

# Polyregular functions

Mikołaj Bojańczyk\*

October 23, 2018

## Abstract

This paper is about certain string-to-string functions, called the *polyregular functions*. These are like the regular string-to-string functions, except that they can have polynomial (and not just linear) growth. The class has four equivalent definitions:

1. deterministic two-way transducers with pebbles;
2. the smallest class of string-to-string functions that is closed under composition, contains all sequential functions as well as:

$$\underbrace{1|23|456|78 \mapsto 1|32|654|87}_{\text{iterated reverse}} \quad \underbrace{1234 \mapsto \underline{1}2341\underline{2}341\underline{2}341\underline{2}34}_{\text{squaring}}$$

3. a fragment of the  $\lambda$ -calculus, which has a list type constructor and limited forms of iteration such as `map` but not `fold`;
4. an imperative programming language, which has `for` loops that range over input positions.

The first definition comes from [MSV03], while the remaining three are new to the author's best knowledge. The class of polyregular functions contains known classes of string-to-string transducers, such as the sequential, rational, or regular ones, but goes beyond them because of super-linear growth. Polyregular functions have good algorithmic properties, such as:

1. the output can be computed in linear time (in terms of combined input and output size);
2. the inverse image of a regular word language is (effectively) regular.

We also identify a fragment of polyregular functions, called the *first-order polyregular functions*, which has additional good properties, e.g. the output can be computed by an  $AC^0$  circuit.

---

\*Supported by the European Research Council under the European Unions Horizon 2020 research and innovation programme (ERC consolidator grant LIPA, agreement no. 683080).

# Contents

<b>0</b>	<b>Introduction</b>	<b>4</b>
0.1	An example: all prefixes in reverse order . . . . .	7
<b>I</b>	<b>Description of the models</b>	<b>11</b>
<b>1</b>	<b>Polyregular functions</b>	<b>11</b>
1.1	Equivalent definitions . . . . .	14
1.2	Regularity preservation. . . . .	16
<b>2</b>	<b>Pebble transducers</b>	<b>18</b>
2.1	Pebble automata . . . . .	18
2.2	Pebble transducers . . . . .	23
<b>3</b>	<b>For-transducers</b>	<b>27</b>
<b>4</b>	<b>Polynomial list functions</b>	<b>30</b>
<b>II</b>	<b>Equivalence of the models</b>	<b>37</b>
<b>5</b>	<b>For-programs to pebble-transducers</b>	<b>38</b>
<b>6</b>	<b>Pebble transducers to polyregular functions</b>	<b>42</b>
6.1	One pebble . . . . .	42
6.2	Many pebbles . . . . .	55
<b>7</b>	<b>Polyregular functions to list functions</b>	<b>57</b>
7.1	Iterated reverse . . . . .	58
7.2	Squaring . . . . .	60
7.3	Sequential functions . . . . .	60
<b>8</b>	<b>List functions to for-transducers</b>	<b>64</b>
8.1	Composition of for-transducers . . . . .	65
8.2	For-transducers implementing $\beta$ -reduction . . . . .	69
<b>III</b>	<b>Algorithms</b>	<b>76</b>
<b>9</b>	<b>Evaluation algorithms</b>	<b>76</b>
9.1	Linear time . . . . .	76
9.2	$AC^0$ . . . . .	80
<b>IV</b>	<b>Appendix</b>	<b>88</b>

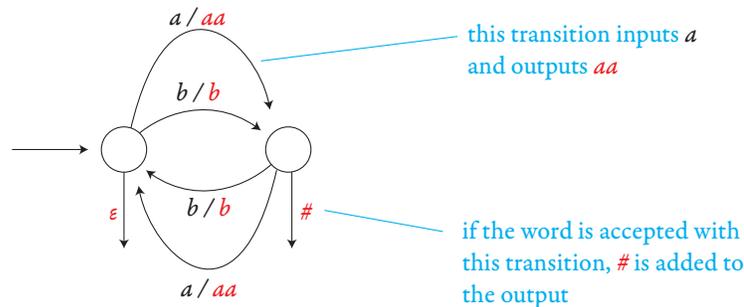
<b>A Haskell Code</b>	<b>88</b>
A.1 Atomic polynomial list programs . . . . .	88
A.2 Iterated reverse . . . . .	89
A.3 Squaring . . . . .	90
<b>B Appendix on <math>\beta</math>-reduction</b>	<b>91</b>
B.1 Values . . . . .	91
B.2 Pictures of the reduction rules . . . . .	93

## 0 Introduction

*The author (along with many other people) has come recently to the conclusion that the functions computed by the various machines are more important—or at least more basic—than the sets accepted by these devices.*  
Dana Scott [Sco67]<sup>1</sup>

This paper is about string-to-string functions that are defined by finite-state devices. There are three main classes of string-to-string functions<sup>2</sup>:

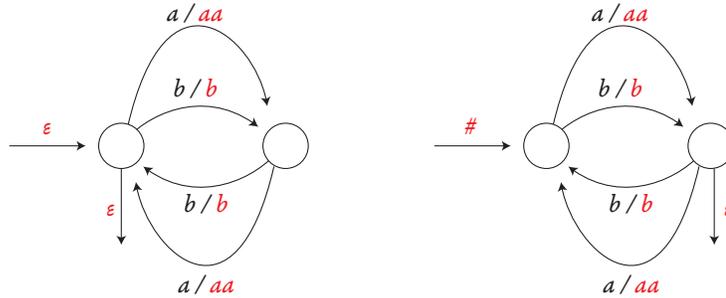
1. **Sequential functions.** The sequential functions are the ones recognised by deterministic finite automata with transitions labelled by output words (all states should be accepting if we care about total functions). Here is an example, which recognises the function that doubles every  $a$ , and appends  $\#$  in case the input has odd length:



2. **Rational functions.** Rational functions are defined like sequential functions, except that the underlying automaton is no longer required to be deterministic, but only *unambiguous*, which means that for every input word it has at most one accepting run (and exactly one accepting run if we care about total functions). Here is an example automaton (with two connected components), which recognises the function that doubles every  $a$ , and prepends  $\#$  in case the input has odd length:

<sup>1</sup>I got this quote from Wolfgang Thomas, who got it from Boris Trakhtenbrot [Tra08, p. 14].

<sup>2</sup>For more on sequential, rational and regular string-to-string functions, including additional references, see the book [STT09], the survey paper [FR16], or [BC, Sections 12, 13].



Apart from the above description, which originates from [Eil74, Chapter IX], there are other equivalent descriptions: regular expressions which use pairs of strings [STT09, Section IV.1], Eilenberg bimachines [Eil74, Chapter XI.7], and unary queries of MSO with associated outputs (see Definition 1.5 later in the paper).

3. **Regular functions.** A *regular string-to-string function* is defined to be one that is recognised by a deterministic two-way automaton with output [AU70]. Equivalent models include string-to-string MSO transductions [EH01], streaming string-transducers [Alu10], and various formalisms that use combinators [AFR14, DGK18, BDK18].

Not only are the three above classes robust (i.e. they have multiple equivalent definitions, using machines, logic, or expressions), but the associated models have the good decidability properties typical for finite automata, as illustrated by the following results. One can minimise automata for sequential [Cho79, Main Theorem], see also [Cho03, Section 3], likewise for and rational functions, see [RS91, Section 4] and [FGL16, Section 3.3]. One can decide if a rational function is already sequential [Cho77, Corollaire 3.5] and one can decide if a regular function is already rational, see [FGRS13, Theorem 1] and [BGMP15, Theorem 4]. Equivalence is decidable for sequential and rational functions (which follows from minimisation), and also for regular functions see [Gur82, Theorem 1] and [AČ11, Theorem 12].

The contribution of this paper is a proposal for fourth class in the list:

4. **Polyregular functions.** These are the string-to-string functions recognised by pebble automata, which were introduced by Globerman and Harel as acceptors [GH96] and by Milo, Suciu and Vianu as transducers [MSV03]. The class has also three other equivalent descriptions; the models in these equivalent descriptions and their equivalence are the contribution of this paper.

One of the distinguishing properties of polyregular functions, and also the reason for the “poly” in the name, is that the output size is polynomial in the input size, as opposed to the linear bounds that hold for the sequential, rational and regular functions.

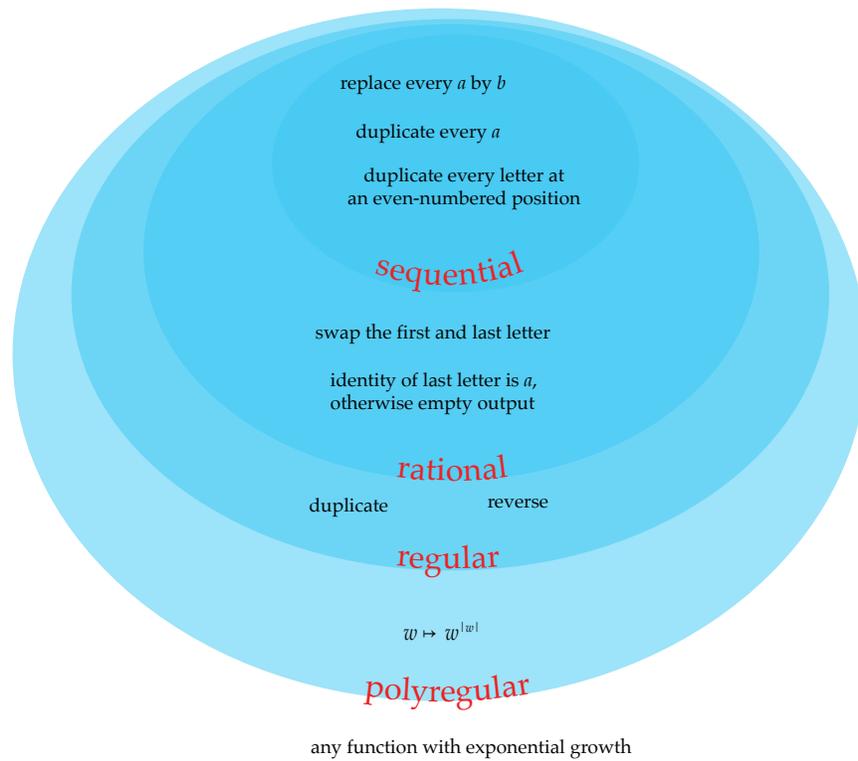


Figure 1: The sequential, rational and regular string-to-string functions.

## 0.1 An example: all prefixes in reverse order

We begin by illustrating the equivalent definitions of polyregular functions (pebble transducers as well as the three new models that are equivalent to them) on a running example. Precise definitions are given in Part I of the paper, the equivalence of the definitions is proved in Part II, while Part III discusses algorithmic questions.

The running example is the function

$$f : \{a, b\}^* \rightarrow \{a, b, |\}^*$$

which maps a word to the reverses of all prefixes, separated by  $|$ , as in this example

$$babaaa \mapsto b|ab|bab|abab|aabab|aaabab|$$

The size of the output is quadratic in the size of the input, which means that  $f$  is not recognised by a deterministic two-way automaton with output. In other words,  $f$  is not regular (and therefore it is also neither rational nor sequential). The function  $f$  is, however polyregular, as demonstrated by the following descriptions, which correspond to the four equivalent models discussed in this paper.

1. **Polyregular Functions, see Section 1.** The first definition is that the polyregular functions are the compositions of certain atomic operations. For the running example  $f$ , the composition uses four steps, illustrated below for the input word

$$babaaa$$

- (a) Append a separator symbol  $|$  giving this result:

$$babaaa|$$

- (b) Take the result of the first step, and for each position  $x$ , produce a copy of the word, with the position  $x$  underlined, giving this result:

$$\underline{b}abaaa|b\underline{a}aaa|ba\underline{b}aaa|bab\underline{a}aa|baba\underline{a}a|babaa\underline{a}|babaaa|$$

- (c) Remove the last block between separators  $|$ . For the remaining blocks, keep only the positions before and including the underlined position, and finally remove the underlines, yielding this result:

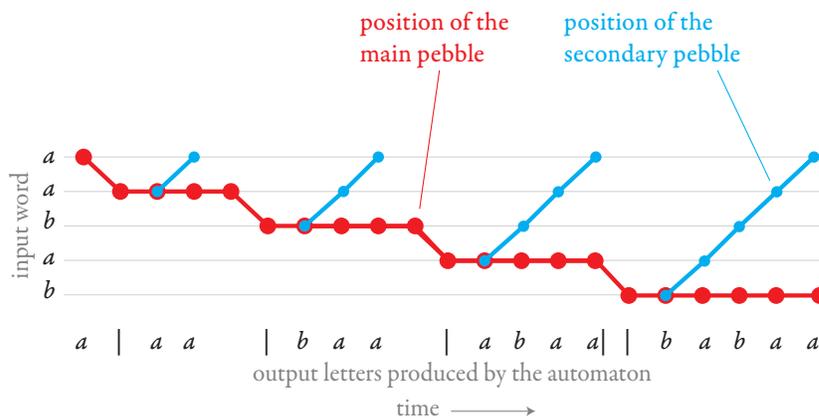
$$b|ba|bab|baba|babaa|babaaa|$$

- (d) Reverse each word between separators  $|$ , yielding this result

$$b|ab|bab|abab|aabab|aaabab|$$

The operations in steps (a) and (c) are rational functions, while the operations in steps (b) and (d) are not (we call these operation *squaring* and *iterated reverse*, respectively). The class of polyregular functions is defined to be the closure under composition of the rational functions, the squaring function, and iterated reverse.

- Pebble transducers, see Section 2.** The second description of the polyregular functions uses pebble transducers. The idea is to have an automaton which runs on the input word, and uses pebbles to mark positions. The pebbles are organised in a stack, of height fixed by the syntax of the automaton, and only the topmost pebble in the stack can be moved. To recognise the function  $f$  from the running example, we use a pebble automaton with two pebbles: a main pebble (first pebble on the stack), and a secondary pebble (second pebble on the stack). The main pebble runs through all input positions in left-to-right order. For each position, the secondary pebble is used to copy the input word from main pebble down to the beginning of the word. Here is a picture of the run of this automaton.



- For-transducers, see Section 3.** The third description uses programs which have variables that range over positions in the input word, as given in the following example.

```

for x in first..last
  for y in last..first
    if x <= y and a(x) then output a
    if x <= y and b(x) then output b
  output |

```

The input positions can be compared for order, and their labels in the input word can be tested. The output word is produced by instructions of

the form `output a`. The programming language also allows Boolean variables, which are useful to simulate the control of a finite state automaton. The programs are easily seen to be a special case of pebble automata. The opposite inclusion is also true, but harder to show, because the loops in a for-transducer can only move first-to-last or last-to-first, while the head in a pebble automaton can alternate between left and right moves an unbounded number of times.

4. **Polynomial List Functions, see Section 4.** The final description of the polyregular functions uses a functional programming language. Define `split` to be the function which inputs a list and outputs all possible ways of splitting it into two parts, as illustrated on the following example

$$\begin{array}{c}
 [1, 2, 3, 4] \\
 \downarrow \\
 [([1, 2, 3, 4], []), ([1, 2, 3], [4]), ([1, 2], [3, 4]), ([1], [2, 3, 4]), ([], [1, 2, 3, 4])]
 \end{array}$$

Let `reverse` be the function which reverses a list, and finally let `fst` be the function which projects a pair to its first coordinate. The function `f` from the running example is then defined using the following Haskell code

```
\x -> (map (\y -> reverse (fst y)) (split x)).
```

Given an input list `x`, the above program first applies the `split` operation. In the resulting list of pairs of lists, one keeps only the reverse of the first coordinate of every pair.

The general idea behind the fourth equivalent description of the polyregular functions is to use functional programs without recursion, which are equipped with certain atomic string manipulators, like `reverse`, and some higher-order combinators, like `map`.

**Structure of the paper.** The paper has three parts.

Part I introduces the four equivalent models which describe the polyregular functions.

Part II proves that the models described in Part I are equivalent. The main insights are that: (a) a pebble transducer can implement  $\beta$ -reduction and therefore evaluate  $\lambda$ -terms; and (b) results from semigroup theory such as the Krohn-Rhodes Theorem and Simon's Factorisation Forest Theorem can be used to decompose the computation of a pebble transducer in a way that can be then simulated by very limited string-to-string transformations.

Part III discusses algorithms for evaluating polyregular functions. The first result is that polyregular functions can be evaluated in linear time, in terms of the combined input and output size. This result uses constant delay enumeration algorithms for first-order queries on strings [KS13]. The second result is that first-order definable polyregular functions can be computed by  $AC^0$  circuits, as

long as the circuits can use an  $\varepsilon$  letter which is ignored when producing the output string.

**Future work.** We are missing a logical characterisation of the polyregular functions, and a streaming (one way) machine model. Such characterisations are left for future work. A natural candidate for the logical characterisation is string-to-string MSO interpretations, i.e. an extension of MSO transductions [CE12, Section 7] where a tuple of input positions can be used to represent a single output position. There are also several algorithmic questions left for future work, including: (a) is equivalence decidable for polyregular functions?; and (b) can one decide if a polyregular function is already regular?

**Acknowledgements.** I would like to thank the following people for many helpful discussions: Jacek Chrzaszcz, Amina Doumane, Sandra Kiefer, Bartek Klin, Anca Muscholl, Nathan Lhote, Aleksy Schubert, Helmut Seidl, Mahsa Shirmohammadi, Paweł Urzyczyn, Igor Walukiewicz, Daria Walukiewicz, James Worrell

## Part I

# Description of the models

In this part, we introduce the four models which describe polyregular functions. Their equivalence will be proved in Part II.

## 1 Polyregular functions

The first definition of the polyregular functions is that these are finite compositions of atomic functions that are either: a sequential function, or two string operations called squaring and iterated reverse. The design objectives for the definition of polyregular functions are:

- the class is closed under composition by definition;
- the atomic functions are as simple as possible.

The minimality of the atomic functions will make it easy to evaluate the polyregular functions, or to prove that the preimage of a regular language is always regular. The minimality will also make the formalism cumbersome to use, which is why the polyregular functions can be seen as a sort of assembly language, as opposed to more user-friendly languages defined in Sections 2, 3 and 4.

We begin by recalling in more detail the definition of sequential functions, and introducing the squaring and iterated reverse operations.

**Sequential functions.** A sequential function is a string-to-string function that arises from a deterministic automaton with outputs on transitions. The idea is that the automaton process the input word from left to right, and the output is produced during this run based only finite state control. The syntax is given in the following definition.

**Definition 1.1 (Sequential function)** *The syntax of a sequential function consists of:*

1. *input and output alphabets  $\Sigma$  and  $\Gamma$ ;*
2. *a deterministic finite automaton with input alphabet  $\Sigma$ ;*
3. *for each transition in the automaton, an associated output in  $\Gamma^*$ ;*
4. *for each state in the automaton, an associated end-of-input word in  $\Gamma^*$ .*

*The semantics of a sequential is a function  $\Sigma^* \rightarrow \Gamma^*$  defined as follows. Given an input word  $w \in \Sigma^*$ , one runs the underlying automaton. When executing a transition, the associated label defined in item 3 is produced. At the end of the run, the output is extended by the end-of-input word associated to the last state reached by the automaton.*

Apart from sequential functions, the polyregular functions use also two string-to-string operations called squaring and iterated reverse, which are described below.

**Squaring.** The squaring operation on strings is illustrated in the following example:

$$1234 \quad \mapsto \quad \underline{1}2341\underline{2}341\underline{2}341\underline{2}34$$

For each position  $x$  in the input word, we produce a copy of the input with  $x$  underlined, and then we concatenate all these copies in left-to-right order of the underlined positions. If the input word has size  $n$ , then its square has length  $n^2$ , which explains the name of the operation. Formally speaking, squaring is a family of operations, with one squaring operation for every choice of input alphabet. The output alphabet for the squaring operation is two copies of the input alphabet: the underlined and the non-underlined letters.

**Iterated reverse.** The iterated reverse operation takes a string, which contains occurrences of a separator symbol, and reverses each block between consecutive separators, but keeps the order of the blocks as it was in the input word, as illustrated in the following example:

$$123|45|678|9 \quad \mapsto \quad 321|54|876|9$$

More formally, iterated reverse is not a single operation, but a family of operations, with one iterated reverse operation for every choice of input alphabet and designated separator symbol.

**Polyregular functions.** We are now ready to give the first definition of the class of polyregular functions.

**Definition 1.2 (Polyregular functions)** *The class of polyregular functions is the smallest class of string-to-string functions which is closed under composition of functions, and contains:*

1. *sequential functions;*
2. *squaring;*
3. *iterated reverse.*

The “regular” in the name polyregular refers to the fact that polyregular functions extend regular functions, i.e. those that are recognised by two-way deterministic automata with output (this fact is most apparent with the definition from Section 2.1 that uses pebble automata). The “poly” in the name polyregular stands for polynomial, because the output of a polyregular is polynomial (possibly super-linear, unlike sequential functions which are linear) in the size of the input.

**The first-order case.** We pay particular attention to the subclass of first-order definable languages, and the corresponding functions. For readers unfamiliar with logic as a means of defining regular languages, a good place to start is [Tho97]. A *first-order definable* language  $L \subseteq \Sigma^*$ , see [Tho97, Section 2.2] is one that can be defined by a formula of first-order logic, which quantifies over positions in the input word, has a binary predicate  $x < y$  for order and unary predicates  $a(x), b(x), \dots$  for testing labels of positions. For example, the formula

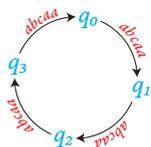
$$\forall x b(x) \Rightarrow \exists y x < y \wedge a(y)$$

says that every position with label  $b$  is followed (not necessarily in the successor position) by a position with label  $a$ . If the alphabet is  $\{a, b\}$ , then the formula defines the regular language  $(a+b)^*a$ . Every first-order sentence defines a regular language of words, but not all regular languages can be defined this way, because in general, set quantification of MSO is needed, see [Tho97, Section 4.1].

A theorem of McNaughton, Papert and Schützenberger<sup>3</sup>, says that the first-order languages can be characterized in terms of the automata that recognize them. Call a deterministic finite automaton *counter-free* if its transition monoid is aperiodic, which means that for every input word  $w \in \Sigma^*$  the following sequence of states is ultimately constant (i.e. from some point on it has only one state appearing in the sequence):

$$qw, qw^2, qw^3, \dots$$

In other words, an automaton is counter-free if it does not have a pattern like this



not all states are the same  
the same input  $w$  is used in each arrow

The McNaughton, Papert and Schützenberger Theorem says that an automaton is counter-free if and only if for every state  $q$ , the set of words which reach state  $q$  is definable in first-order logic.

**Definition 1.3 (First-order sequential and polyregular functions)** A *sequential function* is called *first-order sequential* if it is recognized by a *sequential transducer* where the underlying automaton, i.e. the automaton in item 2 of Definition 1.1, is counter-free. The *first-order polyregular functions* are the special case of *polyregular functions* where the *sequential functions* are required to be *first-order sequential*.

<sup>3</sup>See [Str18, Section 6] for a more in-depth discussion of this result and its history.

## 1.1 Equivalent definitions

One of the building blocks in the class of polyregular functions is the class of sequential functions. This building block could be replaced by other types of functions, without affecting the expressive power of the class, because closure under function composition ensures robustness of the model. We give below two alternatives for sequential functions as building blocks in the polyregular functions, one less expressive and one more expressive, and discuss why – in the presence of closure under composition – the alternatives lead the class polyregular functions.

**Krohn-Rhodes.** Although we claimed that the atomic functions in the definition of polyregular functions are minimal, this is not really the case, because sequential functions can be further decomposed. The Krohn-Rhodes Theorem, see [KR65, Corollary 4.1] or [Str12, Appendix A], says that every first-order sequential function is equal to a composition of finitely many first-order sequential functions where the underlying automaton has two states. For general (not necessarily first-order) sequential functions, one also needs sequential transducers where the underlying automaton is a group, in the sense that each input letter induces a permutation on its states. Therefore, one can replace sequential functions by the more basic building blocks from the Krohn-Rhodes Theorem, as stated in the following theorem.

