal{B}$, for which $\langle u_1, \ldots, u_k, v_1 \rangle \in L(\mathcal{B})$ if and only if $\mathcal{A}$ prints v_1 in some computation $\mathcal{C}$ with price$(\mathcal{C}) \preccurlyeq \langle |u_1| + 1, \ldots, |u_k| + 1 \rangle$. Hence the original generator F could equivalently have been factored into $F' = \sigma_{\mathcal{A}}(\sigma_{\mathcal{B}}(E \times \Sigma^*) \times (\Sigma^*)^{l-1})$ for evaluation. Such factorings of transducers have been studied independently by Weber [101].

Finally, obtaining an exact function in Example 5.3 was no accident: if $\mathcal{W}'_{\mathcal{A}}(m_1, \ldots, m_k) = b \neq -\infty$, then the construction implies that $\mathcal{A}$ does indeed have an accepting computation $\mathcal{C}$ with price$(\mathcal{C}) \preccurlyeq \langle m_1 + 1, \ldots, m_k + 1 \rangle$ that prints a bidirectional output of length exactly b.

5.3 On the Cost of Generator Evaluation

Let us conclude this chapter on evaluating Alignment Calculus queries by considering briefly the cost of evaluating the aforementioned generators, based on already known complexity results from automata theory.

A right-restricted $(1,1)$-EMM $\mathcal{B}$ with its output tape bidirectional can be viewed alternatively as a *checking stack automaton* $\mathcal{D}$ [43, Definition 1.4]: a device, which accepts its unidirectional input string on tape 1 by first guessing the contents for tape 2, and then verifying this guess bidirectionally without altering it. This connection between right-restricted $(1,1)$-EMMs, or two-way finite-state transducers, and checking stack automata has already been noted by Rajlich [67, Theorem 5.1].

Simulating $\mathcal{B}$ on a given u amounts to printing all the stacks v that $\mathcal{D}$ finds acceptable on input u. On the other hand, there are such $\mathcal{D}$ that even determining whether any suitable v exist for a given u is already **NP**-complete [37, Problem AL5]. Although it remains to be seen whether the extra condition $\{1\} \rightsquigarrow \{2\}$ imposed on $\mathcal{B}$ by finite evaluability can ameliorate the situation, it does seem prudent to avoid this type of string restructuring if at all possible.

Therefore we arrive at the following rough guidelines for choosing a string restructuring method in Algorithm 2, if such a method must be chosen.

1. Try first to choose a generator that was considered safe by the heuristic Algorithm 1.

   Verifying the corresponding limitation condition is cheaper than applying Theorem 4.2, and the evaluation method developed in Subsection 5.2.1 can be used, along with its termination detection optimizations.

2. Failing that, try to apply Theorem 4.2 while keeping the output unidirectional.

   Verifying the corresponding limitation condition is now expensive. The evaluation method developed in Subsection 5.2.2 does, however, seem superior to the one developed in Subsection 5.2.3, because the latter uses far less information on the actual current inputs than the former, namely just their lengths instead of their contents.

3. Apply the construction of Subsection 5.2.3 only as a last resort: both limitation condition verification and generator evaluation seem to be costly.

These rules do not take into account the *sizes* of the input and output relations: a better overall result might be obtained by choosing a seemingly more expensive restructuring operation, if this operation rejects more input and produces less output. Limitation functions might provide one such measure with which to compare different query evaluation strategies.

Chapter 6

Expressive Power

Praised be the infinite
Mesh of effects and causes
Which, before it shews me the mirror
In which I shall see no-one or I shall see another,
Grants me now this contemplation pure
Of a language of the dawn.

Jorge Luis Borges [15]

In this chapter we study the definitional power of pure Alignment Calculus. That is, we wish to characterize the kinds of string relations definable with (different sublanguages of) Alignment Calculus. Following Vardi [98], we discuss two distinct measures of expressivity, *data* and *expression* complexity.

However, our aim differs slightly from Vardi's. He namely asks what properties of *relations* are expressible in a given query language. A well-known problem in this field is to ask, "Can I write a query to find out whether a particular relation has an even or odd number of members?" [2, Proposition 17.3.2]. On the other hand, our data items add an internal structure absent from the original relational model, and it is therefore interesting to see what kinds of properties of this *structure* can be expressed using Alignment Calculus. This is why we focus here on *pure* Alignment Calculus and the expressivity and complexity of its string manipulation component only, leaving aside the expressive power of the relational component of Alignment Calculus, or the joint effects of these two components. It is not immediately clear at the outset whether adding new operations, as we have done here, preserves the laws of the unmodified relational model [10], and therefore the analysis of these additional operations is not only interesting on its own but is also a prerequisite for the further analysis of the whole model. Accordingly, we define our expressivity measures as follows.

Data complexity denotes the complexity of the decision problem "is $w \in [\![\phi]\!]$?" measured as a function of the length $|w|$ of the input string (or more generally string tuple) w. That is, we study the complexity of the language described by the formula ϕ, or more generally, a syntactic class Φ of formulae ϕ. Results of this kind are often called *descriptive complexity theory* [27, Chapters 5 and 6].

This corresponds to asking "what kinds of string relations can I construct from my database with queries from Φ?"

Expression complexity denotes the complexity of the same decision problem, now measured as a function of the length $||\phi||$ of the formula ϕ. That is, we study how concisely the class Φ can express these relations.

This in turn corresponds to asking "if I issue a query from Φ, how long does the translation of my query take?"

One could also discuss *combined* complexity by considering both $|w|$ and $||\phi||$ at the same time, but we do not pursue this goal here. This analysis would correspond to asking "how long will the overall processing of a given query take per tuple?"

If we wish to broaden our vision to consider all of Alignment Calculus, we must account for the interplay of its relational and string-oriented components, as noted above. Then a more complex question arises: "if I *encode* the contents of my database into a string, then manipulate this string in some way, and finally *decode* the result back into my database, what kinds of transformations can I then perform?" In a sense, this models the 'unload/reload' utilities found in current database products [24, Chapter 2.11]. We then enter the field of *generic computation* [3, 4, 5, 6][27, Chapter 6.5], which aims to delineate the interplay of a computational device and a relational store. But as stated above, we do not pursue this goal here.

6.1 Data Complexity

We say that the pure Alignment Calculus formula ϕ with one free variable *defines* the language $[\![\phi]\!]$. Furthermore, we say that a class Φ of formulae *represents* a class C of formal languages if and only if $C = \{[\![\phi]\!] : \phi \in \Phi\}$. That is, every $\phi \in \Phi$ defines a language in C, and conversely every language $L \in C$ can be so defined.

Theorem 6.1 *The unidirectional unquantified string formulae of one variable represent exactly the regular languages.*

Proof: Any regular expression A can be rewritten as formula $\phi_A.[\,]_l x_1 = \bot$, where ϕ_A is obtained from A by replacing every occurrence of $c \in \Sigma$ with $[x_1]_l x_1 = c$. Now the set $L(\phi_A)$ of formula words defined by ϕ_A is $\{[x_1]_l x_1 = c_1 \ldots [x_1]_l x_1 = c_n : c_1 \ldots c_n \in L(A)\}$, and therefore a formula word in $L(\phi_A.[\,]_l x_1 = \bot)$ is true in initial alignment A_0 under assignment θ exactly when $\sigma_{A_0}(\theta x_1) \in L(A)$.

Conversely, given any formula ϕ of this kind, Theorem 3.1 yields the corresponding 1-FSA $\mathcal{A}_\phi$, and this is a nondeterministic finite automaton with endmarkers. Hence, the language it accepts is regular. $\square$

Since moving the only tape back and forth does not increase expressivity (as proved implicitly in Lemma 4.1) and since the regular languages are closed under conjunction and intersection, Theorem 6.1 readily generalizes to all unquantified formulae of one variable.

Let us then move into the other end of the expressivity spectrum.

Theorem 6.2 *The formulae $\psi \equiv \exists x_2, x_3.\phi$, where ϕ is a string formula with an unidirectional variable x_1 and two bidirectional variables x_2, x_3, represent exactly the* recursively enumerable *languages* **RE**.

Proof: Consider an arbitrary ψ of this form. Theorem 3.1 provides a 3-FSA $\mathcal{A}_\phi$ corresponding to ϕ, and the problem "$\langle u, v, w\rangle \in L(\mathcal{A}_\phi) = [\![\phi]\!]$?" can be decided by straightforward simulation. Then the original problem "$u \in [\![\psi]\!]$?" can be decided by the following algorithm: given u as input, first guess nondeterministically v and w, and then verify the guess via the simulation above. Hence, $[\![\psi]\!] \in$ **RE**.

