Regular and First-order List Functions

Mikołaj Bojańczyk[1], Laure Daviaud[2], and S. Krishna[3]

1 **MIMUW, University of Warsaw, Poland**
2 **DIMAP, Department of Computer Science, University of Warwick, UK**
3 **Department of Computer Science, IIT Bombay, India**

Abstract.

We define two classes of functions, called *regular (respectively, first-order) list functions*, which manipulate objects such as lists, lists of lists, pairs of lists, lists of pairs of lists, etc. The definition is in the style of regular expressions: the functions are constructed by starting with some basic functions (e.g. projections from pairs, or head and tail operations on lists) and putting them together using four combinators (most importantly, composition of functions). Our main results are that first-order list functions are exactly the same as first-order transductions, under a suitable encoding of the inputs; and the regular list functions are exactly the same as MSO-transductions.

1 Introduction

Transducers, i.e. automata which produce output, are as old as automata themselves, appearing already in Shannon's paper[20, Section 8]. This paper is mainly about string-to-string transducers. Historically, the most studied classes of string-to-string functions were the sequential and rational functions, see e.g. [15, Section 4] or [19]. Recently, much attention has been devoted to a third, larger, class of string-to-string functions that we call "regular" following [13] and [6]. The regular string-to-string functions are those recognised by two-way automata [1], equivalently by MSO transductions [13], equivalently by streaming string transducers [6].

In [5], Alur et al. give yet another characterisation of the regular string-to-string functions, in the spirit of regular expressions. They identify several basic string-to-string functions, and several ways of combining existing functions to create new ones, in such a way that exactly the regular functions are generated. The goal of this paper is to do the same, but with a different choice of basic functions and combinators. Below we describe some of the differences between our approach and that of [5].

The first distinguishing feature of our approach is that, instead of considering only functions from strings to strings, we allow a richer type system, where functions can manipulate objects such as pairs, lists of pairs, pairs of lists etc. (Importantly, the nesting of types is bounded, which means that the objects can be viewed as unranked sibling ordered trees of bounded depth.) This richer type system is a form of syntactic sugar, because the new types can be encoded using strings over a finite alphabet, e.g. $[(a,[b]),(b,[a,a]),(a,[])]$, and the devices we study are powerful enough to operate on such encodings. Nevertheless, we believe that the richer type system allows us to identify a more natural and canonical base of functions, with benign functions such as projection $\Sigma \times \Gamma \to \Sigma$, or append $\Sigma \times \Sigma^* \to \Sigma^*$. Another advantage of the type system is its tight connection with programming: since we use standard types and functions on them, our entire set of basic functions and combinators can be implemented in one screenful of Haskell code, consisting mainly of giving new names to existing operations.

A second distinguishing property of our approach is its emphasis on composition of functions. Regular string-to-string functions are closed under composition, and therefore it is natural to add composition of functions as a combinator. However, the system of Alur et al. is designed so that composition is allowed but not needed to get the completeness result [5, Theorem 15]. In contrast, composition is absolutely essential to our system. We believe that having the ability to simply compose functions – which is both intuitive and powerful – is one of the central appeals of

transducers, in contrast to other fields of formal language theory where more convoluted forms of composition are needed, such as wreath products of semigroups or nesting of languages. With composition, we can leverage deep decomposition results from the algebraic literature (e.g. the Krohn-Rhodes Theorem or Simon's Factorisation Forest Theorem), and obtain a basis with very simple atomic operations.

Apart from being easily programmable and relying on composition, our system has two other design goals. The first goal is that we want it to be easily extensible; we discuss this goal in the conclusions. The second goal is that we want to identify the iteration mechanisms needed for regular string-to-string functions.

To better understand the role of iteration, the technical focus of the paper is on the first-order fragment of regular string-to-string functions [15, Section 4.3]. Our main technical result, Theorem 13, shows a family of atomic functions and combinators that describes exactly the first-order fragment. We believe that the first-order fragment is arguably as important as the bigger set of regular functions. Because of the first-order restriction, some well known sources of iteration, such as modulo counting, are not needed for the first-order fragment. In fact, one could say that the functions from Theorem 13 have no iteration at all (of course, this can be debated). Nevertheless, despite this lack of iteration, the first-order fragment seems to contain the essence of regular string-to-string functions. In particular, our second main result, which characterises all regular string-to-string functions in terms of combinators, is obtained by adding product operations for finite groups to the basic functions and then simply applying Theorem 13 and existing decomposition results from language theory.

Organisation of the paper.

In Section 2, we define the class of first-order list functions. In Section 3, we give many examples of such functions. One of our main results is that the class of first-order list functions is exactly the class of first-order transductions. To prove this, we first show in Section 4 that first-order list functions contain all the aperiodic rational functions. Then, in Sections 5 and 6, we state the result and complete its proof. In Section 7, we generalise our result to deal with MSO-transductions. We conclude the paper with future works in Section 8.

2 Definitions

We use types that are built starting from finite sets (or even one element sets) and using disjoint unions (co-products), products and lists. More precisely, the set of types we consider is given by the following grammar:

$$\mathscr{T} := \text{every one-element set} | \mathscr{T} + \mathscr{T} | \mathscr{T} \times \mathscr{T} | \mathscr{T}^*$$

For example, starting from elements a of type $\Sigma \in \mathscr{T}$ and b of type $\Gamma \in \mathscr{T}$, one can construct the co-product $\{a, b\}$ of type $\Sigma + \Gamma$, the product (a, b) of type $\Sigma \times \Gamma$, and the following lists $[a, a, a]$ of type Σ^* and $[a, a, b, b, a, a, b]$ of type $(\Sigma + \Gamma)^*$.

For Σ in $\mathscr{T}$, we define Σ^+ to be $\Sigma \times \Sigma^*$.

2.1 First-order list functions

The class of functions studied in this paper, which we call *first-order list functions* are functions on the objects defined by the above grammar. It is meant to be large enough to contain natural functions such as projections or head and tail of a list, and yet small enough to have good computational properties (very efficient evaluation, decidable equivalence, etc.). The class is defined by choosing some basic list functions and then applying some combinators.

Definition 1 (First-order list functions)**.** Define the *first-order list functions* to be the smallest class of functions having as domain and co-domain any Σ from $\mathscr{T}$, which contains all the constant functions, the functions from Figure 1 (projection, co − projection and distribute), the functions from Figure 2 (reverse, flat, append, co − append and block) and which is closed under applying the disjoint union, composition, map and pairing combinators defined in Figure 3.

• $\textbf{projection}_1$	• coprojection	• distribute
$\Sigma \times \Gamma \longrightarrow \Sigma$	$\Sigma \longrightarrow \Sigma + \Gamma$	$(\Sigma + \Gamma) \times \Delta \longrightarrow (\Sigma \times \Delta) + (\Gamma \times \Delta)$
$(x, y) \longmapsto x$	$x \longmapsto x$	$(x, y) \longmapsto (x, y)$

Figure 1 Basic functions for product and co-product. To avoid clutter, we write only one of the two projection functions but we allow to use both. The types Σ, Γ, Δ are from $\mathscr{T}$.

3 Examples

Natural functions such as identity, functions on finite sets, concatenation of lists, extracting the first element (head), the last element and the tail of a list,... are first order list functions. In this section, we present those examples and prove that they are first-order list functions. Some of these examples will be used in later constructions.

Example 1. [Identity] For every Σ in $\mathscr{T}$, the identity function: $x \in \Sigma \mapsto x \in \Sigma$ is a first-order list function. This is achieved by induction on the types: the identity function over a one set element is a constant function and thus a first-order list function. For Σ and Γ in $\mathscr{T}$, the identity function over $\Sigma + \Gamma$ is the disjoint union of the co-projections $\Sigma \to \Sigma + \Gamma$ and $\Gamma \to \Sigma + \Gamma$. The identity function over $\Sigma \times \Gamma$ is the pairing of the projections $\Sigma \times \Gamma \to \Sigma$ and $\Sigma \times \Gamma \to \Gamma$. Finally, the identity function over Σ^* is constructed from the identity function over Σ using the Map combinator. □

Example 2. [List unit function] For every Σ in $\mathscr{T}$, the function:

$$x \in \Sigma \mapsto [x] \in \Sigma^*$$

is a first-order list function. This is achieved by first using the pairing combinator which pairs the identity function on Σ: $f(x) = x$ and the constant function g which maps every $x \in \Sigma$ to the empty list [] of type Σ^*. The pairing combinator gives $(x, [])$ from x. This is followed by using the append function on this pair, giving as result $[x]$. □

Example 3. [Functions on finite sets] If Σ, Γ are finite sets, then every function of type $\Sigma \to \Gamma$ is a first-order list function. This is done by viewing Σ as a disjoint union of one-element sets, and then combining constant functions using the disjoint union combinator. □

Example 4. [Heads, Last element and Tails] For every $[x_0, x_1, \ldots, x_n] \in \Sigma^*$, the function $\mathsf{head} : \Sigma^* \to \Sigma$ which extracts the head $x_0 \in \Sigma$ from the list, the function $\mathsf{last} : \Sigma^* \to \Sigma$ which extracts the last element $x_n \in \Sigma$ from the list as well as the function $\mathsf{tail} : \Sigma^* \to \Sigma^*$ which extracts the tail $[x_1, \ldots, x_n]$ from the list are first-order list functions. To see this, head is obtained by first using co − append to $[x_0, x_1, \ldots, x_n]$ obtaining the pair $(x_0, [x_1, \ldots, x_n])$ and then projecting out the first component. Likewise, tail is obtained by projecting the second component. The last element is obtained by reversing the list first, and using head. □

- **Reverse.**

  $\mathsf{reverse}: \quad \Sigma^* \longrightarrow \Sigma^*$

  $[w_1, \ldots, w_n] \longmapsto [w_n, \ldots, w_1]$

- **Flat.**

  $\mathsf{flat}: \quad \Sigma^{**} \longrightarrow \Sigma^*$

  $$[w_1, \ldots, w_n] \longmapsto \begin{cases} [] & \text{if } n = 0 \\ w_1 \cdot \mathsf{flat}([w_2, \ldots, w_n]) & \text{otherwise} \end{cases}$$

  where $\cdot$ denotes concatenation on lists of the same type.
  For example, for a, b, c, d, e of the same type, $[[a, b], [c]] \cdot [[d], [e]] = [[a, b], [c], [d], [e]]$ and:

  $$\mathsf{flat}([[a, b], [c]]) = [a, b] \cdot \mathsf{flat}([[c]]) = [a, b] \cdot [c] \cdot [] = [a, b, c]$$

  Remark that $\cdot$ is not proved yet to be a first-order list function. It is the case, see Example 9 below.

- **Append.**

  $\mathsf{append}: \quad \Sigma \times \Sigma^* \longrightarrow \Sigma^*$

  $(x_0, [x_1, \ldots, x_n]) \longmapsto [x_0, x_1, \ldots, x_n]$

- **Co-append.**

  $\mathsf{co-append}: \quad \Sigma^* \longrightarrow (\Sigma \times \Sigma^*) + \bot$

  $$[x_0, \ldots, x_n] \longmapsto \begin{cases} (x_0, [x_1, \ldots, x_n]) & \text{if } n \geq 1 \\ \bot & \text{otherwise} \end{cases}$$

  where $\bot$ is a new element (also a special type).

- **Block.**

  $\mathsf{block}: \quad (\Sigma + \Gamma)^* \longrightarrow (\Sigma^* + \Gamma^*)^*$

  $x \longmapsto$ the unique list w such that $\mathsf{flat}(\mathsf{w}) = x$ and which alternates between Σ^+ and Γ^+

  For a, b of type Σ and c, d of type Γ,

  $$\mathsf{block}([a, b, c, d, a, d, a, b, c, d]) = [[a, b], [c, d], [a], [d], [a, b], [c, d]]$$

Figure 2 Basic functions for lists. The types Σ, Γ, Δ are from $\mathscr{T}$.

- **Disjoint union.**

$$\frac{f : \Sigma \longrightarrow \Delta \qquad g : \Gamma \longrightarrow \Delta}{f + g : \Sigma + \Gamma \longrightarrow \Delta} \qquad x \mapsto \begin{cases} f(x) & \text{if } x \in \Sigma \\ g(x) & \text{if } x \in \Gamma \end{cases}$$

- **Composition.**

$$\frac{f : \Sigma \longrightarrow \Gamma \qquad g : \Gamma \longrightarrow \Delta}{g \circ f : \Sigma \longrightarrow \Delta} \qquad x \mapsto g(f(x))$$

- **Map.**

$$\frac{f : \Sigma \longrightarrow \Gamma}{f^* : \Sigma^* \longrightarrow \Gamma^*} \qquad [x_1, \ldots, x_n] \mapsto \begin{cases} [f(x_1), \ldots, f(x_n)] & \text{if } n > 0 \\ [] & \text{if } n = 0 \end{cases}$$

- **Pairing.**

$$\frac{f : \Sigma \longrightarrow \Gamma \qquad g : \Sigma \longrightarrow \Delta}{(f, g) : \Sigma \longrightarrow \Gamma \times \Delta} \qquad x \mapsto (f(x), g(x))$$

Figure 3 Combinators of functions. The types Σ, Γ, Δ are from $\mathscr{T}$.