**Theorem 1.4** *The class of polyregular functions is equal to the smallest class of string-functions which is closed under composition, and contains*

1. *sequential functions recognized by:*

  - (a) *two state counter-free automata; or*
  - (b) *automata where every input letter acts as a permutation on the states.*

2. *squaring;*
3. *iterated reverse.*

*The first-order polyregular functions are the special case where 1(b) is not used.*

**Rational functions.** Sequential functions are a left-to-right model. A more expressive, and symmetric, model is the rational functions. There are several ways to define rational functions, including deterministic one way automata with lookahead or nondeterministic unambiguous one way automata with output, as discussed in the introduction. We use a definition of a logical character, which is convenient for describing the first-order fragment of rational functions. In the following, by a *unary query* we mean an MSO formula (which includes the special case of a first-order formula) with one free first-order variable. We apply unary queries to words, and therefore a unary query can be viewed as defining a property of pairs (input word, distinguished position in the input word). For

example, a unary query can say “position  $x$  has label  $a$ , and all later positions have label  $b$ ”.

**Definition 1.5 (Rational function)** *A rational function is given by:*

1. *input and output alphabets  $\Sigma$  and  $\Gamma$ ;*
2. *a finite set  $\mathcal{F}$  of unary MSO queries over the input alphabet  $\Sigma$ , such that for every word in  $\Sigma^*$  and distinguished position, exactly one query from  $\mathcal{F}$  is true;*
3. *for each query from  $\mathcal{F}$ , an associated output in  $\Gamma^*$ ;*
4. *an output word for empty input in  $\Gamma^*$ .*

*The semantics is a function  $f : \Sigma^* \rightarrow \Gamma^*$  defined as follows. Suppose that  $a_1 \dots a_n \in \Sigma^*$  is an input word. If  $n = 0$ , then the output is the word from item 4. Otherwise, the output is the word  $w_1 \dots w_n$ , where  $w_i$  is the word associated in item 3 to the unique query from  $\mathcal{F}$  which selects position  $i$  in the input word.*

*A first-order rational function is the special case when all queries in  $\mathcal{F}$  are defined in first-order logic, i.e. set quantification is disallowed.*

The definition above is not in the same spirit as the definition of sequential functions from Definition 1.1. An equivalent definition of the rational functions would be to use unambiguous automata with output, or Eilenberg bimachines, and the first-order subclass would be recovered by considering an aperiodic restriction on the machines, see [LMSV01, Theorem 3.1].

**Example 1.** Consider the function

$$f : \{a, b\}^* \rightarrow \{a, b\}^* \quad f(w) = \begin{cases} w & \text{if } w \text{ ends with } a \\ \varepsilon & \text{otherwise.} \end{cases}$$

Consider the MSO (in fact, first-order) sentence

$$\varphi = \forall x \exists y \ y \geq x \wedge a(y)$$

which says that the last position has label  $a$ . The set of  $\mathcal{F}$  of queries in item 2 of Definition 1.5 has three queries

$$\underbrace{a(x) \wedge \varphi}_{\alpha(x)} \quad \underbrace{b(x) \wedge \varphi}_{\beta(x)} \quad \underbrace{\neg \varphi}_{\gamma(x)}$$

where the third query  $\gamma(x)$  does not depend on the position  $x$ . The outputs from item 3 of the definition are defined by

$$\alpha(x) \mapsto a \quad \beta(x) \mapsto b \quad \gamma(x) \mapsto \varepsilon,$$

while the output for empty input from item 4 is defined to be  $\varepsilon$ .  $\square$

As shown by Elgot and Mezei, see [EM65, Theorem 7.8], a function is rational if and only if it can be decomposed as a sequential function followed by a reverse sequential function (i.e. reverse, then a sequential function, then reverse again). Since polyregular functions have reverse built in, we get the following result.

**Theorem 1.6** *If in the definition of polyregular functions, one replaces sequential functions by rational functions, the resulting class is the same. Likewise for the first-order case.*

In this paper we use rational functions more often than sequential ones, and hence the definition of polyregular functions that uses rational functions would be more in the spirit of the technical development below.

## 1.2 Regularity preservation.

One advantage of the definition of polyregular functions in terms of composing atomic functions is that if we want to prove that a property is true for all polyregular functions, and that property is preserved under function composition, then it is enough to prove the property for the atomic functions. Here is an example.

**Theorem 1.7 (Regular preimages)** *If  $f : \Sigma^* \rightarrow \Gamma^*$  is polyregular and  $L \subseteq \Gamma^*$  is regular, then the preimage  $f^{-1}(L)$  is regular. Furthermore, if  $f$  is first-order polyregular and  $L$  is first-order definable, then  $f^{-1}(L)$  is first-order definable.*

### Proof

It is enough to prove the theorem when  $f$  is an atomic function, since the property in the statement of the theorem is preserved under function composition. The case for sequential functions is easy to see, see e.g. [Ber13, Corollary 4.2] for a stronger result, which also covers the rational functions.

It remains to deal with iterated reverse and squaring. We only do the case of squaring, the iterated reverse is handled in a similar way. Consider an alphabet  $\Sigma$  and the squaring operation

$$\text{square} : \Sigma^* \rightarrow (\Sigma + \underline{\Sigma})^*$$

We want to show that for every regular language  $L$  over the output alphabet of  $f$ , the inverse image  $\text{square}^{-1}(L)$  is also regular. Suppose that  $L$  is such a language, which is recognized by a monoid homomorphism

$$h : (\Sigma + \underline{\Sigma})^* \rightarrow M.$$

where the monoid  $M$  is finite<sup>4</sup>. Define

$$g : \Sigma^* \rightarrow M^*$$

---

<sup>4</sup>For monoids and homomorphisms as an alternative to recognising regular languages, see [Str12, Chapter V]

to be the function which inputs a word  $w$ , and replaces each position  $x$  in that letter by the value of  $h$  on the word obtained from  $w$  by underlining position  $x$ . Here is a picture:

$$w = \quad a \quad c \quad b \quad a \quad a$$

$$g(w) = \quad h(\underline{a}cbaa) \quad h(a\underline{c}baa) \quad h(ac\underline{b}aa) \quad h(acb\underline{a}a) \quad h(acba\underline{a})$$

It is not hard to see that the following diagram commutes

$$\begin{array}{ccc} \Sigma^* & \xrightarrow{\text{square}} & (\Sigma + \underline{\Sigma})^* \\ g \downarrow & & \downarrow h \\ M^* & \xrightarrow{\text{product in } M} & M \end{array}$$

The language  $\text{square}^{-1}(L)$  is the inverse image, under the function  $h \circ \text{square}$ , of some accepting set of elements  $F \subseteq M$ . Because the diagram commutes,  $\text{square}^{-1}(L)$  is also equal to the inverse image under  $g$  of the language

$$K = \{v \in M^* : \text{the product of } v \text{ is in } F\}.$$

The function  $g$  is rational and the language  $K$  is regular, and therefore the inverse image  $g^{-1}(K)$  is regular, as we have discussed at the beginning of this proof. Furthermore, if  $L$  is first-order definable, then the function  $g$  is first-order rational and the language  $K$  is first-order definable, and therefore  $g^{-1}(K)$  is also first-order definable.  $\square$

The construction in the above theorem is effective. A corollary is that one can effectively check, given a polyregular function, if some output is nonempty (because this is the same as checking if the inverse image of the language  $\{\varepsilon\}$  does not contain all input words), or if the function outputs only words of even length. Another corollary is that a language  $L \subseteq \Sigma^*$  is regular if and only if its characteristic function

$$w \in \Sigma^* \quad \mapsto \quad \begin{cases} 1 & \text{if } w \in L \\ 0 & \text{if } w \notin L \end{cases}$$

is a polyregular function. In other words, for functions with Boolean outputs, the polyregular functions are the same as the regular languages.

**Example 2.** If a function is polyregular then it has (a) polynomial size increase; and (b) preimages of regular languages are regular. One could ask if these properties characterise the polyregular functions, i.e. if the polyregular functions are exactly the string-to-string functions that have properties (a) and (b). Here is a counterexample. For a function  $f : \mathbb{N} \rightarrow \mathbb{N}$ , define

$$\hat{f} : a^* \rightarrow a^* \quad a^n \mapsto a^{f(n)}.$$

If  $f$  grows slow enough, e.g. if it is  $\log \log n$ , then  $\hat{f}$  has polynomial size increase, i.e. it satisfies condition (a). We claim that if  $f$  tends to infinity, then  $\hat{f}$  also satisfies (b). This is because it produces only words of factorial length, i.e. words of the form

$$a^1 \ a^{2!} \ a^{3!} \ \dots,$$

Indeed, it is well known that every regular language  $L \subseteq a^*$  contains either finitely many, or co-finitely many words of factorial length. In particular, if  $f$  tends to infinity, then the preimage under  $\hat{f}$  of every regular language is going to be either finite or co-finite, and therefore also regular. Summing up, if  $f$  tends to infinity slowly enough, then  $\hat{f}$  is going to satisfy conditions (a) and (b). If  $f$  is hard enough, e.g. it is not computable, then  $\hat{f}$  is not polyregular.  $\square$

## 2 Pebble transducers

In this section, we describe a second model for string-to-string functions, which will end up being equivalent to the polyregular functions from the previous section. The model is defined in terms of automata and transducers which have a two-way head and use pebbles to mark positions in an input word. For languages, pebble automata were introduced in Globerman and Harel [GH96], while the transducer version – in the more general setting of trees – comes from Milo, Suciú and Vianu as transducers [MSV03]. Without any restriction on the way that pebbles are placed, one can use pebbles to simulate logarithmic space Turing machine computation [Iba71, Corollary 3.5], and the model has many undecidable properties, e.g. the following problem is undecidable “decide if a pebble automaton accepts at least one input word”. That is why one considers pebbles with a stack discipline [GH96, Definition 4.1]: pebbles are totally ordered, and a pebble can be moved only if all pebbles smaller in the order have been lifted.

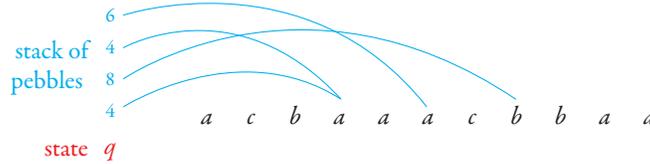
Although we are mainly interested in pebble automata not as acceptors, but as string-to-string transducers, we begin our presentation with automata (i.e. acceptors) in Section 2.1, and only then extend the definition to transducers in Section 2.2.

### 2.1 Pebble automata

The idea behind an  $k$ -pebble automaton is that at any given moment of its computation, it stores a stack of at most  $k$  positions (called pebbles) in the input word. The stack discipline condition says that only the topmost position in the stack – called the *head* – can be modified, by moving it left or right<sup>5</sup>. Also, the automaton can pop the topmost stack position, or push a new pebble

<sup>5</sup>Sometimes, the head is counted separately from the pebbles, e.g. one talks about an automaton with one pebble and one head. In this paper, the head is counted as one of the pebbles, namely the topmost one.

(which initially is equal to the head) Here is a picture of a configuration which has four pebbles:



A 1-pebble automaton is the same thing as a two-way automaton, since it can only move its head and push/pop are disallowed. We do not decorate the input word with end-markers, and therefore a pebble automaton can be run only on a nonempty word. The exact syntax of a pebble automaton is described in the following definition.

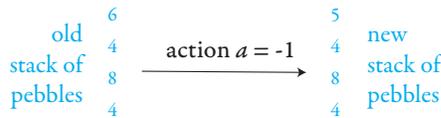
**Definition 2.1 (Pebble automaton)** A  $k$ -pebble automaton consists of:

1. a finite input alphabet  $\Sigma$ ;
2. a finite set  $Q$  of states;
3. two designated states: an initial and final one;
4. a transition function of type

$$\underbrace{Q}_{\text{current state}} \times \underbrace{\Sigma}_{\text{label under head}} \times \underbrace{(\{first, last, 1, \dots, k\}^2 \rightarrow \{\leq, \neq\})}_{\text{the order comparison of the pebbles and the first and last positions}} \rightarrow \underbrace{Q}_{\text{new state}} \times \underbrace{\{-1, 0, 1, push, pop\}}_{\text{pebble action}}.$$

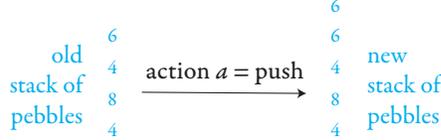
A pebble automaton is a  $k$ -pebble automaton for some  $k$ . A configuration of the automaton consists of: (a) a nonempty *input word* over the input alphabet; (b) a *control state* from the set of states; and (c) a *pebbling* which is a nonempty stack of at most  $k$  positions in the input word. The  $i$ -th position in the pebbling is called the  $i$ -th pebble, with pebble 1 being the bottom of the stack. The topmost position in the stack is called the *head*. For a configuration  $c$ , its *successor configuration*, which might be undefined, is the configuration with the same input word obtained as follows. Apply the transition function of the automaton in the natural way to  $c$ , yielding a new state  $q$  and pebble action  $a$ . The control state in the successor configuration is  $q$  and the pebbling is updated as follows:

- If the pebble action  $a$  is in  $\{-1, 0, 1\}$  then the topmost position on the stack is offset by  $a$ , i.e. the head moves by  $a$ , as in the following picture



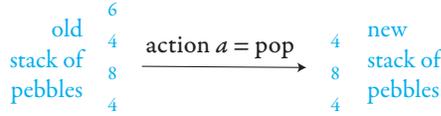
The successor configuration is undefined if adding  $a$  yields something that is not a position, i.e. the automaton moves left from the first position or right from the last position.

- If the pebble action is “push”, then the topmost position on the stack is duplicated, as in the following picture



If “push” is executed when all  $k$  pebbles are already present, then the successor configuration is undefined.

- If the pebble action is “pop”, then the topmost pebble on the stack is removed, and therefore the head is moved to the second-to-last pebble, as in the following picture



If the “pop” action is executed when there is only one pebble, then the successor configuration is undefined.

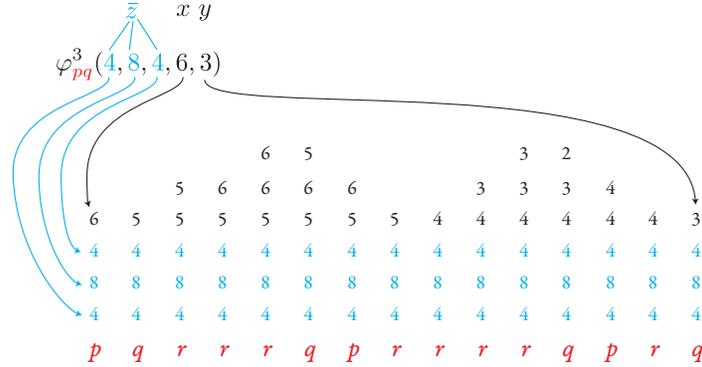
A *run* of the pebble automaton is a sequence of configurations where consecutive configurations are connected by the successor relation on configurations described above. For a given input word, the *initial configuration* of the pebble automaton has the initial state, and the pebbling has one pebble which points to the first position. The automaton accepts a word if there is an *accepting run*, i.e. a run where the first configuration is initial, the last one has an accepting state, and no other configurations have an accepting state. The accepting run, if it exists, is unique, by determinism of the transition function.

**Defining the reachability relation in logic.** We are interested in pebble automata where the reachability (not successor) relation on configurations can be defined in first-order logic, according to the following definition.

**Definition 2.2 (First-order definable pebble automaton)** *A  $k$ -pebble automaton is called first-order definable if for every states  $p$  and  $q$  and numbers  $i, j \in \{1, \dots, k\}$  there is a first-order formula*

$$\varphi_{pq}^{ij}(\underbrace{x_1, \dots, x_i}_{\bar{x}}, \underbrace{y_1, \dots, y_j}_{\bar{y}})$$

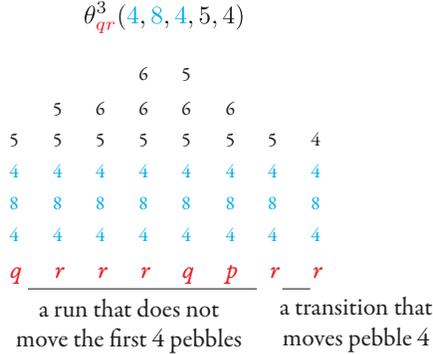




The proof is by induction on  $i$  in the opposite order, i.e. the induction base is when  $i = n - 1$ . The induction base is proved the same way as the induction step, so prove only the induction step. Suppose that we have already proved the claim for  $i + 1$ , and we want to prove it for  $i$ . Using the induction assumption, we can write for every states  $p$  and  $q$  and MSO formula

$$\theta_{pq}(z_1, \dots, z_i, x, y)$$

which describes runs as in the statement of the claim, but with the added requirement that for all configurations used in the run except for the last one, the first  $i + 1$  pebbles are the same, namely  $\bar{z}x$ . In particular, the last transition moves the  $(i + 1)$ -st pebble by some offset in  $\{-1, 0, 1\}$ , as shown in the following picture:



The formula  $\theta$  can be written using the induction assumption because the runs it describes consist of: a run that can be described using the induction assumption, and then a single transition.

To get the formula  $\psi_{pq}^i$  from the statement of the claim, we use MSO to do the following fixpoint computation. Fix an input word and a valuation of the variables  $\bar{z}$ . Consider the least set

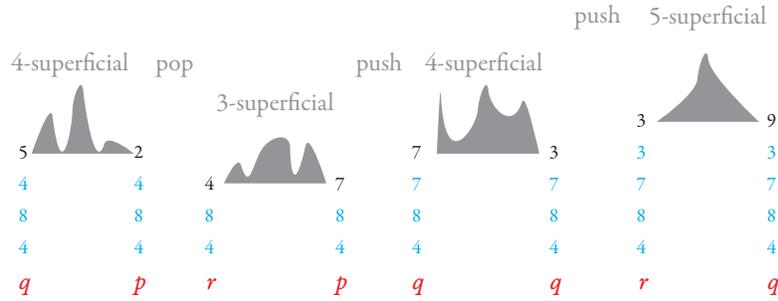
$$X \subseteq \text{states} \times \text{positions}$$

which contains the pair  $(q, x)$  and which has the following closure property:

$$\bigwedge_{r, s \in \text{states}} \quad \forall u \forall v \quad (r, u) \in X \wedge \theta_{rs}(\bar{z}, u, v) \Rightarrow (s, v) \in X$$

The formula  $\psi_{pq}^i$  from the statement of the claim then simply says that  $(q, y)$  belongs to  $X$ . This reasoning can be formalised in MSO, by encoding  $X$  a tuple of sets of positions, one for each control state of the automaton.  $\square$

The above claim shows that MSO can define reachability relation for  $i$ -superficial runs. To define the reachability relation in general, we observe that an arbitrary run can be decomposed into a concatenation of at most  $2k$  superficial runs separated by push/pop transitions, as in the following picture:



$\square$

Motivated by Lemma 2.3, we do not discuss MSO definable pebble automata.

## 2.2 Pebble transducers

We are mainly interested in pebble automata as string-to-string functions (i.e. transducers), rather than as acceptors. To get a string-to-string function, a pebble automaton is extended with output words, as described in the following definition.

**Definition 2.5 (Pebble transducer)** *A  $k$ -pebble transducer is defined to be a  $k$ -pebble automaton, plus:*

1. a finite set  $\Gamma$  called the output alphabet;
2. an output function  $Q \rightarrow \Gamma^*$ .
3. an output word for empty input in  $\Gamma^*$ .

*A pebble transducers is a  $k$ -pebble transducer for some  $k$ . A pebble transducer is called first-order definable if its underlying pebble automaton is such.*

If a pebble transducer has input alphabet  $\Sigma$  and output alphabet  $\Gamma$ , then its semantics is a partial function  $\Sigma^* \rightarrow \Gamma^*$  defined as follows. If the input word is empty, then the output is the empty output word from item 3 in the definition. If the input word is nonempty, then consider the accepting run of the underlying pebble automaton on the input. If this accepting run does not exist (i.e. the underlying pebble automaton rejects), then the output is undefined. If the accepting run exists, then apply the output function from item 2 in the definition to (the state in) each configuration, and concatenate the resulting words. This concatenation is the output of the pebble transducer. We are interested in pebble transducers that define total functions, i.e. the underlying pebble automaton accepts all nonempty inputs.

Note that a 1-pebble transducer is the same thing as a deterministic two-way automaton with output; the class of functions recognised by such devices coincides with string-to-string MSO transductions [EH01, Theorem 31] and with the class of streaming string transducers [Alu10, Theorems 1, 2, 3]. In particular, 1-pebble transducers are closed under composition, because this is true for MSO transductions. Closure under composition also extends to pebble transducers with more pebbles, as shown by Engelfriet and Maneth in [EM02, Theorem 2], see also [Eng15, Theorem 11]:

**Theorem 2.6** *The class of functions recognized by pebble transducers is closed under composition. Likewise for first-order definable pebble transducers.*

Since our syntax for pebble transducers is a little different than in [EM02, Eng15], and since we also need the closure under composition of first-order definable pebble transducers, we present a self-contained proof.

**Proof** (Of Theorem 2.6)

Consider two pebble transducers

$$\Sigma^* \xrightarrow[k \text{ pebbles}]{f} \Gamma^* \xrightarrow[m \text{ pebbles}]{g} \Delta^*$$

We assume without loss of generality that the transducer for  $f$  satisfies the following conditions

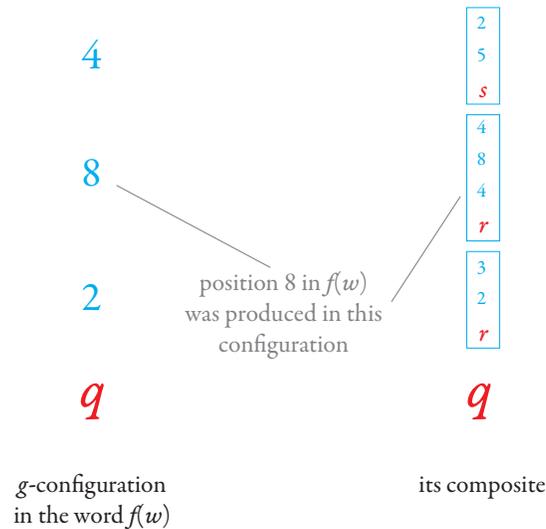
- (a) each transition produces at most one output letter; and
- (b) when a pop action is executed, then the head does not move, i.e. pop is only allowed when the topmost two pebbles are in the same place.

Every pebble transducer can be easily converted into one that satisfies (a) and (b) and produces the same outputs.

We prove below that the composition  $g \circ f$  can be computed by a pebble transducer. Consider an input word  $w \in \Sigma^*$ . In the proof we talk about configurations of  $f$  in  $w$  and about configurations of  $g$  in  $f(w)$ ; we write  $f$ -configurations for the former and  $g$ -configurations for the latter.

The idea is as follows. For an input word  $w \in \Sigma^*$ , define a *composite configuration in  $w$*  to be a state of  $g$  plus a stack of at most  $m$   $f$ -configurations

in the word  $w$ . Since all of the  $f$ -configurations have the same input word, the stack stores only the states and the pebble positions. A composite configuration can be represented using at most  $km$  pebbles. For an input word  $w \in \Sigma^*$ , and a  $g$ -configuration  $c$  over input  $f(w)$ , define its *composite* to be the result of replacing in  $c$  each pebble by the  $f$ -configuration which produced that pebble's position, as in the following picture



The automaton for the composition  $g \circ f$  simulates the run of  $g$  on  $f(w)$ , by computing the composites of the  $g$ -configurations on that run. It remains to show that the composite configurations can be updated, i.e. if we know the composite of a  $g$ -configuration  $c$ , then we can compute the composite of its successor  $g$ -configuration. To perform these updates, we use the toolkit of operations described in the following sublemma.

**Sublemma 2.6.1** *The following operations on composite configurations can be performed by a pebble automaton:*

1. *pop the topmost  $f$ -configuration;*
2. *duplicate the topmost  $f$ -configuration;*
3. *replace the topmost  $f$ -configuration by its successor  $f$ -configuration;*
4. *replace the topmost  $f$ -configuration by its predecessor  $f$ -configuration;*

**Proof**

1. Pop the pebbles at the top of the composite stack.

2. For duplication, suppose that the topmost  $f$ -configuration has  $i \in \{1, \dots, k\}$  pebbles. The simulating automaton does  $i$  left-to-right passes through the input word. In the  $j$ -th pass it waits until it sees the  $j$ -th pebble from the topmost configuration and then pushes a new copy of that pebble.
3. For the successor of the topmost configuration, use the automaton for  $f$ .
4. The predecessor configuration is the hard part. There is a smart solution to this problem that avoids using extra pebbles, and which is based on an idea of Hopcroft and Ullman [UH67], see also [BC, Lemma 13.4]. In the interest of simplicity, we present a less smart idea which uses  $k$  extra pebbles. Suppose that the composite configuration is  $C$ , and let  $c$  be the  $f$ -configuration that is at the top of the stack in  $C$ . Our goal is to produce the predecessor of  $c$ . Recall the assumption (a) described at the beginning of this proof, which says that a pop transition is only allowed when the two topmost pebbles are in the same position. A corollary of this assumption is that if we know  $f$ -configuration  $c$  and a transition  $t$  of the automaton in  $f$ , then there is a unique  $f$ -configuration that goes to  $c$  via  $t$ . Therefore, it is enough to find the transition that was executed by  $f$  when entering configuration  $c$ . To find this transition, one simply restarts the automaton  $f$  from the initial configuration, using at most  $k$  extra pebbles, until the  $f$ -configuration  $c$  is reached. This subcomputation allows us to determine the transition that was executed by  $f$  just before entering configuration  $c$ .