Consider conversely an arbitrary $L \in$ **RE**. Then there exists a Turing machine M that accepts L. Without loss of generality assume that M accepts by entering a unique state f. Let ϕ_M be the corresponding string formula constructed in the proof of Theorem 4.1 and modified to accept only encodings of computations that end in this state f. Hence, choosing $\phi \equiv \exists x_2, x_3.\phi_M$ yields $[\![\phi]\!] = L$. $\square$

Theorem 6.2 implies that the abstract definitional power of pure Alignment Calculus equals the *Arithmetical Hierarchy* [72, Chapter 14]. Note that although string concatenation would have sufficed already [66], Alignment Calculus admits in contrast even a representation, where the quantifier alternation structures of the corresponding formulae match. (Naturally, Theorem 6.2 also implies via diagonalization that pure Alignment Calculus cannot possess any sound, complete, and effective proof system [94]. Hence, it was indeed prudent to exclude queries like Eq. (3.1) on page 47 from consideration.)

In fact, even a stronger version of Theorem 6.2 can be given.

Corollary 6.1 *The formulae* $\exists x_2, x_3.\phi \wedge \psi$, *where* ϕ *and* ψ *are unidirectional string formulae on variables* x_1, x_2, x_3 *represent exactly the recursively enumerable languages* **RE**. *In fact, it even suffices to choose a* ψ *without* x_1.

Proof: The representation of **RE** given by Theorem 6.2 uses the string formulae $\phi_M = \phi_{\text{guess}}.\phi_{\text{rewind}}.\phi_{\text{check}}$ constructed in the proof of Theorem 4.1 (with ϕ_{guess} slightly modified). These can in turn be written equivalently as the formulae $\phi_{\text{guess}} \wedge \phi_{\text{check}}$ of the desired form, because rewinding strings x_2 and x_3 (as well as $x_1 \notin \text{free}(\phi_{\text{check}})$) back into their initial positions can alternatively be accomplished with a conjunction, as pointed out in Example 2.9 of Section 2.3. $\square$

Corollary 6.1 justifies the conjunctivity restriction from Subsection 4.4.1: $\text{basis}(\phi) = \text{basis}(\psi) = \emptyset$, but their conjunction $\phi \wedge \psi$ satisfies the nontrivial finiteness dependency $\{x_1\} \rightsquigarrow \{x_2, x_3\}$ if and only if the language being represented is total. Therefore it is in general undecidable to determine all the finiteness dependencies satisfied by the conjunction $\phi \wedge \psi$, even if the dependencies satisfied by the conjuncts ϕ and ψ are easily obtainable.

A different approach to the computational aspects of sequence queries was taken by Transducer Datalog discussed briefly in Subsection 1.2.2. In this proposal, computational concepts pervade all levels of the language semantics: the string manipulation concept is automata with output, while the relation manipulation concept is recursive logical rules. In contrast, computation in Alignment Calculus arises only as a consequence of a declarative way of specifying constraints between strings. In particular, the concept of transduction arises only at the query evaluation level, as discussed in Section 5.2.

The following result follows from the fact that extended temporal logic equals the regular expressions in their expressive power [29, Theorem 6.5 and Chapter 6.6.1].

Theorem 6.3 *Every set definable in extended temporal logic is definable in pure Alignment Calculus. There are sets definable in pure Alignment Calculus that are not definable in extended temporal logic.*

Proof: Theorem 6.1 established the former, Theorem 6.2 the latter claim. $\square$

Let us now compare string formulae and the sequence predicates of Ginsburg and Wang's sequence logic, which was introduced in Subsection 1.2.2. Because sequence logic employs an infinite alphabet $\mathcal{U}$ of atoms, we postulate a translation injection $e\colon \mathcal{U} \to \Sigma^*$ mapping atoms from $\mathcal{U}$ to strings. Intuitively, this e converts each atom a into its unique name $e(a)$. Given also a new terminator symbol $\sharp$ never output by e, we can extend e to sequences $[a_1, \ldots, a_m]$ of atoms $a_i \in \mathcal{U}$ with $e([a_1, \ldots, a_m]) = e(a_1)\sharp \ldots \sharp e(a_m)\sharp$. With these tools we obtain the following result:

Theorem 6.4 *For each sequence predicate $P \equiv x_{n+1} \in [\![A]\!]_n(x_1, \ldots, x_n)$ of sequence logic, where the variables $x_1, \ldots, x_{n+1}$ are distinct, there exists a unidirectional string formula ϕ_P on the same variables $x_1, \ldots, x_{n+1}$ having the following property: P is true in the assignment mapping x_i to s_i for $i = 1, \ldots, n+1$ if and only if $\langle e(s_1), \ldots, e(s_{n+1})\rangle \in [\![\phi_P]\!]$.*

Proof: Thinking operationally, we build the string formula ϕ_A by replacing each α_i in A with the subformula

$$([x_j, x_{n+1}]_l x_{n+1} = x_j \neq \sharp)^*.[x_j, x_{n+1}]_l x_{n+1} = x_j = \sharp,$$

which implements the copying of the next translated atom from channel i to channel $n+1$. The final ϕ_P is then $\phi_A.[x_1, \ldots, x_{n+1}]_l x_1 = \ldots = x_{n+1} = \perp$. $\square$

The full calculus of Ginsburg and Wang allows sequence constants and repeating variables among the arguments $x_1, \ldots, x_{n+1}$. Repetitions can be eliminated by rewriting each $\phi(\ldots, x_i, \ldots, x_i, \ldots)$ into $\exists y_i.x_i =_s y_i \wedge \phi(\ldots, x_i, \ldots, y_i, \ldots)$ before applying Theorem 6.4, although the result is no longer just a string formula. Similarly, sequence constants c can be eliminated by rewriting $\phi(\ldots, \mathsf{c}, \ldots)$ as $\exists y_\mathsf{c}.\phi_\mathsf{c} \wedge \phi(\ldots, y_\mathsf{c}, \ldots)$, where the unidirectional string formula ϕ_c is true if and only if y_c is $e(\mathsf{c})$; the (effective) existence of these ϕ_c is guaranteed by Theorem 6.1.

Alignment Calculus and sequence logic are clearly related: both add to the relational calculus a string (or sequence) manipulation concept, which is based on the notion of regular expressions. However, these regular expressions denote different things: the string formulae specify regular character-to-character comparisons, while sequence predicates specify regular input-to-output atom transfers. Theorem 6.4 shows how the latter concept can be expressed within the former, but the converse direction is not as obvious, because string formulae allow the same character to take part in multiple comparisons, and even allows returning to compare the same character again by bidirectionality. Transducer Datalog discussed in Subsection 1.2.2 offers these rescanning possibilities by introducing the ability to call another transducer as a subroutine during the execution of the main transducer. This introduction could then account for repeatedly testing a character, because the called subtransducer starts reading the inputs (and current output) of the main transducer again at the beginning. However, these introductions give the language semantics a more procedural flavour. Repeated processing of the same data item is possible also in the functional list language $\mathcal{L}_p$ [19], and the algebraic pomset language [45].

Let us now study the expressive power of the bidirectionally productive Alignment Calculus formulae developed in Subsection 4.4.3. This syntactically safe subclass of ours will be shown to have intimate connections with the *Polynomial-time Hierarchy* **PH** [87]. In particular, the classification of some Alignment Calculus formulae into types Σ_k and Π_k given in Definition 4.4 will be related to the levels Σ_k^p and Π_k^p of the hierarchy. The development is via a sequence of three technical lemmas.

The essence of the first lemma is that the relations specified via classified formulae are in their proper places within the hierarchy. This result can be viewed as adding right-restriction into Lemma 4.5.

Lemma 6.1 *Let ϕ be a pure subformula of a QDNF formula satisfying the following properties:*

- *the type given to ϕ by Definition 4.4 is Σ_k (or Π_k, respectively);*
- required (ϕ) *is both defined and satisfied by Definition 4.5; and*
- *this satisfaction can be verified according to the rules of right-restricted productivity.*

Then $[\![\phi]\!] \in \Sigma_k^p$ (or Π_k^p, respectively).

Proof: We proceed by induction on the structure of ϕ. Note that ϕ might have more than one free variable; therefore we make the natural extra assumption that string tuples $\vec{w} = \langle w_1, \ldots, w_m \rangle$ are represented so that they can be manipulated in polynomial time with respect to the quantity $|\vec{w}| = |w_1| + \cdots + |w_m|$ denoting the size of the tuple contents.