Example 5. [Length up to a threshold] For every set Σ and $n \in \mathbb{N}$,

$$\mathsf{len}_n : \qquad \Sigma^* \to \{0, \ldots, n\}$$
$$[x_1, \ldots, x_i] \mapsto \begin{cases} i & \text{if } i \leq n \\ n & \text{otherwise} \end{cases}$$

is a first-order list function. The proof is by induction on n. The function $\mathsf{len}_0 : \Sigma^* \to 0$ is constant, and therefore it is a first-order list function. For $n > 0$, we first apply tail. This is composed with the function len_{n-1} from the induction assumption, and then further composed with the following function: $x \in \{0, \ldots, n-1\} \mapsto x + 1 \in \{1, \ldots, n\}$ which is a first-order list function by Example 3. For example,

$$\mathsf{len}_2([x_1, x_2, x_3]) = \mathsf{len}_1(\mathsf{tail}([x_1, x_2, x_3])) + 1 = \mathsf{len}_0(\mathsf{tail}([x_2, x_3])) + 1 + 1 = 2.$$

□

Example 6. [Filter] For $\Sigma, \Gamma \in \mathscr{T}$, consider the function:

$$f : (\Sigma + \Gamma)^* \to \Sigma^*$$

which removes the Γ elements from the input list. Let us explain why this is a first-order list function. Consider the function from $\Sigma + \Gamma$ to Σ^*:

$$a \mapsto \begin{cases} [a] & \text{if } a \in \Sigma \\ [] & \text{otherwise} \end{cases} \tag{1}$$

which is the disjoint union of the unit function (Example 2) from Σ and of the constant function which maps every element of Γ to the empty list $[]$ of type Σ^*. Using map, we apply this function to all the elements of the input list, and then by applying the flat function, we obtain the desired

result. For example, if $\Sigma = \{a, b, c\}$ and $\Gamma = \{d, e\}$, consider the list $[a, c, d, e, b, e, d, a]$ in $(\Sigma+\Gamma)^*$. Using map of the above function to this list gives $[[a], [c], [], [], [b], [], [], [a]]$, which after using the function flat, gives $[a, c, b, a]$. Note that for the types to match, it is important to that the empty list in (1) is of type Σ^*. □

Example 7. [List comma function] For Σ, Γ in $\mathscr{T}$, consider the function:

$$(\Sigma + \Gamma)^* \to \Sigma^{**}$$

which groups elements from Σ into lists, with elements of Γ playing the role of list separators, as in the following example, where $\Sigma = \{a, b, c\}$ and $\Gamma = \{\#\}$:

$$[a, b, \#, c, \#, \#, a, \#, \#, \#, b, c, \#] \mapsto [[a, b], [c], [], [a], [], [], [b, c], []]$$

Let us prove that this function is a first order list function. We will freely use the identity function, the disjoint union and the map combinators without necessarily mentioning it.

1. We first apply the block function obtaining:

   $$[[a, b], [\#], [c], [\#, \#], [a], [\#, \#, \#], [b, c], [\#]]$$

2. Next, we apply tail to the elements of Γ^*, obtaining:

   $$[[a, b], [], [c], [\#], [a], [\#, \#], [b, c], []]$$

   Remark that the empty lists have type Γ^*.
3. This is followed by applying the list unit function (Example 2) to the elements of Σ^* obtaining type Σ^{**}. As a result, we have:

   $$[[[a, b]], [], [[c]], [\#], [[a]], [\#, \#], [[b, c]], []]$$

4. Next, we apply the list unit function (Example 2) to all the elements in a list from Γ^* (using two nested map) obtaining type Γ^{**}. This gives:

   $$[[[a, b]], [], [[c]], [[\#]], [[a]], [[\#], [\#]], [[b, c]], []]$$

   Remark that now, the empty lists have type Γ^{**}.
5. Finally, we transform every list of Γ^{**} into a list of Σ^{**}, by mapping every non empty list of Γ^* to the empty list of Σ^*. The example gives:

   $$[[[a, b]], [], [[c]], [[]], [[a]], [[], []], [[b, c]], []]$$

   Remark now that all the elements are of type Σ^{**}.
6. Finally, if the first and last elements are the empty list of Σ^{**}, they are mapped to $[[]]$ of Σ^{**}, by using head, last, append, $\mathsf{co-append}$, the pairing combinator and reversing the list. The example gives then:

   $$[[[a, b]], [], [[c]], [[]], [[a]], [[], []], [[b, c]], [[]]]$$

It is now sufficient to use flat to obtain the desired result. □

Example 8. [Pair to list] We can convert a pair to list of length two as follows: use the pairing combinator on the two following functions:

$(x, y) \mapsto x$	obtained using $\mathsf{projection}_1$
and $(x, y) \mapsto [y]$	obtained by composition of $\mathsf{projection}_2$
	and the list unit function (Example 2)

in order to get $(x, [y])$. Then, append gives $[x, y]$.

To get the converse translation, we use first $\mathsf{co-append}$ on $[x, y]$ to get $(x, [y])$. This is followed by pairing the two functions:

$(x, [y]) \mapsto x$	obtained using $\mathsf{projection}_1$
and $(x, [y]) \mapsto y$	obtained by composition of $\mathsf{projection}_2$ and the function head (Example 4)

to get (x, y).

Note that for the second translation, the type of the output is $(\Sigma \times (\Sigma + \bot)) + \bot$. If we want the output type to be $\Sigma \times \Sigma$, then we can choose some element $c \in \Sigma$ and send the first $\bot$ to c and the second $\bot$ to (c, c). In this case, the resulting function will satisfy:

$$[x_1, \ldots, x_n] \mapsto \begin{cases} (c, c) & \text{for } n = 0 \\ (x_1, c) & \text{for } n = 1 \\ (x_1, x_2) & \text{otherwise.} \end{cases}$$

□

Example 9. [List concatenation] List concatenation is a first-order list function: consider $([x_1, \ldots, x_n],[y_1, \ldots, y_k])$, a pair of lists and apply Example 8 to obtain $[[x_1, \ldots, x_n], [y_1, \ldots, y_k]]$ and then flat to get: $[x_1, \ldots, x_n, y_1, \ldots, y_k]$. □

Example 10. [Windows of size 2] For every Σ in $\mathscr{T}$, the following is a first-order list function:

$$[x_1, \ldots, x_n] \in \Sigma^* \mapsto [(x_1, x_2), (x_2, x_3), \ldots, (x_{n-1}, x_n)] \in (\Sigma \times \Sigma)^*$$

(When the input has length at most 1, then the output is empty.) Let us show how to get the above function. Consider a list $[x_1, \ldots, x_n]$. Using the same ideas as in Example 2, the following function is a first-order list function:

$$x \in \Sigma \qquad \mapsto \qquad [x, \#, x] \in (\Sigma + \{\#\})^*.$$

Apply the above function to every element of the input list, and then use flat on the result, yielding a list of type $(\Sigma + \{\#\})^*$ of the form:

$$[x_1, \#, x_1, x_2, \#, x_2, \ldots, x_n, \#, x_n]$$

Apply the list comma function from Example 7, to get a list of the form:

$$[[x_1], [x_1, x_2], [x_2, x_3], \ldots, [x_{n-1}, x_n], [x_n]]$$

Remove the first and last elements (using head and last), yielding a list of the form:

$$[[x_1, x_2], [x_2, x_3], \ldots, [x_{n-1}, x_n]].$$

Finally, apply the function from Example 8 to each element, yielding the desired list:

$$[(x_1, x_2), (x_2, x_3), \ldots, (x_{n-1}, x_n)].$$

The ideas in this example can be extended to produce windows of size 3,4, etc. □

Example 11. [If then else] Suppose that $f : \Sigma \to \{0, 1\}$ and $g_0, g_1 : \Sigma \to \Gamma$ are first-order list functions. Then $x \mapsto g_{f(x)}(x)$ is also a first-order list function. This is done as follows. On input $x \in \Sigma$, we first apply the pairing of f and the identity function, yielding a result:

$$(f(x), x) \in \{0, 1\} \times \Sigma.$$

Next we apply the function distribute, transforming the type into:

$$(f(x), x) \in (\{0\} \times \Sigma) + (\{1\} \times \Sigma)$$

To this result we apply the disjoint union $h_0 + h_1$ where h_i is defined by $(i, y) \mapsto g_i(y)$, yielding the desired result. □

Example 12. Every function $f : \Sigma \to \Gamma$ can be lifted to a function $f^+ : \Sigma^+ \to \Gamma^+$ in the natural way, and first-order list functions are easily seen to be closed under this lifting by using the map and pairing combinators. □

4 Aperiodic rational functions

The main result of this section is that the class of first-order list functions contains all aperiodic rational functions, see [18, Section IV.1] or Definition 6 below. An important part of the proof is that first-order list functions can compute factorisations as in the Factorisation Forest Theorem of Imre Simon [21, 7]. In Section 4.1, we state that the Factorisation Forest Theorem can be made effective using first-order list functions, and in Section 4.2, we define aperiodic rational functions and prove that they are first-order list functions.

4.1 Computing factorisations

In this section we state the Factorisation Forest Theorem and show how it can be made effective using first-order list functions. We begin by defining monoids and semigroups. For our application, it will be convenient to use a definition where the product operation is not binary, but has unlimited arity. (This is the view of monoids and semigroups as Eilenberg-Moore algebras over monads Σ^* and Σ^+, respectively).

Definition 2. A *monoid* consists of a set M and a product operation $\pi : M^* \to M$ which is associative, ie for all elements $m_1, \dots m_k$ of M and for all $1 \leq \ell_1 < \ell_2 < \cdots < \ell_j < k$:

$$\pi(m_1, \dots, m_k) = \pi(\pi(m_1, \dots, m_{\ell_1}), \pi(m_{\ell_1+1}, \dots, m_{\ell_2}), \dots, \pi(m_{\ell_j+1}, \dots, m_k))$$

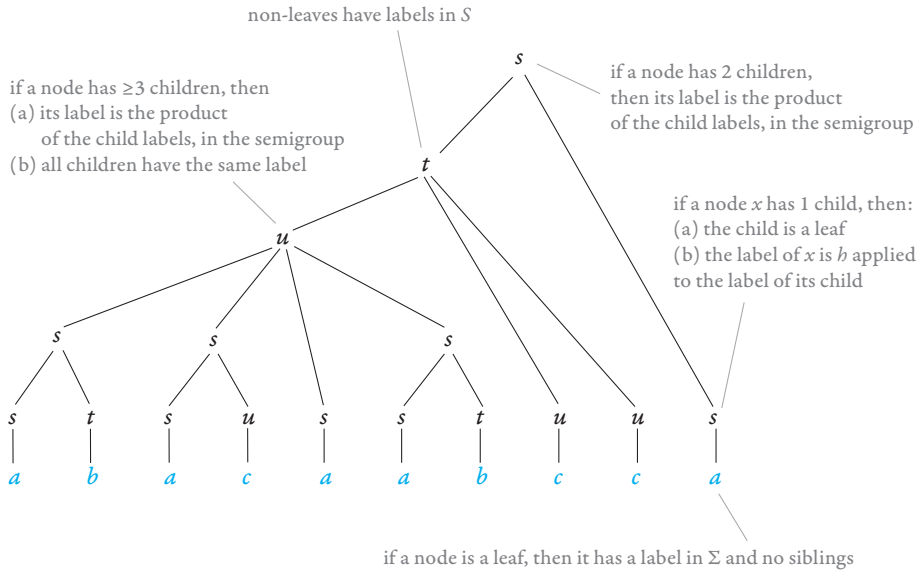
Remark that by definition the empty list of M^* is sent by π to a neutral element in M. A *semigroup* is defined the same way, except that nonempty lists M^+ are used instead of possibly empty ones.

Definition 3. A monoid (or semigroup) is called *aperiodic* if there exists some positive integer n such that $m^n = m^{n+1}$ for every element $m \in M$ where m^n denotes the n-fold product of m with itself.

A *semigroup homomorphism* is a function between two semigroups which is compatible with the semigroup product operation.

Factorisations.

Let $h : \Sigma^+ \to S$ be a semigroup homomorphism (equivalently, h can be given as a function $\Sigma \to S$ and extended uniquely into a homomorphism). An *h-factorisation* is defined to be a sibling-ordered tree which satisfies the following constraints (depicted in the following picture): leaves are labelled by elements of Σ and have no siblings. All the other nodes are labelled by elements from S. The parent of a leaf labelled by a is labelled by $h(a)$. The other nodes have at least two children and are labelled by the product of the child labels. If a node has at least three children then those children have all the same label.



Computing factorisations using first-order list functions.

