String-to-String Interpretations With Polynomial-Size Output

Mikołaj Bojańczyk
Institute of Informatics, University of Warsaw, Poland
bojan@mimuw.edu.pl

Sandra Kiefer
Department of Computer Science, RWTH Aachen University, Germany
kiefer@cs.rwth-aachen.de

Nathan Lhote
Institute of Informatics, University of Warsaw, Poland
nlhote@mimuw.edu.pl

Abstract

String-to-string MSO interpretations are like Courcelle's MSO transductions, except that a single output position can be represented using a tuple of input positions instead of just a single input position. In particular, the output length is polynomial in the input length, as opposed to MSO transductions, which have output of linear length. We show that string-to-string MSO interpretations are exactly the polyregular functions. The latter class has various characterisations, one of which is that it consists of the string-to-string functions recognised by pebble transducers.

Our main result implies the surprising fact that string-to-string MSO interpretations are closed under composition.

2012 ACM Subject Classification Theory of computation → Transducers

Keywords and phrases MSO, interpretations, pebble transducers, polyregular functions

Digital Object Identifier 10.4230/LIPIcs.ICALP.2019.106

Category Track B: Automata, Logic, Semantics, and Theory of Programming

Related Version `https://arxiv.org/abs/1905.13190`, [6]

Acknowledgements The authors would like to thank Benedikt Brütsch for helpful discussions on the topic.

1 Introduction

A string-to-string function is called *regular* if it is computed by a deterministic two-way automaton with output. There are many equivalent models for the same class of functions: string-to-string MSO transductions [11], streaming string transducers [1], and various kinds of combinator-based formalisms [2, 5, 9].

A deterministic two-way automaton can visit each input position at most once in each state, otherwise it would loop forever. This means that the length of the run – and also the size of the output word – is linear in the input string. One way to go beyond linear-sized outputs was proposed by Milo, Suciu and Vianu [17], following earlier work by Globerman and Harel [12]: equip the automaton with k pebbles which can be used to mark positions in the input word. To avoid making the model Turing-powerful, the pebbles are required to observe a so-called stack discipline: they are organised in a stack, and only the top-most pebble can be moved. In [3], it is shown that pebble transducers are equivalent to multiple other models: a higher-order functional programming language [3, Section 4], an imperative programming language with for-loops [3, Section 3], combinators [3, end of Section 4], and

© Mikołaj Bojańczyk, Sandra Kiefer, and Nathan Lhote;
licensed under Creative Commons License CC-BY

compositions of certain simple atomic functions [3, Section 1]. Because of the multitude of models and their polynomial-size outputs, the class of functions recognised by these models is called *polyregular functions*.

The list of models for polyregular functions described in [3] does not include any logical model. In this paper, we fix that omission. As mentioned above, for the regular functions, which have linear-size output, the logical model consists in string-to-string MSO transductions. In an MSO transduction, each position of the output string is interpreted as a single position of the input string. A natural idea to capture polyregular functions is to consider what we call *string-to-string* MSO *interpretations*, where a position of the output string is represented by a k-tuple of positions in the input string. At first glance, this idea looks suspicious: if string-to-string MSO interpretations were equivalent to polyregular functions, then they would be closed under composition, because the class of polyregular functions is. However, composing two string-to-string MSO interpretations

$$\Sigma^* \xrightarrow{f} \Gamma^* \xrightarrow{g} \Delta^*$$

raises the following issue. Suppose that positions of the intermediate word in Γ^* are represented by k-tuples of positions in the input word from Σ^*. If an MSO formula defining g quantifies over a set of positions in the intermediate word to define a property of the output word in Δ^*, then this corresponds to quantifying over a set of k-tuples of positions in the input word. If we assume dimension $k = 1$, then the problem dissolves, and this is why MSO transductions have dimension $k = 1$, whereas dimension $k > 1$ is never used in the context of MSO (as opposed to first-order logic, where the standard notion of transformation, i.e. first-order interpretation, uses higher dimensions).

As our main result, we show that the problems discussed above only invalidate the natural construction for composing MSO interpretations, which uses substitutions of formulas. Still, and surprisingly, for structures that represent strings there exists a (less natural) construction. This follows from our main result, which states that polyregular functions are exactly the string-to-string MSO interpretations. Indeed, corollaries of the main result are that (a) string-to-string MSO interpretations are closed under composition; and (b) for every regular string language, its inverse image under a string-to-string MSO interpretation is also regular. This is because (a) and (b) are true for polyregular functions. Proving (a) and (b) directly for string-to-string MSO interpretations seems hard; in fact an understandable (but wrong) first reaction to the claims (a) and (b) would be that they are false, for the reasons discussed in the previous paragraph.

It is easy to see that every polyregular function is a special case of a string-to-string MSO interpretation. One argument is that a k-pebble automaton can be simulated using a string-to-string MSO interpretation, by representing configurations of the pebble automaton using k-tuples of positions in the input word. The difficulty lies in proving the opposite direction and it comes from the stack discipline required in a pebble automaton. A k-tuple of positions used by an MSO interpretation can of course be viewed as a configuration of a pebble automaton, but there does not seem to be any reason why the resulting pebble automaton should observe stack discipline. It turns out – and this is the main technical insight of this paper – that any MSO formula which defines a linear ordering on k-tuples of positions in strings must necessarily observe an implicit stack discipline, which makes it possible to translate a string-to-string MSO interpretation into a pebble automaton.