□

Using the above toolkit, we can compute the successor on composite configurations, with item 4 used whenever the automaton  $g$  wants to move its head to the previous position. It is not hard to see that if both  $f$  and  $g$  were first-order definable, then the same is true for the composite automaton described in the above construction. □

From Theorem 2.6, we get the following corollary, which says that the polyregular functions, as defined in Section 1, are contained in the functions recognised by pebble transducers (we will also see later on that the opposite inclusion is true as well).

**Corollary 2.7** *Every polyregular function is recognised by a pebble transducer.*

**Proof**

Since pebble transducers are closed under composition, it is enough to show that the basic building blocks of polyregular functions, namely the sequential functions, squaring, and reverse are all recognised by pebble transducers. This is easy to check. A one-way automaton with output, i.e. a sequential function, is a special case of a two-way automaton with output (which is the same as a 1-pebble transducer), and iterated reverse can easily be implemented using a two-way automaton with output. The only place where more than one pebble is needed is to implement squaring; this requires two pebbles. Note that although

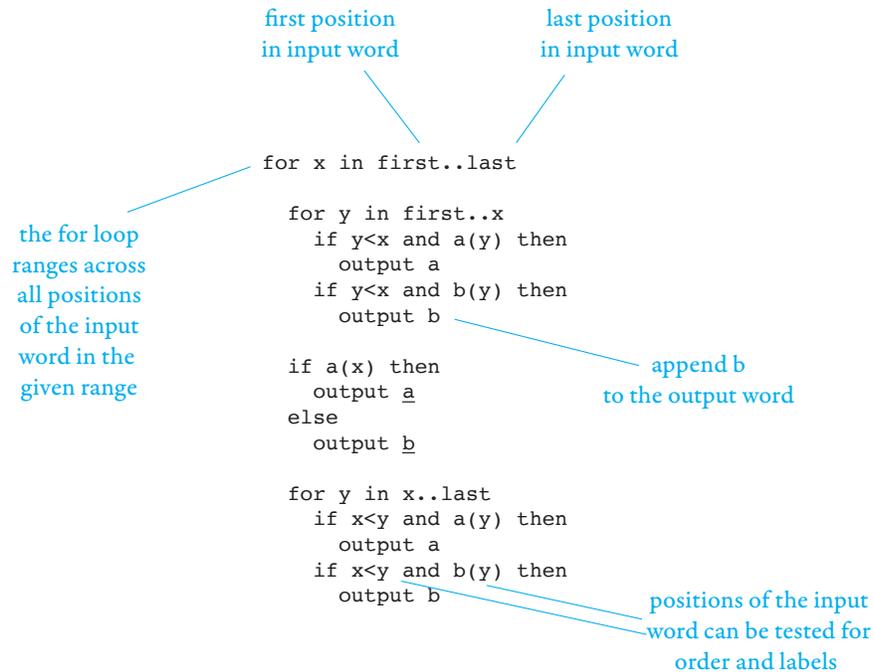
each building block of polyregular functions requires at most two pebbles, the compositions of these building blocks require more pebbles, because the composition of pebble transducers with  $k$  and  $m$  pebbles is implemented by a pebble transducer with at least  $km$  pebbles.

In general, an unbounded number of pebbles is necessary to capture all polyregular functions. This is because for a  $k$ -pebble automaton, if the input has length  $n$ , then the output has length at most  $O(n^k)$ , since the number of configurations is at most  $|Q| \cdot n^k$ . On the other hand, there are polyregular functions (squaring iterated multiple times) where the output size is given by a polynomial of arbitrarily high degree.  $\square$

### 3 For-transducers

The third description of polyregular functions uses a type of imperative programming language. A *for-transducers* is a type of program where loops to range over positions of the input word. The power of the programming language is constrained to the point that it can be simulated by a pebble automaton. We begin with an example for-transducer that computes the squaring function

$$\{a, b\}^* \rightarrow \{a, b, \underline{a}, \underline{b}\}^*$$



The above program illustrates almost all programming constructs allowed in for-transducers: a `for` loop ranging over positions in the input word, an `if` conditional, `a(x)` for checking if position `x` in the input word has label `a`, and an instruction `output a` which appends `a` to the output word. (There is one more feature, namely Boolean variables, which will be explained below.)

The `for` loop is of the form

```
for x in y..z
```

where `x` is the iterating variable which is introduced and bound by the loop, and each of `y` and `z` is either some variable that has been bound in a containing loop, or one of the keywords `first` or `last` representing the first and last positions in the input word. The body of the loop is executed for all positions `x` in the interval from `y` to `z` including both endpoints, in order that is either increasing or decreasing order, depending on whether `y` is smaller or bigger than `z`. Here is an example illustrating the order of positions

the identity function,  
positions `x` are processed  
in increasing order

```
for x in first..last
  if a(x) then
    output a
  else
    output b
```

the reverse function,  
positions `x` are processed  
in decreasing order

```
for x in last..first
  if a(x) then
    output a
  else
    output b
```

The final feature of the language is Boolean variables, which are illustrated in the following program, which computes the parity of length of the input word:

a Boolean variable, all Boolean variables are initialized to false

```
var odd
```

```
for x in first..last
  if odd then
    odd := false
  else
    odd := true
```

check if the  
input word  
has an odd  
number of  
positions

```
if odd then
  for x in last..first
    if a(x) then
      output a
    else
      output b
```

if the input word  
has an odd number  
of positions, then  
reverse it

```
else
  for x in first..last
    if a(x) then
      output a
    else
      output b
```

if the input word  
has an even number  
of positions, then  
output it unchanged

In general, a for-transducer uses two types of variables: *position variables*, which range over positions of the input word, and *Boolean variables* such as `odd` above, which can have value `true` or `false`. Position variables are introduced in `for` loops and one cannot use assignment `:=` to change the value of a position variable. Boolean variables can be declared in any block, e.g. in the scope of a `for` loop, and assignments for Boolean variables are allowed, i.e., one can write `b := true` or `b := false`.

This completes the description of for-transducers. It is straightforward to see that every string-to-string function recognized by for-transducer is also recognized by a pebble transducer; we will discuss this more detail in Section 5. The converse is also true and will follow from the results in Part II. The difficulty in transforming a pebble transducer into a for-transducer is that the loops in a for-transducer have a fixed direction (they sweep the input either from left to right or from right to left), while the head of a pebble transducer can move both left and right.

**Example 3.** Every sequential function is recognised by a for-transducer. The for-transducer does only one for loop of type

```
for x in first..last
```

to scan through all positions in the input word. Boolean variables are used to maintain the state of the automaton underlying the sequential transducer. To simulate rational functions, one would use two nested loops.  $\square$

**First-order definable for-transducers.** There is also a fragment of for-transducers which corresponds to the first-order restriction. Recall that all Boolean variables are initialized to `false`. A for-transducer is called *first-order* if Boolean variables can only go from `false` to `true`, but not back. In other words, the only allowed update for Boolean variables is `q := true`. For the first-order restriction, it is important that Boolean variables can be declared inside for loops, and that they are reinitialized to `false` at each iteration of the loop that they are declared in. This is illustrated in the following example.

```
for x in first..last
  for y in first..last
    var hasa
    for z in x..y
      if a(z) then
        hasa := true
    if hasa then
      output a
```

this Boolean variable  
gets reinitialized to  
false at each iteration  
of the for loop

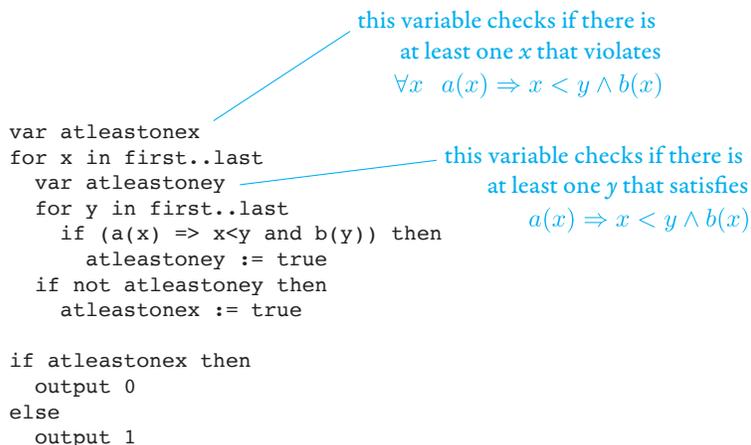
**Example 4.** In Example 3 we showed how to simulate a sequential function using a for-transducer. If we want to simulate a first-order sequential function using a first-order definable for-transducer, we use a different approach, where

the number of nested for loops will correspond to the quantifier depth of the formula. This approach is illustrated below for languages, i.e. functions with yes/no outputs.

Consider a formula of first-order logic defining a language of words. Here is a corresponding for-transducer, which outputs 1 on words in the language and 0 for words outside the language. The for-transducer is obtained using a straight-forward implementation of quantifiers using for loops; and the construction is linear in the size of the formula. For example, if the formula is

$$\forall x \exists y a(x) \Rightarrow x < y \wedge b(x)$$

then the corresponding for-transducer looks like this:



For the above program, it is important that the Boolean variable `atleastoney` gets reinitialized at each iteration of the `for` loop that binds variable `x`. Note that the transducer needs only first-to-last loops; such loops would no longer be sufficient for transducers (as opposed to yes/no formulas).  $\square$

## 4 Polynomial list functions

We now present the fourth definition of the polyregular functions, which uses a functional programming language. Roughly speaking, a polynomial list function is functional program that uses a list data type, some (higher-order) atomic functions for list manipulation like

$$\text{map} : (\tau \rightarrow \sigma) \rightarrow \tau^* \rightarrow \sigma^*$$

and which has no recursion. To model functional programs, we use the  $\lambda$ -calculus. As programming constructs we allow  $\lambda$ -abstraction and application, but no general recursion mechanisms (apart from those implicit in the atomic

functions like `map`). The language is defined so that when restricted to string-to-string functions, the polynomial list functions have the same expressive power as the polyregular functions. In particular, the polynomial list functions have outputs of at most polynomial size.

Before giving the formal definition of polynomial list functions, we begin with some examples that illustrate the programming constructs and atomic operations that are allowed.

**Example 5.** [List duplication] Suppose that  $\tau$  is some type, e.g.

$$\tau = \{1, 2, 3, 4, 5, 6\}$$

One of the atomic functions is

$$\text{concat}^\tau : (\tau^*)^* \rightarrow \tau^*$$

which flattens the input list as illustrated in the following example

$$[[1, 2, 3], [4, 5], [], [6]] \mapsto [1, 2, 3, 4, 5, 6].$$

Using `concat`, we can write a function

$$\lambda x : \tau^* . \text{concat}_\tau [x, x]$$

which duplicates the input list as in the following example

$$[1, 2, 3, 4] \mapsto [1, 2, 3, 4, 1, 2, 3, 4]$$

□

**Example 6.** [Squaring] Another atomic function is

$$\text{map}^{\tau\sigma} : (\tau \rightarrow \sigma) \rightarrow \tau^* \rightarrow \sigma^* \quad \text{for every types } \tau, \sigma$$

which applies the function in the first argument to every element of the list in the second argument. Using the function described above, we can write the following program

$$\lambda x : \tau^* . \text{map}_{\tau\tau^*} (\lambda y : \tau . x) x,$$

which has type  $\tau^* \rightarrow (\tau^*)^*$ , and which replaces each element of the input list by the list itself, as illustrated in the following example

$$[1, 2, 3, 4] \mapsto [[1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4]].$$

□

**Example 7.** [Squaring, continued] The operation in Example 6 is similar to the squaring operation described in Section 1; but it is weaker in the sense that

it does not have the underlines which distinguish consecutive copies of the input list. To recover the squaring function, we use a function

$$\mathbf{split}^\tau : \tau^* \rightarrow (\tau^* \times \tau^*)^* \quad \text{for every type } \tau$$

which splits the input list in all possible ways, as illustrated below:

$$\begin{array}{c} [1, 2, 3, 4] \\ \downarrow \\ [([1, 2, 3, 4], []), ([1, 2, 3], [4]), ([1, 2], [3, 4]), ([1], [2, 3, 4]), ([], [1, 2, 3, 4])] \end{array}$$

If the input list has length  $n$ , then the output contains  $n + 1$  pairs, with the  $i$ -th pair for  $i \in \{0, 1, \dots, n\}$  having the first  $n - i$  elements on the first coordinate, and the remaining  $i$  elements on the second coordinate.  $\square$

**Syntax and semantics** We now give a more formal description of the syntax and semantics of polynomial list functions. We begin by describing the types in the language and their associated semantic domains.

**Definition 4.1 (Types and their domains)** *Every finite set is a type<sup>6</sup>, and if  $\tau, \sigma$  are types, then so are:*

$$\tau^* \quad \tau + \sigma \quad \tau \times \sigma \quad \tau \rightarrow \sigma$$

For a type  $\tau$ , its associated domain  $\llbracket \tau \rrbracket$  is defined as follows:

- if  $\tau$  is a finite set, then  $\llbracket \tau \rrbracket = \tau$ ;
- $\llbracket \tau \rightarrow \sigma \rrbracket$  is the set of all total functions from  $\llbracket \tau \rrbracket$  to  $\llbracket \sigma \rrbracket$ ;
- $\llbracket \tau \times \sigma \rrbracket$  is the product  $\llbracket \tau \rrbracket \times \llbracket \sigma \rrbracket$ , likewise for disjoint union  $+$ ;
- $\llbracket \tau^* \rrbracket$  is the set of all finite lists of elements in  $\llbracket \tau \rrbracket$ .

There is not much difference between  $\tau$  and  $\llbracket \tau \rrbracket$ , except that the former can be viewed as a syntactic expression of finite size, while the latter is a typically infinite set. An important issue is that the functions in  $\llbracket \tau \rightarrow \sigma \rrbracket$  are total, because we only consider always terminating programs.

**Definition 4.2** *Assume some set<sup>7</sup> of variables, each variable having an associated type, such that for every type there are infinitely many variables of that type. The terms and their associated types are generated by the following rules*

<sup>6</sup>An alternative, more minimalistic, approach would be to have only one atomic type, namely the empty set  $\emptyset$ . Then  $\emptyset^*$  would be a type which has only one element  $[]$ , and larger finite sets could be defined using disjoint union, e.g. a three element type could be encoded as

$$(\emptyset^* + \emptyset^*) + \emptyset^*.$$

Since such a representation of finite sets would be cumbersome to use, we choose to define the type system so that it has finite sets as atomic types.

<sup>7</sup>We gloss over the fact that there is no “set of all finite sets”. A more formal approach would require distinguishing only some representative family of finite sets that forms a set.

1. Every variable is a term, of same type as the variable.
2. If  $M : \tau \rightarrow \sigma$  and  $N : \tau$  are terms<sup>8</sup>, then so is  $MN : \sigma$ .
3. For every variable  $x : \tau$ , if  $M : \sigma$  is a term then so is  $(\lambda x : \sigma M) : \tau \rightarrow \sigma$ .
4. If  $\tau$  is a finite set and  $a \in \tau$ , then  $a : \tau$  is a term;
5. If  $M : \tau$  and  $N : \sigma$  are terms, then so is  $(M, N) : \tau \times \sigma$ .
6. If  $\tau_0, \tau_1$  are types and  $M : \tau_i$  is a term, then  $\iota_i M : \tau_0 + \tau_1$  is a term.
7. If  $M_1 : \tau, \dots, M_k : \tau$  are terms, then  $[M_1, \dots, M_k] : \tau^*$  is a term.
8. All atomic programs in Figure 2 are terms.

The pair, list and coprojection constructors in items 5, 7 and 6 could be replaced by extending the operations from Figure 2 with a pairing function, an empty list constant, an append function, and a coprojection function.

**Semantics.** Since there is no recursion, there is no difficulty in providing compositional denotational semantics, i.e. assigning to each term  $M : \tau$  an element of the domain  $[[\tau]]$  by induction on the size of term. For a term  $M : \tau$  with free variables  $x_1 : \tau_1, \dots, x_n : \tau_n$ , its semantics

$$[[M]] \in \underbrace{[[\tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau]]}_{\text{environment}}$$

is defined by induction on the size of  $M$  in the natural way. The semantics of the terms defined in items 1, 2, 3, 4, 5, and 7 should be self-explanatory. The term  $\iota_i$  in item 6 represents the injection of type  $\tau_i$  into the coproduct  $\tau_0 + \tau_1$ . Regarding item 8, the semantics of the atomic programs from Figure 2 is explained in the Haskell code in Section A.1.

**Definition 4.3 (Polynomial list functions)** A polynomial list function is the semantics  $[[M]]$  of some term without free variables. A first-order polynomial list function is one that does not use the group product operations.

The above definition covers functions with higher-order types, like

$$(\{a, b\}^* \rightarrow \{c\}^*) \rightarrow \{c\}^* \rightarrow \{a, b, c\}^*$$

but in the end, we will be interested in programs of type  $\tau^* \rightarrow \sigma^*$ , where  $\tau$  and  $\sigma$  are finite sets. Nevertheless, these programs will typically involve subterms of other types, including higher-order types.

**Example 8.** We write a program

$$\text{headtwo}_\tau : \tau^* \rightarrow (\tau \times \tau) + \perp$$

---

<sup>8</sup>For brevity, we write “ $M : \tau$  is a term” instead of “ $M$  is a term of type  $\tau$ ”.

$\text{is}_a^\tau$	: $\tau \rightarrow \{\mathbf{true}\} + \{\mathbf{false}\}$ <b>true</b> if the input is $a \in \tau$ and <b>false</b> otherwise (defined only when $\tau$ is a finite set)
$\text{proj}_i^{\tau_0\tau_1}$	: $(\tau_0 \times \tau_1) \rightarrow \tau_i$ projection to coordinate $i \in \{0, 1\}$
$\text{case}^{\tau_0\tau_1\sigma}$	: $(\tau_0 \rightarrow \sigma) \rightarrow (\tau_1 \rightarrow \sigma) \rightarrow (\tau_0 + \tau_1) \rightarrow \sigma$ apply first or second argument, according to case of third argument
$\text{map}^{\tau\sigma}$	: $(\tau \rightarrow \sigma) \rightarrow \tau^* \rightarrow \sigma^*$ apply function to all elements in the list
$\text{hd}^\tau$	: $\tau^* \rightarrow (\tau + \{\perp\})$ return first element, or $\perp$ when empty
$\text{tl}^\tau$	: $\tau^* \rightarrow (\tau^* + \{\perp\})$ return all but first element, or $\perp$ when empty
$\text{concat}^\tau$	: $(\tau^*)^* \rightarrow \tau^*$ concatenate list of lists: $[[1,2],[],[3],[4,5]] \mapsto [1,2,3,4,5]$
$\text{split}^\tau$	: $\tau^* \rightarrow (\tau^* \times \tau^*)^*$ return all possible ways of splitting the list in two parts $[1,2,3,4] \mapsto [([], [1,2,3,4]), ([1], [2,3,4]), ([1,2], [3,4]), ([1,2,3], [4]), ([1,2,3,4], [])]$
$\text{group}^G$	: $G^* \rightarrow G$ return the product (in the group) of the input list

Figure 2: Atomic polynomial list programs. For every types  $\tau, \tau_0, \tau_1, \sigma$  and every finite group  $G$ , the above functions are polynomial list programs. The semantics of the functions is explained using Haskell code in Section A.1.

which returns the first two elements of the input list (or the error value if input list has length at most one). The first idea that comes to mind is

$$\lambda x : \tau^* . (\mathbf{hd}_\tau x, \underbrace{\mathbf{hd}_\tau (\mathbf{tl}_\tau x)}_{\text{does not type}})$$

which does not type, because the underlined part applies  $\mathbf{hd}_\tau$  to an argument that has type  $\tau^* + \perp$ . To write a properly typed program, we need error handling. For the error handling, which is admittedly cumbersome, we use **case** and  $\iota$  to implement an error handler, which lifts a function without errors to a function with errors (which is similar to using the error monad in Haskell, but with our syntax being more verbose)

$$\mathbf{err}_{\tau\sigma} : (\tau \rightarrow \sigma) \rightarrow (\tau + \perp) \rightarrow (\sigma + \perp)$$

which is implemented by the following code

$$\lambda f : \tau \rightarrow \sigma . \mathbf{case}^{\tau \perp (\sigma + \perp)} \overbrace{(\lambda y : \tau . \iota_0^{\sigma \perp} (f y))}^{\tau \rightarrow (\sigma + \perp)} \overbrace{(\lambda y : \perp . \iota_1^{\sigma \perp} \perp)}_{\perp \rightarrow (\sigma + \perp)}$$

Using the above error handler, we can write the correct version of the program which outputs the first two elements of a list, namely

$$\lambda x : \tau^* . (\mathbf{hd}_\tau x, (\mathbf{err}_{\tau^*\tau} \mathbf{hd}_\tau) (\mathbf{tl}_\tau x))$$

The last remaining issue is that the output type of the above program is

$$(\tau + \perp) \times (\tau + \perp) \quad \text{instead of} \quad (\tau \times \tau) + \perp.$$

To convert the type on the left into the type on the right, we use **distr**.  $\square$

We end this section with two non-examples of polynomial list functions, namely **fold** and equality checks.

**Example 9.** The language lacks a fold operation, which can be viewed as an evaluator of automata:

$$\mathbf{fold} : \overbrace{(\tau \times \sigma \rightarrow \tau)}^{\text{transition function}} \rightarrow \overbrace{\tau}^{\text{first state}} \rightarrow \overbrace{\sigma^*}^{\text{input word}} \rightarrow \overbrace{\tau}^{\text{last state}}$$

Such an operation would change the expressive power, because it would go beyond polynomial growth, and polynomial list functions have polynomial growth. For example, if we take

$$\mathbf{delta} = \lambda q \lambda a \mathbf{concat}[q, q]$$

to be the function that doubles the list in the first argument regardless of the second argument, then

`fold delta [1]`

will be a program that inputs a list of length  $n$  and outputs a list of ones of length  $2^n$ . In fact, using `fold` one can get functions of primitive recursive growth, see [Hut99, 4.1]. Also, for the same reasons as described in [Hut99], `fold` would break preservation of regularity under preimages, see Theorem 1.7.  $\square$

**Example 10.** Checking equality on lists over a binary alphabet, i.e. a function

$$\text{eq} : \{0, 1\}^* \rightarrow \{0, 1\}^* \rightarrow \{0, 1\}$$

would also change the expressive power of the language, because it would break preservation of regularity under preimages.  $\square$

## Part II

# Equivalence of the models

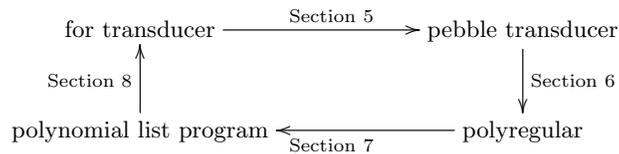
This part shows that all of the models described in Part I are equivalent. There are two variants of the equivalence result: the first-order one, and the general one.

**Theorem 4.4** *The models 1, 2, 3, 4 listed below define the same classes of string-to-string transductions. Same for 1, 2, 3, 4.*

- |   |                                    |
|---|------------------------------------|
| 1. <i>first-order pebble transducers</i>        | 1. <i>pebble transducers</i>       |
| 2. <i>first-order polyregular functions</i>     | 2. <i>polyregular functions</i>    |
| 3. <i>first-order polynomial list functions</i> | 3. <i>polynomial list function</i> |
| 4. <i>first-order for-transducers</i>           | 4. <i>for-transducers</i>          |

Polynomial list programs can define functions other than string-to-string functions. The theorem, in items 3 and 3, talks only about string-to-string functions defined by polynomial list functions, as opposed to the more general class of functions with higher order types that can be defined.