As described above, an h-factorisation is a special case of a tree where leaves have labels in Σ and non-leaves have labels in S. Objects of this type, assuming that there is some bound k on the depth, can be represented using our type system:

$$\begin{aligned} \mathsf{trees}_0(\Sigma, S) &= \Sigma \\ \mathsf{trees}_{k+1}(\Sigma, S) &= \mathsf{trees}_k(\Sigma, S) + S \times (\mathsf{trees}_k(\Sigma, S))^+ \end{aligned}$$

Using the above representation, it is meaningful to talk about a first-order list function computing an h-factorisation of depth bounded by some constant k. This is the representation used in the following theorem. The theorem is essentially the same as the Factorisation Forest Theorem (in the aperiodic case), except that it additionally says that the factorisations can be produced using first-order list functions.

Theorem 4. *Let $\Sigma \in \mathscr{T}$ be a (not necessarily finite) type and let $h : \Sigma \to S$ be a function into the universe of some finite aperiodic semigroup S. If h is a first-order list function then there is some $k \in \mathbb{N}$ and a first-order list function:*

$$f : \Sigma^+ \to \mathsf{trees}_k(\Sigma, S)$$

such that for every $w \in \Sigma^+$, $f(w)$ is an h-factorisation whose yield (i.e. the sequence of leaves read from left to right) is w.

Before giving the proof of this theorem, let us give a corollary.

Corollary 5. *Let $\Sigma \in \mathscr{T}$ be finite. Then the following functions are first-order list functions:*
1. *every semigroup homomorphism $h : \Sigma^+ \to S$ where S is finite aperiodic;*
2. *every regular language over Σ, viewed as a function $\Sigma^* \to \{0, 1\}$.*

Proof (of the corollary 5)
For item 1, because Σ is finite, $h : \Sigma \to S$ is a first-order list function. We can then use Theorem 4, compute a h-factorisation with a first-order list function and then output its root label. For item 2, take some semigroup homomorphism which recognises the language, apply

item 1, and compose with the characteristic function of the accepting set. That is, consider the homomorphism $h : \Sigma^+ \to S$ recognizing $L \subseteq \Sigma^+$ and $P = h(L)$. Compose this with $g : S \to \{0, 1\}$ which maps exactly the elements of P to 1 (which is a first order list function because S is finite). Since the composition of two first-order list functions is a first-order list function, we obtain item 2. However, in item 2, we need to treat separately the case of the empty list on input, but this can be done using Examples 11 and 5. □

Proof (of Theorem 4)
Suppose that $s_1, \ldots, s_n$ are elements of S. In the proof below, we adopt the notational convention that $[s_1, \ldots, s_n] \in S^+$ represents the list of these elements, while $s_1 \cdots s_n \in S$ represents their product. In particular, st denotes the element of S which is the product of two elements s and t.

The proof of the theorem is by induction on the following parameters: (a) the size of S; and (b) the size of the image $h(\Sigma)$. These parameters are ordered lexicographically, i.e. we can call the induction assumption for a smaller semigroup even if the size of $h(\Sigma)$ grows.

1. Consider first the induction base, when the set $h(\Sigma)$ contains only one element, call it s. First, using the function from Example 5, we show that

$$[a_1, \ldots, a_n] \in \Sigma^+ \quad \mapsto \quad \overbrace{s \cdots s}^{n \text{ times}} \in S$$

are first-order list functions. The key observation is that, since S is aperiodic, the above function is constant for lists whose length exceeds some threshold. Pairing the above function with:

$$[a_1, \ldots, a_n] \in \Sigma^+ \mapsto [(s, a_1), \ldots, (s, a_n)] \in (S \times \Sigma)^+$$

we get the conclusion of the theorem.
2. Suppose that there is some $s \in h(\Sigma)$ such that:

$$T = \{ts : t \in S\} \subsetneq S$$

is a proper subset of S. Note that T is a subsemigroup of S. Consider two copies of Σ, Σ_1 the red copy and Σ_2 the blue one, and the function: $f : \Sigma \to \Sigma_1 + \Sigma_2$ which colours red those elements of Σ which are mapped with s, and colours blue the remaining ones (formally speaking, the range of f is a co-product of two copies of Σ). This is a first-order list function, by using the if-then-else construction described in Example 11. To an input list in Σ^*, apply f to all the elements of the list (with map), and then apply the block function, yielding a list: $x \in (\Sigma_1{}^* + \Sigma_2{}^*)^*$. Assume first that x begins with a blue list and ends with a red list and has the form

$$[z_1, y_1, z_2, y_2, \ldots, z_n, y_n],$$
$$z_1, \ldots, z_n \in \Sigma_2{}^+, y_1, \ldots, y_n \in \Sigma_1{}^+$$

Using the window function from Example 10 and discarding pairs that are of type $\Sigma_1{}^+ \times \Sigma_2{}^+$ with the filtering function from Example 6, we can transform the above list into one of the form:

$$[(z_1, y_1), (z_2, y_2), \ldots, (z_n, y_n)].$$

To both $\Sigma_1{}^*$ and $\Sigma_2{}^*$ we can apply the induction assumption on the number of generators. Therefore, using the induction assumption, co-product and map, we can transform the above list into a list of h-factorisations:

$$[u_1, u_2, \ldots, u_n]$$

such that each u_i is an h-factorisation of $z_i y_i$. The key observation is that, since y_i is a nonempty list of elements with value s, it follows that the value of u_i in the semigroup belongs to the set T, which is a smaller semigroup than S. Therefore, we can apply the induction assumption again, to transform the above list into an h-factorisation. We can treat similarly the cases where x does not begin with a blue list or does not end with a red list.

3. If there is some $s \in h(\Sigma)$ such that $T = \{st : t \in S\}$ is a proper subset of S, then we proceed analogously as in the previous case.
4. We claim that one of the above three cases must hold. Indeed, if neither case 2 nor 3 holds, then the functions:

$$f_s : t \mapsto st \qquad \text{and} \qquad g_s : t \mapsto ts$$

are permutations of S for all s. In particular, by the assumption that S is aperiodic, we deduce that f_s and g_s are the identity on the semigroup generated by s and $s^2 = s$. Thus for all t, $st = s^2 t$ and then necessarily f_s is the identity over S. The same thing holds for g_s. Therefore for every $s, t \in h(\Sigma)$ we have $t = st = s$. This means we are in case 1.

□

4.2 Rational functions

For the purposes of this paper, it will be convenient to give an algebraic representation for rational functions. In this section, we will only be interested in the case of aperiodic ones; however we explain in section 7 that our results can be generalised to arbitrary rational functions.

Definition 6 (Rational function)**.** The syntax of a *rational function* is given by:

- input and output alphabets Σ, Γ, which are both finite;
- a monoid homomorphism $h : \Sigma^* \to M$ with M a finite monoid;
- an output function $out : M \times \Sigma \times M \to \Gamma^*$.

If the monoid M is aperiodic, then the rational function is also called *aperiodic*. The semantics is the function:

$$a_1 \cdots a_n \in \Sigma^* \quad \mapsto \quad w_1 \cdots w_n \in \Gamma^*$$

where w_i is defined to be the value of the output function on the triple:

1. value under h of the prefix $a_1 \cdots a_{i-1}$
2. letter a_i
3. value under h of the suffix $a_{i+1} \cdots a_n$.

Note that in particular, the empty input word is mapped to an empty output.

Theorem 7. *Every aperiodic rational function is a first-order list function.*

The rest of Section 4.2 is devoted to showing the above theorem. The general idea is to use factorisations as in Theorem 4 to compute the rational function.

Sibling profiles.

Let $h : \Sigma^* \to M$ be a homomorphism into some finite aperiodic monoid M. Consider an h-factorisation, as defined in Section 4.1. For a non-leaf node x in the h-factorisation, define its *sibling profile* (see Figure 4) to be the pair (s, t) where s is the product in the monoid of the labels in the left siblings of x, and t is the product in the monoid of the labels in the right siblings. If x has no left siblings, then $s = 1$, if x has no right siblings then $t = 1$ (where 1 denotes the neutral element of the monoid M).

The two following lemmas give transformations on trees (which are h-factorisation) that are first-order list functions.

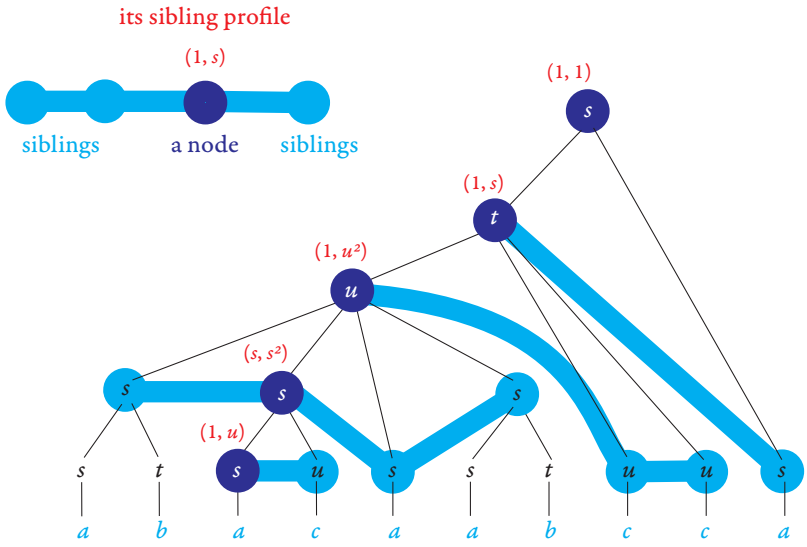


Figure 4 Sibling profiles.

Lemma 8. *Let $k \in \mathbb{N}$ and h be a homomorphism $\Sigma^* \to M$. There is a first-order list function $\mathsf{trees}_k(M, \Sigma) \to \mathsf{trees}_k(M \times M, \Sigma)$ which transforms any h-factorisation by replacing the label of each non-leaf node with its sibling profile.*

Proof
We prove the lemma by induction on k. We use the following claim to deal with nodes of degree at least 3 in the induction step.

Claim 9. *Let $\Delta \in \mathscr{T}$ and let $s \in M$. The function which maps a list $[x_1, \dots, x_n] \in \Delta^*$ to:*

$$[((s^0, s^{n-1}), x_1), \dots, ((s^{i-1}, s^{n-i}), x_i), \dots, ((s^{n-1}, s^0), x_n)] \in (M \times \Delta \times M)^*$$

is a first-order list function.

Proof
Since M is aperiodic, there is some n_0 such that all powers s^n with $n > n_0$ are the same. We use tail (n_0 times), $\mathsf{reverse}$ and again tail (n_0 times) to extract the list consisting of elements that are at distance at least n_0 to both the beginning and end of the list, and apply the function $x \mapsto ((s^{n_0}, s^{n_0}), x)$ to all those elements. We treat one by one the remaining elements, i.e. those at distance at most n_0 from either the beginning or end of the list, because there is at most $2n_0$ such elements, and we can extract them using head and last at most n_0 times. □

We can now give the inductive proof. For $k = 0$, the identity function, which is a first-order list function satisfies the conditions. Let $k > 0$. If the root has degree at most 2, let us write t_1 and t_2 for its two subtrees and s_1 and s_2 for the label of their respective roots. By induction, there exist a first-order list function transforming t_i into t'_i, for $i = 1, 2$, where each node (except the root) is replaced by its sibling profile. One can compose it with a first-order list function which replaces the node corresponding to s_1 by $(1, s_2)$ and s_2 by $(s_1, 1)$. If the root has degree at least 3, the reasoning is similar. Let us write $t_1, \dots, t_n$ for the subtrees and s for the label of their respective roots. By induction, there exist a first-order list function transforming t_i into t'_i, for $i = 1, \dots, n$, where each node (except the root) is replaced by its sibling profile. We can now

use the function from Claim 9, to replace the label of the roots of the subtrees by their sibling profile. □

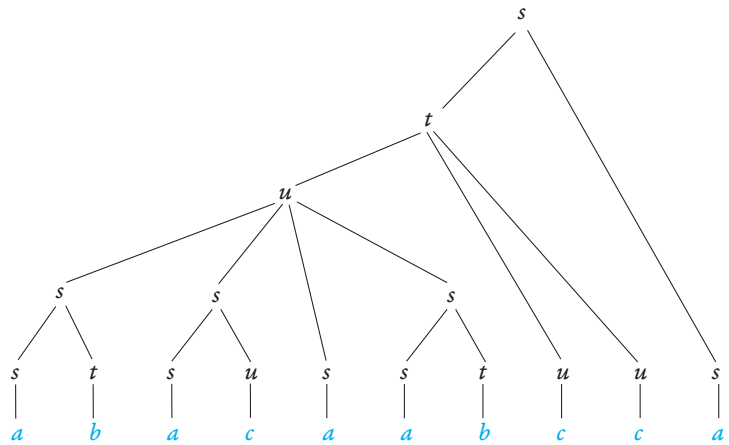
Lemma 10. *Let $k \in \mathbb{N}$ and let Δ be a finite set. Then there is a first-order list function: $\mathsf{trees}_k(\Delta, \Sigma) \to (\Delta^* \times \Sigma)^*$ which inputs a tree and outputs the following list: for each leaf (in left-to-right order) output the label of the leaf plus the sequence of labels in its ancestors listed in increasing order of depth.*

