Regular and First-Order List Functions

Mikołaj Bojańczyk
MIMUW
University of Warsaw
Poland
bojan@mimuw.edu.pl

Laure Daviaud
DIMAP, Department of Computer Science
University of Warwick
UK
l.daviaud@warwick.ac.uk

Shankara Narayanan Krishna
Department of Computer Science
IIT Bombay
India
krishnas@cse.iitb.ac.in

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.

ACM Reference Format:
Mikołaj Bojańczyk, Laure Daviaud, and Shankara Narayanan Krishna. 2018. Regular and First-Order List Functions. In *LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3209108.3209163

1 Introduction

Transducers, i.e. automata which produce output, are as old as automata themselves, appearing already in Shannon's paper[22, 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. [16, Section 4] or [21]. Recently, much attention has been devoted to a third, larger, class of string-to-string functions that we call "regular" following [14] and [5]. The regular string-to-string functions are those recognised by two-way automata [1], equivalently by MSO transductions [14], equivalently by streaming string transducers [5].

In [4], 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 [4].

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 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, and in particular with functional programming languages (see [6] for example): 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 [4, 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. In particular, our main goal is to limit iteration. One of the corollaries of our design is that all of our functions are clearly seen to be computable in linear time, and - for the first-order fragment - then a little more effort shows that the functions can be computed by AC0 circuits. We believe that such results are harder to see in the case of [4], because of the star-like constructions used there.

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 [16, Section 4.3] (see [9, 12, 15] for characterisations of this fragment). Our main technical result, Theorem 4.3, 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
LICS '18, July 9–12, 2018, Oxford, United Kingdom
© 2018 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5583-4/18/07…$15.00
https://doi.org/10.1145/3209108.3209163

as modulo counting, are not needed for the first-order fragment. In fact, one could say that the functions from Theorem 4.3 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 4.3 and existing decomposition results from language theory.

Organisation of the paper. In Section 2, we define the class of first-order list functions and give some 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 3 that first-order list functions contain all the aperiodic rational functions. Then, in Sections 4 and 5, we state the result and complete its proof. In Section 6, we generalise our result to deal with MSO-transductions, and we conclude the paper with future works in Section 7.

2 Definitions and Examples

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:

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

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 2.1 (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.*

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

2.2 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, as well as the three examples below.

$\textbf{projection}_1$	**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.

- **Reverse.**

$$\text{reverse} : \Sigma^* \longrightarrow \Sigma^* \qquad [w_1, \ldots, w_n] \longmapsto [w_n, \ldots, w_1]$$

- **Flat.**

$$\text{flat} : \Sigma^{**} \longrightarrow \Sigma^* \qquad [w_1, \ldots, w_n] \longmapsto \begin{cases} [] & \text{if } n = 0 \\ w_1 \cdot \text{flat}([w_2, \ldots, w_n]) & \text{otherwise} \end{cases}$$

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:

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

- **Append.**

$$\text{append} : \Sigma \times \Sigma^* \longrightarrow \Sigma^* \qquad (x_0, [x_1, \ldots, x_n]) \longmapsto [x_0, x_1, \ldots, x_n]$$

- **Co-append.**

$$\text{co} - \text{append} : \Sigma^* \longrightarrow (\Sigma \times \Sigma^*) + \bot \qquad [x_0, \ldots, x_n] \longmapsto \begin{cases} (x_0, [x_1, \ldots, x_n]) & \text{if } n \geq 1 \\ \bot & \text{otherwise} \end{cases}$$

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

- **Block.**

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

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

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

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

Figure 2. Basic functions for lists.

- **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.

Example 1. [Filter] For $\Sigma, \Gamma \in \mathscr{T}$, the function $f:(\Sigma+\Gamma)^* \to \Sigma^*$ which removes the Γ elements from the input list 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 function $a \mapsto [a]$ from Σ (which can be shown to be a first-order list function) 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 2. [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. □

Example 3. [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.) This example can be extended to produce windows of size 3,4, etc. □

A complete and detailed argument that those functions are first-order list functions can be found in the full version of the paper [8].