Let ϕ first be as in Eq. (4.8). The subformula $\alpha_\phi \wedge \beta_\phi$ has type Π_0, and by purity contains only string formulae. By Cases (1) and (2) of Lemma 2.2 on page 31, and the closure properties of the $\Pi_0^p = \mathbf{P}$ (the class of problems solvable in polynomial time [8, Definition 3.1]), it suffices to show that the membership problem "$\vec{w} \in [\![\psi]\!]$?" is polynomial with respect to $|\vec{w}|$, for any fixed string formula ψ. It is straightforward to perform this membership test in nondeterministic logarithmic space $\mathbf{NLOG} \subseteq \mathbf{P}$ [8, Definition 3.1 and Proposition 3.1 (c)] by guessing the next transition to take and storing only the positions of the tape heads. If $r = 0$, this concludes the base case of the induction.

If on the other hand $r > 0$, then let the type of each subformula D_i be Σ_{k_i}, for $1 \leq i \leq r$. By induction, the membership problem "$\vec{w} \in [\![D_i]\!]$?" belongs to $\Sigma_{k_i}^p$, when measured with respect to $|\vec{w}|$. By Case (1) of Lemma 2.2, the membership problem "$\vec{w} \in [\![\neg D_i]\!]$?" then belongs to $\Pi^p_{k_i}$. By Case (2) of Lemma 2.2, and the closure properties of Π^p_k, where $k = \max(0, k_1, \ldots, k_r)$, the membership problem "$\vec{w} \in [\![\phi]\!]$?" belongs to Π^p_k. In addition, the type of ϕ is indeed Π_k.

If $\phi \equiv \exists z \psi$, and the type of ψ is Σ_k, then the type of ϕ is also Σ_k. Let $\mathrm{pos}(\mathrm{free}(\psi), z) = 1$ for notational convenience. Then by Case (3) of Lemma 2.2 $\vec{w} = \langle w_2, \ldots, w_n \rangle \in [\![\phi]\!]$ if and only if there exists some $w_1 \in \Sigma^*$, for which $\vec{w}' = \langle w_1, \ldots, w_n \rangle \in [\![\phi]\!]$. By induction, the latter membership test belongs to Σ^p_k, when considered with respect to $|\vec{w}'|$. We must show that it suffices to guess some w_1 of polynomial length with respect to $|\vec{w}|$; then the former test can also be carried out within Σ^p_k. Lemma 4.5 provides a function $\mathcal{W}_\phi$ with the following properties:

- Because ϕ is pure, $\mathcal{W}_\phi$ does not refer to the database. That is, the function $\max(R, db)$ of Eq. (3.3) does not occur. Hence, the function has signature $\mathcal{W}_\phi \colon \mathbb{N} \to \mathbb{N}$.

- It suffices to restrict attention to w_1 satisfying

  $$|w_1| \leq \mathcal{W}_\phi \left(\max \left\{ \left| w_{\mathrm{pos}(y, \mathrm{free}(\psi))} \right| : y \in X \right\} \right),$$

  where X is the set on the left side of required (ϕ).

- By right-restricted productivity and Theorem 4.2, all the limitation functions $\mathcal{W}_{a_i}$ (from Definition 3.3) that were used in the construction of $\mathcal{W}_\phi$, are polynomial. The rest of the construction then combines these functions, and selects their maxima. Hence, $\mathcal{W}_\phi$ is indeed polynomial.

Therefore we can indeed restrict attention to polynomially long w_1 with respect to $|\vec{w}|$.

If finally $\phi \equiv \exists z \psi$, but ϕ has type Π_{k-1} instead, then the reasoning above still applies. Now, however, the latter test belongs to Π^p_{k-1}, raising the former one into Σ^p_k. $\square$

The second lemma shows how right-restricted formulae can act as polynomial clocks.

Lemma 6.2 *Let $p \colon \mathbb{N} \to \mathbb{N}$ be a polynomial. There exists a formula δ_p satisfying the following properties:*

- *$|w_1| \leq p(|w_0|)$ implies $\langle w_0, w_1 \rangle \in [\![\delta_p]\!]$;*

- *δ_p is a pure Alignment Algebra formula with type Π_0 or Σ_1;*

- *δ_p satisfies the finiteness dependency $\{x_0\} \rightsquigarrow \{x_1\}$; and*
- *this satisfaction can be verified according to the rules of right-restricted productivity.*

  (In fact, this δ_p can even be chosen right-restricted.)

Proof: Let us without loss of generality assume that p is represented as an expression tree, whose leaves are labelled with constants $m \in \mathbb{N}$ or occurrences of the parameter n, and whose interior nodes are labelled with the binary operators '+' for addition and '·' for multiplication. Denote as $|p|$ the number of nodes in this tree. It suffices to show that we can meet the first of the four requirements by choosing

$$\delta_p \equiv \exists x_2 \ldots \exists x_{|p|}. \bigwedge_{i=1}^{|p|} a_i, \tag{6.1}$$

where each string formula a_i is right-restricted and $\bigcup_{i=1}^{|p|} \operatorname{basis}(a_i) \vdash \{x_0\} \rightsquigarrow \left\{x_1, \ldots, x_{|p|}\right\}$. We proceed by induction on the structure of this tree.

For a constant $m \in \mathbb{N}$ we choose the string formula

$$\delta_m \equiv ([x_1]_l \top)^m.([x_1]_l x_0 = x_1 = \bot).$$

This choice meets the first requirement since $[\![\delta_m]\!] = \{\langle w_0, w_1\rangle : |w_1| \leq m\}$. It also fits Eq. (6.1), because $\{x_0\} \rightsquigarrow \{x_1\} \in \operatorname{basis}(\delta_m)$ and δ_m is unidirectional.

For the parameter n we choose instead the string formula

$$\delta_n \equiv ([x_0, x_1]_l x_0 \neq \bot)^*.([x_1]_l x_1 = \bot),$$

for which $[\![\delta_n]\!] = \{\langle w_0, w_1\rangle : |w_1| \leq |w_0|\}$. The correctness of this choice can be established as in the previous case.

For an addition $q + r$ we proceed as follows. Consider the inductively obtained formula δ_q after the renaming of variable x_k as x_{k+1} for all $1 \leq k \leq |p_1|$. Following Eq. (6.1), it is of the form

$$\exists x_3 \ldots \exists x_{|q|+1}. \bigwedge_{i=1}^{|q|} a_i,$$

and $\bigcup_{i=1}^{|q|} \operatorname{basis}(a_i) \vdash \{x_0\} \rightsquigarrow \left\{x_2, \ldots, x_{|q|+1}\right\}$. Similarly, renaming in δ_r of x_k as $x_{k+|q|+1}$ for all $1 \leq k \leq |r|$ yields the formula

$$\exists x_{|q|+3} \ldots \exists x_{|q|+|r|+1}. \bigwedge_{j=1}^{|r|} b_j,$$

and $\bigcup_{j=1}^{|r|} \text{basis}\,(b_j) \vdash \{x_0\} \rightsquigarrow \left\{x_{|q|+2}, \ldots, x_{|q|+|r|+1}\right\}$. Consider finally the string formula

$$\eta \equiv ([x_1, x_2]_l x_2 \neq \bot)^*.([x_1, x_{|q|+2}]_l x_{|q|+2} \neq \bot)^*.([x_1]_l x_1 = \bot),$$

for which $[\![\eta]\!] = \{\langle w_1, w_2, w_3\rangle : |w_1| \leq |w_2| + |w_3|\}$. η is unidirectional and $\left\{x_2, x_{|q|+2}\right\} \rightsquigarrow \{x_1\} \in \text{basis}\,(\eta)$. Hence it suffices to choose

$$\delta_{q+r} \equiv \exists x_2 \ldots \exists x_{|q|+|r|+1}. \left(\bigwedge_{i=1}^{|q|} a_i\right) \wedge \left(\bigwedge_{j=1}^{|r|} b_j\right) \wedge \eta.$$

Finally for the product $q \cdot r$ we proceed as in the previous case, except that now we choose instead the string formula

$$\begin{aligned}\eta \equiv (&([x_{|q|+2}]_l x_{|q|+2} \neq \bot).([x_1, x_2]_l x_2 \neq \bot)^*.\\ &([x_2]_l x_2 = \bot).\\ &([x_2]_r \top)^*.([x_2]_r x_2 = \bot))^*.([x_1]_l x_1 = \bot),\end{aligned} \tag{6.2}$$

for which $[\![\eta]\!] = \{\langle w_1, w_2, w_3\rangle : |w_1| \leq |w_2| \cdot |w_3|\}$. Note that although η is no longer unidirectional, it still remains right-restricted. $\square$

The third and final lemma shows how alternating polynomial time computations can be expressed in a restricted manner.