Proof
The key assumption here is that the depth of trees is bounded. We will prove the lemma by induction. For $k = 0$, the function $a \in \Sigma \mapsto [([], a)]$ which is a first-order list functions satisfies the conditions in the statement. Let $k > 0$, and $(s, [t_1, \ldots, t_n]) \in \mathsf{trees}_k(\Delta, \Sigma)$. By induction hypothesis, there is a first-order list function f transforming all the t_i into a list as stated in the lemma. Using map (and pairing with identity), we can apply this function to all the subtrees $t_1, \ldots, t_n$ and get a pair $(s, [f(t_1), \ldots, f(t_n)])$. We then just need to concatenate s (which we can extract from the pair using projection) to all the lists of the ancestors already paired with the leaves, which we can do using projection, map and append. We get a pair (s, ℓ) where ℓ is a list of lists of pairs (list of ancestors, leaf). We finally project on the second element and flatten to get the desired list. □

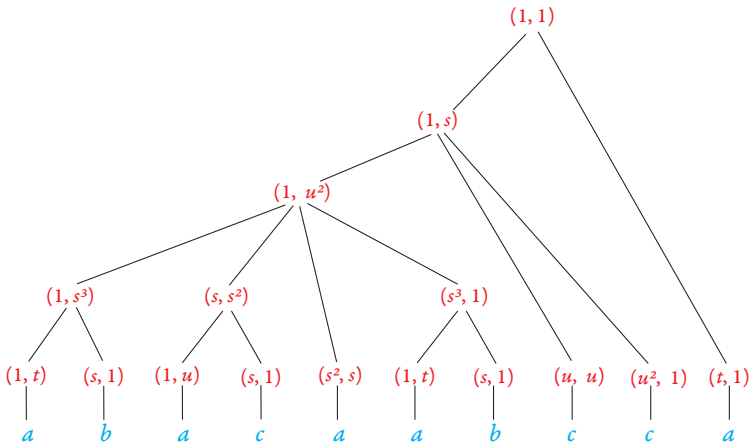
Proof (of Theorem 7)
Let $r : \Sigma^* \to \Gamma^*$ be a rational function, whose syntax is given by $h : \Sigma^* \to M$ and $out : M \times \Sigma \times M \to \Gamma^*$. Our goal is to show that r is a first-order list function. We will only show how to compute r on non-empty inputs. To extend it to the empty input we can use an if-then-else construction as in Example 11. We will define r as a composition of five functions, described below. To illustrate these steps, we will show after each step the intermediate output, assuming that the input is a word from Σ^+ that looks like this:

a b a c a a b c c a

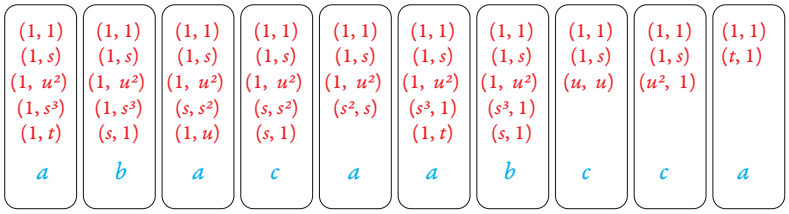
1. Apply Theorem 4 to h, yielding some k and a function: $\Sigma^+ \to \mathsf{trees}_k(M, \Sigma)$ which maps each input to an h-factorisation. After applying this function to our input, the result is an h-factorisation which looks like:



2. To the h-factorisation produced in the previous step, apply the function from Lemma 8, which replaces the label of each non-leaf node with its sibling profile. After this step, the output looks like this:



3. To the output from the previous step, we can now apply the function from Lemma 10, pushing all the information to the leaves, so that the output is a list that looks like this:



4. For k as in the first step, consider the function $g : (M \times M)^* \times \Sigma \to M \times \Sigma \times M + \bot$ defined by:

$$([(s_1, t_1), \ldots, (s_n, t_n)], a) \quad \mapsto \quad \begin{cases} (s_1 \cdots s_n, a, t_n \cdots t_1) & \text{if } n \leq k \\ \bot & \text{otherwise} \end{cases}$$

The function g is a first-order list function, because it returns $\bot$ on all but finitely many arguments. Apply g to all the elements of the list produced in the previous step (using map), yielding a list from $(M \times \Sigma \times M)^*$ which looks like this:

$$\big[(1, a, ts^3u^2s), (t, b, 1s^3u^2s), (s, a, us^2u^2s), (ss, c, u^2u^2s) \quad , \ldots , \quad (u^2, c, .$$

5. In the list produced in the previous step, the i-th position stores the i-th triple as in the definition of rational functions (Definition 6). Therefore, in order to get the output of our original rational function r, it suffices to apply *out* (because of finiteness, *out* is a first-order list function) to all the elements of the list obtained in the previous step (with map), and then use flat on the result obtained.

□

5 First-order transductions

This section states the main result of this paper: the first-order list functions are exactly those that can be defined using first-order transductions (FO-transductions). We begin by describing FO-transductions in Section 5.1, and then in Section 5.2, we show how they can be applied to types from $\mathscr{T}$ by using an encoding of lists, pairs, etc. as logical structures; this allows us to state our main result, Theorem 13, namely that FO-transductions are the same as first-order list functions (Section 5.3). The proof of the main result is given in Sections 5.3, 5.5 and 6.

5.1 FO-transductions: definition

A *vocabulary* is a set (in our application, finite) of relation names, each one with an associated arity (a natural number). We do not use functions. If $\mathcal{V}$ is a vocabulary, then a *logical structure* over $\mathcal{V}$ consists of a universe (a set of elements), together with an interpretation of each relation name in $\mathcal{V}$ as a relation on the universe of corresponding arity.

An FO-*transduction* [9] is a method of transforming one logical structure into another which is described in terms of first-order formulas. More precisely, an FO-transduction consists of two consecutive operations: first, one copies the input structure a fixed number of times, and next, one defines the output structure using a first-order interpretation (we use here what is sometimes known as a *one dimensional interpretation*, i.e. we cannot use pairs or triples of input elements to encode output elments). The formal definitions are given below.

One dimensional FO-interpretation.

The syntax of a one dimensional FO-interpretation (see also [17, Section 5.4]) consists of:

1. Two vocabularies, called the *input* and *output* vocabularies.
2. A formula of first-order logic with one free variable over the input vocabulary, called the *universe formula*.
3. For each relation name R in the output vocabulary, a formula φ_R of first-order logic over the input vocabulary, whose number of free variables is equal to the arity of R.

The semantics is a function from logical structures over the input vocabulary to logical structures over the output vocabulary given as follows. The universe of the output structure consists of those elements in the universe of the input structure which make the universe formula true. A predicate R in the output structure is interpreted as those tuples which are in the universe of the output structure and make the formula φ_R true.

Copying.

For a positive integer k and a vocabulary $\mathcal{V}$, we define k-copying (over $\mathcal{V}$) to be the function which inputs a logical structure over $\mathcal{V}$, and outputs k disjoint copies of it, extended with an additional k-ary predicate that selects a tuple $(a_1, \ldots, a_k)$ if and only if there is some a in the input structure such that $a_1, \ldots, a_k$ are the respective copies of a. (The additional predicate is sensitive to the ordering of arguments, because we distinguish between the first copy, the second copy, etc.)

Definition 11 (FO-transduction)**.** An FO-transduction is defined to be an operation on relational structures which is the composition of k-copying for some k, and of a one dimensional FO-interpretation.

FO-transductions are a robust class of functions. In particular, they are closed under composition. Perhaps even better known are the more general MSO-transductions, we will discuss these at the end of the paper.

An example of FO-transduction.

We give here a simple example of an FO-transduction. Consider the word structure

$$\mathcal{S} = (\mathcal{U}, S, <, (Q_a)_{a \in \Sigma})$$

over a finite alphabet $\Sigma = \{a, b\}$. The universe $\mathcal{U}$ is a finite set of positions $\{0, 1, \ldots, n\}$ in the word. For first order variables x, y, we have the relations $x < y$ and the successor relation $S(x, y)$ with the obvious meanings. We also have the relation $Q_a(x)$ which evaluates to true if x can be

assigned some value $i \in \mathcal{U}$ such that the ith position of the word has an a. For example, the word *ababa* satisfies the formula

$$\begin{aligned}&\exists x[first(x) \wedge Q_a(x)]\\ \wedge&\exists x[last(x) \wedge Q_a(x)]\\ \wedge&\forall x, y[(S(x,y) \wedge Q_a(x) \to \neg Q_a(y)) \wedge (S(x,y) \wedge Q_b(x) \to \neg Q_b(y)]\end{aligned}$$

where $first(x) = \forall y(x \leq y)$ and $last(x) = \forall y(y \leq x)$.

Consider the transduction which transforms a word w into $w_1 w_2$ where w_1 and w_2 respectively are obtained by removing the b's and a's from w. For example *ababa* is transformed into *aaabb*.

1. We make two copies of the input structure. The nodes in the first copy labelled by an a as well as the nodes in the second copy labelled by a b are in the universe of the output word. They are specified by the FO-formula $\varphi^1(x) = Q_a(x)$ and $\varphi^2(x) = Q_b(x)$.
2. The edges between nodes in the first copy are specified by the formula

   $$\varphi^{1,1}(x,y) = x < y \wedge \neg\exists z(x < z < y \wedge Q_a(z))$$

   which allows an edge between an a and the next occurrence of an a. Likewise, edges between nodes in the second copy are specified by the formula

   $$\varphi^{2,2}(x,y) = x < y \wedge \neg\exists z(x < z < y \wedge Q_b(z))$$

3. Finally, we specify edges between the nodes of copy 1 and copy 2. The formula

   $$\varphi^{2,1}(x,y) = false$$

   disallows any edges from the second copy to the first copy, while the formula

   $$\varphi^{1,2}(x,y) = (Q_a(x) \wedge \forall z(z > x \to \neg Q_a(z))) \wedge (Q_b(y) \wedge \forall z(z < y \to \neg Q_b(z)))$$

   enables an edge from the last a (the last position in the first copy) to the first b (the first position in the second copy).

   This results in the word where all the a's in w appear before all the b's in w.

5.2 Nested lists as logical structures

Our goal is to use FO-transductions to define functions of the form $f : \Sigma \to \Gamma$, for types $\Sigma, \Gamma \in \mathscr{T}$. To do this, we need to represent elements of Σ and Γ as logical structures. We use a natural encoding, which is essentially the same one as is used in the automata and logic literature, see e.g. [23, Section 2.1].

Consider a type $\Sigma \in \mathscr{T}$. We represent an element $x \in \Sigma$ as a relational structure, denoted by $\underline{x}$, as follows:

1. The universe $\mathcal{U}$ is the nodes in the parse tree of x (see Figure 5).
2. There is a binary relation $\mathsf{Par}(x,y)$ for the parent-child relation which says that x is the parent of y.
3. There is a binary relation $\mathsf{Sib}(x,y)$ for the transitive closure of the "next sibling" relation. The next sibling relation $\mathsf{NextSib}(x,y)$ is true if y is the next sibling of x: that is, there is a node z which is the parent of x and y, and there are no children of z between x and y (in that order). $\mathsf{Sib}(x,y)$ evaluates to true if x, y are siblings, and y after x.
4. For every node τ in the parse tree of the type Σ (see Figure 6), there is a unary predicate $\mathsf{type}(\tau)$, which selects the elements from the universe of $\underline{x}$, (equivalently the subterms of x) that have the type as τ. For example, for the node τ labeled with $[b]$, $B^*(\tau)$ evaluates to true if $b \in B$.

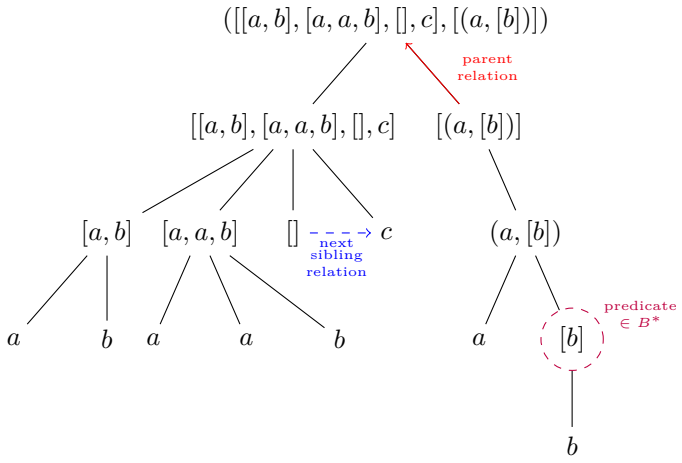


Figure 5 The parse tree of a nested list.