3 Aperiodic rational functions

The main result of this section is that the class of first-order list functions contains all aperiodic rational functions, see [20, Section IV.1] or Definition 3.5 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 [7, 23]. In Section 3.1, we state that the Factorisation Forest Theorem can be made effective using first-order list functions, and in Section 3.2, we define aperiodic rational functions and prove that they are first-order list functions.

3.1 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 3.1. *A* monoid *consists of a set M and a product operation $\pi : M^* \to M$ which is associative, i.e. for all elements $m_1, \ldots m_k$ of M, $\pi(m_1, \ldots, m_k)$ is equal to*

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

for all $1 \le \ell_1 < \ell_2 < \cdots < \ell_j < k$.

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.

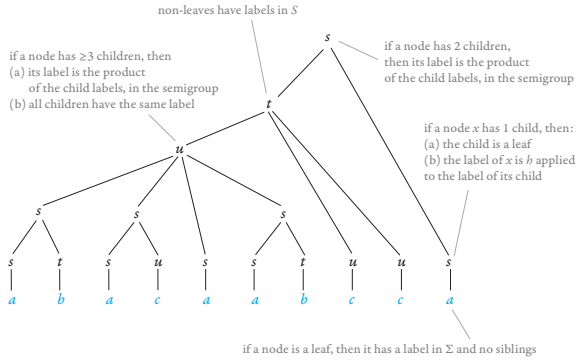
Definition 3.2. *A monoid (or semigroup) is called* aperiodic *if there exists some positive integer n such that*

$$m^n = m^{n+1} \quad \text{for every element } m \in M$$

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:

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

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 3.3. *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 are some $k \in \mathbb{N}$ and a first-order list function:*

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

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. Moreover, given Σ, h and S, one can compute such a k.

One can derive the following corollary.

Corollary 3.4. *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 aperiodic regular language over Σ, viewed as a function $\Sigma^* \to \{0, 1\}$.*

3.2 Rational functions

For the purposes of this paper, it will be convenient to consider an algebraic representation for rational functions, namely in terms of bimachines [19]. In this section, we will only be interested in the case of aperiodic functions; however we explain in section 6 that our results can be generalised to arbitrary rational functions.

Definition 3.5 (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 3.6. *Every aperiodic rational function is a first-order list function.*

The rest of Section 3.2 is devoted to showing the above theorem. The general idea is to use factorisations as in Theorem 3.3 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 3.1. 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).

Lemma 3.7. *Let $k \in \mathbb{N}$. Then there is a first-order list function $\mathrm{trees}_k(M, \Sigma) \to \mathrm{trees}_k(M \times M, \Sigma)$ which inputs a tree, and replaces the label of each non-leaf node with its sibling profile.*

Lemma 3.8. *Let $k \in \mathbb{N}$ and let Δ be a finite set. Then there is a first-order list function: $\mathrm{trees}_k(\Delta, \Sigma) \to (\Delta^* \times \Sigma)^*$ 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 (of Theorem 3.6)
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

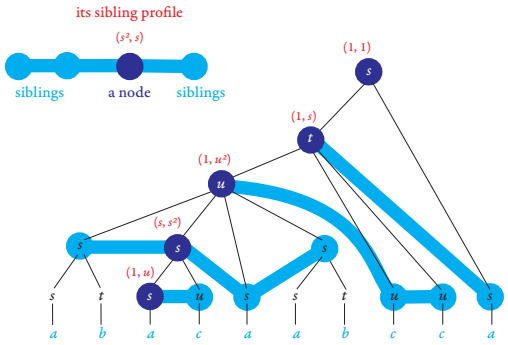
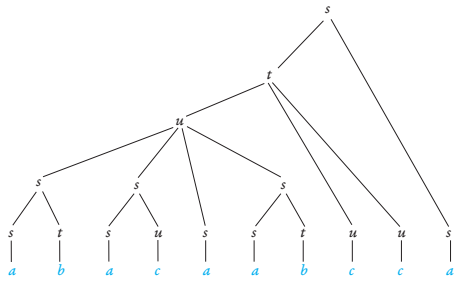


Figure 4. Sibling profiles.