The equivalences are shown according to the following plan



Before showing all of the implications, we give some high level comments on the difficulties involved and the techniques used to solve them.

The transformation from for-transducers to pebble transducers is straightforward, and the only nontrivial part is showing that the first-order restriction in a for-transducer (the Boolean variables can only change value once) translates to the first-order restriction in a pebble transducer (there is a first-order formula defining reachability on configurations). This is proved using the same ideas as in Lemma 2.3 about MSO definability of reachability in pebble automata.

The transformation from a pebble transducer to a polyregular function is the most technical part in the proof of Theorem 4.4. The general idea is that Simon's Factorization Forest Theorem is applied to find repeating patterns in the movement of the head of a pebble transducer; and transducers where the head moves using such repeating patterns can be simulated using the atomic polyregular functions.

The transformation from polyregular functions to polynomial list programs is conceptually quite straightforward: the polyregular functions are designed to have minimal syntax, while polynomial list programs are designed to be a usable programming language. The hard part is simulating a finite automaton (which is the model underlying rational functions, which are one of the atomic

types of polyregular functions) can be simulated by a polynomial list program. The difficulty is that polynomial list programs do not have any explicit iteration mechanisms. To prove this, we use a result from [BDK18], which solved the same kind of problem: simulating a rational function using a restricted functional programming language.

Finally, to transform a polynomial list program into a for-transducer, we use two main ideas. The first main idea is that for-transducers are closed under composition, which is proved using the same kind of stack-of-stacks idea as was used in Theorem 2.6 about closure of pebble transducers under composition. The second main idea is to use a term rewriting semantics of polynomial list programs (namely, normal forms under  $\beta$ -reduction), and then to show this semantics can be implemented using a composition of finitely many for-transducers.

## 5 For-programs to pebble-transducers

In this section we show that for-transducers can be transformed into pebble transducers, as stated in the following lemma.

**Lemma 5.1** *Every string-to-string function recognized by a for-transducer is recognized by a pebble-transducer. Furthermore, if the for-transducer is first-order definable, then so is the pebble transducer.*

Without the first-order restriction, the simulation is completely straightforward: the simulating pebble transducer stores the memory state of the simulated for-program, i.e., the instruction that is about to be executed, and the valuation of the position and Boolean variables. The values of the Boolean variables are stored in the state of the pebble transducer, while the values of the position variables are stored in the pebbles. Stack discipline for the pebbles follows from the nesting of the loops in the for-program. This construction is described in more detail below, in the first-order case, where additional attention is needed.

In the first-order case, we have to show that if the for-transducer is first-order definable (which means that Boolean variables can only go from `false` to `true`) then the simulating pebble transducer is also first-order (which means that the reachability relation on configurations is definable in first-order logic). To prove this, it is convenient to assume that the for-transducer is in *prenex normal form* as defined in the following picture:

for loops can range either over  
first..last or last..first

The program begins with  
a series of nested for loops  
(called the *loop prefix*)  
with associated declarations  
of Boolean variables

*loop prefix*

```

for x in first..last
var p
var q
for y in first..last
var r
for z in first..last
var s
  if b(x) then
    q := true
  else if a(z) and q then
    p := true
    q := true
    output a
  else
    output a

```

Inside the nested for  
loops there is a *kernel*  
which does not have  
any for loops or  
variable declarations.

*kernel*

A straightforward structural induction shows the following result.

**Lemma 5.2** *For every for-program there is a for-program in prenex normal form which recognizes the same function. The construction preserves first-order definability.*

Using prenex normal form, we complete the proof of Lemma 5.1 in the first-order case. Consider a first-order for-transducer  $f$  in prenex normal form. Suppose that  $f$  has  $n$  position variables. Define a *configuration* of  $f$  in an input word  $w$  to be a tuple

$$(q, x_1, \dots, x_n)$$

where  $q$  is a valuation of the Boolean variables and  $x_1, \dots, x_i$  are positions in  $w$ . The idea is that the configuration represents the program state just before executing the kernel of  $f$ , assuming that the variables have values as given in the configuration. The for-transducer begins in the configuration where  $q$  maps all Boolean variables to **false**, and where  $x_i$  is set to either the first or last position, depending on the type of for loop that binds variable  $x_i$ .

The reachability relation on configurations is defined in the natural way: one configuration is reachable from another one (in a fixed input word) if it appears later in the computation. We write

$$w \models (q, x_1, \dots, x_n) \rightarrow^* (p, y_1, \dots, y_n)$$

to say that in the input word  $w$ , the for-transducer can go from configuration  $(q, x_1, \dots, x_n)$  to  $(p, y_1, \dots, y_n)$ . For a configuration, define the *associated output* to be the output produced by the kernel of the for-transducer assuming that the

values of the variables (position and Boolean) are as in the configuration just before executing the kernel. Because the kernel does not change the position variables, and can only test their relative order and labels in the input word, the associated output depends only on  $q$  and the quantifier-free type of the positions  $x_1, \dots, x_n$  with respect to the order and labelling predicates on positions in the input word.

The simulating pebble automaton stores  $q$  in its state and uses at most  $n$  pebbles to store the positions  $x_1, \dots, x_n$ . Therefore, to show that the simulating pebble automaton is first-order definable (and thus complete the proof of Lemma 5.1), it is enough to show that the reachability relation on configurations is definable in first-order logic. This is proved the following lemma.

**Lemma 5.3** *Let  $p, q$  be valuations of the Boolean variables and let  $i \in \{0, \dots, n\}$ . There is a first-order formula*

$$\varphi_{p,q}(\overbrace{x_1, \dots, x_n}^{\bar{x}}, \overbrace{y_1, \dots, y_n}^{\bar{y}})$$

which holds in an input word  $w$  with position tuples  $\bar{x}, \bar{y}$  if and only if

$$w \models (p, \bar{x}) \rightarrow^* (q, \bar{y})$$

**Proof**

This proof follows the same structure as Lemma 2.3. The only difference is that instead of using transitive closure (as was the case in Lemma 2.3), we use the assumption that the variables in a first-order transducer can only go from **false** to **true**.

Let  $i \in \{1, \dots, n\}$ . Define  $X_i$  to be the set of Boolean variables in the for-transducer that are declared in the  $i$ -th for loop or before. Define an  $i$ -configuration in an input word  $w$  to be a tuple of the form

$$(q, x_1, \dots, x_i)$$

where  $x_1, \dots, x_i$  are positions in  $w$  and  $q$  is a valuation of  $X_i$ . This is like a configuration, except that it does not contain the valuations of all variables. Intuitively, an  $i$ -configuration describes the first configuration in an iteration of the  $i$ -th loop. A configuration, in the sense used in the statement of the lemma, is the same thing as an  $i$ -configuration for  $i = n$ . Conversely, an  $i$ -configuration can be viewed as a configuration (i.e., an  $n$ -configuration) by setting all remaining Boolean variables to **false** and setting all positions  $x_{i+1}, \dots, x_n$  to their initial values in their corresponding loops (which are first or last positions in  $w$ , depending on whether the corresponding loops are **first..last** or **last..first**). We write

$$w \models \underbrace{(p, \bar{x})}_{i\text{-configuration}} \rightarrow_i^* \underbrace{(q, \bar{y})}_{i\text{-configuration}}$$

if  $\rightarrow^*$  holds for the corresponding  $n$ -configurations.

**Sublemma 5.3.1** *Let  $i \in \{1, \dots, n\}$  and let  $p, q$  be valuations of  $X_i$ . There is a first-order formula*

$$\varphi_{p,q}(\overbrace{z_1, \dots, z_{i-1}}^{\bar{z}}, x, y)$$

*which holds in an input word  $w$  and a valuation of  $\bar{z}$  and  $y, z$  if and only if*

$$w \models (p, \bar{z}x) \rightarrow_i^* (p, \bar{z}y)$$

**Proof**

Note that in the statement of the sublemma, we do not have to say that the run from  $(p, \bar{z}x)$  to  $(p, \bar{z}y)$  does not change the values of the variables in  $\bar{z}$ ; because this is automatically true in for-transducers, where a position variable can never return to a previously used value once that value has changed. (The same is true for Boolean variables in a first-order for-transducer.) Induction on  $i$ , in the opposite order, i.e. the induction base is when  $i = n$ . Suppose that we want to prove the statement for some  $i \in \{1, \dots, n\}$  and we have already proved it for  $j > i$ . Let us write

$$w \models (p, \bar{z}x) \rightarrow_i (q, \bar{z}y) \tag{1}$$

when there is a run from  $(p, \bar{z}x)$  to  $(p, \bar{z}y)$  which contains no  $i$ -configurations except for the source and target. Note that in the one step case, the variable  $y$  is uniquely determined by  $x$ : it is either the next or previous position, depending on the type of the  $i$ -th for loop. Using the induction assumption, we can write a first-order formula

$$\psi_{pq}(\bar{z}x)$$

which holds whenever (1) is satisfied, with  $y$  being the next/previous position corresponding to  $x$ . Next, consider the case when multiple steps are allowed, but the valuations of the Boolean variables are the same all the time, i.e.

$$w \models (p, \bar{z}x) \rightarrow_i^* (p, \bar{z}y). \tag{2}$$

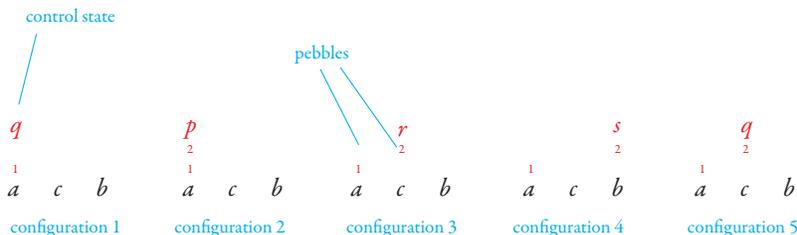
Note that the assumption that  $f$  is a first-order transducer implies that in all intermediate  $i$ -configurations between  $(p, \bar{z}x)$  and  $(p, \bar{z}y)$ , the valuation of the Boolean variables  $X_i$  is also  $p$ , because a Boolean variable can never go from **false** to **true** and then back again. Thanks to this observation, we can define (2) in first-order logic, by saying that  $\psi_{pq}(\bar{z}u)$  holds for all  $u$  between  $x$  and  $y$ . Since the Boolean variables can only grow in a run, the formula in the statement of the lemma is obtained by combining a bounded number of steps of the form (1) and (2).  $\square$

Using the above sublemma, we get Lemma 5.3 easily, by the same reasoning as in the end of the proof of Lemma 2.3. The observation is that an arbitrary run of the for-transducer can be decomposed into a bounded number of steps, which can be described in first-order logic using the sublemma.  $\square$

## 6 Pebble transducers to polyregular functions

In this section we show that every pebble transducer recognises a polyregular function. The converse inclusion was already discussed in Section 2.1. Furthermore, since each of the atomic polyregular functions is recognised by a 2-pebble transducer, it follows that every function recognised by a  $k$ -pebble transducer can be decomposed into finitely many functions recognised by 2-pebble transducers. One pebble is not enough, since 1-pebble transducers, also known as two-way automata with output, are closed under composition, which was proved in [CJ77, Theorem 1], see also [BC, Theorem 12.3].

The main idea is this. A run of a pebble transducer can be viewed as a sequence of words representing the configurations, as in the following picture:



The length is at most

$$(\text{number of states}) \cdot (\text{length of input word})^{1 + \text{number of pebbles}}$$

The  $1 +$  in the exponent is because the input word is copied in each configuration. We show that for every pebble transducer, the word representing the run can be computed using a polyregular function. Once the run has been computed, the output can easily be recovered using a sequential function, which simply replaces each configuration by the output produced in it.

The proof is split into two parts. In Section 6.1, we consider the case of 1-pebble automata (also known as two-way automata), and in Section 6.2, we iterate the construction for 1-pebble automata to get the result for  $k$ -pebble automata. The 1-pebble case in Section 6.1 is the core of the proof, and uses the Factorisation Forest Theorem of Imre Simon [Sim90]. Even in the case of a 1-pebble automaton, runs can have quadratic length, hence the squaring function.

### 6.1 One pebble

In this section, we show that a polyregular function can produce the run of a 1-pebble automaton. Since a 1-pebble automaton is the same thing as a two-way automaton, we use the name two-way automaton for the rest of this section. For the rest of Section 6.1 fix a two-way automaton  $\mathcal{A}$  with input alphabet  $\Sigma$  and states  $Q$ .

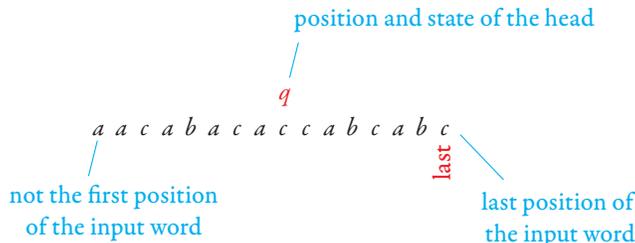
**Some notation.** We begin by fixing some notation on runs and configurations. In the proof we analyse the behaviour of the two-way automaton on infixes of the input word, which do not necessarily begin with the first or end with the last position. We use the name *partial input* for such infixes. A partial input is formally defined to be a word over the alphabet

$$\Sigma \times \mathcal{P}\{\text{first}, \text{last}\}$$

where “first” is only allowed (but not necessary) in the first position, and “last” is only allowed (but not necessary) in the last position. We write **Inputs** for the set of partial inputs; this is a regular language. Define a *partial configuration* to be a partial input with exactly one distinguished position labelled additionally by a state. A partial configuration is formalised as a word over the alphabet

$$(\Sigma \times \mathcal{P}\{\text{first}, \text{last}\}) \times (Q + \varepsilon)$$

where  $Q$  is used exactly once, and erasing the last component leads to a partial input. We write **Confs** for the set of partial configurations; again this is a regular language. Here is a picture:



Since we represent partial configurations as strings, it makes sense to talk about string-to-string transducers that transform them. One example is the successor function

$$\text{suc} : \text{Confs} \rightarrow \text{Confs}$$

which applies a single transition of the automaton. This successor function is undefined if the source partial configuration has the accepting state, or the head is moved outside the partial configuration (which should not be viewed as an error, since it corresponds to the automaton leaving the infix covered by the partial configuration). The successor function is easily seen to be a rational function; but this is not going to be useful, since the difficulty is in iterating the successor function. For example, the successor function for configurations of Turing machines is also rational.

Define a *partial run* to be a word

$$c_1 | \dots | c_n \quad c_1, \dots, c_n \in \text{Confs}$$

which consists of partial configurations, separated by a fresh separator symbol, where consecutive partial configurations are connected by the successor function, and where the last partial configuration has undefined successor (typically,

because the head leaves the part of the input covered by the partial configuration). We write  $\text{Runs}$  for the set of partial runs. Finally, define

$$\text{run} : \text{Confs} \rightarrow \text{Runs}$$

to be the (total) function which maps a partial configuration to the unique partial run which begins in that partial configuration. The main result of this section is the following lemma.

**Lemma 6.1** *The function  $\text{run}$  is polyregular. If the fixed two-way automaton is first-order definable, then  $\text{run}$  is first-order polyregular.*

For a partial input  $w$  and a state  $q$ , we write  $qw$  for the partial configuration which has input  $w$  and the head over the first position in state  $q$ . Likewise, we define  $wq$ , but with the last position used. The following slightly technical definition will be used in the induction proof that comprises the rest of this section.

**Definition 6.2 (Good)** *We say that  $L \subseteq \text{Inputs}$  is good if (a) it is a regular word language; and (b) there is a polyregular function that agrees with  $\text{run}$  on inputs from the set*

$$\{qw, wq : w \in L, q \in Q\}$$

*Furthermore, if the fixed two-way automaton is first-order definable, then we additionally require that (i)  $L$  is first-order definable, and (ii) the polyregular function in (b) is first-order polyregular.*

The main content of Section 6.1 is to prove Lemma 6.3 below, which says that all partial inputs are good. In other words, partial runs can be computed using a polyregular function, at least assuming that the first partial configuration has the head over the first or last position. The implication from Lemma 6.3 to Lemma 6.1 is a straightforward argument using crossing sequences, and is given in Sublemma 6.8.1.

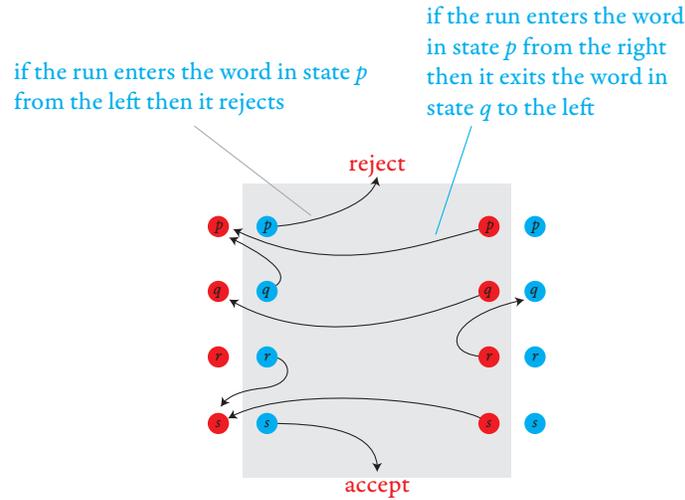
**Lemma 6.3** *The set  $\text{Inputs}$  of all partial inputs is good.*

To prove the above proposition, we apply the Factorisation Forest Theorem (see below) to the semigroup homomorphism that maps a partial input to the behaviour of the two-way automaton. We begin by describing this semigroup in more detail.

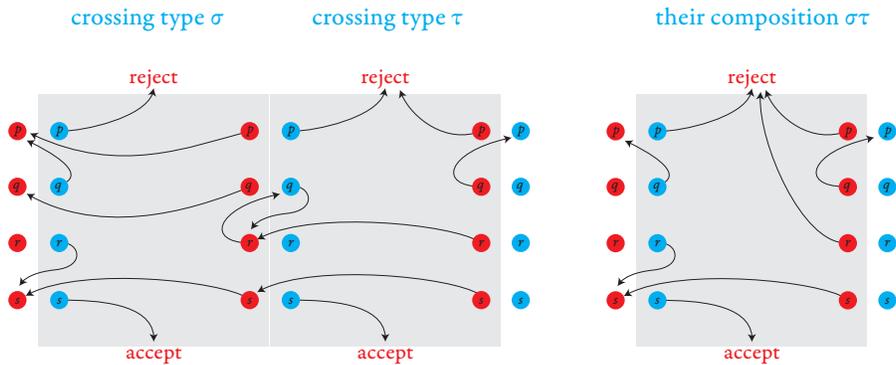
**Crossing types.** Crossing types are the natural information associated to a two-way automaton; since this notion is so classical and natural we only give the notation and intuition, for a more precise description see [CD15]. Define the *crossing type* of  $w \in \text{Inputs}$  to be the function

$$Q \times \{\text{leftmost}, \text{rightmost}\} \rightarrow \{\text{accept}, \text{reject}\} + Q \times \{\text{leftmost}, \text{rightmost}\}$$

which describes the behaviour of the automaton on a partial input, as described in the following picture



Crossing types can be equipped with a semigroup structure such that the function which maps an input to its crossing type becomes a semigroup homomorphism, here is a picture:



Furthermore, by [CD15, Theorem 9], if  $\mathcal{A}$  is a first-order definable two-way automaton, then the semigroup of crossing types is an aperiodic semigroup.

**Proof plan.** We now present the proof strategy for Lemma 6.3, which says that the entire set  $\text{Inputs}$  is good. The plan is to show that good languages can be combined using a form of concatenation and Kleene star with separator symbols to get new good languages. More formally, we show that if  $L, K \subseteq \text{Inputs}$  are good and  $|$  is a fresh separator symbol, then:

1. **Lemma 6.7.** For every state  $q$  there is a polyregular function  $f$  with

$$f(w_1|w_2) = \text{run}(qw_1w_2) \quad \text{for every } w_1 \in L, w_2 \in K.$$

2. **Lemma 6.8.** Assume that all words in  $L$  have the same crossing type. Then for every state  $q$  there is a polyregular function  $f$  with

$$f(w_1|\dots|w_n) = \text{run}(qw_1\dots w_n) \quad \text{for every } n \in \{1, 2, \dots\} \text{ and } w_1, \dots, w_n \in L.$$

Furthermore, if the underlying automaton is first-order definable, then the polyregular functions in the conclusions of the above lemmas are first-order polyregular.

Before proving the above two lemmas, we show how they imply that all partial inputs are good, as required by Lemma 6.3. The idea is to use the Factorisation Forest Theorem, which says that every word can be factorised into several words, and those words can also be factorised, and so on recursively, so that the depth of the recursion is bounded and the factorisations are compatible with Lemmas 6.7 and 6.8. The appropriate definition is given below (think of  $h$  being the homomorphism which maps a word to its crossing sequence).

**Definition 6.4** *Let  $h : \Sigma^+ \rightarrow S$  be a semigroup homomorphism, with  $S$  finite. Define the  $h$ -height of a word  $w \in \Sigma^+$  to be the smallest natural number that can be assigned using the following rules.*

1. every one letter word has  $h$ -height 1;
2. if  $w, v$  have  $h$ -height  $< n$ , then  $wv$  has  $h$ -height  $\leq n$ .
3. if  $w_1, \dots, w_n$  have  $h$ -height  $< n$  and the same value<sup>9</sup> under  $h$ , then  $w_1\dots w_n$  has height  $\leq n$ .

Clearly every word has some  $h$ -height, e.g. at most its length (or even the logarithm of its length). The Factorisation Forest Theorem says that the upper bound on  $h$ -height is actually  $3|S|$ , i.e. there is a finite upper bound that works for all words and depends only on the semigroup  $S$ . The theorem was originally proved by Imre Simon [Sim90, Theorem 9.1] with a bound of  $9|S|$ , while the optimal bound of  $3|S|$  is from [Kuf08, Theorem 1]. What is more, a suitable decomposition can be computed using a rational function, in the following sense.

**Lemma 6.5 ([Col07, BDK18])** *Let  $h : \Sigma^+ \rightarrow S$  be a semigroup homomorphism, and let  $|$  be a fresh separator symbol. For every  $k \in \{2, 3, \dots\}$  there is a rational function which inputs a word  $w \in \Sigma^+$  and outputs a decomposition*

$$w_1|\dots|w_n \quad \text{with } w = w_1\dots w_n$$

*such that*

---

<sup>9</sup>Often one assumes that this same value is an idempotent. We do not make this assumption, which plays a role when computing factorisations in first-order logic, see Footnote 10.

1.  $w_1, \dots, w_n$  have  $h$ -height strictly smaller than  $w$ ; and
2. either  $n = 2$  or all  $w_1, \dots, w_n$  have the same value under  $h$ .

Furthermore, if  $S$  is aperiodic, then a first-order rational function is enough<sup>10</sup>.

**Proof** (of Lemma 6.3)

Let  $h$  be the homomorphism which maps an input word to its crossing type with respect to our fixed two-way automaton. Since the semigroup of crossing types is finite, the Factorisation Forest Theorem implies that every partial input has  $h$ -height bounded by a constant that depends only on the fixed two-way automaton, and not on the length of the word. By induction on  $k$ , we show that the set of all words in `Inputs` with  $h$ -height at most  $k$  is good. In the induction step, we use the rational function from Lemma 6.5, and then apply either Lemma 6.7 or 6.8 to the result, depending on the number of factors produced.  $\square$

It remains to prove Lemmas 6.7 and 6.8; the rest of Section 6.1 is devoted to proving these lemmas. Since the proofs use only first-order polyregular functions, we neglect to always add that “if the underlying automaton is first-order definable, then ...”. There is one exception, Sublemma 6.8.3, where special care is needed in the first-order case.

**If-then-else.** We begin by showing that polyregular functions can be combined using an “if then else” construction.

**Lemma 6.6** *If  $L \subseteq \Sigma^*$  is a regular language, and  $f, g : \Sigma^* \rightarrow \Gamma^*$  are polyregular functions, then the following function is also polyregular*

$$w \in \Sigma^* \quad \mapsto \quad \begin{cases} f(w) & \text{for } w \in L \\ g(w) & \text{for } w \notin L \end{cases}$$

**Proof**

The basic idea is to decompose the natural parallel implementation of “if then else” into a sequential one. The main step is given in the following sublemma.