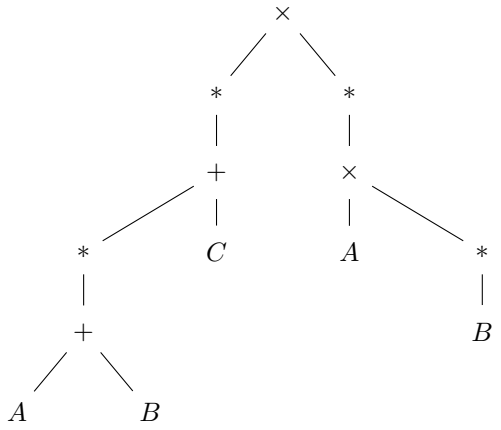
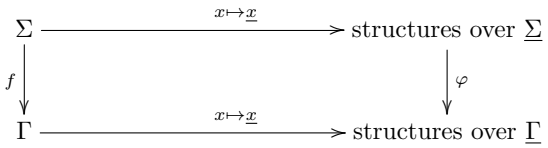


Figure 6 The parse tree of a type in $\mathscr{T}$.

We write $\underline{\Sigma}$ for the relational vocabulary used in the structure $\underline{x}$. This vocabulary has two binary relations, as described in items 2 and 3, as well as one unary relation for every node in the parse tree of the type Σ.

Definition 12. Let $\Sigma, \Gamma \in \mathscr{T}$. We say that a function $f : \Sigma \to \Gamma$ is *definable by an* FO*-transduction* if it is an FO-transduction under the encoding $x \mapsto \underline{x}$; more formally, if there is some FO-transduction φ which makes the following diagram commutes:



It is important that the encoding $x \mapsto \underline{x}$ gives the transitive closure of the next sibling relation. For example when the type Σ is $\{a, b\}^*$, our representation allows a first-order transduction to access the order $<$ on positions, and not just the successor relation. For first-order logic (unlike

for MSO) there is a significant difference between having access to order vs successor on list positions.

5.3 Main result

Below is one of the main contributions of this paper.

Theorem 13. *Let $\Gamma, \Sigma \in \mathscr{T}$. A function $f : \Sigma \to \Gamma$ is a first-order list function if and only if it is definable by an* FO*-transduction.*

Before proving Theorem 13, let us note the following corollary: the equivalence of first-order list functions (i.e. do they give the same output for every input) is decidable. Indeed, we will see below that any first-order list function can be encoded into a string-to-string first-order list function. Using this encoding and Theorem 13, the equivalence problem of first-order list functions boils down to deciding equivalence of string-to-string FO-transductions; which is decidable [16].

Proof of the left-to-right implication of Theorem 13.

The proof of the left-to-right implication of Theorem 13 is by induction following the definition of first-order list functions. The basic functions $\mathsf{projection}$, $\mathsf{co-projection}$ and $\mathsf{distribute}$ are clearly definable by FO-transductions. We prove now that $\mathsf{reverse}$, flat, append, $\mathsf{co-append}$ and block are also defined by FO-transduction.

In all of the following cases, let $\mathsf{root}(z)$ be a macro for the root node ($\mathsf{root}(z) = \neg\exists z' \mathsf{Par}(z', z)$). In all cases below, for a list τ, let the structure of τ be $\underline{\tau}$, and recall that we have $\mathsf{Par}, \mathsf{NextSib}, \mathsf{Sib} \in \underline{\Delta}$, the vocabulary of $\underline{\tau}$.

1. $\mathsf{Reverse}$. Given a list τ, the first-order list function $\mathsf{reverse}(\tau)$ can be implemented using an FO-transduction as follows: The nodes in the parse tree of $\mathsf{reverse}(\tau)$ is specified by the universe formula $\varphi^1(x) = \mathit{true}$, selecting all the nodes from the parse tree of τ. The parent-child relations are left unchanged but the next sibling relations are reversed.
2. Append. Given $\tau = (x_0, [x_1, \ldots, x_n]) \in \Sigma \times \Sigma^*$, the first-order list function $\mathsf{append}(\tau)$ resulting in $[x_0, x_1, \ldots, x_n]$ is implemented using an FO-transduction as follows:
   a. The nodes in the parse tree of $\mathsf{append}(\tau)$ is specified by the universe formula which selects all nodes y in the parse tree of τ which are not the second child of the root. Remind that the second child of the root here represents the entire list $[x_1, \ldots, x_n]$.

$$\varphi^1(y) = \neg[\mathsf{Par}(x, y) \wedge \mathsf{root}(x) \wedge \neg\exists z \mathsf{NextSib}(y, z)]$$

   Note that in the parse tree of τ, the root node has two children, and in the constructed parse tree, we omit this second child.
   b. We now specify the parent child relation. This retains the leftmost child of the root in the tree of τ as a child in $\mathsf{append}(\tau)$, and in addition, adds all the children of the second child of the root in τ as the children of the root in $\mathsf{append}(\tau)$.

$$\begin{aligned}\varphi^{1,1}(x, y) = \{\neg\mathsf{root}(x) \wedge \mathsf{Par}(x, y)\} \vee [\mathsf{root}(x) \wedge \mathsf{Par}(x, y) \wedge \neg\exists z \mathsf{NextSib}(z, y)] \\ \vee[\mathsf{root}(x) \wedge \{\exists z'[\mathsf{Par}(x, z') \wedge \exists z''(\mathsf{NextSib}(z'', z)) \wedge \mathsf{Par}(z', y)]\}]\end{aligned}$$

3. $\mathsf{Co-append}$. Given $\tau = [x_0, x_1, \ldots, x_n] \in \Sigma^*$, the first-order list function $\mathsf{co-append}(\tau)$ resulting in $(x_0, [x_1, \ldots, x_n])$ if $n \geq 1$ and undefined otherwise, is implemented using an FO-transduction as follows: To obtain the parse tree of $\mathsf{co-append}(\tau)$, we make two copies of the parse tree of τ.

a. The nodes in the first copy are specified by the formula $\varphi^1(y) = true$ selecting all the nodes of τ. The second child of the root in the parse tree of τ is selected in the second copy. Note that the existence of a second child checks the condition that $n \geq 1$, without which the function is not defined. The formula

$$\varphi^2(y) = \exists x[\mathsf{root}(x) \wedge \mathsf{Par}(x,y) \wedge \exists z.[\mathsf{Par}(x,z) \wedge \mathsf{NextSib}(z,y) \wedge \neg\exists z'\mathsf{NextSib}(z',z)]]$$

selects the second child of the root in the parse tree of τ.

b. The parent child relation in the first copy is defined as follows: It allows all the edges already present except the parent-child relation between the root and the nodes which are not the first child of the root.

$$\varphi^{1,1}(x,y) = \psi_1 \vee \psi_2$$

where

$$\psi_1 = \mathsf{root}(x) \wedge \mathsf{Par}(x,y) \wedge \neg\exists z[\mathsf{NextSib}(z,y) \wedge \mathsf{Par}(x,z)]$$

$$\psi_2 = \neg\mathsf{root}(x) \wedge \mathsf{Par}(x,y)$$

The parent child relation in the second copy is defined by $\varphi^{2,2}(x,y) = false$, since there is a unique node in the second copy.

c. There is one edge from the first copy to the second which makes the root of the first copy the parent of the unique node in the second copy. The unique node in the second copy is a parent to all the non-leftmost children of the root. This is given by formulae $\varphi^{1,2}(x,y) = \mathsf{root}(x)$ and

$$\varphi^{2,1}(x,y) = \exists z[\mathsf{root}(z) \wedge \mathsf{Par}(z,y) \wedge \exists z'[\mathsf{Par}(z,z') \wedge \mathsf{NextSib}(z',y)]]$$

See Figure 7 where this is illustrated on an example.

4. Flat. Given a list τ, the first-order list function $\mathsf{flat}(\tau)$ can be implemented using an FO-transduction as follows:

a. The nodes in the parse tree of $\mathsf{flat}(\tau)$ are all nodes y which are not the children of the root node. Note that we do not need any copying of the input structure here. It is given by the formula

$$\varphi^1(y) = [\mathsf{Par}(x,y) \wedge \neg\mathsf{root}(x)] \vee \mathsf{root}(y)$$

b. To specify the parent child relation, all nodes other than the root have the same parent child relation as before. We also connect the grandchildren of the root to the root. This is specified by

$$\varphi^{1,1}(x,y) = \{\neg\mathsf{root}(x) \wedge \mathsf{Par}(x,y)\} \vee [\mathsf{root}(x) \wedge \{\exists z'[\mathsf{Par}(x,z') \wedge \mathsf{Par}(z',y)]\}]$$

which says that a non-root node has the same children as before, while the root's children in $\mathsf{flat}(\tau)$ are its grandchildren in τ.

5. Block. Let's now consider the function block. Let Σ and Γ be types and $\Delta = (\Sigma + \Gamma)^*$. Let $\tau \in \Delta$. Let $\underline{\tau}$ be the relational structure for τ. To obtain $\mathsf{block}(\tau)$ as an FO-transduction, we make two copies of the parse tree of τ. The first copy has all the nodes. All edges except those defining the parent-child relation between the root and its children are present in the first copy. The second copy consists of nodes which are the children of the root, and whose next sibling is of a different type. The only exception is when dealing with the last child of the root, which is always added. There are no edge relations between nodes in the second copy. We add a parent-child relation between the root of the first copy and all nodes in the second copy. Likewise, add a parent-child relation between a node α in the second copy with node α in the first copy and all the following siblings of α who have the same type as α.

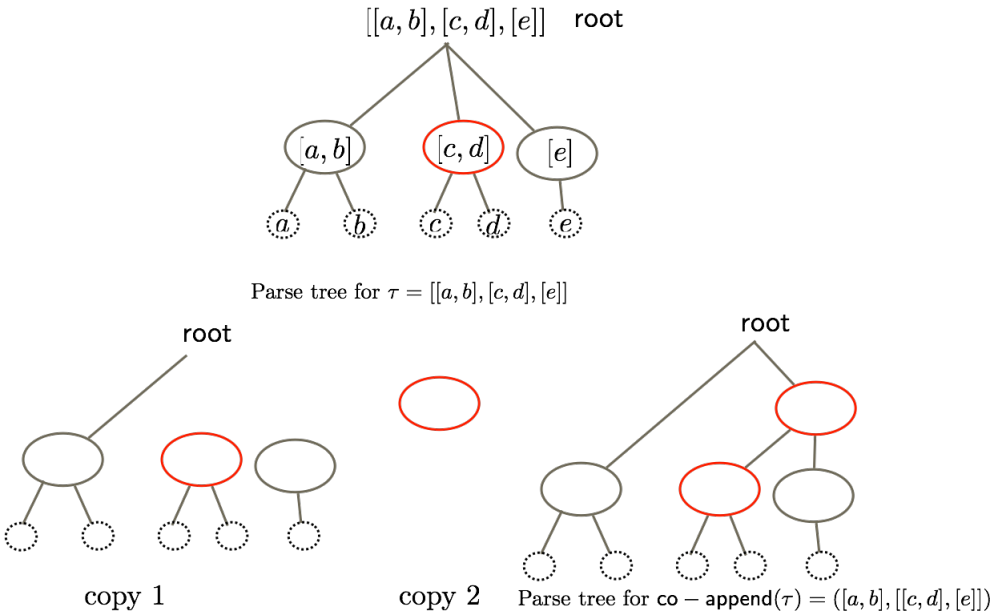


Figure 7 We start with $\tau = [[a, b], [c, d], [e]] \in \Sigma^{**}$. On the left is the parse tree of τ. Copy 1 has all the nodes in the parse tree of τ, while copy 2 only has the red circled node from the parse tree of τ. The edges in copy 1 include all original edges except the one from the root in the parse tree of τ to children having a left sibling. The parse tree of $\mathsf{co-append}(\tau)$ is obtained by drawing edges from the root in copy 1 to the only node in copy 2, and from the node in copy 2 to all non-leftmost children of the root, as illustrated.

a. The universe formula describing the nodes in the first copy is given by $\varphi^1(x) = true$, including all the nodes.

b. The edges between nodes in the first copy is given by $\varphi^{1,1}(x, y) = \neg\mathsf{root}(x) \wedge \mathsf{Par}(x, y)$ which retains all parent-child relations other than between the root and its children.

c. Assume that we have finitely many type predicates $\mathsf{type}_1, \ldots, \mathsf{type}_n$ in the structure $\underline{\tau}$. That is, $\mathsf{type}_1, \ldots, \mathsf{type}_n, \mathsf{Par}, \mathsf{NextSib}, \mathsf{Sib} \in \underline{\Delta}$, the vocabulary of $\underline{\tau}$. The universe formula describing nodes in the second copy is given by

$$\varphi^2(x) = \exists z.[\mathsf{Par}(z, x) \wedge \mathsf{root}(z)] \wedge \{\bigvee_{i=1}^{n} [\mathsf{type}_i(x) \wedge \mathsf{NextSib}(x, y) \rightarrow \neg\mathsf{type}_i(y)]\}$$

This selects all children of the root which either does not have a next sibling (last child) or whose next sibling has a different type.

d. The edge relation between nodes in the second copy is $\varphi^{2,2}(x, y) = false$, thereby disallowing any edges.

e. The edges from nodes in the first copy to the second copy is given by

$$\varphi^{1,2}(x, y) = \mathsf{root}(x) \wedge \mathsf{Par}(x, y)$$

which enables edges from the root of the first copy to all nodes in the second copy. Recall that $\mathsf{Par}(x, y)$ is true in τ for all nodes y in the second copy and the root x.

f. The edges from nodes in the second copy to nodes in the first copy is given by

$$\varphi^{2,1}(x, y) = \psi_1 \wedge \psi_2$$

where

$$\psi_1 = [\mathsf{Sib}(y,x) \vee (x = y)] \wedge [x \neq y \to \bigvee_{i=1}^{n} [\mathsf{type}_i(y) \leftrightarrow \mathsf{type}_i(x)]]$$

$$\psi_2 = \neg\exists z'(\mathsf{Sib}(z',x) \wedge \mathsf{Sib}(y,z') \wedge \bigvee_{i=1}^{n} [\mathsf{type}_i(y) \leftrightarrow \neg\mathsf{type}_i(z')])$$

ψ_1 collects all the siblings to the left of x (and itself) which have the same type as x, while ψ_2 ensures that the chosen nodes are contiguous and of the same type. This ensures that we "block" nodes of the same type and assign it one parent, and a change of type results in a different parent. Figure 8 illustrates this on an example.