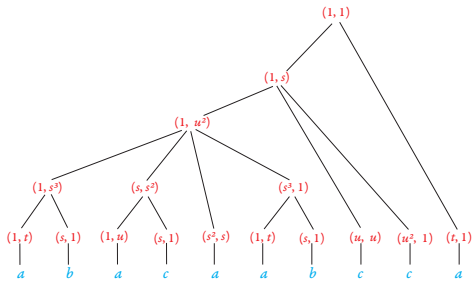
r on non-empty inputs. To extend it to the empty input we can use an if-then-else construction as in Example 2. We will define r as a composition of five functions, described below. To illustrate these steps, we will show after each step the intermediate output, assuming that the input is a word from Σ^+ that looks like this:

$$a \quad b \quad a \quad c \quad a \quad a \quad b \quad c \quad c \quad a$$

1. Apply Theorem 3.3 to h, yielding some k and a function from Σ^+ to $\mathrm{trees}_k(M, \Sigma)$ 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 3.7, which replaces the label of each non-leaf node with its sibling profile. After this step, the output looks like this:

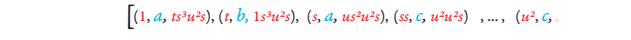


3. To the output from the previous step, we can now apply the function from Lemma 3.8, pushing all the information to the leaves, so that the output is a list that looks like this:

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

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

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:



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 3.5). 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.

□

4 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 4.1, and then in Section 4.2, 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 4.3, namely that FO-transductions are the same as first-order list functions (Section 4.3). The proof of the main result is given in Sections 4.3, 4.5 and 5.

4.1 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* [11] 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 [18, 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 4.1 (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.

4.2 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. [25, 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 τ. For example, for $\tau = B^*$, $\mathsf{type}(\tau)([b])$ evaluates to true if $b \in B$.

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 4.2. *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*

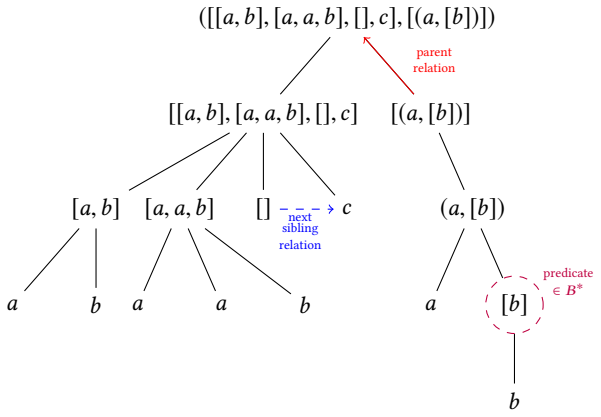


Figure 5. The parse tree of a nested list.

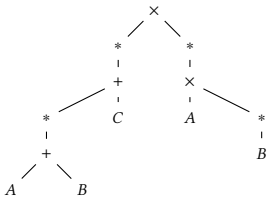
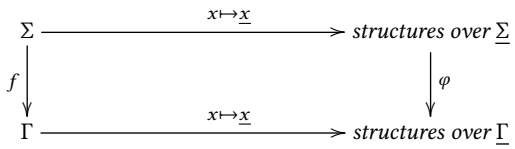


Figure 6. The parse tree of a type in $\mathscr{T}$.

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.

4.3 Main result

Below is the main contribution of this paper.

Theorem 4.3. *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 4.3, 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 4.3, the equivalence problem of first-order list functions boils down to deciding equivalence of string-to-string FO-transductions; which is decidable [17].

The proof of the left-to-right implication of Theorem 4.3 is a straightforward induction. 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 4.4. *To prove the right-to-left implication of Theorem 4.3, 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 4.3, it suffices to show that (a) and (b) can be done by first-order list functions. Step (a) is Theorem 3.6. Step (b) is described in Section 4.5. Before tackling the proofs, we need to generalise slightly the definition of first-order list functions.

4.4 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 4.5 (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 $\mathscr{T}_{FO}$ denote first-order definable subsets of types in $\mathscr{T}$.*

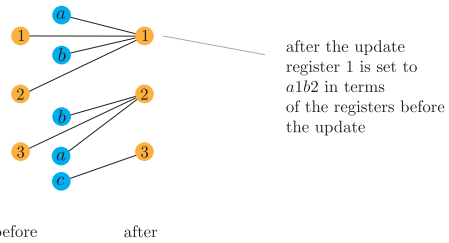
The *generalised first-order list functions* are defined to be first-order list functions as defined previously where the domains and co-domains are in $\mathscr{T}_{FO}$.