**Sublemma 6.6.1** *For every polyregular function  $h : \Sigma^* \rightarrow \Gamma^*$  and every alphabet  $\Delta$  disjoint from  $\Sigma$  and  $\Gamma$ , there is a polyregular function*

$$h^\Delta : (\Sigma + \Delta)^* \rightarrow (\Gamma + \Delta)^*$$

*which agrees with  $h$  on inputs from  $\Sigma^*$  and is the identity on inputs from  $\Delta^*$ . We have no requirements for inputs that use both  $\Sigma$  and  $\Delta$ .*

<sup>10</sup>To get a first-order rational function, it is important that we do not require values to be idempotent in item 3 of Definition 6.4. To see how this is important, consider the semigroup  $\{1, 2\}$  where all products have value 2, i.e. this is the syntactic semigroup of the language “words of length exactly 1”. This semigroup is clearly aperiodic, and yet a rational function cannot produce decompositions with idempotent values, since this would require grouping letters into groups (say, of size two).

**Proof**

If the conclusion of the sublemma is true for two polyregular functions, then it is also true for their composition, by setting

$$(h_1 \circ h_2)^\Delta = h_1^\Delta \circ h_2^\Delta$$

Therefore, it remains to show the sublemma for the atomic functions, namely the sequential functions, squaring, and iterated reverse. The case of rational functions is straightforward, by taking a union of automata. The case of squaring is also easy: first apply squaring, and if the result contains at least one letter from  $\Delta$ , then use a rational function to recover the original input. The most interesting case is when  $h$  is iterated reverse operation, which is implemented as follows.

1. Apply the following rational function:

- (a) if the input is in  $\Sigma^*$ , e.g. it is

123|45|67|8

then leave the input unchanged.

- (b) if the input is in  $\Delta^*$ , e.g. it is

*abcdef*

then add the separator | between every two positions, like this:

*a|b|c|d|e|f*

- (c) otherwise, output the empty word

2. apply iterated reverse, with the separator being |
3. if the output contains letters from  $\Delta$  remove all separators |.

□

Having proved Sublemma 6.6.1, we return to the proof of Lemma 6.6 about an if-then-else construction for polyregular functions. Let us write  $\Sigma$  for a disjoint copy of the alphabet  $\Sigma$ , and for a word  $w \in \Sigma^*$  let us write  $w \in \Sigma^*$  for the corresponding word over the copied alphabet. The function in the statement of the lemma is implemented by the following sequence of operations (we assume without loss of generality that  $\Sigma$  and  $\Gamma$  are disjoint):

1. if the input is not in  $L$ , replace  $w$  by  $w$ ;
2. apply  $f^\Sigma$  as defined in Sublemma 6.6.1;
3. swap  $\Sigma$  with  $\Sigma$ ;
4. apply  $g^\Gamma$  as defined in Sublemma 6.6.1;

The functions items 1 and 3 are clearly rational, while the functions in items 2 and 4 are polyregular thanks to Sublemma 6.6.1. □

**Concatenation.** We now show the first of the two lemmas needed in Lemma 6.3, namely that two good languages can be combined via concatenation.

**Lemma 6.7** *If  $L, K \subseteq \text{Inputs}$  are good, then for every state  $q$  there is a polyregular function  $f$  with*

$$f(w_1|w_2) = \text{run}(qw_1w_2) \quad \text{for every } w_1 \in L, w_2 \in K.$$

**Proof**

Using the “if then else” construction from Lemma 6.6, we can assume without loss of generality that all words in  $L$  have the same crossing type, call it  $\sigma$ , and all words in  $K$  have the same crossing type, call it  $\tau$ . The following sublemma, which follows almost immediately from the definitions, shows how the run of a two-way automaton on input  $vw$  can be reconstructed in a compositional way from its runs on the factors  $v$  and  $w$ . In the lemma, we use the following notation: for  $w \in \text{Inputs}$  and  $\rho \in \text{Runs}$ , we write  $w \odot \rho \in \text{Runs}$  for the partial run obtained prepending  $w$  to the left of every partial configuration appearing in  $\rho$ . Similarly we define  $\rho \odot w$ . Here is a picture:

$$\begin{array}{ccc}
 \rho & & ab \odot \rho \\
 \\
 \begin{array}{cccc}
 q & p & q & r \\
 acb | acb | acb | acb |
 \end{array} & & \begin{array}{cccc}
 q & p & q & r \\
 abacb | abacb | abacb | abacb |
 \end{array}
 \end{array}$$

We will show in Sublemma 6.7.3 below that  $\odot$  can be implemented by a polyregular function. We begin though by showing how a run over  $w_1w_2$  decomposes into a bounded number of runs over  $w_1$  or  $w_2$ ; this result is a simple application of crossing types. (The sublemma talks about runs that begin in the leftmost position; a symmetric construction deals with runs that begin in the rightmost position.)

**Sublemma 6.7.1** *For every crossing types  $\sigma, \gamma$  and every  $q \in Q$  there exist  $k \in \{0, 1, \dots\}$  and states*

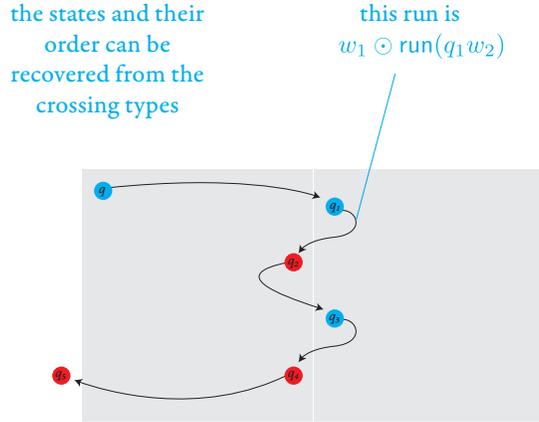
$$q_1, \dots, q_k \in Q$$

*such that partial inputs  $w_1, w_2$  with crossing types  $\sigma, \tau$  respectively satisfy*

$$\text{run}(qw_1w_2) = \rho_0 \cdots \rho_k \quad \text{where } \rho_i = \begin{cases} \text{run}(qw_1) \odot w_2 & \text{when } i = 0 \\ w_1 \odot \text{run}(q_iw_2) & \text{when } i \in \{1, 3, 5, \dots\} \\ \text{run}(w_1q_i) \odot w_2 & \text{when } i \in \{2, 4, 6, \dots\} \end{cases}$$

**Proof**

The proof is best seen in a picture:



□

To finish the proof of the lemma, we need to show that the construction described in Sublemma 6.7.1 can be implemented by a polyregular function. The first challenge is implementing the concatenation  $\rho_1 \cdots \rho_k$ . Note that the number of concatenated blocks  $k$  is constant in the sense that it depends only on the fixed crossing types  $\sigma, \tau$  and not on the input words  $w_1, w_2$ . The following sublemma shows that polyregular functions are closed under this type of concatenation.

**Sublemma 6.7.2** *If  $f, g : \Sigma^* \rightarrow \Gamma^*$  are polyregular, then so is  $w \mapsto f(w)g(w)$ .*

**Proof**

The sublemma follows from the two observations below.

1. As in Sublemma 6.6.1, let us write  $\Sigma$  for a disjoint copy of the alphabet  $\Sigma$ , and for  $w \in \Sigma^*$  let us write  $w$  for the corresponding word in  $\Sigma^*$ . First observe that the function

$$w \mapsto ww$$

is polyregular. This is done as follows. Suppose that the input is

1234

Add a fresh separator at the end of  $w$ , yielding

1234|

and then apply squaring, yielding

1234|1234|1234|1234|1234|

Remove the underlines, colour the letters before the first separator black, the letters between the first and second separator red, and remove the remaining letters, giving the desired output

12341234.

2. The second observation is that if  $h : \Sigma^* \rightarrow \Gamma^*$  is polyregular, then the same is true for the function  $\bar{h}$  defined by

$$w \in (\Sigma + \Sigma)^* \mapsto \begin{cases} h(w_1)w_2 & \text{if } w = w_1w_2 \text{ for } w_1 \in \Sigma^+ \text{ and } w_2 \in \Sigma^+ \\ \varepsilon & \text{otherwise} \end{cases}$$

As was the case in the proof of Sublemma 6.6, it is enough to show that  $\bar{h}$  is polyregular whenever  $h$  is one of the atomic functions, and the straightforward proof of that is left to the reader.

The sublemma follows immediately: we first duplicate the input word as in item 1 above, then apply item 2 with  $h = f$ , and finally apply a symmetric version of item 2 with  $h = g$ .  $\square$

We now finish the proof of the lemma. Suppose that the input is  $w_1|w_2$ . Thanks to Sublemmas 6.7.1 and 6.7.2, to finish the proof of the lemma, it is enough to show that for every state  $q$ , the functions

$$w_1|w_2 \mapsto \text{run}(qw_1) \odot w_2 \quad w_1|w_2 \mapsto w_1 \odot \text{run}(qw_2)$$

are polyregular (and likewise for runs where  $q$  is at the end of the input word). By symmetry, we only do the first function. By assumption that  $L$  is good and closure of polyregular functions under concatenation (Sublemma 6.7.2), the function

$$w_1|w_2 \mapsto \text{run}(qw_1)|w_2$$

is polyregular. To finish the proof, we use the following construction, which allows us to implement  $\rho|w \mapsto \rho \odot w$  using a polyregular function.

**Sublemma 6.7.3** *For every alphabet  $\Sigma$  and separator  $|$  not in  $\Sigma$ , the function*

$$v_1|\dots|v_n|v \mapsto v_1v|\dots|v_nv$$

*is polyregular (in the above, we assume that  $v_1, \dots, v_n, v$  do not use  $|$ ).*

**Proof**

Suppose that the input looks like this:

$$12|345|6|78$$

The number of blocks, as separated by  $|$ , is unbounded. Using a sequential function, append a fresh endmarker, say a comma

$$12|345|6|78,$$

and then apply squaring. In the result, keep only the maximal blocks in  $\Sigma$  where either the first position is underlined, or the block is directly followed by a comma and the closest underlined position to the left is the first in its block, giving a result like this:

$$\underline{1}2|78, \underline{3}45|78, \underline{6}78,$$

Finally, swap  $|$  and the comma, and then remove the commas.  $\square \square$

**Homogeneous Kleene star.** To finish the proof of Lemma 6.3, we show that good languages are closed under a variant of the Kleene star, where the starred language contains only words of the same crossing type and the different copies of the starred language are separated by  $|$ .

**Lemma 6.8** *Suppose that  $L$  is good, and all words in  $L$  have the same crossing type. Let  $|$  be a fresh separator symbol. Then for every state  $q$  of the fixed two-way automaton there is a polyregular function  $f$  with*

$$f(w_1|\dots|w_n) = \text{run}(qw_1\dots w_n) \quad \text{for every } n \in \{1, 2, \dots\} \text{ and } w_1, \dots, w_n \in L.$$

**Proof**

Suppose that the input is

$$w_1|\dots|w_n \quad \text{with } w_1, \dots, w_n \in L$$

Let  $q$  be a state as in the assumption of the lemma, and let us write

$$\rho \stackrel{\text{def}}{=} \text{run}(qw_1\dots w_n)$$

For  $i \in \{1, \dots, n\}$ , define

$\rho_i$  is sequence of configurations from  $\rho$  that begins in the configuration with the first visit in  $w_i$  and ends just before the first visit in  $w_{i+1}$ ; and

$q_i$  is the state used in the first visit in  $w_i$ .

If  $\rho$  never visits  $w_i$ , then  $\rho_i$  is empty and  $q_i$  is undefined. If  $\rho$  never visits  $w_{i+1}$ , or  $i = n$ , then  $\rho_i$  is a suffix of  $\rho$ . Note that the first visit in  $w_i$  necessarily is in the leftmost position, since  $\rho$  begins in the leftmost position of  $w_1|\dots|w_n$ . By the assumption that all of the words  $w_1, \dots, w_n$  have the same crossing type, it follows that the sequence  $q_1, \dots, q_n$  behaves in a certain cyclic way, this will be explained in the proof of Sublemma 6.8.3 below, but not used. By definition we have

$$\rho = \rho_1 \dots \rho_n$$

The goal is to show that the function  $w_1|\dots|w_n \mapsto \rho$  is polyregular, where the number  $n$  of words is not fixed, but their crossing type is.

The following lemma is a version of Lemma 6.7 which allows us to compute runs on concatenations of two inputs from good languages, but with the head positioned between the inputs and not over the first position. If  $w, v$  are partial inputs and  $p$  is a state, then we write  $w(pv)$  to denote the partial configuration where the head is in state  $p$  over the first position of the input word  $v$ ; we need the parenthesis since  $(wp)v$  means something else, namely that the head is in the last position of  $w$ .

**Sublemma 6.8.1** *If  $L, K$  are good then there is a polyregular function which agrees with run on partial configurations of the form*

$$w(pv) \quad \text{where } w \in K, v \in L, p \in Q.$$

**Proof**

The same reasoning, using crossing sequences, as in the proof of Lemma 6.7.  $\square$

The above sublemma also gives the implication from Lemma 6.3 to Lemma 6.1. By unfolding the definition of  $\rho_i$ , we see that

$$\rho_i = \text{run}(w_1 \cdots w_{i-1}(qw_i)) \odot w_{i+1} \cdots w_n$$

Using Sublemma 6.8.1, we can compute  $\rho_i$  for any fixed value of  $i$ , since the language  $L^{i-1}$  is good by Lemma 6.7 applied several times. However, this approach yields a polyregular function which depends on  $i$ , while we need a uniform construction that does not depend on  $i$ , since the number runs  $\rho_1, \dots, \rho_n$  is unbounded. To get the uniform construction, the following observation is crucial.

**Sublemma 6.8.2** *For every  $w_1, \dots, w_n \in L$  and every  $i \in \{1, \dots, n\}$ , the run  $\rho_i$  visits only words  $w_j$  with  $i - j \leq |Q|$ .*

**Proof**

A pumping argument, which uses the assumption that all words in  $L$  have the same crossing type. Suppose that  $\rho_i$  visits  $w_{i-k}$  for some  $k \in \{1, 2, \dots\}$  such that  $i - k > 1$ . By the assumption that all of the words  $w_1, \dots, w_n$  have the same crossing type, also  $\rho_{i-1}$  visits  $w_{i-k-1}$ . A corollary is that if  $\rho_i$  visits  $w_1$ , then the same is true for  $\rho_{i-1}$ . Since every input position can be visited at most once in each state, it follows that  $\rho_i$  cannot visit the first position when  $i$  exceeds the number of states, and the result follows<sup>11</sup>.  $\square$

For words  $w_1, \dots, w_n$  and  $i \in \{1, \dots, n\}$ , consider the decomposition of the word  $w_1 | \cdots | w_n$  into four parts shown below

$$\underbrace{w_1 | \cdots | w_{j-1}}_{x_i} | \underbrace{w_j | \cdots | w_{i-1}}_{y_i} | \underbrace{w_i}_{w_i} | \underbrace{w_{i+1} | \cdots | w_n}_{z_i} \quad j = \max(1, i - |Q|).$$

In other words,  $y_i w_i$  describes a window of the  $\leq |Q|$  blocks that contain the head positions of the run  $\rho_i$ . By Sublemma 6.8.2,

$$\rho_i = x_i \odot \text{run}(y_i q_i w_i) \odot z_i \tag{3}$$

The partial input  $y_i$  comes from a good language (after removing the separators), thanks to Lemma 6.7 iterated at most  $|Q|$  times. (We need to use closure of good languages under unions, because the number of iterations could be  $\leq |Q|$ , but closure under unions is an easy corollary of the if-then-else construction in Sublemma 6.6.) Therefore, we can use Sublemma 6.8.1 to compute the run  $\rho_i$ , assuming that the decomposition into  $x_i, y_i, w_i, z_i$  is given and the state  $q_i$  is known. This decomposition can indeed be computed, thanks to the following result (as usual, we use red to denote a disjoint copy of the alphabet).

<sup>11</sup>If we assume that the fixed crossing type of all words in  $L$  is idempotent, then we could get a stronger conclusion, namely that  $\rho_i$  visits only  $w_i$  and  $w_{i-1}$ . However, as explained in Footnote 10, the assumption on idempotency cannot be made in the first-order definable case.

**Sublemma 6.8.3** *The following function is polyregular*

$$w_1| \cdots |w_n \mapsto x_1 y_1 q_1 w_1 z_1 | \cdots | x_n y_n q_n w_n z_n$$

where  $w_1, \dots, w_n \in L$  and  $|$  is a fresh separator symbol.

**Proof**

Similar to Sublemma 6.7.3: first take a square, and then apply rational post-processing. The states  $q_1, \dots, q_n$  can be computed using a rational function (which is a first-order rational function in the case when the two-way automaton is first-order definable). Actually, the assumption that all words in  $L$  have the same crossing type can be used to obtain a stronger result, namely that the sequence  $q_1, q_2, \dots$  is a lasso, in the following sense. There exist  $k, k_0 \in \mathbb{N}$ , which depend only on the crossing type of  $L$ , such that

$$q_i = q_{i+k} \quad \text{for all } i \geq k_0.$$

Furthermore, if the two-way automaton is first-order definable, then  $k = 1$ , since otherwise there would be a counter.  $\square$

To complete the proof of the lemma, the final piece is the following result, which shows that polyregular functions can be iterated over blocks in an input word with separators.

**Sublemma 6.8.4** *If  $f : \Sigma^* \rightarrow \Gamma^*$  is polyregular then the same is true for*

$$w_1| \cdots |w_n \mapsto f(w_1)| \cdots |f(w_n)$$

where  $|$  is a fresh separator symbol not appearing in  $w_1, \dots, w_n$ .

**Proof**

It is enough to prove the lemma for the atomic polyregular functions. For sequential functions, the construction is natural, likewise for iterated reverse. For squaring, we illustrate the construction on an example. Suppose that the input is like this:

$$12|3|45$$

We first use a sequential function to add a marker at the end (a comma), and then apply squaring, yielding a result like this:

$$\underline{1}2|3|45, \underline{1}2|3|45, 12|3|45, 12|\underline{3}|45, 12|3|45, 12|3|45, 12|3|45, 12|3|45,$$

Use a rational function to colour red every block (defined as maximal infix which has neither  $|$  nor  $,$ ) which contains an underlined position:

$$\underline{1}2|3|45, \underline{1}2|3|45, 12|3|45, 12|\underline{3}|45, 12|3|45, 12|3|45, 12|3|45, 12|3|45,$$

Keep only the red letters, and for every two consecutive red blocks where the underlined position goes from first to last, separate the blocks using  $|$ .  $\square$

Let us complete the proof of the lemma. Suppose that the input is

$$w_1 | \cdots | w_n \quad \text{with } w_1, \dots, w_n \in L.$$

Using Sublemma 6.8.3, compute the word

$$x_1 y_1 q_1 w_1 z_1 | \cdots | x_n y_n q_n w_n z_n$$

By Sublemma 6.8.1, the function

$$x_i y_i q_i w_i z_i \mapsto \rho_i$$

is polyregular. Applying Sublemma 6.8.4 to this function, compute

$$\rho_1 | \cdots | \rho_n.$$

Finally, we remove the separators.  $\square$

## 6.2 Many pebbles

In Section 6.1, we showed that a polyregular function can compute the run of a 1-pebble automaton. In this section, we lift the result to  $k$ -pebble automata, by reducing to the 1-pebble case. The reduction is given in following lemma.

**Lemma 6.9** *Let  $k > 1$ . For every*

$$f : \Sigma^* \rightarrow \Gamma^*$$

*recognised by a  $k$ -pebble transducer there exist:*

1. *a rational<sup>12</sup> function  $g : \Sigma^* \rightarrow \Delta^*$ ;*
2. *a 1-pebble automaton  $\mathcal{A}$  with input alphabet  $\Delta$ ;*
3. *a  $(k-1)$ -pebble transducer  $h$  that inputs runs of  $\mathcal{A}$ ;*

*such that the following diagram commutes:*

$$\begin{array}{ccc}
 \Sigma^* & \xrightarrow{f} & \Gamma^* \\
 g \downarrow & & \uparrow h \\
 \Delta^* & \xrightarrow{\text{run}} & \text{Runs}
 \end{array}$$

*where **run** is the function from Lemma 6.1 that maps an input word of  $\mathcal{A}$  to the corresponding run of  $\mathcal{A}$ , as described in Section 6.1. Furthermore, if the automaton for  $f$  is first-order definable, then the same is true for  $g$ ,  $\mathcal{A}$  and  $h$ .*

<sup>12</sup>Actually, the rational function in item 1 could be avoided using Lemma [UH67, Lemma 3], which says that a two-way automaton can simulate regular lookahead.

Before proving the lemma, let us use it to show that every pebble transducer recognises a polyregular function. The proof is by induction on the number of pebbles. For the induction base, we use Lemma 6.1, which says that a polyregular function can transform an input word into the run of a given 1-pebble automaton; once the run is given, the output of the run can be recovered using a rational function. For the induction step, we apply Lemma 6.9, and observe that all three red arrows in the diagram from the lemma are polyregular:  $g$  is polyregular because it is rational,  $\text{run}$  is polyregular by Lemma 6.1, and  $h$  is polyregular by induction assumption. It remains to prove the lemma.

**Proof** (of Lemma 6.9)

The basic idea is that a run of a  $k$ -pebble automaton can be decomposed into configurations that have only pebble 1, and intermediate subcomputations that do not move pebble 1. The intermediate subcomputations can be simulated using  $k - 1$  pebbles. The rational function  $g$  is used to determine how pebble 1 is moved, without entering into the subcomputations that use more pebbles. A more formal proof is given below.

Define a *main configuration* in a  $k$ -pebble automaton to be a configuration where only pebble 1 is present.

**Sublemma 6.9.1** *Let  $\mathcal{B}$  be a  $k$ -pebble automaton with input alphabet  $\Sigma$ . There is a letter-to-letter rational function*

$$g : \Sigma^* \rightarrow \Delta^*$$

*such that the following is true for every  $w \in \Sigma^*$ . Let  $(q_i, x_i)$  be the state and head position in the  $i$ -th main configuration of the automaton in the run of  $\mathcal{B}$  on input  $w$ . Then*

$$\underbrace{x_{i+1} - x_i}_{\text{and offset in } \{-1, 0, 1\}} \quad \text{and} \quad q_{i+1}$$

*depend only on  $q_i$  and the label of  $g(w)$  in position  $x_i$ . Furthermore, if  $\mathcal{B}$  is first-order definable, then  $g$  is first-order rational.*

**Proof**

Thanks to Lemma 2.3, for every state  $q, p$  and offset  $\delta \in \{-1, 0, 1\}$  there is an MSO sentence  $\varphi_{q, \delta, p}(x)$  with one free first-order variable such that

$$w \models \varphi_{q, \delta, p}(x)$$

if and only if the automaton  $\mathcal{B}$ , when in main configuration  $(q, x)$ , does a subcomputation such that the next main configuration is  $(p, x + \delta)$ . The function  $g$  simply labels each position  $x$  with the set

$$\{(p, \delta, q) : w \models \varphi_{p, \delta, q}(x)\}$$

This function is rational, thanks to compositionality of MSO. In the case when  $\mathcal{B}$  is first-order definable, we do not need to use Lemma 2.3, but we just appeal to the definition of a first-order definable pebble automaton.  $\square$

Using the function  $g$  from the above sublemma, it is not hard to design a 1-pebble automaton  $\mathcal{A}$ , which has the same states as  $\mathcal{B}$ , with the following property. For every input word  $w$ , the  $i$ -th configuration of  $\mathcal{A}$  (which is also the  $i$ -th main configuration, since all configurations are main configurations in a 1-pebble automaton) in the word  $g(w)$  has the same state and head position as the  $i$ -th main configuration of  $\mathcal{B}$  on input  $w$ . Finally, to recover the output of  $f$ , we apply to each configuration of  $\mathcal{A}$  the transducer from the following sublemma.