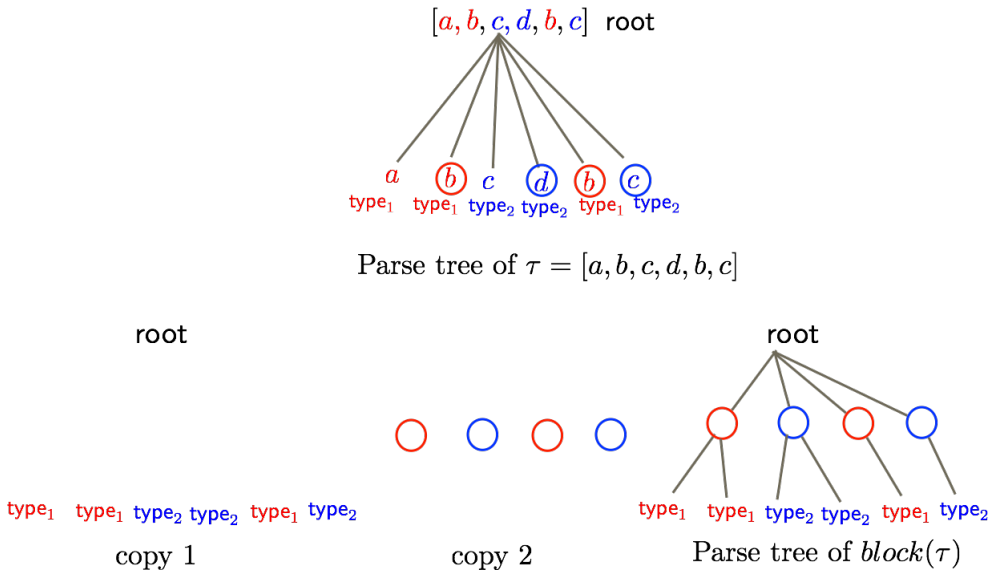


Figure 8 We start with $\tau = [a,b,c,d,b,c] \in (\Sigma + \Gamma)^*$ where $a, b \in \Sigma$ and $c, d \in \Gamma$. On the left is the parse tree of τ. The type type_1 represents Σ while type_2 represents Γ. Copy 1 has all the nodes in the parse tree of τ, while copy 2 only has the circled nodes from the parse tree of τ. The parse tree of $\mathsf{block}(\tau)$ is obtained by drawing edges from copy 1 to copy 2, and back as illustrated.

6. Putting together all the basic list functions using combinators : Finally, we are left to prove that FO-transductions are closed under the four combinators. It is clear that FO-transductions are closed under disjoint union and composition. Moreover, map and pairing can be handled the same way. Consider two FO-transductions f and g and a relational structure representing a list or a pair. Map of f (resp. pairing f and g) is defined by applying f to all the subtrees of the root in the relational structure (resp. applying f to the left subtree of the root and g to the right one). It is clear that this can be done with an FO-transduction making a number of copies equal to the maximum of the number of copies required for f and g and linking the roots of the subtrees obtained by applying f and g to one unique new root.

Proof of the right-to-left implication of Theorem 13.

The more challenging right-to-left implication is described in the rest of this section. First, by using an encoding of nested lists of bounded depth via strings, e.g. XML encoding (both the encoding and decoding are easily seen to be both first-order list functions and definable by FO-transductions), we obtain the following lemma:

Lemma 14. *To prove the right-to-left implication of Theorem 13, it suffices to show it for string-to-string functions, i.e. those of type* $\Sigma^* \to \Gamma^*$ *for some finite sets* Σ, Γ.

Without loss of generality, we can thus only consider the case when both the input and output are strings over a finite alphabet. This way one can use standard results on string-to-string transductions (doing away with the need to reprove the mild generalisations to nested lists of bounded depth).

Every string-to-string FO-transduction can be decomposed as a two step process: (a) apply an aperiodic rational transduction to transform the input word into a sequence of operations which manipulate a fixed number of registers that store words; and then (b) execute the sequence of operations produced in the previous step, yielding an output word. Therefore, to prove Theorem 13, it suffices to show that (a) and (b) can be done by first-order list functions. Step (a) is Theorem 7. Step (b) is described in Section 5.5. Before tackling the proofs, we need to generalise slightly the definition of first-order list functions.

5.4 Generalised first-order list functions

In some constructions below, it will be convenient to work with a less strict type discipline, which allows types such as "lists of length at least three" or

$$\{[x_1, \ldots, x_n] \in \{a, b\}^* : \text{every two consecutive elements differ}\}$$

Definition 15 (First-order definable set)**.** Let $\Sigma \in \mathscr{T}$. A subset $P \subseteq \Sigma$ is called *first-order definable* if its characteristic function $\Sigma \to \{0, 1\}$ is a first-order list function. Let $\mathscr{T}_{\text{FO}}$ denote first-order definable subsets of types in $\mathscr{T}$.

When Σ is of the form Γ^* for some finite alphabet, the above notion coincides with the usual notion of first-order definable language, as in the Schützenberger-McNaughton-Papert Theorem. The *generalised first-order list functions* are defined to be first-order list functions as defined previously where the domains and co-domains are in $\mathscr{T}_{\text{FO}}$.

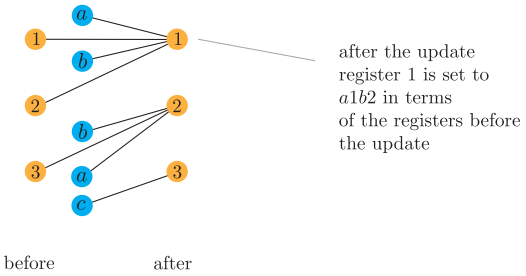
Definition 16 (Generalised first-order list functions)**.** A function $f : \Sigma \to \Gamma \quad \text{with} \quad \Sigma, \Gamma \in \mathscr{T}_{\text{FO}}$ is said to be a *generalised first-order list function* if it is obtained by taking some first-order list function (definition 1) and restricting its domain and co-domain to first-order definable subsets so that it remains a total function.

5.5 Registers

To complete the proof of Theorem 13, it will be convenient to use a characterisation of FO-transductions which uses registers, in the spirit of streaming string transducers [3].

Registers and their updates.

Let M be a monoid, not necessarily finite, and let $k \in \{1, 2, \ldots\}$. We define a *k-register valuation* to be a tuple in M^k, which we interpret as a valuation of registers called $\{1, \ldots, k\}$ by elements of M. Define a *k-register update* over M to be a parallel substitution, which transforms one k-valuation into another using concatenation, as in the following picture.



Formally, a k-register update is a k-tuple of words over $M \cup \{1, \ldots, k\}$. In particular, if M is in $\mathscr{T}_{\text{FO}}$ then also the set of k-register updates is in $\mathscr{T}_{\text{FO}}$, and therefore it is meaningful to talk about (generalised) first-order list functions that input and output k-register valuations. If η is a k-register update, we use the name *i-th right hand side* for the i-th coordinate of the k-tuple η.

There is a natural right action of register updates on register valuations: if $v \in M^k$ is a k-register valuation, and η is a k-register update, then we define $v\eta \in M^k$ to be the k-register valuation where register i stores the value in the monoid M obtained by taking the i-th right hand side of η, substituting each register name j with its value in v, and then taking the product in the monoid M. This right action can be implemented by a generalised first-order list function, as stated in the following lemma, which is easily proved by inlining the definitions.

Lemma 17. *Let k be a non-negative integer, let $v \in M^k$ and assume that M is a monoid whose universe is in $\mathscr{T}_{FO}$ and whose product operation is a generalised first-order list function. Then the function which maps a k-register update η to the k-register valuation $v\eta \in M^k$ is a generalised first-order list function.*

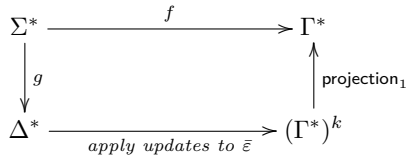
Proof
Consider a k-register update encoded as a k-tuple of words, each of which are encoded by a list of elements from M and from $\{1, \ldots, k\}$. First, because k is fixed and using projection and pairing, we can only consider the case of a single word. Let v as in the lemma. Because v is fixed, the function from the finite set $\{1, \ldots, k\}$ associating with i the ith component of v is a first-order list function. The function identity over M is also a generalised first-order list function, so by using disjoint union first and then map, the function which transforms a list of elements from $M + \{1, \ldots, k\}$ into a list of element from M by replacing i by the ith component of v is a generalised first-order list function. Finally, by using the product operation in M which is a generalised first-order list function, one can compute the product of the elements of the list. □

Non duplicating monotone updates.

A k-register update is called *nonduplicating* if each register appears at most once in the concatenation of all the right hand sides; and it is called *monotone* if after concatenating the right hand sides (from 1 to k), the registers appear in strictly increasing order (possibly with some registers missing). We write $M^{[k]}$ for the set of nonduplicating and monotone k-register updates.

Lemma 18 says that every string-to-string FO-transduction can be decomposed as follows: (a) apply an aperiodic rational transduction to compute a sequence of monotone nonduplicating register updates; then (b) apply all those register updates to the empty register valuation (which we denote by $\bar{\varepsilon}$ assuming that the number of registers k is implicit), and finally return the value of the first register.

Lemma 18. *Let Σ and Γ be finite alphabets. Every* FO*-transduction $f : \Sigma^* \to \Gamma^*$ can be decomposed as:*



for some positive integer k, where Δ is a finite subset of $(\Gamma^*)^{[k]}$ (i.e. a finite set of k-register updates that are monotone and nonduplicating), $g : \Sigma^* \to \Delta^*$ is an aperiodic rational function and $\mathsf{projection}_1$ *is the projection of a k-tuple of $(\Gamma^*)^k$ on its first component.*

Proof

We use an equivalent characterisation of FO-transduction in terms of streaming string transducers (SST) (first shown for MSO-transduction in [2]). By [14], an FO-transduction $f : \Sigma^* \to \Gamma^*$ can be computed by an SST $\mathbf{F}$ whose register updates are nonduplicating and whose transition monoid is aperiodic. A possible definition for the transition monoid of an SST is given in [11] (called *substitution transition monoid*). Elements of the monoid are functions mapping a state p of the SST to a pair (q, r) formed with a state q of the SST and a k-register update r containing only names of registers (and no element of Γ). Every word u is mapped to such a function g_u such that $g_u(p) = (q, r)$ if there is a run in the SST on u from state p to state q where the registers are updated according to r with possibly elements of Γ inserted in the products.

We will first transform $\mathbf{F}$ to add a regular look-ahead which will guess which registers are output at the end of the computation on a given word and in which order. More precisely, there is a rational function $\tilde{g}$ taking as input a state p of $\mathbf{F}$ and a word v and which outputs the sequence of registers $\tilde{g}(p, v) = (r_1, r_2, \cdots, r_\ell)$ which appears (respecting the order) in the update of the register output in q in r where $g_v(p) = (q, r)$. Note that this sequence contains at most k registers, all distinct. Because the transition monoid of $\mathbf{F}$ is aperiodic, so is $\tilde{g}$.

Now, let $\mathbf{G}$ be the SST constructed from $\mathbf{F}$ and $\tilde{g}$ as follows:

States are pairs of a state p of $\mathbf{F}$ and a sequence of at most k distinct registers which corresponds to $\tilde{g}(p, v)$ for some word v. Transitions are similar as the ones in $\mathbf{F}$, such that:

- from a state $(p, \tilde{g}(p, uv))$, one can reach the state $(q, \tilde{g}(q, v))$ when reading u, where q is the state reached from p by reading u in $\mathbf{F}$,
- the register updates are modified in such a way that the registers are reordered according to $\tilde{g}(p, v)$ so as to maintain monotone updates all along the computation.