Definition 4.6 (Generalised first-order list functions)**.** *A function $f : \Sigma \to \Gamma$ with $\Sigma, \Gamma \in \mathscr{T}_{FO}$ is said to be a* generalised first-order list function *if it is obtained by taking some first-order list function (definition 2.1) and restricting its domain and co-domain to first-order definable subsets so that it remains a total function.*

4.5 Registers

To complete the proof of Theorem 4.3, it will be convenient to use a characterisation of FO-transductions which uses registers, in the spirit of streaming string transducers [2]. We use here a similar notion as the *substitution transition monoid* introduced in [12].

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 the alphabet $M \cup \{1, \ldots, k\}$. 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 4.7. *Let k be a non-negative integer, let $v \in M^k$ and assume that M is a monoid whose universe is in $\mathscr{T}_{\text{FO}}$ 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.*

Non duplicating monotone updates. A k-register update is said *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 4.8 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 4.8. *Let Σ and Γ be finite alphabets. Every* FO*-transduction $f : \Sigma^* \to \Gamma^*$ can be decomposed as:*

$$\begin{array}{ccc} \Sigma^* & \xrightarrow{\quad f \quad} & \Gamma^* \\ \downarrow{\scriptstyle g} & & \uparrow{\scriptstyle \text{projection}_1} \\ \Delta^* & \xrightarrow[\text{apply updates to } \bar{\varepsilon}]{} & (\Gamma^*)^k \end{array}$$

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 projection_1 *is the projection of a k-tuple of $(\Gamma^*)^k$ on its first component.*

Thanks to Lemma 4.4 and the closure of first-order list function under composition, it is now sufficient to prove that the bottom three functions of Lemma 4.8 are first-order list functions, in order to complete the proof of Theorem 4.3. This is the case of the function projection_1 by definition and of the aperiodic rational function g by Theorem 3.6. We are thus left to prove the following lemma, which is the subject of Section 5.

Lemma 4.9. *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.*

5 The register update monoid

The goal of this section is to prove Lemma 4.9. 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 5.1 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 4.5, since being monotone and nonduplicating are first-order definable properties.

Theorem 5.1. *Let M be a monoid whose universe is in $\mathscr{T}_{\text{FO}}$ 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 4.9 follows from the above theorem applied to $M = \Gamma^*$, and from Lemma 4.7. 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 5.1. The function from Lemma 4.9 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 4.7 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 4.9 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 4.9.

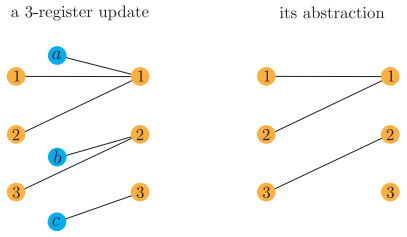
It remains to prove Theorem 5.1. We do this using factorisation forests, with our proof strategy encapsulated in the following lemma, using the notion of *homogeneous lists*: a list $[x_1, \ldots, x_n]$ is said to be homogeneous under a function h if $h(x_1) = \cdots = h(x_n)$.

Lemma 5.2. *Let M be a monoid whose universe is in $\mathscr{T}_{\text{FO}}$. The following conditions are sufficient for the product operation of $M^{[k]}$ to be a generalised first-order list function:*

1. *the binary product $M \times M \to M$ is a generalised first-order list function; and*
2. *there is a monoid homomorphism $h : M \to T$, with T a finite aperiodic monoid, and a generalised first-order list function*

$M^* \to M$ that agrees with the product operation of M on all lists that are homogeneous under h.