**Sublemma 6.9.2** *There is a  $(k-1)$ -pebble transducer which inputs a configuration  $c$  of  $\mathcal{B}$ , and returns the output of  $\mathcal{B}$  in the subcomputation that starts in  $c$  and ends in the next main configuration (not including the output produced in the next main configuration).*

**Proof**

By stack discipline, the subcomputation does not move pebble 1.  $\square \square$

## 7 Polyregular functions to list functions

In this section we prove that every polyregular function (i.e. any composition of sequential functions, squaring and iterated reverse, as described in Section 1) is a polynomial list function (i.e. can be defined in the functional programming language from Section 4), and the construction preserves first-order definability. Polynomial list functions (and their first-order fragment) are closed under function composition: if  $M$  and  $N$  are polynomial list function, then their composition is the program

$$\lambda x.M(Nx).$$

Therefore, it remains to show that the basic building blocks of polyregular functions are polynomial list functions, which we do in the following sections:

- Iterated reverse in Section 7.1;
- Squaring in Section 7.2;
- Sequential functions in Section 7.3.

The main challenge is the sequential functions, since this requires showing that polynomial list functions can simulate the state updates in a finite state automaton. For this, we use a result from [BDK18], which in turn requires implementing a function called `block`, which separates a list which has two types of elements into a list of lists which have only one type of elements, as shown in the following example:

$$[1, 2, a, b, c, 3, d, e, 4, 5, 6] \quad \mapsto \quad [[1, 2], [a, b, c], [3], [d, e], [4, 5, 6]]$$

The code for the polynomial list functions described in this section, written in Haskell notation, is given in Appendix A, and therefore this section only illustrates the programs on examples. The interested reader is invited to read (or run) the code from the appendix.

## 7.1 Iterated reverse

We begin by showing that iterated reverse is a first-order polynomial list function. We first implement (not iterated) reverse (Lemma 7.1), and then we implement `block` (Lemma 7.2).

**Lemma 7.1** *Reverse is a first-order polynomial list function.*

### Proof

The corresponding Haskell code is in Appendix A.2, so we just show the construction on an example. Suppose that the input list is

$$[1, 2, 3, 4]$$

Apply `split`, yielding a list like this:

$$[[1, 2, 3, 4], []], ([1, 2, 3], [4]), ([1, 2], [3, 4]), ([1], [2, 3, 4]), ([], [1, 2, 3, 4])]$$

Project every pair in the above list to its second coordinate, yielding

$$[[], [4], [3, 4], [2, 3, 4], [1, 2, 3, 4]]$$

To each element of the above list, apply the function

$$[a_1, \dots, a_n] \mapsto \begin{cases} [] & \text{if } n = 0 \\ [a_1] & \text{otherwise} \end{cases}$$

yielding a list like this:

$$[[], [4], [3], [2], [1]].$$

Finally, apply `concat`, yielding the desired list

$$[4, 3, 2, 1].$$

□

We now show that polynomial list functions can implement the `block` function mentioned at the beginning of Section 7. More formally, for types  $\tau$  and  $\sigma$ , define `block $\tau\sigma$`  to be the function which maps a list  $l \in (\tau + \sigma)^*$  to the unique list in  $(\tau^* + \sigma^*)^*$  which: (a) yields  $l$  after applying `concat`; and (b) alternates between lists in  $\tau^+$  and  $\sigma^+$ , in particular contains only nonempty lists. The function `block` is one of the basic building blocks in the programming language from [BDK18], but it turns out to be implementable in the programming language from this paper.

**Lemma 7.2** *For every types  $\tau$  and  $\sigma$ , the function  $\text{block}_{\tau\sigma}$  is a first-order polynomial list function.*

**Proof**

The corresponding Haskell code is in Appendix A.2, so we just show the construction on an example. Suppose that the input list is

$$[1, 2, 3, a, b, 4, 5, 6, 7, c, 8, d, e, f]$$

Apply `split`, and in the resulting list keep only those pairs  $(x, y)$  such that  $y$  is nonempty, and either  $x$  is empty, or the last element of  $x$  has a different type ( $\tau$  vs  $\sigma$ ) than the first element of  $y$ , yielding a result like this:

$$\begin{aligned} &([1, 2, 3, a, b, 4, 5, 6, 7, c, 8], [d, e, f]), \\ &([1, 2, 3, a, b, 4, 5, 6, 7, c], [8, d, e, f]), \\ &([1, 2, 3, a, b, 4, 5, 6, 7], [c, 8, d, e, f]), \\ &([1, 2, 3, a, b, 4], [5, 6, 7, c, 8, d, e, f]), \\ &([1, 2, 3], [a, b, 4, 5, 6, 7, c, 8, d, e, f]), \\ &([], [1, 2, 3, a, b, 4, 5, 6, 7, c, 8, d, e, f]) \end{aligned}$$

Reverse the list, see Lemma 7.1, and keep only the second coordinates:

$$\begin{aligned} &[[1, 2, 3, a, b, 4, 5, 6, 7, c, 8, d, e, f], \\ & \quad [a, b, 4, 5, 6, 7, c, 8, d, e, f], \\ & \quad \quad [5, 6, 7, c, 8, d, e, f], \\ & \quad \quad \quad [c, 8, d, e, f], \\ & \quad \quad \quad \quad [8, d, e, f], \\ & \quad \quad \quad \quad \quad [d, e, f]] \end{aligned}$$

Finally, for each element in the result, keep only the prefix that agrees on type ( $\tau$  vs  $\sigma$ ) with the first element of the list, yielding the desired result

$$\begin{aligned} &[[1, 2, 3], \\ & \quad [a, b], \\ & \quad \quad [5, 6, 7], \\ & \quad \quad \quad [c], \\ & \quad \quad \quad \quad [8], \\ & \quad \quad \quad \quad \quad [d, e, f]] \end{aligned}$$

□

Iterated reverse is obtained by applying the block function from the above lemma (with  $\tau$  being the separator and  $\sigma$  being the remaining letters), then using map and reverse from Lemmas 7.1, and finally applying `concat`.

## 7.2 Squaring

We now show that squaring is a first-order polynomial list function.

**Lemma 7.3** *Squaring is a first-order polynomial list function.*

**Proof**

The corresponding Haskell code is in Appendix A.3, so we only illustrate the code using an example. Suppose that the input is a list like this

$$[1, 2, 3, 4]$$

Apply `split`, yielding a list like this

$$[[1, 2, 3, 4], []], ([1, 2, 3], [4]), ([1, 2], [3, 4]), ([1], [2, 3, 4]), ([], [1, 2, 3, 4])]$$

To every pair in the above list, apply the function

$$(a, b) \mapsto \begin{cases} [] & \text{if } b \text{ is empty} \\ [(a, \text{head of } b, \text{tail of } b)] & \text{otherwise} \end{cases}$$

yielding a list like this

$$[[[]], [([], 1, [2, 3, 4])], [([1], 2, [3, 4])], [([1, 2], 3, [4])], [([1, 2, 3], [4], [])]]$$

Flatten the above list using `concat`, yielding

$$[[[]], [1, [2, 3, 4]], [1, 2, [3, 4]], [1, 2, 3, [4]], [1, 2, 3, [4], []]]$$

Next, apply a reformatting function to yield a list

$$[[\underline{1}, 2, 3, 4], [1, \underline{2}, 3, 4], [1, 2, \underline{3}, 4], [1, 2, 3, \underline{4}]]$$

where  $\underline{n}$  stands for copy of letter  $n$  in a disjoint copy of the alphabet.  $\square$

## 7.3 Sequential functions

We are left with showing that every sequential function can be implemented using a polynomial list function, and if the sequential function is first-order rational then a first-order polynomial list function is enough.

There are two ways to solve this problem.

The first way is to use Theorem 1.4, which says that we only need to do the construction for sequential functions where the underlying automaton either: (a) has two states and is counter-free; or (b) has a transformation monoid which is a group. Both kinds of functions can be easily implemented using polynomial list functions, and the construction for (a) needs only a first-order polynomial list function. The construction for (a) is implemented using the `block` function discussed in Lemma 7.2.

The second way is to use regular list functions defined in [BDK18]. We discuss the second way in more detail, because it highlights the relationship between the polynomial list functions from this paper, and the class of regular list functions, which can be seen as the linear growth fragment of polynomial list functions. We begin by defining the class of functions considered in [BDK18].

**Definition 7.4 (Regular and first-order list functions)** *The class of regular list functions is the smallest class of functions which*

- *contains all functions described in Figure 3;*
- *is closed under the combinators described in Figure 4.*

*The first-order list functions are the ones that can be constructed without using the group products in Figure 3.*

By definition, every regular list function (in particular, every first-order one), has a type of the form  $\tau \rightarrow \sigma$  where  $\tau$  and  $\sigma$  are arrow-free, i.e. are constructed from finite sets using only the type constructors  $\tau + \sigma$ ,  $\tau \times \sigma$  and  $\tau^*$ . In this sense, regular and first-order list functions cannot use higher-order types. Another key design property is that the atomic functions from Figure 3 have linear growth and the combinators in 4 preserve this property. This is because regular list functions are designed to capture the regular string-to-string functions, which have linear growth. The linear growth also explains why  $\lambda$ -abstraction or `split` are not allowed in regular list functions, since they could be used to generate functions of super-linear growth, see Example 6.

**Lemma 7.5** *Every regular list function is a polynomial list function. Likewise for the first-order fragment.*

**Proof**

Clearly polynomial list functions and their first-order fragment have the closure properties described in Figure 4. The atomic functions in Figure 3 are either already atomic polynomial list functions (the ones in black) or can be implemented using polynomial list functions (the ones in blue). The implementations of `reverse` and `block` were given in Lemmas 7.1 and 7.2, while the implementations of the remaining functions

$$\text{distr}_{\tau\sigma\pi} \quad \text{const}_a \quad \text{append}_{\tau} \quad \text{group}_G^*$$

are straightforward and left to the reader. Note the difference of the group operations

$$\text{group}_G : G^* \rightarrow G \quad \text{group}_G^* : G^* \rightarrow G^*$$

that are used in Figures 2 and 3 respectively. These can be defined in terms of each other: `groupG` can be defined in terms of `groupG*` using `head` and `reverse`, and a converse construction can be done using `split`. However, `split` is not

$\text{proj}_0^{\tau\sigma}$	:	$(\tau \times \sigma) \rightarrow \tau$
$\text{proj}_1^{\tau\sigma}$	:	$(\tau \times \sigma) \rightarrow \tau$
$\iota_0^{\tau\sigma}$	:	$\tau \rightarrow (\tau + \sigma)$
$\iota_1^{\tau\sigma}$	:	$\tau \rightarrow (\tau + \sigma)$
$\text{concat}^\tau$	:	$(\tau^*)^* \rightarrow \tau^*$
$\text{hd}^\tau$	:	$\tau^* \rightarrow (\tau + \{\perp\})$
$\text{tl}^\tau$	:	$\tau^* \rightarrow (\tau^* + \{\perp\})$
$\text{distr}^{\tau\sigma\pi}$	:	$\tau \times (\sigma + \pi) \rightarrow (\tau \times \sigma) + (\tau \times \pi)$
		distribute + across $\times$
$\text{reverse}^\tau$	:	$\tau^* \rightarrow \tau^*$
		reverse the input list
$\text{const}^a$	:	$\tau \rightarrow \sigma$
		return $a$ for every argument
$\text{append}^\tau$	:	$(\tau \times \tau^*) \rightarrow \tau^*$
		add first argument to the left of the list in the second argument
$\text{block}^{\tau\sigma}$	:	$(\tau + \sigma)^* \rightarrow (\tau^* + \tau^*)^*$
		group into maximal blocks of type $\tau^*$ or $\sigma^*$
		$[1,a,3,4,b,c] \mapsto [[1],[a],[3,4],[b,c]]$
$\text{group}_G^*$	:	$G^* \rightarrow G^*$
		the $i$ -th element of the output list is the product
		of the first $i$ elements in the input list

Figure 3: Atomic linear first-order list functions. The types  $\tau, \sigma$  are required to be arrow-free, i.e. are constructed from finite sets using  $\tau + \sigma$ ,  $\tau \times \sigma$  and  $\tau^*$ . The functions in black are already present in Figure 2, while the functions in blue are not present, but can be derived.

$$\begin{array}{c}
\frac{f : \tau \rightarrow \sigma \quad g : \sigma \rightarrow \pi}{f \circ g : \tau \rightarrow \pi} \quad x \mapsto f(g(x)) \\
\\
\frac{f_1 : \tau_1 \rightarrow \sigma \quad f_2 : \tau_2 \rightarrow \sigma}{\langle f_1, f_2 \rangle : (\tau_1 + \tau_2) \rightarrow \sigma} \quad x \mapsto \begin{cases} f_1(x) & \text{if } x \in \tau_1 \\ f_2(x) & \text{if } x \in \tau_2 \end{cases} \\
\\
\frac{f_1 : \tau \rightarrow \sigma_1 \quad f_2 : \tau \rightarrow \sigma_2}{(f_1, f_2) : \tau \rightarrow (\sigma_1 \times \sigma_2)} \quad x \mapsto (f_1(x), f_2(x)) \\
\\
\frac{f : \tau \rightarrow \sigma}{f^* : \tau^* \rightarrow \sigma^*} \quad [x_1, \dots, x_n] \mapsto [f(x_1), \dots, f(x_n)]
\end{array}$$

Figure 4: Closure properties of linear list functions.

available in Figure 3, and hence the more powerful  $\mathbf{group}_G^*$  is used as an atomic function in Figure 3. One can, in fact, show that replacing  $\mathbf{group}_G^*$  by  $\mathbf{group}_G$  in Figure 4 would lead to a weaker class of functions.  $\square$

The main result about regular list functions and their first-order fragment is that they correspond to the class of regular string-to-string functions. Consider string-to-string functions, i.e. functions of type<sup>13</sup>

$$f : \sigma^* \rightarrow \tau^* \quad \text{where } \sigma, \tau \text{ are finite sets.}$$

When restricted to string-to-string functions, the regular list functions are exactly the same as MSO string-to-string transductions [BDK18, Theorem 6.1]. Since MSO string-to-string transductions are the same as functions recognised by two-way transducers (i.e. 1-pebble transducers) [EH01, Theorem 13], it follows that for string-to-string transducers, the regular list functions are the same as 1-pebble transducers. This equivalence also works for the first-order case: the first-order list functions are exactly the same as first-order transductions [BDK18, Theorem 4.3], and first-order string-to-string transductions are the same as functions recognised by first-order definable 1-pebble automata [CD15]. Putting these results together, we get the following theorem.

**Theorem 7.6** *For string-to-string transducers, i.e. functions of type*

$$f : \sigma^* \rightarrow \tau^* \quad \text{where } \sigma, \tau \text{ are finite sets,}$$

<sup>13</sup>There is a notation clash here. Finite sets in automata theory are denoted using uppercase letters  $\Sigma, \Gamma$ , while  $\lambda$ -calculus typically uses lowercase letters  $\sigma, \tau$  for types, which includes the case of finite sets. We use lowercase when typing programs from the  $\lambda$ -calculus, and uppercase letters in other situations.

*the regular list functions are exactly the same as 1-pebble transducers. The first-order list functions are exactly the same as the first-order definable 1-pebble transducers.*

Putting together Lemma 7.5 and Theorem 7.6, we see that every 1-pebble transducer is a polynomial list function, and this construction preserves first-order definability. Since sequential functions are recognised by 1-pebble transducers, we obtain that every rational function is a polynomial list function (and the construction preserves first-order definability). This argument would have also worked for iterated reverse, but the main work in Section 7.1 consisted of coding non-iterated reverse and `block`, which were necessary to get Lemma 7.5.

This completes the proof that all polyregular functions are polynomial list functions (and the corresponding first-order result).

**Combinators.** The regular list functions (i.e. the ones corresponding to 1-pebble automata) from Definition 7.4 are *combinatory* in the sense that they do not have any variables and  $\lambda$  construction. We can also identify a combinatory syntax for polynomial list functions: define a *combinatory polynomial list function* to be any function generated by the rules from Definition 7.4 plus `split`. It is not hard to see that squaring is a combinatory polynomial list function; and the same is true for iterated reverse and sequential functions (here `split` is not needed) as we have seen above. It follows that

$$\begin{array}{l} \text{polyregular functions} \quad \subseteq \\ \text{combinatory polynomial list functions} \quad \subseteq \\ \text{polynomial list functions} \end{array}$$

Furthermore, since the first and third lines above are equal, it follows that the polynomial list functions collapse to their combinatorial fragment, i.e.  $\lambda$  abstraction can be eliminated without affecting the power of the programming language, at least as long as string-to-string functions are concerned.

## 8 List functions to for-transducers

In this section, we show that every string-to-string function computed by a polynomial list function can also be computed by a for-transducer, and the translation preserves first-order definability.

In the first part of the proof, Section 8.1, we show that string-to-string functions computed by for-transducers are closed under composition. The idea is similar to the composition closure for pebble transducers: if  $f, g$  are for-transducers, then a for-transducer computing the composition  $f \circ g$  is the same as the for-transducer  $f$ , except that instead of positions it uses configurations of  $g$ .

In the second and main part of the proof, Section 8.2, we use a semantics of polynomial list functions based on term rewriting. The key idea is to do  $\beta$ -reduction in parallel, e.g. if there is a list of terms, then one step of  $\beta$ -reduction

can be applied in parallel to each term on the list. By doing  $\beta$ -reduction in parallel, only a bounded number of rewriting steps is needed to compute the value of a term. Since a single parallel  $\beta$ -reduction step can be computed by a for-transducer, and for-transducers are closed under composition, it follows that a for-transducer can compute the value of a polynomial list program.

## 8.1 Composition of for-transducers

The goal of this section is to prove the following lemma.

**Lemma 8.1** *String-to-string functions recognized by for-transducers are closed under composition. The construction preserves first-order definability.*

### Proof

Consider two for-transducers

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

Our goal is to show that the composition

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

is also a for-transducer, and if both  $f, g$  were first-order, then the same is true for  $g \circ f$ . Recall the prenex normal form that was defined in Section 5. Using Lemma 5.2, we can assume that both  $f$  and  $g$  are in prenex normal form. An example of  $f$  and  $g$  is given in Figure 5, the for-transducer for their composition is illustrated in Figure 6.

Suppose that the input to the composition is some word  $w \in \Sigma^*$ . Our goal is to compute  $g(f(w))$  using a single for-transducer that simulates the run of  $g$  over an input of the form  $f(w)$ . The idea is the same as in the proof of Theorem 2.6: use the same code as  $g$ , except that positions in the intermediate word are represented by configurations of  $f$  that were used to produce them.

Consider the first for-transducer  $f$ , and let  $n$  be the number for loops that it uses. For an input word  $w \in \Sigma^*$  and an  $n$ -tuple of positions  $x_1, \dots, x_n$ , define

$$f(w, x_1, \dots, x_n) \in \Gamma^*$$

to be the letters that are output by the kernel of  $f$  (recall that the kernel of a for-transducer in prenex normal form is the part that does not use for loops) in the iteration of the for loops where the position variables are set to  $x_1, \dots, x_n$ . Since the kernel has no loops, the above word has fixed length. To simplify the proof of Lemma 8.1, we assume that this fixed length is at most one, i.e. in each iteration of the innermost loop of  $f$  at most one letter is produced. This assumption means that each position in the intermediate word  $f(w)$  is represented uniquely by a tuple of position variables of  $f$  in the input word  $w$ . The proof without this additional assumption requires additional notation, but follows the same lines, and is left to the reader.

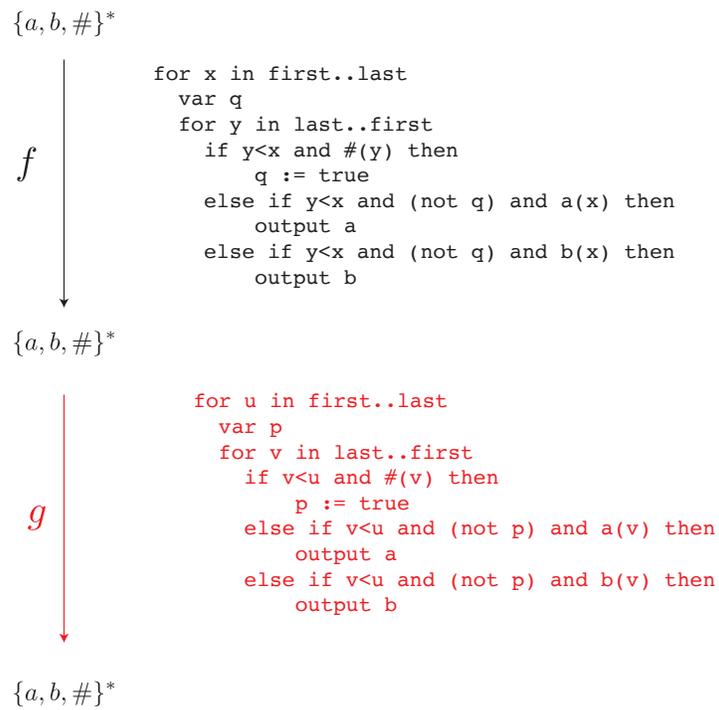


Figure 5: Example instance of composition of two for-transducers. In this example, both for-transducers define the same function, namely iterated reverse, only using renamed variables.

As in the proof of Lemma 5.3, we order  $n$ -tuples of positions in the input word  $w$  using a lexicographic order  $\leq$ , with the  $i$ -th coordinate being ordered first-to-last or last-to-first depending on the type of the  $i$ -th for loop. By definition

$$f(w) = \prod_{x_1, \dots, x_n} f(w, x_1, \dots, x_n)$$

where  $\prod$  stands for concatenation, with  $n$ -tuples  $(x_1, \dots, x_n)$  ordered by  $\leq$ .

The code of the for-transducer  $g \circ f$  is the same as the one for  $g$ , except that the for loops, the order tests  $\mathbf{x} < \mathbf{y}$  and the label tests  $\mathbf{a}(\mathbf{x})$  are modified as follows, to account for the representation of positions in  $f(w)$  via  $n$ -tuples of positions in  $w$ .

1. Suppose that  $g$  does a loop of one of the two types below (which are the only types allowed in a for-transducer in prenex normal form):

$$\underbrace{\text{for } u \text{ in first..last}}_{\text{first to last}} \quad \underbrace{\text{for } u \text{ in last..first}}_{\text{last to first}} \quad (4)$$

To simulate this loop, the for-transducer computing  $g \circ f$  enumerates through all  $n$ -tuples of positions in the input word  $w$ , ordered according to the ordering  $\leq$  – in the first-to-last case – or the opposite of  $\leq$  – in the last-to-first case. This enumeration is done using  $n$  nested for loops. For each  $n$ -tuple of positions  $(x_1, \dots, x_n)$  from this enumeration, the for-transducer computing  $g \circ f$  checks if the output

$$f(w, x_1, \dots, x_n) \in \Gamma^*$$

is nonempty. This check is done by running a fresh copy of  $f$  from scratch, and waiting until it reaches configuration  $(x_1, \dots, x_n)$ . If the above output is empty, then the body of the for loop in (4) is skipped, otherwise it is executed, with the label and order tests simulated as described below.

2. Suppose that  $g$  tests the order  $x \leq y$  on two positions in the intermediate word  $f(w)$ . In the for-transducer computing  $g \circ f$ , these positions are represented as two  $n$ -tuples  $(x_1, \dots, x_n)$  and  $(y_1, \dots, y_n)$ , and the corresponding comparison is  $\leq$ , which is a Boolean combination of comparisons on the positions  $x_i$  and  $y_i$ , and therefore can be implemented by a for-transducer.
3. Suppose that  $g$  tests the label  $a(x)$  of a position  $x$  in the intermediate word. In the for-transducer  $g \circ f$ , this position is represented as an  $n$ -tuple  $(x_1, \dots, x_n)$ . To determine the label of this position, the transducer  $g \circ f$  does the same trick as in item 1: it runs a fresh copy of  $f$  from scratch and waits until it reaches configuration  $(x_1, \dots, x_n)$ .