Outline. After describing string-to-string MSO interpretations in Section 2, we revise polyregular functions via the formalism of for-programs in Section 3. In Section 4, we show that the models are equivalent.

Due to space limits, we omit some of the technical details and proofs in this article. For a full version, we defer the reader to [6]

2 Interpretations

In this section, we revise first-order and MSO interpretations, which are transformations of relational structures using formulas.

2.1 Logic and interpretations

Relational vocabularies and logic. A *(relational) vocabulary* is a set of relation names, each one associated with a natural number called its *arity*. For short, we refer to relational vocabularies simply as *vocabularies*. A *structure* over a vocabulary σ consists of a set called the *universe* and for each relation name of σ a corresponding relation of the same arity over the universe. To define properties of relational structures, we use monadic second-order logic and its first-order fragment with the usual syntax and semantics [19]. We use the convention that lower-case variables x, y, z range over elements and upper-case variables X, Y, Z range over sets of elements.

Interpretations. Intuitively speaking, an interpretation is a function from relational structures to relational structures where each element of the universe of the output structure is a tuple of elements of the input structure, and the relations of the output structure are defined using formulas evaluated over the input structure.

▶ **Definition 1** (Interpretations over general structures). *For $k \geq 1$, the syntax of a k-*dimensional first-order interpretation *consists of:*
1. *two vocabularies, called the* input vocabulary *and the* output vocabulary
2. *an* FO *formula over the input vocabulary with k free variables, called the* universe formula*.*
3. *for each n and each n-ary relation name R of the output vocabulary, an associated* FO *formula φ_R over the input vocabulary, with $k \cdot n$ free variables.*

MSO interpretations *are defined analogously, except that formulas of* MSO *are used, but the free variables still range over elements and not over sets.*

The semantics of an interpretation is a function from structures over the input vocabulary to structures over the output vocabulary, defined as follows.
- The universe of the output structure is the set of k-tuples of elements in the universe of the input structure which satisfy the universe formula from item 2 in Definition 1.
- An n-ary relation name R of the output vocabulary is interpreted as the set of n-tuples of k-tuples from the input structure for which (a) each k-tuple is in the output universe, and (b) the entire $(n \cdot k)$-tuple satisfies the formula φ_R in item 3 in Definition 1.

Composition. First-order interpretations are closed under composition [14, p. 218]. Let us recall the proof. Suppose that we want to compose interpretations

$$\text{structures over } \sigma_1 \xrightarrow{\mathcal{I}_1} \text{structures over } \sigma_2 \xrightarrow{\mathcal{I}_2} \text{structures over } \sigma_3$$

of dimensions k_1 and k_2, respectively. The $(k_1 \cdot k_2)$-dimensional composition is obtained from $\mathcal{I}_2$ as follows: (a) quantification over elements of $\mathcal{I}_2$ is replaced by a quantification over k_1-tuples of elements; and (b) relation names from σ_2 that appear in the input of $\mathcal{I}_2$ are replaced by the corresponding formulas from $\mathcal{I}_1$. This idea does not work for MSO in general, since set quantification in $\mathcal{I}_2$ would need to be replaced by quantification over sets of k_1-tuples. It does work when $k_1 = 1$. This essentially corresponds to Courcelle's transductions, for which closure under composition follows naturally [8, Theorem 7.14]. To recover closure under composition for $k_1 \geq 2$, one can use (not necessarily monadic) second-order logic, which by Fagin's Theorem [16, Corollary 9.9] corresponds to the polynomial hierarchy of computational complexity and is outside the scope of this paper.

2.2 String-to-string interpretations

In this paper we are interested in interpretations that transform structures which represent strings. While there are two natural ways to model strings as relational structures, namely with an order relation or with a successor relation, only the order relation is useful in our context.

▶ **Definition 2** (String-to-string interpretations). *For a string $w \in \Sigma^*$, its* ordered model *is defined to be the following relational structure, denoted by $\underline{w}$:*

- *the universe consists of the positions in the string, i.e., natural numbers;*
- *there is a binary relation for the natural order on positions;*
- *for each $a \in \Sigma$ there is a unary relation which is satisfied by every position with label a.*

For alphabets Σ and Γ, a function $f : \Sigma^* \to \Gamma^*$ is called first-order string-to-string interpretation *if the corresponding transformation on ordered models is a first-order interpretation for strings with length at least two*[1]*. Likewise we define* MSO string-to-string interpretations.

▶ **Example 3.** Consider the function $f\colon \{a, b\}^* \to \{a, b\}^*$ which maps a word to the concatenation of all of its reversed prefixes, as in the following example (with prefixes grouped for better readability):

$$abbb \quad \mapsto \quad \underbrace{a}\underbrace{ba}\underbrace{bba}\underbrace{bbba}.$$

This transformation is the running example in [3]. We show that f can be seen as a string-to-string first-order interpretation. The dimension is 2, i.e. positions in the output word represent pairs of positions in the input word. A pair (x_1, x_2) of positions in the input word is used in the output word if it satisfies the universe formula $x_2 \leq x_1$. The idea is that x_1 represents the length of the prefix, while x_2 is the position in that prefix. The label of a position (x_1, x_2) is inherited from the second coordinate, as expressed by the formulas corresponding to labels on the output structure:

$$\varphi_a(x_1, x_2) = a(x_2) \quad \varphi_b(x_1, x_2) = b(x_2)$$

The order on the positions of the output word is defined by the formula

$$\varphi_{\leq}(\underbrace{x_1, x_2}_{\substack{\text{a position of}\\ \text{the output word}}}, \underbrace{x_1', x_2'}_{\substack{\text{another position of}\\ \text{the output word}}}) = (x_1 < x_1') \vee (x_1 = x_1' \wedge x_2 \geq x_2').$$

[1] A typical operation we want to model is string duplication. When the input length is at least two, one can represent additional copies of the input string using a higher dimension. For input length $n \leq 1$, the output length will be $n^k \leq 1$ regardless of the dimension k. Another solution to this issue would be to have duplication built into the definition of interpretations.