In order to prove Theorem 5.1, it suffices to show that if a monoid M satisfies the assumptions of Theorem 5.1, then the monoid $M^{[k]}$ satisfies conditions 1 and 2 in Lemma 5.2. Let us fix for the rest of this section a monoid M which satisfies the assumptions of Theorem 5.1, i.e. its universe is in $\mathscr{T}_{\text{FO}}$ and its product operation is a generalised first-order list function. Condition 1 of Lemma 5.2 for $M^{[k]}$ is easy, using the fact that the updates are non duplicating. We focus on condition 2, i.e. showing that the product operation can be computed by a generalised 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 5.2 is satisfied when using the abstraction homomorphism.

Lemma 5.3. *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-homogenous.*

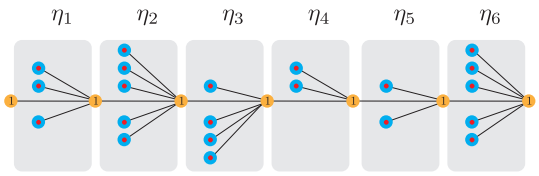
Since there are finitely many abstractions, and a generalised first-order list function can check if a list is τ-homogeneous, the above lemma yields item 2 of Lemma 5.2 using a case disjunction as described in Example 2. Therefore, proving the above lemma finishes the proof of Theorem 5.1, and therefore also of Theorem 4.3. The rest of the section is devoted to the proof of Lemma 5.3. In Section 5.1, we prove the special case of Lemma 5.3 when $k = 1$, and in Section 5.2, we deal with the general case (by reducing it to the case $k = 1$).

5.1 Proof of Lemma 5.3: One register

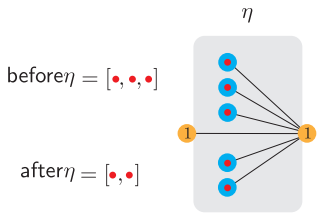
Let us first prove the special case of Lemma 5.3 when $k = 1$. In this case, there are two possible abstractions:

register 1 is used register 1 is not used

We only do the proof for the more interesting left case; fix τ to be the left abstraction above. Here is a picture of a list $[\eta_1, \ldots, \eta_n] \in M^{[1]}$ which is τ-homogeneous:



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



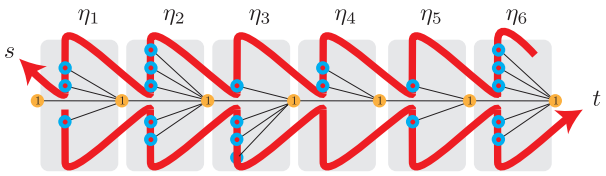
Using the block operation and flattening, we get:

Claim 5.4. *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

$$[\text{before}\eta_n, \ldots, \text{before}\eta_1] \quad \text{and} \quad [\text{after}\eta_1, \ldots, \text{after}\eta_n].$$

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

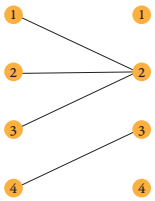


The function $[\eta_1, \ldots, \eta_n] \mapsto (s, t)$ is a generalised first-order list function, using Claim 5.4, reversing and flattening. The product of the 1-register valuations $[\eta_1, \ldots, \eta_n]$ is the register valuation where the (only) right hand side is the concatenation of $s, [1], t$. Therefore, this product can be computed by a generalised first-order list function.

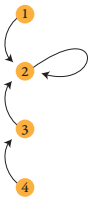
This completes the proof of Lemma 5.3 in the case of $k = 1$, in particular we now know that Theorem 5.1 is true for $k = 1$.

5.2 Proof of Lemma 5.3: More registers

We now prove the general case of Lemma 5.3. 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 homogenity to reduce to the case of 1 register, which was considered in the previous section. As a running example (for the proof in this section) we use the following τ:

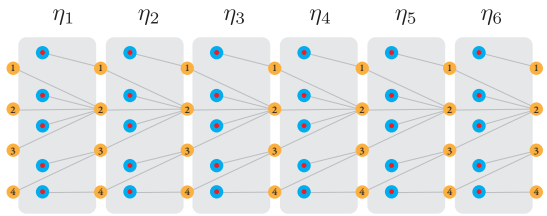


Define G to be a directed graph where the vertices are the registers $\{1, \ldots, k\}$ and which contains an edge $i \leftarrow j$ if the abstraction τ is such that $\tau(i)$ contains register j, i.e. the new value of register i after the update uses register j. Here is a picture of the graph G for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