□

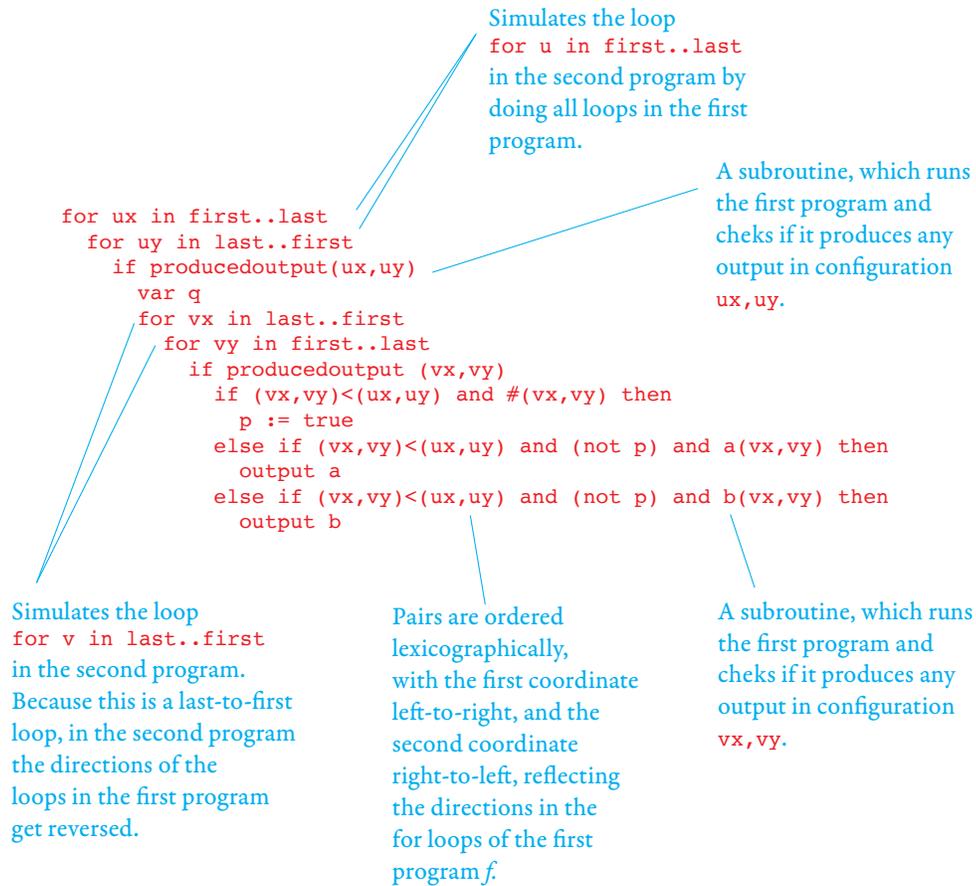
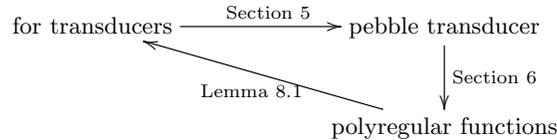


Figure 6: The for-transducer computing the composition of the functions described in Figure 5.

Before continuing with the proof that for-transducers can simulate polynomial list functions, we observe that the composition result above can be used to show equivalence of for-transducers with pebble transducers and polyregular functions. For-transducers are easily seen to recognize all of the atomic polyregular functions, i.e. iterated reverse, squaring and sequential functions. Also, first-order for-transducers are enough when only first-order sequential functions are used, see e.g. Example 4. Therefore, the composition closure result from Lemma 8.1 implies that for-transducers recognize all polyregular functions, and the conversion preserves first-order definability. Combining this with the results from the previous two sections, we get the equivalence of the following three models:



## 8.2 For-transducers implementing $\beta$ -reduction

In this section we complete the proof that polynomial list programs can be simulated by for-transducers. The idea is to use a term rewriting approach to the semantics of polynomial list functions, and to show that this approach can be simulated by a for-transducer.

**Definition 8.2 ( $\beta$ -reduction)** *Define  $\beta$ -reduction to be the binary relation on terms, denoted by*

$$M \rightarrow_{\beta} N,$$

*which holds if  $N$  can be obtained from  $M$  by applying one of the following re-*

duction rules to a subterm:

$$\begin{array}{lcl}
\text{is}_a b & \xrightarrow{\text{is}} & \begin{cases} \text{true} & \text{if } a = b \\ \text{false} & \text{if } a \neq b. \end{cases} \\
(\lambda x M)N & \xrightarrow{\lambda} & M[x := N] \quad \text{if } x \text{ is not bound in } M \\
\text{map } M[N_1, \dots, N_k] & \xrightarrow{\text{map}} & [MN_1, \dots, MN_k] \\
\text{proj}_i (M_0, M_1) & \xrightarrow{\text{proj}} & M_i \quad \text{for } i \in \{0, 1\} \\
\text{case } M_0 M_1 (\iota_i N) & \xrightarrow{\text{case}} & M_i N \quad \text{for } i \in \{0, 1\} \\
\text{split } [M_1, \dots, M_n] & \xrightarrow{\text{split}} & [([], [M_1, \dots, M_n]), \dots, ([M_1, \dots, M_n], [])] \\
\text{hd } [M_1, \dots, M_n] & \xrightarrow{\text{hd}} & \iota_1 M_1 \quad \text{for } n \geq 1 \\
\text{hd } [] & \xrightarrow{\text{hd}} & \iota_0 \perp \\
\text{tl } [M_1, \dots, M_n] & \xrightarrow{\text{tl}} & \iota_1 [M_2, \dots, M_n] \quad \text{for } n \geq 1 \\
\text{tl } [] & \xrightarrow{\text{tl}} & \iota_0 \perp
\end{array}$$

In the rules above, the blue colour highlights the first term of the left hand side of a rule, which is called the *leading term* of the rule (we use leading terms to define a reduction strategy).

It is easy to see that  $\beta$ -reduction changes neither the type nor the semantics of a term. Define a *redex* in a term  $M$  to be a subterm to which a reduction rule can be applied, and define the *type* of a redex to be the type of its main term. An example of a term and its redexes is shown in Figure 7.

The *normal form* of a term  $M$  is a term that has no redexes, and which can be obtained from  $M$  by finitely many steps of  $\beta$ -reduction and renaming bound variables (the latter process is called  $\alpha$ -conversion)<sup>14</sup>. For terms which represent strings – i.e. terms that have type  $\tau^*$  where  $\tau$  is a finite set – there

<sup>14</sup>One can show that  $\beta$ -reduction for our terms is well-founded and has the Church-Rosser property; the reason is that our calculus can be embedded in System  $\mathbf{F}$ , see Section 11.3 in [SU06]. This implies that each term has a unique normal form up to renaming of bound variables, and that this normal form is reached regardless of the order in which  $\beta$ -reduction is applied. However, the uniqueness of normal forms and strong normalisation are not needed for this section, so we do not prove them. Nevertheless, we acknowledge the uniqueness of normal forms, by talking about “the” normal form of a term as opposed to “a” normal form.

is no difference between a value – i.e. a string – and a term in normal form, as given in the following lemma. The lemma also implies the uniqueness of normal forms for terms representing strings.

**Lemma 8.3** *Let  $M$  be a term in normal form without free variables, whose type is  $\tau^*$  for some finite set  $\tau$ . Then  $M = [a_1, \dots, a_n]$  for some  $a_1, \dots, a_n \in \tau$ .*

**Proof**

A standard and folklore case analysis, see Appendix B.1.  $\square$

To compute the output of a polynomial list program  $M : \tau^* \rightarrow \sigma^*$  on an input string  $[a_1, \dots, a_n] \in \tau^*$ , we do the following:

1. produce the term  $M [a_1, \dots, a_n]$  by prepending  $M$  to the input word;
2. convert the term from step 1 into normal form;

Since  $\beta$ -reduction does not change the semantics of terms, and normal form terms of string type are the same as strings thanks to Lemma 8.3, it follows that the normal form of  $M [a_1, \dots, a_n]$  is the same as the output of  $M$  on input  $[a_1, \dots, a_n]$ , as defined in Section 4. To prove that the above steps can be simulated by a for-transducer, we show that there is a reduction strategy (a choice of which redexes to reduce first, with possibly several redexes being reduced in parallel) such that (a) one step of the reduction strategy can be implemented by a for-transducer; and (b) there is a constant  $k$  depending only on  $M$  such that for every input string  $[a_1, \dots, a_n]$  at most  $k$  steps of the reduction strategy are needed to convert  $M [a_1, \dots, a_n]$  into normal form. Since for-transducers are closed under composition by Lemma 8.1, it follows that every string-to-string function recognised by a polynomial list program is also recognised by a for-transducer. We begin by explaining how terms are represented as strings so that they can be handled by a for-transducer.

**For-transducers manipulating terms.** The difficulties with representing terms as strings come from: (a) renaming bound variables, (b) having complicated types in subterms, and (c) dealing with the tree structure of a term. To get around these difficulties, we assume a bound on the depth of terms and on the set of types allowed in subterms, as given in the following definition.

**Definition 8.4** *For a finite set of types  $\Gamma$  and  $k \in \{1, 2, \dots\}$ , define*

$$\text{Terms}(\Gamma, k)$$

*to be the terms which satisfy the following conditions:*

1. *the depth (length of the longest path in its syntax tree) is at most  $k$ ;*
2. *all subterms have types in  $\Gamma$ ;*
3. *for each type  $\tau \in \Gamma$  there are at most  $k$  variable names, call them  $x_1^\tau, \dots, x_k^\tau$ .*



Note that the size (number of nodes in the syntax tree) of a term can be unbounded even for terms of depth two, since the depth of a list of terms is one plus the maximal depth of terms on the list. The restriction on the variable names in item 3 of Definition 8.4 is not important for the semantics, because bound variables in a term of depth at most  $k$  can be renamed so that there are at most  $k$  variable names of each type (or even  $k$  variables altogether, if we would allow the same variable name to be used for different types). Nevertheless, having a finite set of variable names makes it possible to represent the terms as strings over a finite alphabet, which consists of the variable names, the atomic functions,  $\lambda$ , parentheses and commas. Because the depth of terms is bounded, a finite automaton can check if a string represents some term, in particular it can check if the parentheses are well matched. Using such a string representation, we can talk about a function

$$f : \text{Terms}(\Gamma, k) \rightarrow \text{Terms}(\Gamma, m) \quad \text{for } k, m \in \mathbb{N}$$

being recognized by a string-to-string device, such as a for-transducer. This notion is used in the following lemma, which is the main result of this section.

**Lemma 8.5** *For every finite set of types  $\Gamma$  and  $k \in \{1, 2, \dots\}$  there exists*

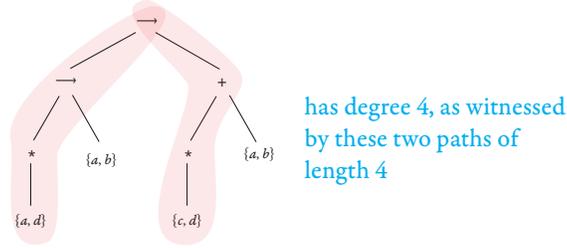
$$m \in \{1, 2, \dots\} \quad \text{and} \quad f : \text{Terms}(\Gamma, k) \rightarrow \text{Terms}(\Gamma, m)$$

*such that  $f$  is recognized by a for-transducer, and maps every input term to its normal form. For terms without the group product operation, a first-order for-transducer is enough.*

In particular, the lemma says that the depth of normal forms is bounded by a constant  $m$ , once the  $k$  and  $\Gamma$  have been fixed. This constant is a tower of exponentials, whose height is linear in the size of  $\Gamma$ , and this cannot be avoided, see Section 3.7 in [SU06].

The above lemma, together with Lemma 8.3 about normal forms of terms of string type, completes the proof that for-transducers can simulate polynomial list programs. The rest of Section 8 is devoted to proving the above lemma. We define a specific reduction strategy and show that it terminates in a constant number of steps. The key point is that the reduction strategy is parallel, i.e., multiple redexes are reduced at the same time.

**The reduction strategy.** The reduction strategy is adapted from Theorem 3.5.1 in [SU06], with the modification that it uses parallel reduction when possible. Likely other reduction strategies would work as well, e.g. reducing all innermost or all outermost redexes. Fix  $\Gamma$  and  $k$  in the assumptions of Lemma 8.5. Define the *degree* of a type to be the depth of its syntax tree, as illustrated below:



Define the *type* of a redex in a term to be the type of its leading term, see the remarks at the end of Definition 8.4 and the example in Figure 7. Define a *maximal redex* in a term  $M$  to be any redex such that (a) the type  $\tau$  of the redex has maximal degree among the types of other redexes in the same term; (b) the redex is maximally deep in the sense that there are no redexes of the same degree in its subtree. A term might have multiple maximal redexes, but they are incomparable in the tree ordering thanks to condition (b), so it makes sense to reduce them in parallel. Our reduction strategy is to reduce all maximal redexes in parallel: for a term  $M$ , define  $\text{red}M$  to be the result of reducing in parallel all maximal redexes (and doing nothing if there are no redexes). It is easy to see that the depth of  $\text{red}M$  is at most twice the depth of  $M$ . The doubling of depth occurs when reducing application with  $\lambda$ , other redexes increase the depth by at most 2.

The following lemma, together with the closure of for-transducers under composition from Lemma 5, completes the proof of Lemma 8.5, and therefore also the proof that polynomial list-programs can be simulated by for-transducers.

**Lemma 8.6** *Let  $\Gamma$  be a finite set of types and let  $k \in \{1, 2, \dots\}$ .*

1. *One step of the reduction strategy, i.e. the function*

$$\text{red} : \text{Terms}(\Gamma, k) \rightarrow \text{Terms}(\Gamma, 2k)$$

*is recognised by a for-transducer. For terms without group products, a first-order for-transducer is enough.*

2. *There exists some  $m \in \{1, 2, \dots\}$  such that*

$$\overbrace{\text{red} \circ \dots \circ \text{red}}^{m \text{ times}}(M)$$

*is in normal form for every  $M \in \text{Terms}(\Gamma, k)$ .*

**Proof**

The first item is proved by formalizing the  $\beta$ -reduction rules in a straightforward manner. We concentrate on the second item, about normalization in a bounded number of steps.

**Sublemma 8.6.1** *Let  $M$  be a term where the maximal degree of redexes is  $d$ . Then all redexes in  $\text{red}M$  have degree  $\leq d$  and*

$$|\text{red}M|_d < |M|_d$$

*where  $|M|_d$  is the maximal number of redexes of degree  $d$  that can be found on root-to-leaf paths in the syntax tree of  $M$ .*

**Proof**

By inspecting the reduction rules in the definition of  $\beta$ -reduction. This inspection is easier by looking at pictures of the reduction rules as tree transformations, see Appendix B.2.  $\square$

A corollary of the above sublemma is that if a term  $M$  has depth  $k$  and its maximal redexes have degree  $d$ , then  $k$  applications of  $\text{red}$  will eliminate all redexes of degree  $d$ , yielding a term where all redexes have degree  $< d$ . Furthermore, each application of  $\text{red}$  at most doubles the depth of a term, and thus eliminating all redexes of degree  $d$  in a term of depth  $k$  leads to a term of depth at most  $k \cdot 2^k$ . Putting these observations together, we see that a term normalises after a number of steps which depends only on the depth of the initial term and the maximal degree of types appearing in it. This dependence is a tower of exponentials, with the height of the tower being the maximal degree, see Section 3.7 in [SU06] for why this bound is tight.  $\square$

## Part III

# Algorithms

This part is about algorithms for polyregular functions. It has only one section, Section 9, which shows that polyregular functions can be efficiently evaluated. A future version might include an algorithm (or an undecidability proof) for the equivalence problem, which is left open for now:

**Open Problem.** Is equivalence decidable for polyregular functions, i.e., can one decide if two given polyregular functions produce the same outputs on every input?

## 9 Evaluation algorithms

In Section 9.1, we show that for every polyregular function there is a linear time algorithm for computing its values, in the sense that the running time is linear in the combined input and output size. In Section 9.2, we focus on another aspect of efficient evaluation, namely parallelization. We show that every first-order polyregular function can be computed in  $AC^0$ , i.e., by circuits of polynomial size and constant depth (assuming that the output contains a special empty letter  $\varepsilon$  which can be ignored).

### 9.1 Linear time

Polyregular functions can be evaluated in time linear in the combined input and output size.

**Theorem 9.1** *Every polyregular function can be evaluated in time*

$$\mathcal{O}(\text{length of input string} + \text{length of output string}).$$

A natural idea for evaluating a polyregular function would be to simply implement algorithmically the semantics of any one of the devices (compositions of atomic functions, polynomial list functions, for-transducers, or pebble automata). Unfortunately, in each case, the naive algorithm would have running time that is super-linear in the output size; the reason being that all of the devices mentioned above can do many steps before producing any output. Therefore, we need to run an optimisation in the algorithm such that subcomputations which do not produce any output are simulated in constant time.

The rest of Section 9.1 is devoted to finding such an optimisation. Our proof shows a slightly stronger result, namely that after a precomputation that is linear in the input string, one can start producing the output string with constant delay between positions. The ideas in the proof are closely based constant delay enumeration algorithms for MSO on strings, see [Bag06] and especially [KS13].

The only difference between Theorem 9.1 and [Bag06, KS13] is that in Theorem 9.1, we have to list tuples according to a given order, while the tuples in [Bag06, KS13] can be listed in an order chosen by the algorithm (e.g. lexicographic). This difference is not very important for the solution.

Let  $f$  be a polyregular function, which is recognised by a  $k$ -pebble transducer  $\mathcal{A}$ . A configuration of the  $k$ -pebble transducer is called *productive* if its state outputs at least one letter. The general idea in the linear time algorithm from Theorem 9.1 is to enumerate the productive configurations, with constant delay between two consecutive ones. By Lemma 2.3, for every states  $p$  and  $q$  and numbers  $i, j \in \{1, \dots, k\}$  there is an MSO formula

$$\varphi_{pq}^{ij}(\underbrace{x_1, \dots, x_i}_{\bar{x}}, \underbrace{y_1, \dots, y_j}_{\bar{y}})$$

such that for every nonempty word  $w$  over the input alphabet,

$$w \models \varphi_{pq}^{ij}(\bar{x}, \bar{y})$$

holds if and only if the automaton  $\mathcal{A}$  has a run from configuration  $p(\bar{x})$  to configuration  $q(\bar{y})$ . Using the above formulas, we can express in MSO that the automaton has a run from  $p(\bar{x})$  to  $q(\bar{y})$  which uses only non-productive configurations except for the source and target. In other words, the “next productive configuration” function can be defined in MSO. The following lemma shows that, using a data structure that can be computed in linear time with respect to the input, one can evaluate in constant time every MSO definable partial function from tuples of positions to tuples of positions, in particular the “next productive configuration” function. Therefore, to prove Theorem 9.1 it remains to prove the lemma.

**Lemma 9.2** *Consider an MSO formula*

$$\varphi(\overbrace{x_1, \dots, x_i, y_1, \dots, y_j}^{\text{first-order variables}})$$

which uses order  $<$  and label predicates  $a(-)$  for  $a \in \Sigma$ . There is an algorithm which does the following for every input string  $w \in \Sigma^*$ .

1. Computes a data structure in time linear in  $|w|$ ;
2. Using the data structure, answers the following queries in constant time:
  - **Input.** A tuple of positions  $\bar{x} = x_1, \dots, x_i$  in  $w$ ;
  - **Output.** The lexicographically least tuple of positions  $\bar{y} = y_1, \dots, y_j$  such that  $w \models \varphi(\bar{x}, \bar{y})$ , or an answer that no such tuple exists.

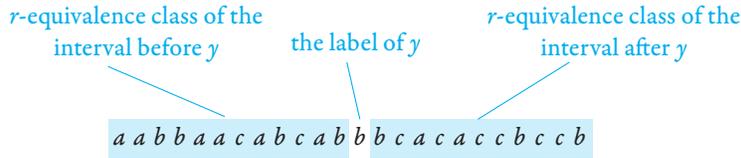
**Proof**

The idea is to use factorisation forests and compositionality of MSO, in the same

way as in [Col07], [KS13] or [Boj09, Section 2.1]. Since the proof is a minor adaptation of [Col07, Boj09, KS13], we only give a rough sketch.

We assume without loss of generality that  $j = 1$ , i.e. the output tuple  $\bar{y}$  consists of only one position. The case of  $j > 1$  can be easily reduced to the case of  $j = 1$ , by computing successively the coordinates of the output tuple  $\bar{y}$ .

For  $r \in \{1, 2, \dots\}$  define  $r$ -equivalence to be the equivalence relation on  $\Sigma^*$  which identifies two words when they satisfy the same MSO sentences of quantifier rank at most  $r$ . Compositionality for MSO says that  $r$ -equivalence classes can be equipped with a monoid structure so that mapping a word to its  $r$ -equivalence class becomes a monoid homomorphism (see e.g. [Tho97, proof of Lemma 4.1]). Furthermore, for every position  $y$  and every MSO formula  $\psi(y)$  of quantifier rank at most  $r$ , whether or not  $w \models \psi(y)$  depends only on the following information (an interval is a connected set of positions in a word):



A similar result is true for queries with more free variables.

The data structure from item 1 in the lemma will be a factorisation as in the Factorisation Forest Theorem; we have already used a similar data structure in Section 6. Let us begin with some terminology. A *factorisation* of an interval  $I$  is a sequence of intervals  $I_1, \dots, I_m$  which partitions  $I$ , listed in left-to-right order. When talking about the  $r$ -equivalence class of an interval  $I$  in a word  $w$ , we mean the  $r$ -equivalence class of the word  $w[I]$  which is obtained from  $w$  by keeping only positions from  $I$ .

**The data structure.** We now define the data structure from item 1 of the lemma. Define an  $r$ -factorisation of an input word  $w \in \Sigma^*$  to be a tree (an unranked tree with ordered siblings) with nodes labelled by intervals in  $w$  such that:

1. the root is labelled by the full interval containing all positions;
2. leaves are labelled by intervals with one position only;
3. if a node has label  $I$  and its children have labels  $I_1, \dots, I_m$  then
  - (a)  $m \geq 2$  and the intervals  $I_1, \dots, I_m$  are a factorisation of  $I$ ;
  - (b) if  $m > 2$ , then  $I_1, \dots, I_m$  have the same  $r$ -equivalence class, which is idempotent in the semigroup of  $r$ -equivalence classes<sup>15</sup>.

<sup>15</sup>In Section 6 we did not assume idempotence, because we wanted the factorisation to be defined in first-order logic. Here we do not need first-order definability, which allows us to use idempotence.

Let  $r$  be the quantifier rank of the formula  $\varphi$  in the assumption of the lemma. One can compute in linear time an  $(r + 1)$ -factorisation  $t$  of  $w$  whose depth is constant, i.e. depends only on  $r$  and not on  $w$ . The algorithm that computes  $t$  is implicit in the proof of the Factorisation Forest Theorem [Sim90], see also [BP10] for an explicit description of this algorithm and its more efficient versions. The  $(r + 1)$ -factorisation  $t$  is the data structure from item 1 in the statement of the lemma. We also assume that each interval  $I$  in the factorisation is labelled by its  $(r + 1)$ -equivalence class of  $I$ ; these labels can be computed in linear time.

**Using the data structure.** It remains to show that the data structure  $t$  can be used to answer in constant time the queries from item 2 in the statement of the lemma. The key step is given in the following sublemma. To get the result from the statement of the lemma, one applies a routine compositionality argument, see [KS13, Theorem 3.1]. Therefore, we only prove the sublemma.

**Sublemma 9.2.1** *Using  $t$ , the following can be answered in constant time:*

- **Input.** *An interval  $I$  and an MSO query  $\psi(y)$  of quantifier rank  $r$ ;*
- **Output.** *The first position  $y \in I$  such that  $y$  satisfies  $\psi$  in  $w[I]$ , or an answer that there is no such position.*

**Proof**

Distinct nodes in  $t$  are labelled by distinct intervals, and therefore we can identify nodes in  $t$  with the intervals that label them. Choose the node in the tree which represents an interval  $J \subseteq I$  that contains  $I$  and is smallest inclusion-wise for this property. This node can be computed in constant time, because the depth of the tree is constant. The algorithm works by induction on the height of the subtree of  $J$ . If  $J$  is a leaf, then  $w[I]$  has one letter only, and therefore one only needs to check if the unique position  $y \in I$  satisfies  $\psi(y)$  in  $w[I]$ , which can be done in constant time.