Lemma 6.3 *Let $A \subseteq \Sigma^*$ be any language in Σ_k^p, where $k > 0$. There exists a formula ϕ satisfying the following properties:*

- *ϕ is a pure Alignment Calculus formula with type Σ_k;*
- *$[\![\phi]\!] = A$;*
- *ϕ satisfies the finiteness dependency $\{x_0\} \rightsquigarrow \emptyset$ by Definition 4.5; and*
- *this satisfaction can be verified according to the rules of right-restricted productivity.*

  (In fact, this ϕ can even be chosen right-restricted.)

Proof: Every $A \in \Sigma_k$ has the following representation [8, Theorem 8.3]: there exists some $B \in \mathbf{P}$ and some polynomial $p \colon \mathbb{N} \to \mathbb{N}$, for which $w_0 \in A$ if and only if the statement $Q_0(w_0)$ holds, where

- the statement $Q_k(w_0, \ldots, w_k)$ is "$\langle w_0, \ldots, w_k\rangle \in B$"; and

- each statement $Q_i(w_0, \ldots, w_i)$, where $0 \leq i < k$, is "there exists some $v \in \Sigma^*$, for which $|v| \leq p(|w_0|)$ and the statement $Q_{i+1}(w_0, \ldots, w_i, v)$ *fails*".

Our goal is to express this statement $Q_0(w_0)$ with the formula ϕ_0. More specifically, we construct the formulae ϕ_j, $0 \leq j < k$, for which the following five requirements are met:

1. ϕ_j is a pure Alignment Calculus formula with type Σ_{k-j};
2. $\text{free}(\phi_j) = \{x_0, \ldots, x_j\}$;
3. $\langle w_0, \ldots, w_j \rangle \in [\![\phi]\!]$ if and only if the statement $Q_j(w_0, \ldots, w_j)$ holds;
4. ϕ satisfies the limitation condition $\text{free}(\phi_j) \rightsquigarrow \emptyset$; and
5. this satisfaction can be verified according to the rules of right-restricted productivity.

The formula ϕ_{k-1} is obtained as follows. (We omit ϕ_k, and accordingly state this lemma only for $k > 0$, because the construction introduces new quantifiers, and there are none in Q_0 originally.) Let M be a Turing machine that accepts the *complement* $\overline{B}$ of B in polynomial time with respect to the size of its input $\langle w_0, \ldots, w_k \rangle$; recall that $\mathbf{P}$ is closed under complement [8, Proposition 3.1 (a)]. Without loss of generality we further assume this M to have $k+1$ distinct unidirectional read-only input tapes with endmarkers, and one work tape, whose alphabet is also Σ^*. Then by a straightforward generalization of Theorem 6.2, there is a string formula β with $\text{free}(\beta) = \{x_0, \ldots, x_{k+2}\}$, for which $\overline{B} = [\![\exists x_{k+1} \exists x_{k+2} \beta]\!]$. However, even though β does satisfy the required limitation condition $\{x_0, \ldots, x_k\} \rightsquigarrow \{x_{k+1}, x_{k+2}\}$, β is not right-restricted. (Moreover, applying Corollary 6.1 instead would not help either, since this would violate the conjunctivity restriction in turn.) Nevertheless, a closer review of the construction in Theorem 4.1 reveals that x_{k+1} and x_{k+2} encode the sequence of *computation steps* taken by M on the current input $\langle w_0, \ldots, w_k \rangle$. Therefore there is a polynomial $p_M \colon \mathbb{N}^{k+1} \to \mathbb{N}$, for which $\langle w_0, \ldots, w_{k+2} \rangle \in [\![\beta]\!]$ implies $|w_{k+1}|, |w_{k+2}| \leq p_M(|w_0|, \ldots, |w_k|)$; this p_M is essentially the product of the (polynomial) time and space complexities of M, and the extra costs incurred during the encoding. Recalling that $|w_1|, \ldots, |w_k| \leq p(|w_0|)$, we furthermore see that there exists some polynomial $q \colon \mathbb{N} \to \mathbb{N}$, for which $\langle w_0, \ldots, w_{k+2} \rangle \in [\![\beta]\!]$ implies $|w_{k+1}|, |w_{k+2}| \leq q(|w_0|)$. Hence, we can choose as ϕ_{k-1} the QDNF form of the formula $\exists x_k \exists x_{k+1} \exists x_{k+2}. \delta_p^{x_1 \mapsto x_k} \wedge \delta_q^{x_1 \mapsto x_{k+1}} \wedge \delta_q^{x_1 \mapsto x_{k+2}} \wedge \beta$, where the subformulae δ are supplied by Lemma 6.2. Requirement 1 follows from the construction and the fact that the subformulae ϕ also have type Σ_1 by Lemma 6.2. Requirement 2 follows immediately from the construction. Requirement 3 is met, because subformula δ_p can produce all the possible choices for $w_k \in \Sigma^*$ by Lemma 6.2, and similarly the subformulae δ_q for w_{k+1} and w_{k+2}. Requirement 4 is met, because the subformulae δ limit the choices they produce by Lemma 6.2. Finally, Requirement 5 is also met by analogous considerations to Lemma 6.2.

(If ϕ is further required to be right-restricted, we can proceed as above, except that instead of obtaining β from a generalization of Theorem 6.2, we obtain an equivalent unidirectional $\beta_1 \wedge \beta_2$ from a similar generalization of Corollary 6.1.)

The formula ϕ_j, where $0 \leq j < k-1$, is then obtained as the QDNF form of the formula $\exists x_{j+1}.\delta_p^{x_1 \mapsto x_{j+1}} \wedge \neg\phi_{j+1}$, where the subformula δ_p is again supplied by Lemma 6.2. The five restrictions are met by a similar argument as above. $\square$

Now we are finally in a position to state our main result. Intuitively it says that given any language A in the Polynomial-time Hierarchy **PH**, we can write a corresponding formula ϕ that defines A, and the conjunction $R(x_0) \wedge \phi$ is right-restricted productive. In other words, ϕ can extract from the unary input relation R those strings w that belong to A, and this extraction can be carried out via the computational operations described in Chapter 5. Moreover, the type of ϕ matches the hierarchy level of A, except that level 0 is not separated from level 1.

Theorem 6.5 *Let $k > 0$. Language A belongs to Σ_k^p (to Π_k^p, respectively), if and only if there exists some pure Alignment Calculus formula ϕ with type Σ_k (type Π_k, respectively), for which* free $(\phi) = \{x_0\}$, *$A = [\![\phi]\!]$, and the QDNF form of $R(x_0) \wedge \phi$ is right-restricted productive. (In fact, this ϕ can even be chosen right-restricted.)*

Proof: Let $A \in \Pi_k^p$, and consider the formula ϕ corresponding to the complement $\overline{A} \in \Sigma_k^p$ of A from Lemma 6.3. Then $\neg\phi$ is of type Π_k by Definition 4.4, and $[\![\neg\phi]\!] = A$ by Case (1) of Theorem 1. Moreover, the QDNF form of $\chi \equiv R(x_0) \wedge \neg\phi$ is χ itself. This χ is then a right-restricted productive QDNF formula, as desired.

Let conversely ϕ be a pure Alignment Calculus formula with type Π_k, for which free $(\phi) = \{x_0\}$ and $R(x_0) \wedge \phi$ is right-restricted productive. Then ϕ satisfies the requirements of Lemma 6.1, and therefore $[\![\phi]\!] \in \Pi_k^p$.

Let $A \in \Sigma_p^k$, and consider the formula corresponding to A provided by Lemma 6.3. This formula has type Σ_k, and it is of the form

$$\psi \equiv \exists x_1 \ldots \exists x_m.\phi, \tag{6.3}$$

where ϕ is as in Eq. (4.8). In addition, this ψ satisfies the limitation condition $\{x_0\} \rightsquigarrow \emptyset$ according to the rules of right-restricted productivity; this is in turn equivalent to having both *(a)* $\bigcup_{i=1}^{p} \text{basis}\,(a_i) \vdash \{x_0\} \rightsquigarrow \{x_1, \ldots, x_m\}$, where $\text{basis}\,(a_i) = \emptyset$ whenever a_i is not right-restricted, and *(b)* each D_k satisfies required (D_k) again according to the rules of right-restricted productivity. The QDNF form of $R(x_0) \wedge \psi$ is then

$$\chi \equiv \exists x_1 \ldots \exists x_m . R(x_0) \wedge \phi, \tag{6.4}$$

and this χ is right-restricted productive by Definition 4.5.