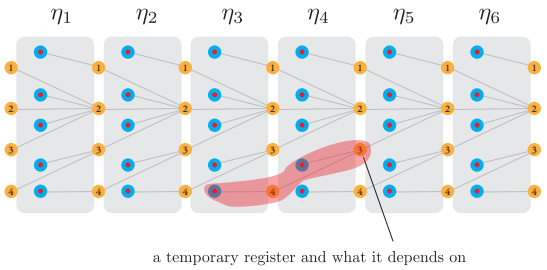


Every vertex in the graph has outdegree at most one (because τ is nonduplicating) and the only types of cycles are self-loops (because τ is monotone). Because registers that are in different weakly connected components do not interact with each other and then can be treated separately, without loss of 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. In our running example, the temporary registers are 1,3 and 4. Because of monotonicity, if a register is temporary, then all incoming edges are also from temporary vertices. The key observation about temporary registers is that their value depends only on the last k updates, as shown in the following picture:



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 5.5 below). 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 results on one register proved in the main part of the paper.

Claim 5.5. *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 3 and the binary product. □

6 Regular transductions

In Theorem 4.3, 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 4.1), except that the interpretations are allowed to use the logic MSO instead of only first-order logic. 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 $[g_1, \ldots, g_i]$. Define the *regular list functions* to be defined the same way as the first-order list functions (Definition 2.1), except that for every finite group G, we add its prefix multiplication function to the base functions.

Admittedly the group product operation may not seem at first glance an obvious programming construction (although it might look better for solvable groups, like counting modulo 2). The main point of our combinators is to have a language without any combinators that do iteration (like Kleene star, and variants of star for transducers that are used in the work Alur et al.). In the first-order case we managed to remove all iteration, and in the MSO-case we have identified the sole place where iteration is used, namely groups.

Theorem 6.1. *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 sketch. 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 [10, 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 3.6, 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 [13], every rational function can be decomposed as: (a) a sequential function [16, 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 [24, 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 3.6) or a group (in which case we use the prefix multiplication functions for groups).

An alternative approach to proving the top-down implication would be to revisit the proof of Theorem 4.3, with the only important change being a group case needed when computing a factorisation forest for a semigroup that is not necessarily aperiodic. □

7 Conclusion and future work

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. [3]. 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 aribtrary monads?

Acknowledgements

This research has been supported by the European Research Council (ERC) under the European Union Horizon 2020 research and innovation programme (ERC consolidator grant LIPA, agreement no. 683080) and the EPSRC grant EP/P020992/1 (Solving Parity Games in Theory and Practice).