Note that the above formula defines the lexicographic ordering on pairs of positions, with the first coordinate being used in increasing order, and the second coordinate being used in decreasing order. This, as it will turn out, is not a coincidence, since our main technical result says that it is impossible to define a linear order on tuples of positions without implicitly using some kind of lexicographic ordering.

Successor instead of order. When modelling a string as a relational structure, we use the order on positions. An alternative solution would be to use just the successor relation. The difference between the two solutions is that it is harder to define an order on k-tuples of positions than it is to define a successor relation. It turns out that the difference is crucial, and functions that output strings with successor can be ill-behaved. Note that whether or not the input string is equipped with an order or a successor relation makes no difference, since the order on the positions of the input string can be recovered in MSO, which can compute the transitive closure of binary relations on positions.

Define the *successor model* of a string in the same way as the ordered model from Definition 2, except that a binary relation for successor is used instead of the order. Define a *successor-*MSO *string-to-string interpretation* to be a string-to-string function which is computed by an MSO interpretation, assuming that strings are represented by their successor models. Likewise, we define successor-first-order string-to-string interpretations. These are closed under composition, because first-order interpretations are closed under composition. On the other hand, successor-MSO string-to-string interpretations are not closed under composition and lead to undecidability, as summarised in the following theorem.

▶ **Theorem 4.**

1. *The class of successor-*MSO *string-to-string interpretations is not closed under composition, and strictly contains the class of (order-)*MSO *string-to-string interpretations.*
2. *The following is undecidable: given a successor-first-order string-to-string interpretation f and a regular language L over the output alphabet, decide if $f^{-1}(L)$ is nonempty.*

3 Polyregular functions

Here we describe the class of polyregular functions. It has several equivalent characterisations, see [3, Theorem 4.4], one of which consists in the afore-mentioned pebble transducers. For the purposes of this paper, it will be most convenient to use a slightly more abstract characterisation in terms of for-programs, a machine model for string-to-string functions. We just explain the formalism on short examples, for a more detailed description see [3].

```
for x in first..last
  for y in last..first
    if y≤x and a(y) then
      output a
    if y≤x and b(y) then
      output b
```

(a) A for-program for the function in Example 3.

```
for x in first..last
  var P : Bool
  for y in last..first
    if y≥x then
      P := not P
  if P and a(x) then
    output a
  if P and b(x) then
    output b
```

(b) A for-program with a Boolean variable P.

```
for y in first..last
  if x1≤y and y≤x2 and a(y)
      P := true
```

(c) A for-program which checks if there is an a between the positions x1 and x2.

Figure 1 Example for-programs.

Most of the syntactic constructions that can be used in a for-program are illustrated in Figure 1a: (1) variables that range over positions in the input word; (2) for-loops in which a variable iterates over all positions in the input word in increasing or decreasing order; (3) if-statements which depend on the order/labels of variables; (4) instructions which output letters. Position variables cannot be declared or written to, they are implicitly declared by for-loops and their only updates are the iterations performed by the for-loops.

The only feature of for-programs that is not used in Figure 1a is (5) Boolean variables. Figure 1b shows a program that outputs only those letters in the input word which have even distance to the last position. In the program, the Boolean variable `P` is declared in the scope of a for-loop. On each iteration of the loop, the variable is reinitialised to false.

A for-program is called *first-order definable* if Boolean variables can only be updated from `false`, which is their initial value upon declaration, to `true`. In other words, the only allowed update for Boolean variables is `P := true`. For the first-order restriction, it is important that Boolean variables can be declared inside for-loops, and that they are reinitialised to `false` at each iteration of the loop that they are declared in. The reason for the name "first-order definable" is that one can define in first-order logic the reachability relation on program states of the for-program, see [3, Lemma 5.3].

▶ **Definition 5.** *A string-to-string function is called* polyregular *if it is computed by a for-program. It is called* first-order polyregular *if it is computed by a first-order definable for-program.*

The class of polyregular functions has other characterisations, including the string-to-string pebble transducers introduced by Milo, Suciu and Vianu [17], as well as a higher-order functional programming language [3, Section 4]. The main result of this paper, Theorem 7 in the next section, adds a logical characterisation, namely string-to-string MSO interpretations.

Evaluating first-order formulas. The for-programs described above take as input strings and also output strings. One can also consider for-programs which input a string with distinguished positions and which output a Boolean value, as in Figure 1c. The distinguished positions are represented by free variables (here `x1` and `x2`), while the output value is taken from some distinguished Boolean variable, here `P`.

▶ **Lemma 6.** *Let $\varphi(x_1, \dots, x_k)$ be an* FO *formula over strings. There is a first-order for-program which computes the following.*

- ***Input.*** *A word $w \in \Sigma^*$ and positions $x_1, \dots, x_k$ in w;*
- ***Output.*** *Yes or No, depending on whether w satisfies $\varphi(x_1, \dots, x_k)$.*

Proof. The for-program implements the semantics of an FO formula. For each quantifier, it loops over all possible values for the quantified position, and a Boolean variable is used to remember if some value has already been found which renders the formula true. ◀

A similar result is true for MSO formulas, but the proof for that statement uses automata.