Let finally ψ be a pure Alignment Calculus formula with type Σ_k, for which $\text{free}\,(\phi) = \{x_0\}$ and the QDNF form χ of $R(x_0) \wedge \psi$ is right-restricted productive. This ψ is as in Eq. (6.3), and χ as in Eq. (6.4). Then the right-restricted productivity of χ implies that both conditions *(a)* and *(b)* in the preceding case are also met. Hence ψ satisfies the limitation condition $\Gamma = \{x_0\} \rightsquigarrow \emptyset$ according to the rules of right-restricted productivity. Moreover, this Γ is a possible required (ψ), for example in the formula $\neg\psi$. Therefore $[\![\psi]\!] \in \Sigma_k^p$ by Lemma 6.1. □

In fact, Theorem 6.5 holds not only for right-restricted but also for heuristic productivity. Therefore, although these two classes of safe formulae were seen to differ *syntactically*, they coincide *semantically*, as already pointed out on page 80.

Corollary 6.2 *Theorem 6.5 continues to hold even after right-restricted productivity is replaced with heuristic productivity.*

Proof: Reviewing the constructions leading to Theorem 6.5 reveals that only the following two modifications are necessary.

Consider first the limitation functions $\mathcal{W}_{a_i}$ from the proof of Lemma 6.1. We must now establish their polynomiality, when a_i was seen to satisfy the corresponding limitation property by the heuristic Algorithm 1, instead of appealing to Theorem 4.2 for a right-restricted a_i. This, however, is given by Theorem 4.4.

The other modification to consider is Eq. (6.2) in the proof of Lemma 6.2. We must show that Algorithm 1 can indeed infer the required limitation property $\{2, 3\} \rightsquigarrow \{1\}$ of the 3-FSA $\mathcal{A}_\eta$ corresponding to the string formula η by Corollary 3.1. $\mathcal{A}_\eta$ is terminal on the output tape 1 (corresponding to the variable x_1), because of the terminating subformula $([x_1]_l x_1 = \bot)$ in η, thereby satisfying Condition 2 of Definition 4.3. Consider then all the loops advancing tape 1 in $\mathcal{A}_\eta$. The first, innermost loop L_1 is generated by the subformula $([x_1, x_2]_l x_2 \neq \bot)^*$ in η. Tape 2 (corrsponding to x_2) is strictly advancing in L_1, thereby satisfying Condition 1 of Definition 4.3.

The second, outermost loop L_2 is in turn generated by the whole Kleene closure in η preceding the terminating subformula. Tape 3 (corresponding to $x_{|q|+2}$) is strictly advancing in L_2. $\mathcal{A}_\eta$ has no other loops, and therefore Algorithm 1 does indeed perform as desired. □

Note that Theorem 6.5 does not provide a syntactic representation for $\Sigma_0^p = \Pi_0^p = \mathbf{P}$ distinct from the representations of $\Sigma_1^p = \mathbf{NP}$ and $\Pi_1^p =$ **co-NP** [8, Definition 1.12]. Representations for **P** have been obtained in Transducer Datalog discussed in Subsection 1.2.2 [61, Theorems 3 and 5]; these results, however, build on the introduction of either a Datalog fixpoint semantics [13, Theorem 3] or the extension of ordinary transducers with subroutines [13, Theorem 5]. However, these introductions do carry a procedural flavour. Similarly, the functional list processing language $\mathcal{L}_p$ [19, Chapter 4] and the pomset algebra on lists [45, Theorem 6.5] express exactly **P**, but again the flavour of the language is procedural, because list recursion must be carefully controlled to attain the result. In fact, all these **P** expressivity results utilize a similar polynomial clocking construction to Lemma 6.2.

Representations for the levels of **PH** other than **P** have not been reported for any of the aforementioned languages. On the other hand, representations have been found for complexity classes far above **PH**, such as the elementary functions [45, Theorem 7.2][59, Theorem 9], while such representations seem difficult to attain by syntactic restrictions of pure Alignment Calculus.

6.2 Expression Complexity

Let us now turn from data to expression complexity. Because this complexity measure disregards the lengths $|w|$ of the input strings w, we can restrict attention to the case where there are none; in other words, we consider only *closed* formulae ϕ, or ones without free variables. Then the problem is to determine whether they are are true or false; in other words, we study the satisfiability of pure Alignment Calculus formulae ϕ.

We know from Theorem 6.2 that two bidirectional variables in a string formula suffice to make expression complexity undecidable. Furthermore, Corollary 6.1 tells us the same thing happens in conjunctions of two unidirectional string formulae with at least two variables. The case of just one variable is decidable by Theorem 6.1. It remains then to examine the case of right-restricted string formulae. This problem turns out to be complete for **PSPACE**, the problems solvable in polynomial space [8, Definition 3.1].

Theorem 6.6 *Let the number of variables k be fixed beforehand. Then the* *set of true Alignment Calculus formulae of the form $\exists x_1, \ldots, x_k.\phi$, where ϕ is a right-restricted string formula on variables $x_1, \ldots, x_k$, is complete for* **PSPACE***.*

Proof: The membership in **PSPACE** can be seen as follows. Consider an arbitrary formula $\phi \equiv \exists x_1, \ldots, x_k.\phi$, and assume for notational simplicity that its variable x_1 is the bidirectional one (if any). The algorithmic construction of the right-restricted k-FSA $\mathcal{A}_\phi$ corresponding to ϕ given in Theorem 3.1 and Corollary 3.1 is polynomial in $||\phi||$ and $(|\Sigma|+2)^k$. Because not only Σ but also k is now fixed, this construction is now polynomial in $||\phi||$. By condition 4 of Corollary 3.1 we can omit the unidirectional tapes $2, \ldots, k$ from $\mathcal{A}_\phi$ to form a right-restricted 1-FSA $\mathcal{B}$: $\mathcal{B}$ accepts input w_1 if and only if $\mathcal{A}_\phi$ accepts some input of the form $\langle w_1, w_2, \ldots, w_k\rangle$. (In other words, condition 4 allows *unidirectional quantifier elimination* in front of string formulae; this is a special case of a more general result by Harrison and Ibarra [47, Theorem 2.3].) As in Section 5.3, this $\mathcal{B}$ can easily be augmented into a checking stack automaton [43, Definition 1.4] $\mathcal{C}$ with an empty input by considering (the only) tape 1 as the stack, whose contents $\mathcal{C}$ first guesses nondeterministically, and then proceeds to browse (but not alter) this guess to see if $\mathcal{B}$ finds the guess acceptable. Then ϕ is true if and only if $\mathcal{C}$ accepts, and this problem is in **PSPACE** [35, Second half of Case (1) of Theorem 2].

Because the problem used in the previous case (namely whether a given checking stack automaton accepts an empty input or not) is even **PSPACE**-hard [35, First half of Case (1) of Theorem 2], it would be tempting to invert this reduction. Theorem 3.2 does not, however, yield a *polynomial* reduction of FSAs to string formulae. (Indeed, as it is known that automata can be exponentially more succinct than regular expressions [28, Theorem 4.1], this approach fails no matter what construction is used in proving Theorem 3.2.) Therefore we reduce instead the *acceptance problem for Linear Bounded Automata (LBAs)* [49, Chapter 9.3] to the truth of formulae $\psi \equiv \exists x_1 \phi$ of the required kind. We essentially follow the construction of Theorem 4.1; here we can do without the second bidirectional quantifier, because the lengths of the LBA configurations do not change. Let M be an LBA with

- alphabet Γ, including both the input and tape alphabets;
- left and right endmarkers $\triangleright$ and $\triangleleft$, respectively;
- states $p_0, \ldots, p_m$ where p_0 is the start state and p_m the final state which has no outgoing transitions; and
- input string $I = c_1 \ldots c_n$.

We write a string formula ϕ on alphabet $\Gamma' = \Gamma \cup \{\triangleright, \triangleleft, p_0, \ldots, p_m\}$ of length $||\phi|| = O(nt\,|\Gamma'|)$, where t denotes the number of transitions in M. This formula ϕ will be true if and only if the value of x_1 is (an encoding of) an accepting computation of M on I.

1. The subformula

$$\psi(n,a,b) \equiv [\,]_l x_1 = a.([x_1]_l x_1 \neq \bot)^{n+1}.[x_1]_l x_1 = b.([x_1]_r \top)^{n+1}$$

   verifies that the current position contains the symbol a, that the $(n+2)$nd position to its right contains the symbol b, and returns to the right neighbour of a. In other words, $\psi(n,a,b)$ verifies that the next configuration has a b in the same place as this one has the current a.