We can moreover assume that the output register is always the first one.

The set Δ is now defined as the finite set of register updates on the transitions in $\mathbf{G}$, which are nonduplicating and monotone. The function g is the function mapping a word u to the sequence of register updates performed while reading u in $\mathbf{G}$ from the initial state to the state $(q, \tilde{g}(q, \varepsilon))$, where q is the state reached in $\mathbb{F}$ by reading u from the initial state. The function g is thus aperiodic rational.

Because $\mathbf{F}$ and $\mathbf{G}$ are equivalent, the function f can be decomposed into the three functions as in the statement of the lemma. □

Thanks to Lemma 14 and the closure of first-order list function under composition, it is now sufficient to prove that the bottom three functions of Lemma 18 are first-order list functions, in order to complete the proof of Theorem 13. This is the case of the function $\mathsf{projection}_1$ by definition and of the aperiodic rational function g by Theorem 7. We are thus left to prove the following lemma, which is the subject of Section 6.

Lemma 19. *Let $\Gamma \in \mathscr{T}$, let k be a positive integer and let Δ a finite subset of $(\Gamma^*)^{[k]}$. The function from Δ^* to $(\Gamma^*)^k$ which maps a list of nonduplicating monotone k-register updates to*

the valuation obtained by applying these updates to the empty register valuation is a first-order list function.

6 The register update monoid

The goal of this section is to prove Lemma 19. We will prove a stronger result which also works for a monoid other than Γ^*, provided that its universe is a first-order definable set and its product operation is a generalised first-order list function. This result is obtained as a corollary of Theorem 20 below. In order to state this theorem formally, we need to view the product operation: $(M^{[k]})^* \to M^{[k]}$ as a generalised first-order list function. The domain of the above operation is in $\mathscr{T}_{\text{FO}}$ from Definition 15, since being monotone and nonduplicating are first-order definable properties.

Theorem 20. *Let M be a monoid whose universe is in $\mathscr{T}_{FO}$ and whose product operation is a generalised first-order list function. Then the same is true for $M^{[k]}$, for every $k \in \{0, 1, \ldots\}$.*

Lemma 19 follows from the above theorem applied to $M = \Gamma^*$, and from Lemma 17. Indeed, given $\Gamma \in \mathscr{T}$, the universe of ${\Gamma^*}^{[k]}$ is in $\mathscr{T}_{\text{FO}}$ and its product operation is a generalised first-order list function by Theorem 20. The function from Lemma 19 is then the composition of the product operation in Δ^* (which corresponds to the product operation in ${\Gamma^*}^{[k]}$), which transforms a list of updates from Δ into an update of ${\Gamma^*}^{[k]}$, and the evaluation of this update on the empty register valuation. This last function is a generalised first-order list function by Lemma 17 with $v = \bar{\varepsilon}$. Implicitly, we use the fact that the right action is compatible with the monoid structure of $M^{[k]}$, i.e.

$$v(\eta_1\eta_2) = (v\eta_1)\eta_2 \qquad \text{for } v \in M^k \text{ and } \eta_1, \eta_2 \in M^{[k]}.$$

Summing up, we have proved that the function of type $\Delta^* \to (\Gamma^*)^k$ discussed in Lemma 19 is a generalised first-order list function. Since its domain and co-domain are in $\mathscr{T}$, it is also a first-order list function. This completes the proof of Lemma 19.

It remains to prove Theorem 20. We do this using factorisation forests, with our proof strategy encapsulated in the following lemma, using the notion of *homogeneous lists*: a list $[x_1, \ldots, x_n]$ is said to be homogeneous under a function h if $h(x_1) = \cdots = h(x_n)$.

Lemma 21. *Let P be a monoid whose universe is in $\mathscr{T}_{FO}$. The following conditions are sufficient for the product operation to be a generalised first-order list function:*

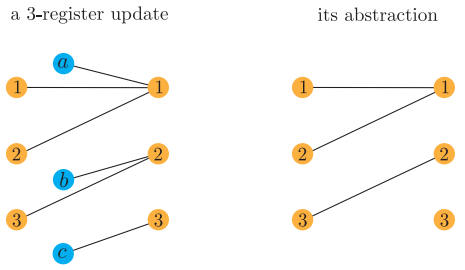
1. *the binary product $P \times P \to P$ is a generalised first-order list function; and*
2. *there is a monoid homomorphism $h : P \to T$, with T a finite aperiodic monoid, and a generalised first-order list function $P^* \to P$ that agrees with the product operation of P on all lists that are homogeneous under h.*

Proof
Our goal is to compute the product of a list $x \in P^*$. Consider h and T as given in condition **2.** and compute an h-factorisation of x with a first-order list function using Theorem 4. The depth of such a tree is bounded by a constant depending only on T. Then, by induction on the depth, one can prove that there is a generalised first-order list function computing the product of the labels of the leaves, using condition **1.** to deal with nodes of degree 2 and condition **2.** to deal with nodes of degree at least 3. By composition, we get that the product operation of P^* is a generalised first-order list function. □

In order to prove Theorem 20, it suffices to show that if a monoid M satisfies the assumptions of Theorem 20, then the monoid $M^{[k]}$ satisfies conditions 1 and 2 in Lemma 21. Let us fix for the rest of this section a monoid M which satisfies the assumptions of Theorem 20, i.e. its universe

is in $\mathscr{T}_{\mathrm{FO}}$ and its product operation is a generalised first-order list function. Condition 1 of Lemma 21 for $M^{[k]}$ is easy. Indeed, consider two elements of $M^{[k]}$, say $u = [u_1, u_2, \ldots, u_k]$ and $v = [v_1, v_2, \ldots, v_k]$. The u_i's and v_i's are lists of elements in $M \cup \{1, \ldots, k\}$. To obtain the product, we need to replace every occurrence of $j \in \{1, \ldots, k\}$ in the v_i's by u_j. Because $\{1, \ldots, k\}$ is finite, using the if-then-else construction, one can prove that the function from $\{1, \ldots, k\} \times M^{[k]}$ associating (i, u) with the register update $\mathsf{projection}_i(u)$ is a generalised first-order list function. Then, as a first step, replace every element s of M in the v_i's by the singleton list $[s]$. Then, replace every element j of $\{1, \ldots, k\}$ in the v_i's by the pair (j, u). Finally apply the generalised first-order function, as defined above, to those elements. The desired list is obtained by flattening.

We focus now on condition 2, i.e. showing that the product operation can be computed by a first-order list function, for lists which are homogeneous under some homomorphism into a finite monoid. For this, we need to find the homomorphism h. For a k-register update η, define $h(\eta)$, called its *abstraction*, to be the same as η, except that all monoid elements are removed from the right hand sides, as in the following picture:



Intuitively, the abstraction only says which registers are moved to which ones, without saying what new monoid elements (in blue in the picture) are created. Having the same abstraction is easily seen to be a congruence on $M^{[k]}$, and therefore the set of abstractions, call it T_k, is itself a finite monoid, and the abstraction function h is a monoid homomorphism. We say that a list $[x_1, \ldots, x_n]$ in $M^{[k]}$ is τ-homogeneous for some τ in T_k if it is homogeneous under the abstraction h and $\tau = h(x_1) = \cdots = h(x_n)$. We claim that item 2 of Lemma 21 is satisfied when using the abstraction homomorphism.

Lemma 22. *Given a non-negative integer k and $\tau \in T_k$, there is a generalised first-order list function from $(M^{[k]})^*$ to $M^{[k]}$ which agrees with the product in the monoid $M^{[k]}$ for arguments which are τ-homogeneous.*

Since there are finitely many abstractions, and a generalised first-order list function can check if a list is τ-homogeneous, the above lemma yields item 2 of Lemma 21 using a case disjunction as described in Example 11. Therefore, proving the above lemma finishes the proof of Theorem 20, and therefore also of Theorem 13. The rest of the section is devoted to the proof of Lemma 22. In Section 6.1, we prove the special case of Lemma 22 when $k = 1$, and in Section 6.2, we deal with the general case (by reducing it to the case $k = 1$).

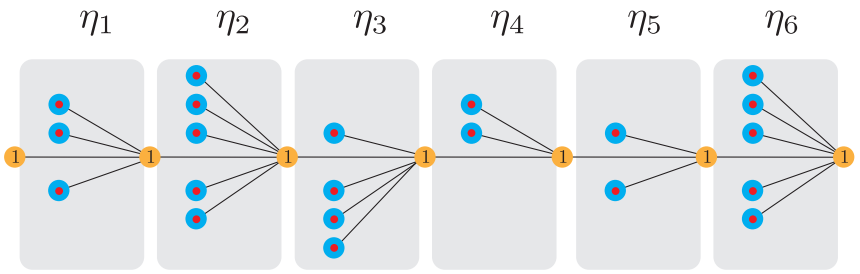
6.1 Proof of Lemma 22: One register

Let us first prove the special case of Lemma 22 when $k = 1$. In this case, there are two possible abstractions:

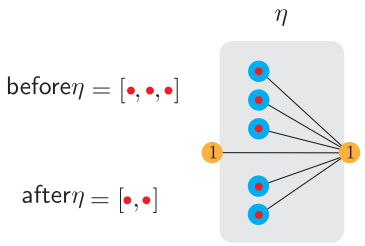
register 1 is used register 1 is not used

The right case is easy to deal with: in that case, the product of a sequence of elements in $(M^{[k]})^*$ which are τ-homogeneous, is equal to the last element of the list. This can be obtained using the first-order list function last from Example 4.

Let us consider now the more interesting left case; fix τ to be the left abstraction above. Here is a picture of a list $[\eta_1, \dots, \eta_n] \in (M^{[1]})^\star$ which is τ-homogeneous:



Our goal is to compute the product of such a list, using a generalised first-order list function. For $\eta \in M^{[1]}$ define $\mathsf{before}(\eta)$ (respectively, $\mathsf{after}(\eta)$) to be the list in M^* of monoid elements that appear in η before (respectively, after) register 1. Here is a picture



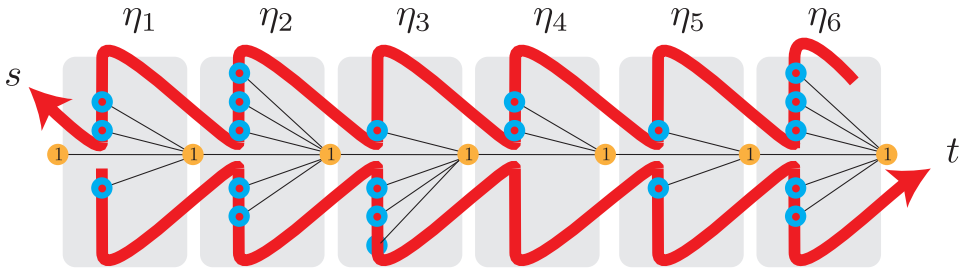
Using the list comma function from Example 7, one can transform η by grouping the elements of M in lists, using registers as separators. This way, η is transformed into the list of lists $[\mathsf{before}(\eta), \mathsf{after}(\eta)]$. Using head and last, we get:

Claim 23. *Both* before *and* after *are generalised first-order list functions* $M^{[1]} \to M^*$.

Let $s, t \in M^*$ be the respective flattenings of the lists

$$[\mathsf{before}(\eta_n), \dots, \mathsf{before}(\eta_1)] \quad \text{and} \quad [\mathsf{after}(\eta_1), \dots, \mathsf{after}(\eta_n)].$$

Note the reverse order in the first list. Here is a picture:



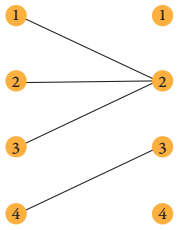
The function $[\eta_1, \dots, \eta_n] \mapsto (s, t)$ is a generalised first-order list function, using Claim 23, map, reversing, flattening and pairing. The product of the 1-register valuations $[\eta_1, \dots, \eta_n]$ is the

register valuation where the (only) right hand side is the concatenation of $s, [1], t$. Therefore, this product can be computed by a generalised first-order list function.

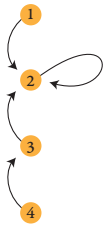
This completes the proof of Lemma 22 in the case of $k = 1$, in particular we now know that Theorem 20 is true for $k = 1$.