4 Equivalence

We show that the models defined in Sections 2 and 3 are equivalent.

▶ **Theorem 7.**

1. *String-to-string* MSO *interpretations are exactly the polyregular functions.*
2. *First-order string-to-string interpretations are exactly the first-order polyregular functions.*

Since the class of polyregular functions is closed under composition[2], we obtain:

▶ **Corollary 8.** *String-to-string* MSO *interpretations are closed under composition.*

By using Theorem 7, the proof of the corollary passes through for-programs. We are not aware of any direct proof that does not exploit the equivalence to polyregular functions.

The rest of this paper is dedicated to the proof of Theorem 7. We begin with a reduction of the first to the second item. This reduction illustrates a general phenomenon, namely that results about first-order polyregular functions often imply results about general polyregular functions, despite the latter class being larger. The reason behind this phenomenon is the following lemma, which says that for every polyregular function, all of the behaviour that is not first-order definable can be pushed into a simple preprocessing step. Define a *rational function*, see [4, Section 13.2], to be a string-to-string function which is recognised by a nondeterministic automaton, where every transition is labelled by a pair consisting of a letter from the input alphabet and a string over the output alphabet, and which is unambiguous in the sense that every input string admits exactly one accepting run.

▶ **Lemma 9.**

1. *A function is polyregular if and only if it is a composition consisting of:*
   a. *a (letter-to-letter) rational function; followed by*
   b. *a first-order polyregular function.*
2. *A function is an* MSO *string-to-string interpretation if and only if it is a composition consisting of:*
   a. *a (letter-to-letter) rational function; followed by*
   b. *a first-order string-to-string interpretation.*

The proof of Lemma 9 can be found in [6]. It is based on ideas from [7, 15, 3] and uses factorisation forests. With the lemma, we show that item 2 in Theorem 7 implies item 1, i.e. if first-order string-to-string interpretations are exactly the first-order polyregular functions, then MSO interpretations are exactly the polyregular functions:

$$\begin{aligned} \text{polyregular} &= && \text{by item 1 of Lemma 9}\\ (\text{first-order polyregular}) \circ \text{rational} &= && \text{by item 2 of Theorem 7}\\ (\text{first-order interpretations}) \circ \text{rational} &= && \text{by item 2 of Lemma 9}\\ \text{MSO interpretations} & && \end{aligned}$$

It remains to prove item 2 in Theorem 7, i.e. that first-order string-to-string interpretations are exactly the first-order polyregular functions. The right-to-left inclusion follows immediately from [3, Lemma 5.3], which says that a formula in first-order logic can define the reachability relation on program states in first-order for-programs. We are left with the left-to-right-inclusion:

$$\text{first-order string-to-string interpretations} \subseteq \text{first-order definable for-programs} \qquad (1)$$

The rest of the paper is devoted to showing the above inclusion. When simulating a first-order interpretation by a for-program, we will mainly be concerned with the universe of the output string (which is a set of k-tuples of positions in the input string) and its ordering. The labelling of the k-tuples can then be recovered using the for-program from Lemma 6.

[2] Closure under composition was proved for pebble transducers in [10, Theorem 11] and for the class of for-programs in [3, Section 8.1] as a step in proving equivalence with the other models of polyregular functions.

The main result is that every first-order definable linear ordering on tuples of positions can be implemented by a for-program. To be able to speak about this result, we introduce some notation for devices that produce lists of tuples of positions.

Enumerators. Let $k \in \mathbb{N}$. A *k-enumerator* over an alphabet Σ is a function of the following form:

- **Input.** A string $w \in \Sigma^*$;
- **Output.** A list of k-tuples of positions in w, that is nonrepeating[3].

We compare the following two ways of implementing k-enumerators:

1. A k-enumerator is called *definable* if there are two FO formulas: one with k variables, which says when a tuple is part of the output list, and one with $2k$ variables, which defines a total order on the tuples selected by the first formula.
2. A k-enumerator is called *programmable* if its output can be computed by a first-order for-program that instead of outputting letters uses instructions of the form `output (x1,...,xk)` where `x1`,...,`xk` are position variables.

For definable k-enumerators, the order on tuples in the output list is given explicitly by the formula φ, while in programmable ones, the order is implicit from the order in which the output instructions are executed during the computation.

▶ **Example 10.** We present an enumerator based on Example 3. Consider the 2-enumerator which outputs all pairs of positions (x_1, x_2) with $x_2 \leq x_1$, listed in lexicographic order, where x_1 is ordered in increasing order and x_2 is ordered in decreasing order. Here is an example:

$$abbb \quad \mapsto \quad (1,1),(2,2),(2,1),(3,3),(3,2),(3,1),(4,4),(4,3),(4,2),(4,1)$$

This enumerator is definable, as witnessed by the formula $\varphi_{\leq}$ in Example 3. The formula $\varphi_{\leq}$ is quantifier-free, but in general, quantifiers are allowed. Here is a for-program that computes the same function:

```
for x1 in first..last
  for x2 in last..first
    if x2 ≤ x1 then
      output (x1,x2)
```

The following lemma is the main technical result of this paper.