2. As in the proof of Theorem 4.1, we encode as $\delta_M \equiv \underset{t\in T}{+}\,\delta_t$ the transition table T of M. Let t be a transition from state p_i to state p_j on character a that writes character b and moves the head right. Then

$$\delta_t \equiv \underset{d\in\Gamma\cup\{\triangleright,\triangleleft\}}{+} \psi(n,d,d).\psi(n,p_i,b).\psi(n,a,p_j).$$

   If, on the other hand, t moves the head left, then

$$\delta_t \equiv \underset{d\in\Gamma\cup\{\triangleright,\triangleleft\}}{+} \psi(n,d,p_j).\psi(n,p_i,d).\psi(n,a,b).$$

3. One computation step can then be expressed with the subformula

$$\chi \equiv \psi(n,\triangleright,\triangleright).\psi'(\triangleright).\delta_M.\psi'(\triangleleft).\psi(n,\triangleleft,\triangleleft), \text{ where}$$
$$\psi'(b) \equiv \left(\underset{a\in\Gamma\cup\{b\}}{+} \psi(n,a,a)\right)^*.$$

4. Finally the whole formula states that the first configuration is the initial one, loops applying the rules and finally checks that the last configuration accepts:

$$\begin{aligned}\phi \equiv \exists x_1([x_1]_l x_1 = \triangleleft.\\ &[x_1]_l x_1 = p_0.[x_1]_l x_1 = \triangleright.[x_1]_l x_1 = c_1 \ldots [x_1]_l x_1 = c_n.\\ &[x_1]_l x_1 = \triangleleft.([x_1]_r x_1 \neq \bot)^*.\chi^*.([x_1]_l \top)^*.[x_1]_l x_1 = p_m).\end{aligned}$$

   (The first '$\triangleleft$' is included so that the first configuration would not form a special case with respect to δ_M.)

This proves **PSPACE**-hardness, once Γ' is encoded into Σ in ϕ as explained in Theorem 4.2 (with logarithmic overhead). $\square$

The decidability result in Theorem 6.6 admits easy proofs for certain other (non)closure properties of string formulae. String formulae are trivially closed under disjunction, because $\phi \vee \psi$ is equivalent to $\phi + \psi$. Let us then study the power of conjunction.

Corollary 6.3 *There exist unidirectional string formulae ϕ and ψ over three variables x_1, x_2, x_3, for which $\phi \wedge \psi$ cannot be expressed with any formula of the form $\exists x_4, \ldots, x_k.\chi$ where χ is a right-restricted string formula.*

Proof: Let L be any recursively enumerable language. By Corollary 6.1, L can be represented with $\exists x_2, x_3.\phi \wedge \psi$ with unidirectional ϕ and ψ. If the claim did not hold, L would have then a representation $\exists x_2, \ldots, x_k.\chi$ with χ right-restricted. Then the membership problem of L could be solved by the following algorithm P. First, there exists a right-restricted k-FSA $\mathcal{A}_\chi$ corresponding to χ; let it be encoded in P. Then it is straightforward to see that on input w_1, P can form a right-restricted $(k-1)$-FSA $\mathcal{B}$ which accepts $\langle w_2, \ldots, w_k \rangle$ if and only if $\mathcal{A}_\chi$ accepts $\langle w_1, \ldots, w_k \rangle$. Theorem 3.2 shows how P can construct effectively a corresponding right-restricted string formula $\phi_\mathcal{B}$. Finally, Theorem 6.6 provides a method by which P can determine the truth of $\psi \equiv \exists x_1, \ldots, x_k.\phi_\mathcal{B}$. By construction, $w_1 \in L$ if and only if this ψ turns out to be true. $\square$

Corollary 6.3 shows that there are string formulae in which rewinding two variables is required; only one does not suffice, not even when quantified. Let us now examine the effect the number of variables has on the problem. The unquantified string formulae with a single variable were seen to be closed under conjunction in Theorem 6.1. The following result discusses the remaining case of two variables.

Corollary 6.4 *If Corollary 6.3 fails when there are only two variables x_1 and x_2, then the mapping $\phi \wedge \psi \mapsto \exists x_3, \ldots, x_k.\chi$ is uncomputable.*

Proof: Assume conversely that the mapping f in question is computable and defined for all ϕ and ψ. Let M be any Turing machine. Note that building the representation $\exists x_2, x_3(\phi \wedge \psi)$ for M in Corollary 6.1 is effective. Then the formula $\exists x_1, x_2, x_3.f(\phi, \psi)$ expressing "M accepts some input" can be effectively constructed and is of a form, whose truth is decidable by Theorem 6.6. $\square$

Corollaries 6.3 and 6.4 justify our choice of a layered language by showing that it would be quite difficult to give an operational semantics to a language where conjunctions and modalities over more than one string were allowed to mix freely.

We are now in a position to assess whether to extend Alignment Calculus into *infinite* strings by allowing semi-infinite intervals K_i^A of the form $\{j, j+1, j+2, \ldots\}$ in Eq. (2.1) of Section 2.1; operational semantics would then be obtained by reconsidering the k-FSAs as (e.g. Büchi) automata over infinite words [90, Part I.1]. For Wolper's extended temporal logic the corresponding extension has been carried out by Vardi and Wolper [99]. In this extension each relation symbol R would no longer be interpreted as a finite relation but instead as an $a(R)$-FSA representing finitely the possibly infinite strings belonging to the relation. Corollaries 6.3 and 6.4, however, show that even the most elementary query evaluation would in general be impossible; in the case of single strings, the extension to infinite words was possible precisely because there the corresponding classes remained closed under the boolean operations. This is true for the alphabets we consider; for the *unary* alphabet case, this extension can indeed be developed for example by applying arithmetical techniques [53].

Another dimension that might be enlarged to infinity is $|\Sigma|$; that is, to turn Alignment Calculus from string to (linear) list query language. (In fact, this extension is already underway as an application of Alignment Calculus for processing lists of compex objects [9].) The logical semantics in Chapter 2 needs no changes. To modify Aligment Algebra in Section 3.3 accordingly, it suffices to extend the k-FSAs introduced in Section 3.1 to encompass the infinite alphabet case; one suggestion in this direction has been made by Kaminski and Francez [54]. The crux is, however, whether an analogy of Theorem 4.2 can still be proved. If so, the further safety analysis in Section 4.4 would provide us the desired language.

The results obtained in this chapter (Chapter 6) are summarized as Table 6.1.

Table 6.1 Summary of expressivity results.

FORMULA CLASS	DATA COMPLEXITY	EXPRESSION COMPL.
String formulae of one variable.	The regular languages, by Theorem 6.1.	Polynomial-time for uni- and **PSPACE**-complete for bidirectional formulae, by [37, Problem AL2] and Theorem 6.6.
Formulae of form $\exists x_2, x_3 \phi$, where ϕ is either a string formula or a conjunction of two unidirectional string formulae.	The recursively enumerable languages, by Theorem 6.2 and Corollary 6.1.	Undecidable.
Right-restricted or heuristically productive type Σ_k (Π_k) formulae, where $k > 0$.	Level Σ_k^p (Π_k^p) of the Polynomial-time Hierarchy, by Theorem 6.5 and Corollary 6.2.	Decidable by Chapter 5. Type Σ_1 may already be **PSPACE**-hard, by Theorem 6.6.

Chapter 7

Conclusions

Adding support for structured types is an important problem in the database field. One such type is strings drawn from a finite alphabet fixed beforehand: at times they should be considered as indivisible entities, while at other times their internal structure should become visible for processing. A major application field for these string databases is computational molecular biology, where it is desirable to be able to model the stored nucleotide strands directly. Moreover, this field needs cutting-edge database tools, both because the amount of incoming data grows exponentially, and because modelling this data requires application-specific data type support. One particular important subfield of molecular data processing is studying the grammatical structure of the stored nucleotide sequences.

This work has introduced Alignment Calculus, a new declarative query language for string databases. This language seems well suited for the aforementioned grammatical sequence studies in bioinformatics. The string handling capabilities of Alignment Calculus are provided by string formulae, which emerged as an extension of linear-time extended temporal logic into multiple string comparisons; to our knowledge, there are no other extensions of this kind in the literature. These string formulae can also be seen as a specification language for grammatical kinds of multiple string alignments; in turn, the concept of aligning multiple strings to each other is fundamental to computational molecular biology.

The full Alignment Calculus is then derived by adding these string formulae into the relational calculus. That is, we suggest an extension of the well-known relational data model into string data. However, the string formulae are in no way tied to the relational model, and have indeed been used in an object-oriented model as well. Alignment Calculus supports not only pattern matching in strings, but also user-defined comparison of multiple strings, even in the case when some of the strings are not known in advance and have to be generated to match the comparison condition. In this way, Alignment Calculus can be considered a string constraint database language.