6.2 Proof of Lemma 22: More registers

We now prove the general case of Lemma 22. Let $\tau \in T_k$ be an abstraction. We need to show that a generalised first-order list function can compute the product operation of $M^{[k]}$ for inputs that are τ-homogeneous. Our strategy is to use homogeneity to reduce to the case of one register, which was considered in the previous section. As a running example (for the proof in this section) we use the following τ:

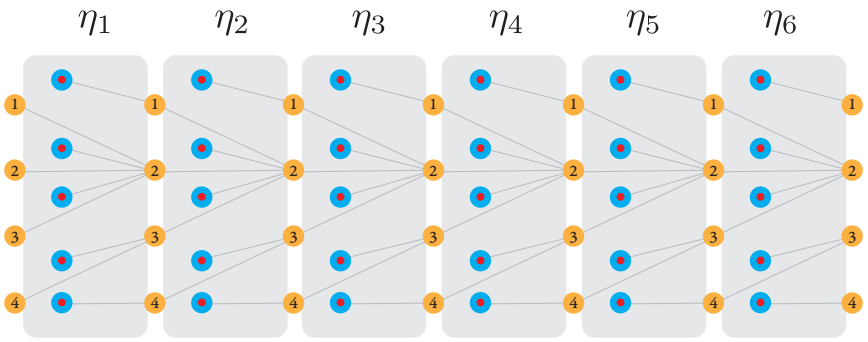


Define G to be a directed graph where the vertices are the registers $\{1, \ldots, k\}$ and which contains an edge $i \leftarrow j$ if the i-th element of the abstraction τ contains register j, i.e. the new value of register i after the update uses register j. Here is a picture of the graph G for our running example:

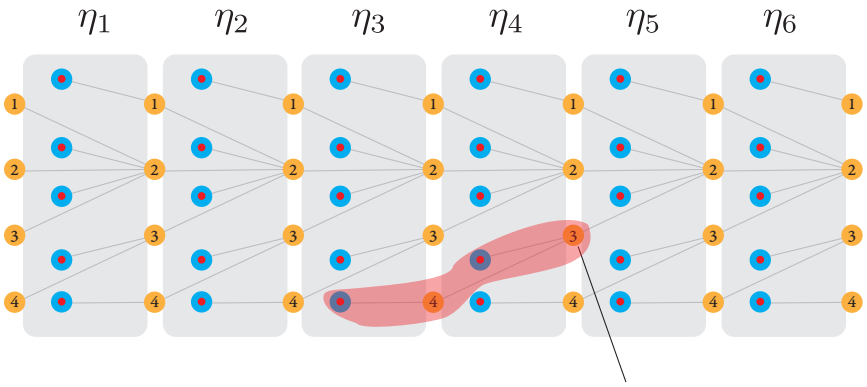


Every vertex in the graph has outdegree at most one (because τ is nonduplicating) and the only types of cycles are self-loops (because τ is monotone). Because registers that are in different weakly connected components do not interact with each other and then can be treated separately, without loss of generality we can assume that G is weakly connected (i.e. it is connected after forgetting the orientation of the edges).

Consider a τ-homogeneous list $[\eta_1, \ldots, \eta_n]$ of k-register updates. Here is a picture for our running example:



A register $i \in \{1, \ldots, k\}$ is called *temporary* if it does not have a self-loop in the graph G (we will also say that the vertex is temporary). In our running example, the temporary registers are 1,3 and 4. Because the outdegree of all the vertices in G is at most 1, if a vertex has an incoming edge from a different vertex in the graph then this latter must be temporary. The key observation about temporary registers is that their value depends only on the last k updates, as shown in the following picture:



a temporary register and what it depends on

Indeed, an incoming edge in a temporary vertex i must come from a different temporary vertex, so the value in i depends only on the values of the temporary registers corresponding to the vertices in simple paths without self-loop reaching i, and thus which occur in the last k updates.

Because the temporary registers depend only on the recent past, the values of temporary registers can be computed using a generalised first-order list function (as formalised in Claim 24 below).

Claim 24. *Assume that $\tau \in T_k$ is such that all the registers are temporary. Consider the function:*

$$(M^{[k]})^* \cap \tau\text{-homogeneous} \xrightarrow{\quad f \quad} (M^{[k]})^*$$

which maps an input $[\eta_1, \ldots, \eta_n]$ to the list $[\eta'_1, \ldots, \eta'_n]$ where η'_i is equivalent to the product of the prefix $\eta_1 \cdots \eta_i$. Then f is a generalised first-order list function.

Proof

If all registers are temporary, then the product of a τ-homogeneous list is the same as the product of its last k elements. Therefore, we can prove the lemma using the window construction from Example 10 (for a window of size k) and the binary product. □

If the graph G is connected, as supposed, then there is at most one register that is not temporary. For this register, we use the result on one register proved in the previous section. Indeed, let $[\eta_1, \ldots, \eta_n]$ be a τ-homogeneous list and let r be the only register which is not temporary. Each η_i is a list of k register updates. Let us denote it by: $\eta_i = [w_1^i, \ldots, w_k^i]$. Recall that $w_j^i \in (M \cup \{1, \ldots k\})^\star$. By using Claim 24 and map, there is a generalised first-order list function which computes $[\eta'_1, \ldots, \eta'_n]$ where $\eta'_i = [w_1'^i, \ldots, w_k'^i]$ with $w_j'^i$ equivalent to j-th right hand-side in the product $\eta_1 \cdots \eta_i$ if $j \neq r$ and equal to w_r^i if $j = r$. The list $w_j'^i$ for $j \neq r$ depends only on elements of M and possibly on the initial valuation of the registers. Then, the list $[\eta'_1, \ldots, \eta'_n]$ can be treated as a list of updates, using only one register (register r), which is solved in the previous section.

7 Regular list functions

In Theorem 13, we have shown that FO-transductions are the same as first-order list functions. In this section, we discuss the MSO version of the result.

An MSO-transduction is defined similarly as an FO-transduction (see Definition 11), except that the interpretations are allowed to use the logic MSO instead of only first-order logic[1]. When restricted to functions of the form $\Sigma^* \to \Gamma^*$ for finite alphabets Σ, Γ, these are exactly the regular string-to-string functions discussed in the introduction.

To capture MSO-transductions, we extend the first-order list functions with product operations for finite groups in the following sense. Let G be a finite group. Define its *prefix multiplication function* to be

$$[g_1, \cdots, g_n] \in G^* \qquad \mapsto \qquad [h_1, \ldots, h_n] \in G^*$$

where h_i is the product of the list $[g_1, \ldots, g_i]$. Let the *regular list functions* to be defined the same way as the first-order list functions (Definition 1), except that for every finite group G, we add its prefix multiplication function to the base functions.

Theorem 25. *Given $\Sigma, \Gamma \in \mathscr{T}$ and a function $f : \Sigma \to \Gamma$, the following conditions are equivalent:*

1. *f is defined by an MSO-transduction;*
2. *f is a regular list function.*

Proof. The bottom-up implication is straightforward, since the group product operations are seen to be MSO-transductions (even sequential functions).

For the top-down implication, we use a number of existing results to break up an MSO-transduction into smaller pieces which turn out to be regular list functions. By [8, Theorem 2] applied to the special case of words (and not trees), every MSO-transduction can be decomposed as a composition of (a) a rational function; followed by (b) an FO-transduction. Since FO-transductions are contained in regular list functions by Theorem 7, and regular list functions are closed under composition, it is enough to show that every rational function is a regular list function. By Elgot and Mezei [12], every rational function can be decomposed as: (a) a sequential function [15, Section 2.1]; followed by (b) reverse; (c) another sequential function; (d) reverse again. Since regular list functions allow for reverse and composition, it remains to deal with sequential functions. By the Krohn-Rhodes Theorem [22, Theorem A.3.1], every sequential function is a composition of sequential functions where the state transformation monoid of the underlying automaton is either aperiodic (in which case we use Theorem 7) or a group (in which case we use the prefix multiplication functions for groups).

[1] This definition differs slightly from MSO-transductions as defined in [10, Section 1.7], because it does not allow guessing a colouring, but for functional transductions on objects from our type system $\mathscr{T}$, the colourings are superfluous and can be pushed into the formulas from the interpretation.

An alternative approach to proving the top-down implication would be to revisit the proof of Theorem 13, with the only important change being a group case needed when computing a factorisation forest for a semigroup that is not necessarily aperiodic. □

8 Conclusion

The main contribution of the paper is to give a characterisation of the regular string-to-string transducers (and their first-order fragment) in terms of functions on lists, constructed from basic ones, like reversing the order of the list, and closed under combinators like composition.

One of the principal design goals of our formalism is to be easily extensible. We end the paper with some possibilities of such extensions, which we leave for future work.

One idea is to add new basic types and functions. For example, one could add an infinite atomic type, say the natural numbers $\mathbb{N}$, and some functions operating on it, say the function $\mathbb{N} \times \mathbb{N} \to \{0, 1\}$ testing for equality. Is there a logical characterisation for the functions obtained this way?

MSO-transductions and FO-transductions are linear in the sense that the size of the output is linear in the size of the input; and hence our basic functions need to be linear and the combinators need to preserve linear functions. What if we add basic operations that are non-linear, e.g.

$$(a, [b_1, \ldots, b_n]) \quad \mapsto \quad [(a, b_1), \ldots, (a, b_n)]$$

which is sometimes known as "strength"? A natural candidate for a corresponding logic would use interpretations where output positions are interpreted in pairs (or triples, etc.) of input positions.

Finally, our type system is based on lists, or strings. What about other data types, such as trees, sets, unordered lists, or graphs? Trees seem particularly tempting, being a fundamental data structure with a developed transducer theory, see e.g. [4]. Lists and the other data types discussed above can be equipped with a monad structure, which seems to play a role in our formalism. Is there anything valuable that can be taken from this paper which works for arbitrary monads?

References

1 Alfred V Aho and Jeffrey D Ullman. A Characterization of Two-Way Deterministic Classes of Languages. *J. Comput. Syst. Sci.*, 4(6):523–538, 1970.

2 Rajeev Alur and Pavol Cerný. Expressiveness of streaming string transducers. In Kamal Lodaya and Meena Mahajan, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2010, December 15-18, 2010, Chennai, India*, volume 8 of *LIPIcs*, pages 1–12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

3 Rajeev Alur and Pavol Černý. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '11*, page 599, New York, New York, USA, 2011. ACM Press.

4 Rajeev Alur and Loris D'Antoni. Streaming Tree Transducers. *J. ACM*, 64(5):1–55, 2017.

5 Rajeev Alur, Adam Freilich, and Mukund Raghothaman. Regular combinators for string transformations. *CSL-LICS*, pages 1–10, 2014.

6 Alur, Rajeev and Cerný, Pavol. Expressiveness of streaming string transducers. *FSTTCS*, 2010.

7 M. Bojańczyk. *Factorization forests*, volume 5583 LNCS. 2009.

8 Thomas Colcombet. A Combinatorial Theorem for Trees. In *Automata, Languages and Programming*, pages 901–912. Springer, Berlin, Heidelberg, Berlin, Heidelberg, July 2007.

9 B. Courcelle and J. Engelfriet. *Graph Structure and Monadic Second-Order Logic - A Language-Theoretic Approach*, volume 138 of *Encyclopedia of mathematics and its applications*. CUP, 2012.

10 Bruno Courcelle and Joost Engelfriet. *Graph Structure and Monadic Second-Order Logic - A Language-Theoretic Approach*, volume 138 of *Encyclopedia of mathematics and its applications*. Cambridge University Press, 2012.

11 Luc Dartois, Ismaël Jecker, and Pierre-Alain Reynier. Aperiodic string transducers. In Srecko Brlek and Christophe Reutenauer, editors, *Developments in Language Theory - 20th International Conference, DLT 2016, Montréal, Canada, July 25-28, 2016, Proceedings*, volume 9840 of *Lecture Notes in Computer Science*, pages 125–137. Springer, 2016.

12 C C Elgot and J E Mezei. On Relations Defined by Generalized Finite Automata. *IBM Journal of Research and Development*, 9(1):47–68, 1965.

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

14 Emmanuel Filiot, Shankara Narayanan Krishna, and Ashutosh Trivedi. First-order definable string transformations. In Venkatesh Raman and S. P. Suresh, editors, *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014, December 15-17, 2014, New Delhi, India*, volume 29 of *LIPIcs*, pages 147–159. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014.

15 Emmanuel Filiot and Pierre-Alain Reynier. Transducers, logic and algebra for functions of finite words. *SIGLOG News*, 2016.

16 Eitan M. Gurari. The equivalence problem for deterministic two-way sequential transducers is decidable. *SIAM Journal on Computing*, 11(3):448–452, 1982.

17 Wiflrid Hodges. *Model Theory*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1993.

18 Jacques Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, 2009.

19 Jacques Sakarovitch and Reuben Thomas. *Elements of Automata Theory*. Cambridge University Press, Cambridge, 2009.

20 Claude E Shannon. A mathematical theory of communication, Part I, Part II. *Bell Syst. Tech. J.*, 27:623–656, 1948.

21 Imre Simon. Factorization Forests of Finite Height. *Theor. Comput. Sci.*, 72(1):65–94, 1990.

22 Howard Straubing. *Finite Automata, Formal Logic, and Circuit Complexity*. Springer Science & Business Media, December 2012.

23 Wolfgang Thomas. Languages, Automata, and Logic. In *Handbook of Formal Languages*, pages 389–455. 1997.