▶ **Lemma 11.** *Every definable k-enumerator is also programmable.*

Our proof of Lemma 11 uses two fundamental ingredients. The first is by now standard: this is Simon's factorisation forest theorem [18], which roughly says that every string can be cut into pieces that are similar to each other. The second ingredient is new: the Domination Lemma, presented in Section 4.1, roughly says that if a string is cut into pieces that are similar to each other, then any first-order definable linear order on tuples of positions must observe an implicit stack discipline. These two results are combined in Section 4.2 to prove Lemma 11. Before we proceed with the proof of Lemma 11, we use it to complete the proof of Theorem 7.

[3] Every tuple appears at most once, but positions can appear in multiple tuples. We need this for the existence of the formulas stated in the following definitions.

Proof of Theorem 7, second part. The only part of Theorem 7 that has not been proved yet is that every first-order string-to-string interpretation is polyregular. Suppose that f is a k-dimensional first-order string-to-string interpretation. Consider the k-enumerator that inputs a string w and outputs the list of k-tuples of positions in w used to represent output positions of $f(w)$, in the appropriate order. Apply Lemma 11 to obtain a first-order for-program g which computes the same list. To compute the original function f, we use a for-program which behaves as g, except that instead of outputting a k-tuple of positions like g, it uses the program described in Lemma 6 as a subroutine to check what is the output letter that should be produced for this tuple, and outputs that letter. ◀

4.1 The Domination Lemma

In this section we present the Domination Lemma, which says that if $\prec$ is a first-order definable linear order on k-tuples of positions in a string, then there is an implicit stack discipline in the following sense. For every type (see below) t of tuples of positions there is a coordinate $d \in \{1, \ldots, k\}$ such that for the subset of k-tuples of positions formed by all of type t, the order $\prec$ is uniquely determined by the order of the d-th coordinates in the string.

We begin by explaining the notions of types. For $r \in \{0, 1, \ldots\}$, the *rank r type* of a structure $\mathfrak{A}$ with k distinguished positions $\bar{x} := (x_1, \ldots, x_k)$ is defined to be the set of first-order formulas of quantifier rank at most r and with k free variables that are true in $\mathfrak{A}, \bar{x}$. The number k is the *arity* of the type. For arity 0, we talk about the rank r *type of the structure* $\mathfrak{A}$. If the structure $\mathfrak{A}$ is implicit from the context, then we talk about the rank r type of the tuple $\bar{x}$. For every finite vocabulary, there are finitely many types of given arity and rank. We write $\equiv_r$ for the equivalence relation on structures with distinguished elements of having the same rank r type. For a binary relation R, its *inverse* is the set $\{(v, u) \mid (u, v) \in R\}$. For $p \in \{1, -1\}$, define R^p to be either R or its inverse, depending on the value of p.

▶ **Lemma 12** (Domination Lemma). *For all $k, m, r \in \{1, 2, \ldots\}$, there is an $\omega \in \{1, 2, \ldots\}$ with the following property. Let $n \in \{1, 2, \ldots\}$, let $w_1, \ldots, w_n$ be strings over some alphabet Σ and let $\mathfrak{A}$ be the ordered structure of the concatenation $w_1 \cdots w_n$ extended with the* block order *defined by*

$$x \sqsubset y \qquad \textit{if} \qquad x \textit{ is a position in } w_i \textit{ and } y \textit{ is a position in } w_j \textit{ for some } i < j.$$

Let $\prec$ be a linear order on k-tuples in $\mathfrak{A}$ defined by a first-order formula of quantifier rank r, and let t be a k-ary rank ω type over the vocabulary of $\mathfrak{A}$. If

$$w_i \equiv_\omega w_{i+1} \qquad \textit{holds for all } i \in \{1, \ldots, n-1\} \textit{ with at most } m \textit{ exceptions,}$$

then there is a $d \in \{1, \ldots, k\}$, called the dominating coordinate*, and a $p \in \{-1, 1\}$, called the* polarity*, such that*

$$x_d \sqsubset^p y_d \quad \textit{implies} \quad (x_1, \ldots, x_k) \prec (y_1, \ldots, y_k) \qquad \textit{for all } \underbrace{x_1, \ldots, x_k}_{\textit{of type } t}, \underbrace{y_1, \ldots, y_k}_{\textit{of type } t} \textit{ in } \mathfrak{A}.$$

The Domination Lemma is the technical heart of this paper. The full proof can be found in [6]. To explain some of the ideas that we use, we treat a special case in detail. In the Domination Lemma, the structure $\mathfrak{A}$ consists of blocks organised in a linear way. A very simple linear order – although infinite – is the natural one on the rational numbers; one reason for its simplicity is that quantifiers can be eliminated (see [13, Section 5.6.2]). Because of this, it is quite easy to prove a version of the Domination Lemma for the rational numbers and still its proof bears some similarity to the proof of the general case.

▶ **Lemma 13** (Rational Domination Lemma). *Let $\prec$ be a linear ordering on k-tuples of rational numbers defined by a quantifier-free (equivalently, first-order) formula using only the usual ordering $<$ on rational numbers. Then there is a coordinate $d \in \{1, \ldots, k\}$ and a polarity $p \in \{-1, 1\}$ such that*

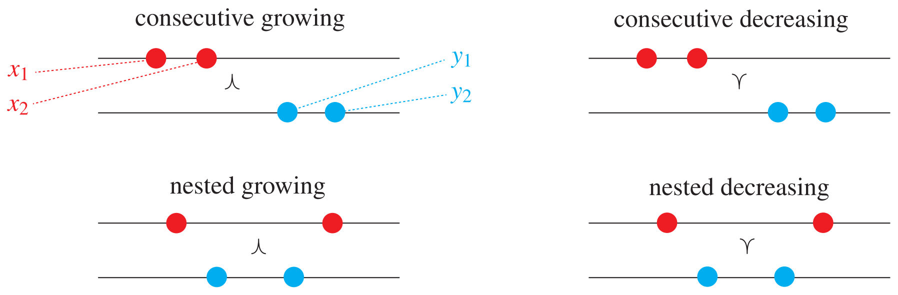
$$x_d <^p y_d \quad \textit{implies} \quad (x_1, \ldots, x_k) \prec (y_1, \ldots, y_k)$$

for all tuples of rational numbers satisfying $x_1 < \cdots < x_k$ and $y_1 < \cdots < y_k$.