Otherwise  $J$  has at least two children whose represent a factorisation

$$J = J_1 \cup \dots \cup J_m.$$

We only deal with the more interesting case of  $m > 2$ , in which case all of the intervals  $J_1, \dots, J_m$  have the same  $(r + 1)$ -equivalence class  $e$ , which is idempotent. Choose  $first, last \in \{1, \dots, m\}$  so that  $J_{first}$  contains the first position of  $I$  and  $J_{last}$  contains the last position of  $I$ . By assumption on minimality of  $J$ , we have  $first < last$ . The key observation is that if there is some position  $y \in I$  which satisfies  $\psi(y)$  in the word  $w[I]$ , then the leftmost position with this property belongs to one of the intervals

$$J_{first}, J_{first+1}, J_{last-1}, J_{last}. \tag{5}$$

This is because of the assumption on the idempotence of the  $(r + 1)$ -type  $e$ , which tells us when an interval contains at least one position satisfying a given

rank  $r$  query  $\psi(y)$ . (This is the same kind of observation as was used in Sublemma 6.8.2.) Therefore, we can use the induction assumption to search for the position  $y$  in one of the four intervals in (5).  $\square \square$

We finish Section 9.1 with some questions for future work. As mentioned at the beginning of the proof of Theorem 9.1, our proof shows that after a precomputation that is linear in the input size, one can start producing the output word with constant delay between positions. This construction leaves some space for improvement. One improvement would be to lower the constants in the  $\mathcal{O}$  notation. In the current algorithm, the constants in the linear time are towers of exponentials in the size of the query, and therefore the algorithm is not likely to be practical. Maybe the constants can be made polynomial in the size of a pebble automaton recognising the function? Another improvement would be to have random access to the output in the following sense: after a precomputation (linear time, or maybe  $n \log n$ ) one can answer in constant time queries of the form “give the  $i$ -th letter of the output word”. Yet another direction is other kinds of access to the output word, e.g., pattern matching. These questions are left for future work.

## 9.2 $AC^0$

This section is about evaluating polyregular functions using  $AC^0$  circuits (i.e., constant depth and polynomial size). There are two caveats: this can only be done for first-order polyregular functions, and the circuits are allowed to produce blank letters which do not count in the output. The takeaway is that evaluation of first-order polyregular functions can be done efficiently in parallel.

**Circuits defining string-to-string functions.** For definitions of circuits and the class  $AC^0$ , see [Str12, Section VIII]. A circuit with  $n$  input gates and  $k$  output gates defines a Boolean function  $2^n \rightarrow 2^k$ .

**Definition 9.3 (Functions in  $AC^0$ )** *A function  $f : 2^* \rightarrow 2^*$  is said to be in  $AC^0$  if there is a family of constant depth and polynomial size circuits  $\{C_n\}_{n \in \mathbb{N}}$  such that for input words  $w$  of length  $n$ , the output  $f(w)$  is obtained by applying the circuit  $C_n$  and reading the output gates. We extend the definition of  $AC^0$  functions to functions of type  $\Sigma^* \rightarrow \Gamma^*$ , for finite alphabets  $\Sigma, \Gamma$ , by coding letters as bit strings of fixed length.*

Under the above definition, the length of the output is uniquely determined by the length of the input (call such functions *fixed output length*), because it is determined by the number of output gates in the circuit appropriate to the input length. We want to use circuits to compute functions where the output length is not determined by the input length (call such functions *variable output length*), e.g. the homomorphism

$$h : \{a, b\}^* \rightarrow \{a\}^*$$

which erases all  $b$ 's does not have fixed output length but is polyregular. To extend the definition of  $\text{AC}^0$  to functions of variable length, we use the class of functions which can be decomposed as

$$\Sigma^* \xrightarrow{f} (\Gamma + \varepsilon)^* \xrightarrow{h} \Gamma^*$$

where  $f$  is in  $\text{AC}^0$  as in Definition 9.3 (and therefore has fixed output length) and  $h$  is the function that erases symbol  $\varepsilon$ . We use  $\text{homAC}^0$  to denote the resulting class of functions. An equivalent definition of the class, which motivates its name, would be to consider compositions of an  $\text{AC}^0$  function followed by an arbitrary string-to-string homomorphism (i.e. a homomorphism of free monoids), and not just the special homomorphism that erases  $\varepsilon$ .

Here is the main result of Section 9.2.

**Theorem 9.4** *Every first-order polyregular function is in  $\text{homAC}^0$ .*

The assumption on first-order is crucial, because the product operation in the two-element group is polyregular (even sequential), but not in  $\text{AC}^0$ , see [FSS84, Theorem 4.3].

**Example 11.** An alternative idea for dealing with variable output lengths is to pad the output with a symbol  $\#$ . The resulting class, call it *padding  $\text{AC}^0$* , would not contain some polyregular functions. Consider the homomorphism

$$h : \{a, b\}^* \rightarrow \{a\}^*$$

which erases all letters  $b$ . This function is clearly polyregular. If  $h$  were in padding  $\text{AC}^0$ , then there would be a family of  $\text{AC}^0$  circuits, in the sense of Definition 9.3, computing the function

$$w \in \{a, b\}^* \mapsto a^n \#^m$$

where  $n$  is the number of  $a$ 's in  $w$  and  $m$  is the number of  $b$ 's. By composing a circuit for the above function with a circuit that checks if the first  $\#$  is on an even numbered position, we would get an  $\text{AC}^0$  family of circuits for parity, contradicting [FSS84, Theorem 4.3].  $\square$

A natural idea for proving Theorem 9.4 would be to show that the class of functions  $\text{homAC}^0$  is closed under composition, and then show that the atomic first-order polyregular functions (first-order sequential functions, iterated reverse and squaring) are in  $\text{homAC}^0$ . Unfortunately,  $\text{homAC}^0$  is not closed under composition, which can be shown using the same reasoning as in Example 11. Therefore, we need a different approach.

We say that a sequential function has *fixed block size* if there is some  $c \in \{0, 1, 2, \dots\}$  such that for every transition in the underlying automaton, the associated output word (see item 3 in Definition 1.5) has length exactly  $c$ . In particular, for nonempty inputs, the output is exactly  $c$  times longer than the input.

**Lemma 9.5** *Every first-order polyregular function can be decomposed as*

$$h \circ f_1 \circ \dots \circ f_n$$

where

1.  $h$  is a homomorphism where letters are mapped to strings of length  $\leq 1$ ;
2. each of  $f_1, \dots, f_n$  is squaring, iterated reverse, or a first-order sequential function with fixed block size.

**Proof**

Let us write  $H$  for the functions as in the first item of the lemma, and  $F$  for the functions as in the second item of the lemma. If  $*$  denotes closure under composition, then statement of the lemma is that

$$\text{polyregular} \subseteq H \circ F^*.$$

(The converse inclusion is clearly seen to be true.) It is easy to see that the atomic first-order polyregular functions are all in  $H \circ F^*$ . Therefore, to prove the lemma, it remains to show that  $H \circ F^*$  is closed under composition. Since both  $F$  and  $H^*$  are closed under composition, it is enough to show

$$F \circ H \subseteq H \circ F^*,$$

which is left as an easy exercise for the reader.  $\square$

**Proof** (Proof of Theorem 9.4)

Apply Lemma 9.5, yielding a decomposition

$$h \circ f_1 \circ \dots \circ f_n.$$

Since the class  $\text{AC}^0$  from Definition 9.3 is closed under composition, to prove the lemma, it suffices to show that all of the functions  $f_1, \dots, f_n$  are in the class  $\text{AC}^0$ . In other words, we need to show that the class  $\text{AC}^0$  contains the squaring function, iterated reverse and also every first-order sequential function with fixed block size. For squaring, the circuit is easy to construct, and it has depth one since it essentially amounts to copying the input without really reading it. For first-order sequential functions with fixed block size  $c$ , we observe that the  $i$ -th bit of the output depends a first-order property of the first  $i/c$  input letters, and such first-order properties can be computed in  $\text{AC}^0$ , see [Str12, Theorem IX.2.1].

We are left with showing that iterated reverse is in  $\text{AC}^0$ . Without loss of generality we consider the case of two letters  $\{0, 1\}$  and a separator  $\#$ , as in this example

$$011\#1000\#00111 \quad \mapsto \quad 110\#0001\#11100$$

Suppose that the input has length  $n$ , and therefore also the output will have length  $n$ . For an output position  $i \in \{1, \dots, n\}$ , the  $i$ -th output letter is:

1. # if the  $i$ -th input letter is #;
2.  $a \in \{0, 1\}$  if there exist positions  $j < i < k$  such that:
  - (a) the input word has no separator strictly between positions  $j$  and  $k$ ;
  - (b)  $j = 0$  or the input word has a separator on position  $j$ ;
  - (c)  $k = n + 1$  or the input word has a separator on position  $k$ ;
  - (d) the  $(k - i + j)$ -th letter of the input is  $a$ .

The above conditions can be formalised in an  $AC^0$  circuit.

□

## References

- [AČ11] Rajeev Alur and Pavol Černý. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *ACM SIGPLAN Notices*, volume 46, pages 599–610. ACM, 2011.
- [AFR14] Rajeev Alur, Adam Freilich, and Mukund Raghothaman. Regular combinators for string transformations. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, page 9. ACM, 2014.
- [Alu10] Rajeev Alur. Expressiveness of streaming string transducers. 2010.
- [AU70] Alfred V Aho and Jeffrey D Ullman. A characterization of two-way deterministic classes of languages. *Journal of Computer and System Sciences*, 4(6):523–538, 1970.
- [Bag06] Guillaume Bagan. Mso queries on tree decomposable structures are computable with linear delay. In *International Workshop on Computer Science Logic*, pages 167–181. Springer, 2006.
- [BC] Mikołaj Bojańczyk and Wojciech Czerwiński. Automata toolbox, <https://www.mimuw.edu.pl/~bojan/upload/reduced-may-25.pdf>.
- [BDK18] Mikołaj Bojańczyk, Laure Daviaud, and Shankara Narayanan Krishna. Regular and first-order list functions. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 125–134, 2018.
- [Ber13] Jean Berstel. *Transductions and context-free languages*. Springer-Verlag, 2013.

- [BGMP15] Félix Baschenis, Olivier Gauwin, Anca Muscholl, and Gabriele Puppis. One-way definability of sweeping transducers. In *35th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'15)*, 2015.
- [Boj09] Mikołaj Bojańczyk. Factorization forests. In *International Conference on Developments in Language Theory*, pages 1–17. Springer, 2009.
- [BP10] Mikołaj Bojańczyk and Paweł Parys. Efficient evaluation of non-deterministic automata using factorization forests. In *International Colloquium on Automata, Languages, and Programming*, pages 515–526. Springer, 2010.
- [CD15] Olivier Carton and Luc Dartois. Aperiodic Two-way Transducers and FO-Transductions. *CSL*, 2015.
- [CE12] Bruno Courcelle and Joost Engelfriet. *Graph structure and monadic second-order logic: a language-theoretic approach*, volume 138. Cambridge University Press, 2012.
- [Cho77] Christian Choffrut. Une caractérisation des fonctions séquentielles et des fonctions sous-séquentielles en tant que relations rationnelles. *Theoretical Computer Science*, 5(3):325–337, 1977.
- [Cho79] Christian Choffrut. A generalization of ginsburg and rose’s characterization of gsm mappings. In *International Colloquium on Automata, Languages, and Programming*, pages 88–103. Springer, 1979.
- [Cho03] Christian Choffrut. Minimizing subsequential transducers: a survey. *Theoretical Computer Science*, 292(1):131–144, 2003.
- [CJ77] Michal P Chytil and Vojtěch Jákl. Serial composition of 2-way finite-state transducers and simple programs on strings. In *International Colloquium on Automata, Languages, and Programming*, pages 135–147. Springer, 1977.
- [Col07] Thomas Colcombet. A combinatorial theorem for trees. In *International Colloquium on Automata, Languages, and Programming*, pages 901–912. Springer, 2007.
- [DGK18] Vrunda Dave, Paul Gastin, and Shankara Narayanan Krishna. Regular transducer expressions for regular transformations. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 315–324, 2018.

- [EH01] Joost Engelfriet and Hendrik Jan Hoogeboom. Mso definable string transductions and two-way finite-state transducers. *ACM Transactions on Computational Logic (TOCL)*, 2(2):216–254, 2001.
- [Eil74] Samuel Eilenberg. *Automata, languages, and machines*. Academic press, 1974.
- [EM65] Calvin C Elgot and Jorge E Mezei. On relations defined by generalized finite automata. *IBM Journal of Research and Development*, 9(1):47–68, 1965.
- [EM02] Joost Engelfriet and Sebastian Maneth. Two-way finite state transducers with nested pebbles. In *International Symposium on Mathematical Foundations of Computer Science*, pages 234–244. Springer, 2002.
- [Eng15] Joost Engelfriet. Two-way pebble transducers for partial functions and their composition. *Acta Informatica*, 52(7-8):559–571, 2015.
- [FGL16] Emmanuel Filiot, Olivier Gauwin, and Nathan Lhote. First-order definability of rational transductions: An algebraic approach. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 387–396. ACM, 2016.
- [FGRS13] Emmanuel Filiot, Olivier Gauwin, Pierre-Alain Reynier, and Frédéric Servais. From two-way to one-way finite state transducers. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 468–477, 2013.
- [FR16] Emmanuel Filiot and Pierre-Alain Reynier. Transducers, logic and algebra for functions of finite words. *ACM SIGLOG News*, 3(3):4–19, 2016.
- [FSS84] Merrick Furst, James B Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical systems theory*, 17(1):13–27, 1984.
- [GH96] Noa Globberman and David Harel. Complexity Results for Two-Way and Multi-Pebble Automata and their Logics. *Theor. Comput. Sci.*, 169(2):161–184, 1996.
- [Gur82] Eitan M Gurari. The equivalence problem for deterministic two-way sequential transducers is decidable. *SIAM Journal on Computing*, 11(3):448–452, 1982.
- [Hut99] Graham Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, 1999.

- [Iba71] Oscar H Ibarra. Characterizations of some tape and time complexity classes of turing machines in terms of multihead and auxiliary stack automata. *Journal of Computer and System Sciences*, 5(2):88–117, April 1971.
- [KR65] Kenneth Krohn and John Rhodes. Algebraic theory of machines. i. prime decomposition theorem for finite semigroups and machines. *Transactions of the American Mathematical Society*, 116:450–464, 1965.
- [KS13] Wojciech Kazana and Luc Segoufin. Enumeration of monadic second-order queries on trees. *ACM Trans. Comput. Log.*, 14(4):1–12, 2013.
- [Kuf08] Manfred Kufleitner. The height of factorization forests. In *International Symposium on Mathematical Foundations of Computer Science*, pages 443–454. Springer, 2008.
- [LMSV01] Clemens Lautemann, Pierre McKenzie, Thomas Schwentick, and Heribert Vollmer. The descriptive complexity approach to logcfl. *Journal of Computer and System Sciences*, 62(4):629–652, 2001.
- [MSV03] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for xml transformers. *Journal of Computer and System Sciences*, 66(1):66–97, 2003.
- [RS91] Christophe Reutenauer and Marcel-Paul Schutzenberger. Minimization of rational word functions. *SIAM Journal on Computing*, 20(4):669–685, 1991.
- [Sco67] Dana Scott. Some definitional suggestions for automata theory. *Journal of Computer and System Sciences*, 1(2):187–212, 1967.
- [Sim90] Imre Simon. Factorization forests of finite height. *Theoretical Computer Science*, 72(1):65–94, 1990.
- [Str12] Howard Straubing. *Finite automata, formal logic, and circuit complexity*. Springer Science & Business Media, 2012.
- [Str18] Howard Straubing. First-order logic and aperiodic languages: A revisionist history. *SIGLOG News*, 6(1), 2018.
- [STT09] Jacques Sakarovitch, Reuben Thomas, and Reuben Thomas. *Elements of automata theory*, volume 6. Cambridge University Press Cambridge, 2009.
- [SU06] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*, volume 149. Elsevier, 2006.
- [Tho97] Wolfgang Thomas. Languages, automata, and logic. In *Handbook of formal languages*, pages 389–455. Springer, 1997.

- [Tra08] Boris A Trakhtenbrot. From logic to theoretical computer science—an update. In *Pillars of computer science*, pages 1–38. Springer, 2008.
- [UH67] JD Ullman and JE Hopcroft. An approach to a unified theory of automata. *Bell System Technical Journal*, 46(8):1793–1829, 1967.

## Part IV

# Appendix

## A Haskell Code

This section contains Haskell code for the polynomial list programs used in the paper.

### A.1 Atomic polynomial list programs

```
data Errortype = Error

-- projections
first :: (a, b) -> a
first (x,y) = x
second :: (a, b) -> b
second (x,y) = y

-- coprojections are built into Haskell
-- Left :: a -> Either a b
-- Right :: b -> Either a b

-- case
cases :: (a1 -> b) -> (a2 -> b) -> Either a1 a2 -> b
cases f g (Left x) = f x
cases f g (Right x) = g x

-- map is built into Haskell
-- map :: (a -> b) -> [a] -> [b]

-- concat is built into Haskell
-- concat :: [[a]] -> [a]

-- similar to Haskell head, but with an explicit error in the type
hd :: [a] -> Either Errortype a
hd (h:_) = Right(h)
hd [] = Left(Error)

-- similar to Haskell tail, but with an explicit error in the type
tl :: [a] -> Either Errortype [a]
tl (h:t) = Right t
tl [] = Left (Error)

-- input a list and return all ways of splitting it into two parts
split :: [a] -> [[a], [a]]
split [] = [[], []]
split (h:t) = (map (\(x,y) -> (h:x,y)) (split t)) ++ [[], h:t]
```

## A.2 Iterated reverse

```
-- like map, but only keeps the non-error values
errmap :: (a -> ( Either Errortype b)) -> [a] -> [b]
errmap f l = concat(map
  (\e -> cases (\e -> []) (\e -> [e]) ( f e)) l)

-- we can use reverse then else because it could be coded as such:
-- reverse l = errmap (\x -> hd (second x)) (split l)

-- empty test for lists
empty :: [a] -> Bool
empty l = cases (\x -> True) (\x -> False) (hd l)

-- errcatch: if there is an error, then replace it by some default value
errcatch :: (Either Errortype a) -> a -> a
errcatch a b = cases (\x -> b) id a

-- is there some element on the list that satisfies f?
exists :: (a -> Bool) -> [a] -> Bool
exists f l = not (empty (filter f l))

-- do all elements of the list satisfy f?
forall :: (a -> Bool) -> [a] -> Bool
forall f l = (empty (filter (not.f) l))

-- return the longest prefix of the list where all elements satisfy f
fprefix :: (a -> Bool) -> [a] -> [a]
fprefix f l =
  let
    onlyfs = filter (forall f) (map first (split l))
  in
    errcatch (hd (map (filter f) onlyfs)) []

-- which part of the coproduct is used?
isleft :: Either a b -> Bool
isleft (Left a) = True
isleft _ = False

-- we can use if then else because it could be coded as such:
-- ifthenelse :: Bool -> t -> t -> t
-- ifthenelse c t e = if (c == True) then t else e

-- we can use filter, because it could be coded as such:
-- filter :: (a -> Bool) -> [a] -> [a]
-- filter f l = concat (map (\x -> ifthenelse (f x) [x] [])) l

-- blocks a list of coproducts into maximal blocks of same type
-- for example
-- [Left 1, Left 2, Right 'a', Right 'b', Left 3, Right 'c', Right 'd']
```

```

-- will be mapped to
-- [Left [1,2], Right ['a','b'], Left[3], Right['c','d']]

block 1 =
  let
    --f inputs a prefix/suffix pair, and outputs the sametype prefix of b (see below)
    --if the last and first elements of a,b disagree on Left/Right type
    f (a,b) =
      let
        --sametype is the list prefix of b with same Left/Right type as first element
        sametype =
          let
            leftprefix =
              Left(
                concat (map (cases (\x -> [x]) (\x -> [])) (fprefix isleft b))
              )
            rightprefix =
              Right(
                concat (map (cases (\x -> []) (\x -> [x])) (fprefix (not.isleft) b))
              )
          in do
            {
              x <- hd b;
              return (cases (\_ -> leftprefix) (\_ -> rightprefix) x)
            }
          in
            if (empty a) then
              return sametype
            else
              do {
                lasta <- hd (reverse a);
                firstb <- hd b;
                if ((isleft firstb) && ((not.isleft) lasta)) then return sametype
                else if ((not.isleft) firstb) && (isleft lasta) then return sametype
                else Left Error
              }
          in
            reverse (errmap f (split 1))

```

### A.3 Squaring

```

-- maps [1,2,3] to [[1,2,3],[]],[1,2],[3]],[1],[2,3]],[[],[1,2,3]]
revsplit :: [a] -> [[a], [a]]
revsplit l = map (\x -> (reverse (second x), reverse (first x))) (split (reverse l))

-- for each element of the input list,
-- output a triple (before the element, the element, after the element)
triples :: [t] -> [[t], (t, [t])]
triples l = errmap
  (\p -> do

```

```

    { h <- hd (second p);
      t <- tl (second p);
      return (first p, (h,t))
    }
  )
  (revsplit l)

-- squaring
square :: [a] -> [Either a a]
square l =
  let
    reformat x =
      concat [map Left (first x),
              [Right (first (second x))],
              map Left (second (second x))]
  in
    concat (map reformat (triples l))

```

## B Appendix on $\beta$ -reduction

### B.1 Values

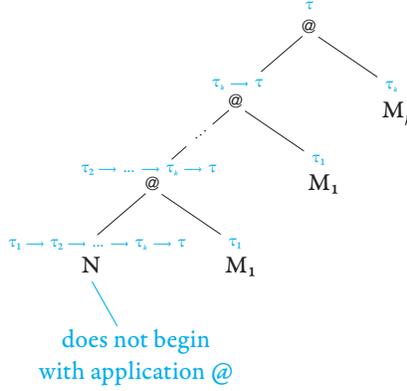
In this part of the appendix, we analyse the shape of terms in normal form. Define a *function type* to be a type where the outermost type constructor is  $\rightarrow$ , i.e. the type has form  $\tau \rightarrow \sigma$ . Similarly, we talk about *list types*  $\tau^*$ , *product types*  $\tau \times \sigma$ , *co-product types*  $\tau + \sigma$  and *finite set types*. The following lemma says that normal form terms with non-function types and no free variables can only be built using their appropriate term constructors. A corollary of the lemma is that a normal form term with an arrow-free type and no free variables can only be built using elements of finite sets, pairing, co-pairing and lists. In particular, the lemma below implies Lemma 8.3.

**Lemma B.1** *Let  $M : \tau$  be a term in normal form without free variables. Then:*

1. *If  $\tau$  is a finite set type, then  $M$  is an element of  $\tau$ .*
2. *If  $\tau$  is a coproduct type, then  $M$  is a coprojection of a normal form term;*
3. *If  $\tau$  is a product type, then  $M$  is a pair of normal form terms.*
4. *If  $\tau$  is a list type, then  $M$  is a list of normal form terms;*

#### Proof

Induction on the size of the term. Suppose that the type  $\tau$  is not a function type. Consider the leftmost branch of the syntax tree of  $M$ , and take the prefix of that path that uses only applications, leading to a decomposition of  $M$  as in the following picture:



The number  $k$  could be 0, in the case when the outermost operation in  $M$  is not an application. Consider the possibilities for the term  $N$ , according to the items in Definition 4.2.

1. *Variable.* This case cannot hold, because  $M$  has no free variables.
2. *Application.* This case cannot hold, by choice of  $N$ .
3. *Abstraction.* For  $k = 0$  this case cannot hold since otherwise  $\tau$  would be a function type. For  $k > 0$  this case cannot hold since otherwise there would be a redex.
4. *Element of a finite set.* In this case  $k = 0$  and item 1 in the conclusion of the lemma is satisfied.
5. *Pair of terms.* In this case  $k = 0$  and item 2 in the conclusion of the lemma is satisfied.
6. *Co-projection.* In this case  $k = 0$  and item 3 in the conclusion of the lemma is satisfied.
7. *List of terms.* In this case  $k = 0$  and item 4 in the conclusion of the lemma is satisfied.
8. *Atomic program.* This case cannot happen, which follows from a case analysis on the atomic programs:

**is** Since **is** takes an argument of finite set type, we must have  $k \geq 1$  since otherwise  $\tau$  would be a function type. By induction assumption,  $N_1$  is an element of the finite set in the input type for **is**. Therefore there is a redex, contradicting the assumption on normal form.

**proj<sub>i</sub>** Since **proj<sub>i</sub>** takes an argument of pair type, we must have  $k \geq 1$  since otherwise  $\tau$  would be a function type. By induction assumption,  $N_1$  is a pair of terms. Therefore there is a redex, contradicting the assumption on normal form.

**case** Since **case** takes three arguments, we must have  $k \geq 3$  because otherwise  $\tau$  would be a function type. In particular, the term  $N_3$  is defined, and by induction assumption it is a coprojection. Therefore there is a redex, contradicting the assumption on normal form.

- The remaining cases of **map**, **hd**, **tl**, **concat** and **split** are dealt with in the same way.

□

## B.2 Pictures of the reduction rules

This appendix contains pictures of the reduction rules from Definition 8.2.