Because Alignment Calculus is a declarative language, it is not clear at first sight how to provide an operational interpretation for it, and therefore a blueprint for an actual computer implementation. The operational counterpart to string formulae is provided by multitape nondeterministic finite state automata with endmarkers. Alignment Algebra, the operational counterpart of the whole language was then derived from relational algebra as follows. First, an explicit domain symbol was added. And second, selection conditions were expressed with these automata.

However, because the added domain symbol denoted an infinite set, further measures had to be taken to ensure the finiteness of query evaluation. This forced a reconsideration of the well-known safety problem of the relational calculus. Indeed, it did transpire that safety was already undecidable in general for the string formulae, the basic constituent of the language. Fortunately, a subset of string formulae with decidable safety properties could also be identified; in these "right-restricted" string formulae at most one of the strings is scanned back and forth. A syntactically safe sublanguage of full Alignment Calculus could then be developed based on these right-restricted string formulae. This sublanguage, the right-restricted productive formulae, is then our candidate for a practical string database query language. A major principle in designing our sublanguage was to restrict the generation of new strings during query evaluation as tightly as possible. Hence, the construction of our sublanguage, which is not specific to string databases, may be of interest in other constraint database formalisms with costly constraint satisfaction components.

These right-restricted productive formulae were then mapped into a subclass of Alignment Algebra, which admits a natural finite evaluation strategy. In this subclass, every occurrence of the domain symbol is immediately enclosed by a selection, which is known to always return finite results only – that is, the corresponding right-restricted string formula is known to satisfy the corresponding finiteness dependency. Moreover, algorithms were given for evaluating the finite result the enclosing selection in finite time, thereby providing the desired strategy. Therefore our string extension should be straightforward to add into existing query engines by omitting the domain symbol and introducing instead these finite evaluation algorithms. Indeed, we are currently experimenting with such an extension.

There are languages in the literature with a similar approach to generating new strings, such as sequence logic and Transducer Datalog discussed in Subsection 1.2.2. However, these languages state the concept of generation explicitly by means of string processing automata with inputs and outputs. In contrast, input and output directions arise only during the translation of a given Alignment Calculus query into the Algebra. Moreover, the semantics of Alignment Calculus can be formulated without recourse to computational terminology, whereas the semantics of these other languages have a more procedural feel, because their basic string handling constructs are inherently procedural in nature.

Finally, the expressive power of Alignment Calculus was studied in order to determine what kinds of string handling tasks could be solved by combining string formulae with logical connectives. The central result was that the right-restricted string formulae equal the Polynomial-time Hierarchy; therefore, our candidate language can express all feasible string handling tasks. In fact, we even identified another syntactically safe sublanguage of Alignment Calculus with the same expressive power, but a less complex safety analysis. It remains to be seen whether the way our language expresses these tasks is considered practical; the necessary theoretical groundwork has now been laid.

Bibliography

[1] Abiteboul, S., Herr, L., and Van der Bussche, J. Temporal versus first-order logic to query temporal databases. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems* (1996), pp. 49–57.

[2] Abiteboul, S., Hull, R., and Vianu, V. *Foundations of Databases*. Addison-Wesley, 1995.

[3] Abiteboul, S., Papadimitriou, C., and Vianu, V. The power of reflective relational machines (extended abstract). In *Logic in Computer Science* (1994), pp. 230–240.

[4] Abiteboul, S., Vardi, M., and Vianu, V. Computing with infinitary logic. *Theoretical Computer Science 149* (1995), 101–128.

[5] Abiteboul, S., Vardi, M., and Vianu, V. Fixpoint logics, relational machines, and computational complexity. *Journal of the ACM 44*, 1 (1997), 30–56.

[6] Abiteboul, S., and Vianu, V. Computing with first-order logic. *Journal of Computer and System Sciences 50* (1995), 309–335.

[7] Aho, A., Sethi, R., and Ullman, J. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[8] Balcázar, J., Díaz, J., and Gabarró, J. *Structural Complexity*, vol. I. Springer-Verlag, 1988.

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

[10] Benedikt, M., and Libkin, L. Languages for relational databases over interpreted structures. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems* (1997), pp. 87–96.

[11] BIERCE, A. *The Enlarged Devil's Dictionary*. Victor Gollancz ltd. London, 1967.

[12] BISKUP, J., AND KLUCK, A. A new approach to inferences of semantic constraints. In *Advances in Databases and Information Systems* (1997), Springer Electronic Workshops in Computing, pp. 72–79.

[13] BONNER, A., AND MECCA, G. Sequence, Datalog and transducers. *Journal of Computer and System Sciences* (1997). (To appear.).

[14] BORGES, J. *The Book of Imaginary Beings*. Penguin Books, 1974.

[15] BORGES, J. *Dreamtigers*. University of Texas Press, 1987.

[16] BRAUER, W., AND LANGE, K.-J. Non-deterministic two-tape automata are more powerful than deterministic ones. In *Symposium on Theoretical Aspects of Computer Science* (1985), pp. 71–79.

[17] CHAUDHURI, S., AND SHIM, K. Optimization of queries with user-defined predicates. In *Very Large Data Bases Conference* (1996), pp. 87–98.

[18] CLOCKSIN, W., AND MELLISH, C. *Programming in Prolog*, 3rd, revised and extended ed. Springer-Verlag, 1987.

[19] COLBY, L., ROBERTSON, E., SAXTON, L., AND VAN GUCHT, D. A query language for list-based complex objects. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems* (1994), pp. 179–189.

[20] COLLADO-VIDES, J. 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 (1991), 321–326.

[21] COLLADO-VIDES, J. Grammatical model of the regulation of gene expression. *Proceedings of the National Academy of Sciences of the USA 89* (1992), 9405–9409.

[22] COLLADO-VIDES, J. A linguistic representation of the regulation of transcription initiation. I. an ordered array of complex symbols with distinctive features. *BioSystems 29* (1993), 87–104.

[23] COLLADO-VIDES, J. A linguistic representation of the regulation of transcription initiation. II. distinctive features of sigma 70 promoters and their regulatory binding sites. *BioSystems 29* (1993), 105–128.

[24] DATE, C. *An Introduction to Database Systems*, 6th ed. Addison-Wesley, 1994.

[25] DENNING, P. Computing, applications, and computational science. *Communications of the ACM 34*, 10 (1991), 129–131.

[26] DONG, S., AND SEARLS, D. Gene structure prediction by linguistic methods. *Genomics 23* (1994), 540–551.

[27] EBBINGHAUS, H.-D., AND FLUM, J. *Finite Model Theory*. Springer-Verlag, 1995.

[28] EHRENFEUCHT, A., AND ZEIGER, P. Complexity measures for regular expressions. *Journal of Computer and System Sciences 12* (1976), 134–146.

[29] EMERSON, E. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed., vol. B: Formal Models and Semantics. Elsevier, 1990, ch. 16, pp. 994–1072.

[30] ESCOBAR-MOLANO, M., HULL, R., AND JACOBS, D. Safety and translation of calculus queries with scalar functions (extended abstract). In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems* (1993), pp. 253–264.

[31] ESCOBAR-MOLANO, M., HULL, R., AND JACOBS, D. Safety and translation of calculus queries with scalar functions. Tech. rep., Computer Science Department, University of Southern California, March 1993. Abstracted in [30].

[32] ETZOLD, T., AND ARGOS, P. SRS – an indexing and retrieval tool for flat file data libraries. *Computer Applications in the Biosciences 9*, 1 (1993), 49–57.

[33] ETZOLD, T., AND ARGOS, P. Transforming a set of biological flat file libraries to a fast access network. *Computer Applications in the Biosciences 9*, 1 (1993), 59–64.

[34] EUROPEAN MOLECULAR BIOLOGY NETWORK (EMBNET). *Strategies in Bioinformatics: a European Perspective* (1994).

[35] GALIL, Z. Hierarchies of complete problems. *Acta Informatica 6* (1976), 77–88.

[36] GANGULY, S., AND NOORDEWIER, M. PROXIMAL: a database system for the efficient retrieval of genetic information. *Computers in Biology and Medicine 26*, 3 (1996), 199–207.

[37] GAREY, M., AND JOHNSON, D. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, 1979.

[38] GINSBURG, S., AND WANG, X. 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* (1992), pp. 293–300.

[39] GOLDBERG, R. Finite state automata from regular expression trees. *The Computer Journal 36*, 7 (1993), 623–630.

[40] GRAHNE, G., AND NYKÄNEN, M. Safety, translation and evaluation of Alignment Calculus. In *Advances in Databases and Information Systems* (1997), Springer Electronic Workshops in Computing, pp. 295–304.

[41] GRAHNE, G., NYKÄNEN, M., AND UKKONEN, E. Reasoning about strings in databases. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems* (1994), pp. 303–312.