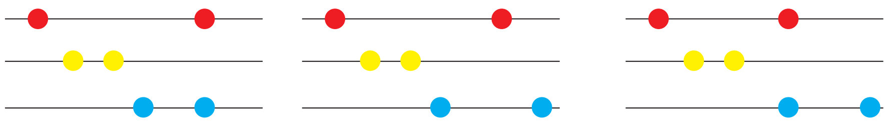
Proof. We first prove the statement for $k = 1$ and $k = 2$ and then we deduce the general case.

1. When $k = 1$, then the formula defining $\prec$ must be either $x < y$ or $x > y$.
2. For $k = 2$, we do a case analysis. Note that whether $\bar{x} \prec \bar{y}$ or $\bar{y} \prec \bar{x}$ holds depends only on the order relationship of the positions in $\bar{x}$ and $\bar{y}$ in the rational numbers and not on the precise values in $\bar{x}$ and $\bar{y}$.

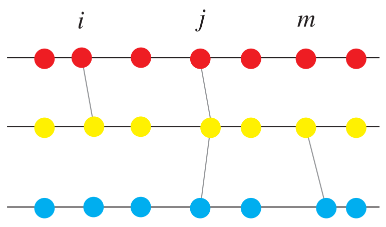
   The following picture shows the two possible relationships for two pairs $\bar{x}$ and $\bar{y}$ when they are "consecutive" and the two possible relationships when they are "nested":

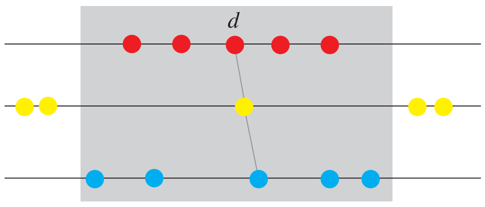
   Suppose we are given a pair $\bar{x}$ and without loss of generality, assume the "consecutive growing" case for some second pair $\bar{y}$. We only show the proof for the case that there is a pair $\bar{y}'$ such that $\bar{x}$ and $\bar{y}$ are "nested growing" ("nested decreasing" works analogously). We prove that $d = 1$ is dominating for $\prec$ with polarity $p = 1$. Consider all three remaining configurations of pairs $\bar{x}$ and $\bar{y}$ with $x_1 < y_1$. In all cases, $\bar{x} \prec \bar{y}$ is proved by finding an intermediate pair (drawn in yellow), whose order with respect to $\bar{x}$ and $\bar{y}$ follows from the assumptions "consecutive/nested growing" (in the pictures below, we assume that lower lines represent bigger tuples in the ordering $\prec$):

3. Consider the case $k > 2$. Fix a "growing" tuple of k rational numbers, i.e. a tuple $\bar{z}$ such that for $1 \leq i < j \leq k$, it holds that $z_1 \leq z_i < z_j \leq z_k$. Define $\prec_{ij}^{\bar{z}}$ to be the restriction of $\prec$ to tuples that agree with $\bar{z}$ on coordinates from $\{1, \ldots, k\} \setminus \{i, j\}$. Using the reasoning from the previous item, the ordering $\prec_{ij}^{\bar{z}}$ must admit some dominating coordinate $d \in \{i, j\}$ and one of the cases "growing" or "decreasing". This must hold for every choice of $\bar{z}$ and i, j. Furthermore, the dominating coordinate d depends only on i and j and not on $\bar{z}$, likewise for the choice of "growing" or "decreasing". Let us write $i \to j$ if j dominates, otherwise we write $j \to i$. The reasoning in the following picture shows that $\to$ is transitive, i.e. $i \to j$ and $j \to m$ implies $i \to m$:


i
j
m
Step 1. Move coordinate i to its final position, and move j so that the tuple grows
Step 2. Move coordinate j back to the initial, position and move m to its final position

Therefore, $\to$ is in fact a total order on $\{1, \dots, k\}$. Let d be the maximum with respect to this order. The following picture explains why d is the dominating coordinate d from the statement of Lemma 12.


d
Step 1. For every $i \neq d$, use $i \to d$ to move coordinates $i \neq d$ outside the gray interval (the gray interval contains $\bar{x}$ and $\bar{y}$)
Step 2. Move coordinates $i \neq d$ to their final positions

Suppose without loss of generality that we are in the "growing" case for each pair of coordinates. Then we can first move all coordinates apart from d to positions smaller than $\min\{x_1, y_1\}$ or bigger than $\max\{x_k, y_k\}$ and then use the dominations $i \to d$ to move them, one by one, to their final positions (always increasing the d-th coordinate slightly to a value in the open interval (x_d, y_d)). ◀

4.2 Proof of Lemma 11

We now return to Lemma 11, i.e., we prove that every definable k-enumerator is also programmable. In the proof, we use the following version of the Factorisation Forest Theorem. We use the term *interval* for a connected set of positions in a string.