References

[1] Alfred V Aho and Jeffrey D Ullman. 1970. A Characterization of Two-Way Deterministic Classes of Languages. *J. Comput. Syst. Sci.* 4, 6 (1970), 523–538.
[2] Rajeev Alur and Pavol Černý. 2011. 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*. ACM Press, New York, New York, USA, 599. https://doi.org/10.1145/1926385.1926454
[3] Rajeev Alur and Loris D'Antoni. 2017. Streaming Tree Transducers. *J. ACM* 64, 5 (2017), 1–55.
[4] Rajeev Alur, Adam Freilich, and Mukund Raghothaman. 2014. Regular combinators for string transformations. *CSL-LICS* (2014), 1–10.
[5] Alur, Rajeev and Cerný, Pavol. 2010. Expressiveness of streaming string transducers. *FSTTCS* (2010).
[6] Richard S. Bird. 1987. An Introduction to the Theory of Lists. In *Logic of Programming and Calculi of Discrete Design*, M. Broy (Ed.). Springer-Verlag, 3–42. NATO ASI Series F Volume 36. Also available as Technical Monograph PRG-56, from the Programming Research Group, Oxford University.
[7] M. Bojańczyk. 2009. *Factorization forests*. Vol. 5583 LNCS. https://doi.org/10.1007/978-3-642-02737-6_1
[8] Mikołaj Bojańczyk, Laure Daviaud, and Krishna Shankara Narayanan. 2018. Regular and First Order List Functions. *CoRR* abs/1803.06168 (2018). arXiv:1803.06168 http://arxiv.org/abs/1803.06168
[9] Olivier Carton and Luc Dartois. 2015. Aperiodic Two-way Transducers and FO-Transductions. In *24th EACSL Annual Conference on Computer Science Logic, CSL 2015, September 7-10, 2015, Berlin, Germany (LIPIcs)*, Stephan Kreutzer (Ed.), Vol. 41. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 160–174. https://doi.org/10.4230/LIPIcs.CSL.2015.160
[10] Thomas Colcombet. 2007. A Combinatorial Theorem for Trees. In *Automata, Languages and Programming*. Springer, Berlin, Heidelberg, Berlin, Heidelberg, 901–912.
[11] Bruno Courcelle and Joost Engelfriet. 2012. *Graph Structure and Monadic Second-Order Logic - A Language-Theoretic Approach*. Encyclopedia of mathematics and its applications, Vol. 138. CUP. I–XIV, 1–728 pages.
[12] Luc Dartois, Ismaël Jecker, and Pierre-Alain Reynier. 2016. Aperiodic String Transducers. In *Developments in Language Theory - 20th International Conference, DLT 2016, Montréal, Canada, July 25-28, 2016, Proceedings (Lecture Notes in Computer Science)*, Srecko Brlek and Christophe Reutenauer (Eds.), Vol. 9840. Springer, 125–137. https://doi.org/10.1007/978-3-662-53132-7_11
[13] C C Elgot and J E Mezei. 1965. On Relations Defined by Generalized Finite Automata. *IBM Journal of Research and Development* 9, 1 (1965), 47–68.
[14] Joost Engelfriet and Hendrik Jan Hoogeboom. 2001. MSO definable string transductions and two-way finite-state transducers. *ACM Transactions on Computational Logic* 2, 2 (April 2001), 216–254.
[15] Emmanuel Filiot, Shankara Narayanan Krishna, and Ashutosh Trivedi. 2014. First-order Definable String Transformations. In *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014, December 15-17, 2014, New Delhi, India (LIPIcs)*, Venkatesh Raman and S. P. Suresh (Eds.), Vol. 29. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 147–159. https://doi.org/10.4230/LIPIcs.FSTTCS.2014.147
[16] Emmanuel Filiot and Pierre-Alain Reynier. 2016. Transducers, logic and algebra for functions of finite words. *SIGLOG News* (2016).
[17] Eitan M. Gurari. 1982. The Equivalence Problem for Deterministic Two-Way Sequential Transducers is Decidable. *SIAM J. Comput.* 11, 3 (1982), 448–452. https://doi.org/10.1137/0211035 arXiv:https://doi.org/10.1137/0211035
[18] Wiflrid Hodges. 1993. *Model Theory*. Cambridge University Press. https://books.google.pl/books?id=Rf6GWut4D30C
[19] John Rhodes and Pedro V. Silva. 2008. Turing machines and bimachines. *Theor. Comput. Sci.* 400, 1-3 (2008), 182–224. https://doi.org/10.1016/j.tcs.2008.03.019
[20] Jacques Sakarovitch. 2009. *Elements of Automata Theory*. Cambridge University Press. http://www.cambridge.org/uk/catalogue/catalogue.asp?isbn=9780521844253
[21] Jacques Sakarovitch and Reuben Thomas. 2009. *Elements of Automata Theory*. Cambridge University Press, Cambridge.
[22] Claude E Shannon. 1948. A mathematical theory of communication, Part I, Part II. *Bell Syst. Tech. J.* 27 (1948), 623–656.
[23] Imre Simon. 1990. Factorization Forests of Finite Height. *Theor. Comput. Sci.* 72, 1 (1990), 65–94. https://doi.org/10.1016/0304-3975(90)90047-L
[24] Howard Straubing. 2012. *Finite Automata, Formal Logic, and Circuit Complexity*. Springer Science & Business Media.
[25] Wolfgang Thomas. 1997. Languages, Automata, and Logic. In *Handbook of Formal Languages*. 389–455. https://doi.org/10.1007/978-3-642-59126-6_7 arXiv:arXiv:1011.1669v3