[42] GRAHNE, G., NYKÄNEN, M., AND UKKONEN, E. Reasoning about strings in databases. *Journal of Computer and System Sciences* (1997). To appear. Abstracted in [41].

[43] GREIBACH, S. Checking automata and one-way stack languages. *Journal of Computer and System Sciences 3* (1969), 196–217.

[44] GRIES, D. *The Science of Programming.* Springer-Verlag, 1981.

[45] GRUMBACH, S., AND MILO, T. An algebra for pomsets (extended abstract). In *International Conference on Database Theory* (1995), pp. 191–207.

[46] GUSFIELD, D. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[47] HARRISON, M., AND IBARRA, O. Multi-tape and multi-head pushdown automata. *Information and Control 13* (1968), 433–470.

[48] HELGESEN, C., AND SIBBALD, P. PALM - a pattern language for molecular biology. In *International Conference on Intelligent Systems in Molecular Biology* (1993), pp. 172–180.

[49] HOPCROFT, J., AND ULLMAN, J. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[50] HULL, R. Counterexample to claims of van Gelder and Topor. Unpublished note, January 25th 1994.

[51] HULL, R., AND SU, J. Domain independence and the relational calculus. *Acta Informatica 31* (1994), 513–524.

[52] JAGADISH, H., MENDELZON, A., AND MILO, T. Similarity-based queries. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems* (1995), pp. 36–45.

[53] KABANZA, F., STÉVENNE, J.-M., AND WOLPER, P. Handling infinite temporal data. *Journal of Computer and System Sciences 51* (1995), 3–17.

[54] KAMINSKI, M., AND FRANCEZ, N. Finite-memory automata. *Theoretical Computer Science 134* (1994), 329–363.

[55] KEKKONEN, M., KOSKELAINEN, J., NIEMI, J., TUONONEN, T., VIHERVAARA, A., AND VUOLASTO, J. Design document for the BiO_2 system. Tech. rep., Department of Computer Science, University of Helsinki, Finland, 10th April 1997. In Finnish.

[56] KRISHNAMURTHY, R., RAMAKRISHNAN, R., AND SHMUELI, O. A framework for testing safety and effective computability. *Journal of Computer and System Sciences 52* (1996), 100–124.

[57] LEFEBVRE, F. A grammar-based unification of several alignment and folding algorithms. In *International Conference on Intelligent Systems in Molecular Biology* (1996), pp. 143–155.

[58] LLOYD, J. *Foundations of Logic Programming*, 2nd, extended ed. Springer-Verlag, 1987.

[59] MECCA, G. *From Datalog to Sequence Datalog.* PhD thesis, Università di Roma “La Sapienza”, Dipartimento di Informatica e Sistemistica, 1996.

[60] MECCA, G. personal communication, 27th February 1997.

[61] MECCA, G., AND BONNER, A. Sequences, Datalog and transducers. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems* (1995), pp. 23–35.

[62] MILLER, W., SCHWARTZ, S., AND HARDISON, R. A point of contact between computer science and molecular biology. *IEEE Computational Science and Engineering 1*, 1 (1994), 69–78.

[63] PELCZAR, JR., M., CHAN, E., AND KRIEG, N. *Microbiology*, 5th ed. McGraw-Hill, 1986.

[64] PINKER, S. *The Language Instinct.* William Morrow and Company, Inc., 1994.

[65] PISTOR, P., AND TRAUNMÜLLER, R. A database language for sets, lists and tables. *Information Systems 11*, 4 (1986), 323–336.

[66] QUINE, W. Concatenation as a basis for arithmetic. *The Journal of Symbolic Logic 11*, 4 (1946), 105–114.

[67] RAJLICH, V. Absolutely parallel grammars and two-way finite-state transducers. *Journal of Computer and System Sciences 6* (1972), 324–342.

[68] RAMAKHRISNAN, R., BANCILHON, F., AND SILBERSCHATZ, A. Safety of recursive Horn clauses with infinite relations (extended abstract). In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems* (1987), pp. 328–339.

[69] RICE, J. Computational science and the future of computing research. *IEEE Computational Science and Engineering 2*, 4 (1995), 35–41.

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

[71] RODRIGUEZ-TOMÉ, P., STOEHR, P., CAMERON, G., AND FLORES, T. The European Bioinformatics Institute (EBI) databases. *Nucleic Acids Research 24*, 1 (1996), 6–12.

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

[73] ROSENBLUETH, D., THIEFFRY, D., HUERTA, A., SALGADO, H., AND COLLADO-VIDES, J. Syntactic recognition of regulatory regions in *Escherichia Coli. Computer Applications in the Biosciences 12*, 5 (1996), 415–422.

[74] ROZOY, B. About two-way transducers. In *Foundations of Computation Theory* (1985), pp. 371–379.

[75] SACKS, O. *The Man Who Mistook His Wife for a Hat.* Picador, 1986.

[76] SCHWEIGERT, S., HERDE, P., AND SIBBALD, P. Issues in incorporation semantic integrity in molecular biological object-oriented databases. *Computer Applications in the Biosciences 11*, 4 (1995), 339–347.

[77] SEARLS, D. The linguistics of DNA. *American Scientist 80* (1992), 579–591.

[78] SEARLS, D. Formal grammars for intermolecular structure. In *Intelligence in Neural and Biological Systems* (1995), pp. 30–37.

[79] SEARLS, D. String Variable Grammar: A logic grammar formalism for the biological language of DNA. *Journal of Logic Programming* (1995), 73–102.

[80] SEARLS, D., AND MURPHY, K. Automata-theoretic models of mutation and alignment. In *International Conference on Intelligent Systems in Molecular Biology* (1995), pp. 341–349.

[81] SESHADRI, P., LIVNY, M., AND RAMAKRISHNAN, R. SEQ: A model for sequence databases. In *IEEE International Conference on Data Engineering* (1995), pp. 232–239.

[82] SESHADRI, P., LIVNY, M., AND RAMAKRISHNAN, R. The design and implementation of a sequence database system. In *Very Large Data Bases Conference* (1996), pp. 99–110.

[83] SILBERSCHATZ, A., STONEBRAKER, M., AND ULLMAN, J. Database systems: Achievements and opportunities. *Communications of the ACM 34*, 10 (1991), 110–120.

[84] SILBERSCHATZ, A., STONEBRAKER, M., AND ULLMAN, J. Database research: Achievements and opportunities into the 21st century. *ACM SIGMOD Record 25*, 1 (1996), 52–63.

[85] SIPPU, S., AND SOISALON-SOININEN, E. *Parsing Theory*, vol. I: Languages and Parsing. Springer-Verlag, 1988.

[86] Sohn, K., and van Gelder, A. Termination detection in logic programs using argument sizes (extended abstract). In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems* (1991), pp. 216–226.

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

[88] Stoesser, G., Sterk, P., Tuli, M., and Cameron, G. The EMBL nucleotide sequence database. *Nucleic Acids Research 25*, 1 (1997), 7–13.

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

[90] Thomas, W. Automata on infinite objects. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed., vol. B: Formal Models and Semantics. Elsevier, 1990, ch. 4, pp. 133–192.

[91] Tsonis, A., Elsner, J., and Tsonis, P. Is DNA a language? *Journal of Theoretical Biology 184* (1997), 25–29.

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

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

[94] Väänänen, J. *Matemaattinen logiikka*, 2nd ed. Gaudeamus, 1987.

[95] Valdéz-Pérez, R. A scientific basis for computational science. Tech. Rep. CMU-CS-93-162, School of Computer Science, Carnegie-Mellon University, USA, 1993.

[96] van Gelder, A. Deriving constraints among argument sizes in logic programs. Tech. Rep. UCSC-CRL-89-41, University of Califormia, Santa Cruz, 1989. Appears also in the *Annals of Mathematics and Artificial Intelligence*.

[97] van Gelder, A., and Topor, R. Safety and translation of relational calculus queries. *ACM Transactions on Database Systems 16*, 2 (1991), 235–278.

[98] VARDI, M. The complexity of relational query languages. In *ACM Symposium on Theory of Computing* (1982), pp. 137–145.

[99] VARDI, M., AND WOLPER, P. Reasoning about infinite computations. *Information and Computation 115* (1994), 1–37.

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

[101] WEBER, A. On the valuedness of finite transducers. *Acta Informatica 27* (1990), 749–780.

[102] WEBER, A. On the lengths of values in a finite transducer. *Acta Informatica 29* (1992), 663–687.

[103] WOLPER, P. Temporal logic can be more expressive. *Information and Control 56* (1983), 72–99.

Index

ISSN 1238-8645
ISBN 951-45-7859-7
Helsinki 1997
Helsinki University Printing House