▶ **Theorem 14** (Factorisation Forest Theorem, aperiodic variant). *Let $h \colon \Sigma^+ \to S$ be a semigroup homomorphism, where S is finite and aperiodic. Then there exists a function which assigns to each string in Σ^+ a partition of the positions into intervals (so-called* blocks*) such that:*

1. *All blocks are nonempty, and for each string in Σ^+ of length at least 2, there are at least two blocks.*
2. *If a string has at least three blocks, then all of the blocks have the same value under h.*
3. *There exists $M \in \mathbb{N}$ such that all strings have height at most M, where the height of a string is defined as follows: letters have height 1, for other strings the height is the maximum of the heights of its blocks + 1.*
4. *There is a first-order formula φ such that for every string w, the positions satisfying $\varphi(x)$ are exactly the first positions of the blocks of w.*

Apart from the Factorisation Forest Theorem and the Domination Lemma, our proof uses the following straightforward result on combining outputs of two for-programs. As a convention, if ψ is a first-order formula with k free variables and f is a k-enumerator, then $f|\psi$ denotes the k-enumerator where the output list of f is filtered so that it contains only tuples satisfying ψ.

▶ **Lemma 15** (Merging Lemma). *Let f be a definable k-enumerator. Let Φ be a finite set of* FO *formulas ψ, each one with k free variables, such that every k-tuple of positions satisfies at least one formula from Φ. Then f is programmable if and only if every $f|\psi$ is programmable.*

We are now ready to prove Lemma 11. Let f be a definable k-enumerator. We need to describe a for-program that outputs the same list of tuples as f. Let r be the maximal quantifier rank of the first-order formulas used in the definition of f. Apply the Domination Lemma to k, $m := 5k$, and r, yielding a constant ω. Define h to be the function which maps a string $w \in \Sigma^+$ to the rank ω type of the corresponding ordered model of w. Compositionality of first-order logic (see [16, Section 3.4]) on strings says that the image of h, the set of rank ω types of strings, is a finite aperiodic semigroup and h is a semigroup homomorphism. Apply the Factorisation Forest Theorem to h, yielding a function that partitions each string into blocks and an upper bound M on heights of strings. By abuse of notation, we lift notions about strings to intervals inside strings: the height of an interval X in a string w is defined to be the height (in the sense of item 3 in Theorem 14) of the infix of w induced by X. Likewise, we define the blocks of X as the blocks of the infix induced by X, viewed as intervals contained in X.

To show that f is also programmable, we use an induction over heights in factorisation forests. More precisely, we prove that for every $i \in \mathbb{N}$ there is a for-program which computes the following:

- **Input.** A string $w \in \Sigma^+$ with distinguished nonempty intervals $X_1, \ldots, X_k$ that are pairwise equal or disjoint, and such that the sum of their heights (in the sense of Theorem 14) is at most i. Each interval is represented by its first and its last position.
- **Output.** The list $f(w)$ restricted to tuples in $X_1 \times \cdots \times X_k$.

By item 3 in Theorem 14, the for-program with parameter $i := kM$ will work for every choice of pairwise equal or disjoint intervals, in particular when all of the intervals are the entire string. The induction base $i = k$ (where every interval has the height 1) is straightforward: each interval is a singleton, and the for-program simply checks if the unique tuple in $X_1 \times \cdots \times X_k$ belongs to the output of f by using the subroutines from Lemma 6. The rest of the proof is devoted to the induction step, more specifically, to producing the correct order of the tuples: whether a tuple belongs to the output or not can again be checked using the subroutines from Lemma 6.

Let $X_1, \ldots, X_k$ be intervals in an input string w that are pairwise disjoint or equal. Define $\mathcal{X}$ to be the coarsest partition of the positions in the input string into intervals that satisfies $X_1, \ldots, X_k \in \mathcal{X}$. This partition uses at most $2k + 1$ intervals. Consider a factorisation

$$w = w_1 \cdots w_n$$

where each w_j is a block of one of the elements of $\mathcal{X}$. Define $\mathfrak{A}$ as in the Domination Lemma, i.e. as the ordered structure of w extended with an extra order $\sqsubset$ that describes the partition into factors $w_1, \ldots, w_n$. By item 4 of the Factorisation Forest Theorem, the order $\sqsubset$ can be defined by a first-order formula which uses the input string and the endpoints of the intervals $X_1, \ldots, X_k$. It follows that for every k-ary rank ω type t over the vocabulary of $\mathfrak{A}$, there is a corresponding first-order formula that selects the k-tuples of positions in w that have type t in $\mathfrak{A}$. Since there are finitely many choices of t, it follows from the Merging Lemma that it is enough to show that for every t, there is a for-program which outputs the tuples of type t.

Let t be a k-ary rank ω type over the vocabulary of $\mathfrak{A}$. We show a for-program which outputs all tuples in

$$T := \{\bar{x} \in X_1 \times \cdots \times X_k : \bar{x} \text{ has type } t \text{ and is in the output of } f(w)\}$$

according to their order given by $f(w)$, call this order $\prec$.

If an interval from $\mathcal{X}$ has more than two blocks, then, by item 2 of the Factorisation Forest Theorem, all of these blocks have the same image under h, i.e., the same rank ω type. Since there are at most $2k+1$ intervals, it follows that with at most $2(2k+1)-1 = 4k+1 < 5k$ exceptions, consecutive strings w_j and w_{j+1} have the same rank ω type. Hence, for the order $\prec$ defined by $f(w)$, the Domination Lemma yields $d \in \{1,\dots,k\}$ and $p \in \{-1,1\}$ such that

$$x_d \sqsubset^p y_d \quad \text{implies} \quad (x_1,\dots,x_k) \prec (y_1,\dots,y_k) \qquad \text{for all } \underbrace{x_1,\dots,x_k}_{\text{of type } t}, \underbrace{y_1,\dots,y_k}_{\text{of type } t} \text{ in } \mathfrak{A}.$$

This means that the tuples in T are $\prec$-ordered as $T_1 \prec^p T_2 \prec^p \cdots \prec^p T_s$, where s is the number of blocks in X_d and T_j consists of the tuples from T where the coordinate x_d is in the j-th block of X_d. Our for-program can simply loop over all the blocks of X_d – in increasing or decreasing order depending on the choice of p – because the endpoints of each block can be identified in first-order logic due to item 4 of the Factorisation Forest Theorem. In each iteration of the loop, the for-program outputs the tuples in the corresponding T_j using the following claim, thus completing the proof of the lemma.

▷ Claim 16. There is a for-program which inputs the i-th block of X_d, given by its endpoints, and outputs the tuples from T_j ordered according to $\prec$.

Proof of the claim. The general idea is to replace X_d with its j-th block (call this block X) and use the induction assumption. However, if there is some $j \neq d$ such that $X_j = X_d$, then replacing X_d with X would violate the assumption that the intervals are pairwise disjoint or equal (since $X \subsetneq X_j$). To overcome this issue, we use the following simple case disjunction. For each of the 3^k possible values of

$$v \in \{\text{positions before } X,\ X,\ \text{positions after } X\}^k$$

construct a for-program that outputs all tuples from $Y_1 \times \cdots \times Y_k$, where Y_j is the intersection of X_j with the j-th entry of v. Since each Y_j is a union of blocks of X_j, it is empty or its height is at most the height of X_j. Furthermore, if Y_d is nonempty, then it is X, which is a block of X_d, and therefore its height is strictly smaller than the height of X_d. It follows that the induction assumption can be applied to produce all tuples in $Y_1 \times \cdots \times Y_k$, for any given choice of v. These choices can be combined using the Merging Lemma. ◁

References

1 Rajeev Alur and Pavol Cerný. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 599–610, 2011. doi:10.1145/1926385.1926454.

2 Rajeev Alur, Adam Freilich, and Mukund Raghothaman. Regular combinators for string transformations. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, page 9. ACM, 2014.

3 Mikołaj Bojańczyk. Polyregular Functions, 2018. arXiv:1810.08760.

4 Mikołaj Bojańczyk and Wojciech Czerwiński. Automata Toolbox. URL: https://www.mimuw.edu.pl/~bojan/upload/reduced-may-25.pdf.

5 Mikołaj Bojańczyk, Laure Daviaud, and Shankara Narayanan Krishna. Regular and First-Order List Functions. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 125–134, 2018. doi:10.1145/3209108.3209163.

6 Mikołaj Bojańczyk, Sandra Kiefer, and Nathan Lhote. String-to-string interpretations with polynomial-size output, 2019. arXiv:1905.13190.

7 Thomas Colcombet. A combinatorial theorem for trees. In *International Colloquium on Automata, Languages, and Programming*, pages 901–912. Springer, 2007.

8 Bruno Courcelle and Joost Engelfriet. *Graph Structure and Monadic Second-Order Logic*. A Language-Theoretic Approach. Cambridge University Press, June 2012.

9 Vrunda Dave, Paul Gastin, and Shankara Narayanan Krishna. Regular Transducer Expressions for Regular Transformations. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 315–324, 2018. doi:10.1145/3209108.3209182.

10 Joost Engelfriet. Two-way pebble transducers for partial functions and their composition. *Acta Informatica*, 52(7-8):559–571, 2015.

11 Joost Engelfriet and Hendrik Jan Hoogeboom. MSO definable string transductions and two-way finite-state transducers. *ACM Transactions on Computational Logic (TOCL)*, 2(2):216–254, 2001.

12 Noa Globerman and David Harel. Complexity Results for Two-Way and Multi-Pebble Automata and their Logics. *Theor. Comput. Sci.*, 169(2):161–184, 1996.

13 Erich Grädel, Phokion G. Kolaitis, Leonid Libkin, Maarten Marx, Joel Spencer, Moshe Y. Vardi, Yde Venema, and Scott Weinstein. *Finite Model Theory and Its Applications*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2007. doi:10.1007/3-540-68804-8.

14 Wilfrid Hodges. *Model Theory*. Cambridge University Press, Cambridge, March 1993.

15 Wojciech Kazana and Luc Segoufin. Enumeration of monadic second-order queries on trees. *ACM Transactions on Computational Logic (TOCL)*, 14(4):1–12, 2013.

16 Leonid Libkin. *Elements of finite model theory*. Springer Science & Business Media, 2013.

17 Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for XML transformers. *Journal of Computer and System Sciences*, 66(1):66–97, 2003.

18 Imre Simon. Factorization forests of finite height. *Theoretical Computer Science*, 72(1):65–94, 1990.

19 Wolfgang Thomas. Languages, Automata, and Logic. In *Handbook of Formal Languages, Volume 3: Beyond Words*, pages 389–455. Springer, 1997. doi:10.1007/978-3-642-59126-6_7.