# Transducers of polynomial growth

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

## ABSTRACT

The polyregular functions are a class of string-to-string functions that have polynomial size outputs, and which can be defined using finite state models. There are many equivalent definitions of this class, with roots in automata theory, programming languages and logic. This paper surveys recent results on polyregular functions. It presents five of the equivalent definitions, and gives self-contained proofs for most of the equivalences. Decision problems as well as restricted subclasses of the polyregular functions are also discussed.

**ACM Reference Format:**
Mikołaj Bojańczyk. 2022. Transducers of polynomial growth. In *37th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS) (LICS '22), August 2–5, 2022, Haifa, Israel.* ACM, New York, NY, USA, 27 pages. https://doi.org/10.1145/3531130.3533326

Transducers, i.e. automata that produce outputs beyond "yes" or "no", have been around since the beginnings of automata theory. In fact, the early authors often understood automata to have string outputs, see e.g. Burks and Wang [10, Definition 2] or Trakhtenbrot [36, p. 1005]. The language approach gained ascendance; for example in their famous paper [31, p. 114], Rabin and Scott advocated moving to the language approach, writing that they were "doing away with a complicated output function and having our machines simply give *yes* or *no* answers". However, a few years later, Scott [33, Section 5] seems to have returned to the functional perspective, writing: "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." This paper also takes the functional perspective, with automata computing string-to-string functions, and languages being the special case when there are two possible output strings "yes" and "no".

The early transducer models were one-way automata equipped with some simple output mechanism, e.g. each transition could append a letter to the output string. Although such models are hardly different from automata, the fact that they are functions – and as such can be composed – means that certain things can be said about them that cannot be said about Boolean valued automata. A famous early example is the Krohn-Rhodes Decomposition Theorem [25, p. 454], which can be stated as follows: every one-way transducer is a composition of finitely many prime transducers, which are either reversible or have two states; see later in this paper for a full statement.

In later years, transducers have grown in sophistication, reaching a point where they resemble programming languages, with constructs such as numerical variables, loops, recursion or higher-order functions. This means that modern models of transducers can now be actually useful programs, as opposed to being idealized and radically simplified models of useful programs. All of this while retaining the good decidability properties of automata (transducers are not supposed to be Turing complete, which is framed here as an advantage: the halting problem is decidable) and their attractive mathematical theory.

In the last two decades, transducer theory has been energized by the study of a class of string-to-string functions that was first mentioned already in the bronze age of computer science: deterministic two-way automata with output (alternatively, Turing machines with a read-only input tape and no work tape) [34, Note 4]. More recent research revealed that this model has many different equivalent descriptions, originating in different fields, including: monadic second-order transductions [17, Section 4], streaming string transducers [1, Section 3], certain kinds of regular expressions [2, Section 2], and a calculus of functions based on combinators [8, Theorem 6.1]. Because of its remarkable robustness and good mathematical theory, Engelfriet and Hoogeboom proposed to call this class *the regular functions*, asserting its status as the string-to-string counterpart of the classical string-to-Boolean regular languages. From the perspective of this paper, one might want to use a more specific name, the *regular functions of linear growth*, since we describe here another class of string-to-string functions that has polynomial


*LICS '22, August 2–5, 2022, Haifa, Israel*

ACM ISBN 978-1-4503-9351-5/22/08…$15.00
https://doi.org/10.1145/3531130.3533326

growth, and yet seems to be equally robust as the functions of linear growth[1].

The functions described in the previous paragraph have linear growth. Given the importance of polynomial time, it is rather mysterious that polynomial functions have been neglected in the transducer literature. This paper is about such functions, which are called *polyregular*. The first appearance of the polyregular functions goes back to Engelfriet and Maneth [16, 20], who observed that if one restricts the pebble transducer model of Milo et al. [28] from trees to strings, then the resulting class of string-to-string functions is closed under composition [16, Theorem 11]. Also, this model has polynomial output sizes. The polyregular functions were further studied in [5, 9], where they were shown to have many alternative descriptions rooted in logic, automata theory, or programming languages. The present paper is meant to be an introduction to the polyregular functions, based on [5, 6, 9, 14, 15, 30], with a description of the alternative models, and proofs or proof sketches for some of the main results.

We finish this introduction with a polyregular function and its representation in three of the equivalent models defining polyregular functions. These three models, and several others, will be described in more detail later in the paper.

**Example 1.** Consider the function

$$f : \{a, b\}^* \to \{a, b\}^*,$$

of quadratic size output, which copies the input string as many times as it has letters, as in the following example:

$$abbb \quad \mapsto \quad abbbabbbabbbabbb.$$

Here are three ways of describing the function:

**Logic, see Section 5.** The output string $f(w)$ can be defined in terms of the input string $w$ using logic. Here, a string is viewed as a logical structure, whose universe consists of the string positions, and which has unary predicates for the letters and a binary predicate for order. In the case of our example function $f$, the positions of the output string are pairs of positions $(x, y)$ in the input string. The label – in the output string – for such a pair is inherited from the second coordinate $y$, while the order on pairs is lexicographic, which can be described using a formula:

$$\varphi(x_1, y_1, x_2, y_2) \stackrel{\text{def}}{=} \underbrace{x_1 < x_2 \vee (x_1 = x_2 \wedge y_1 \leq y_2)}_{\substack{\text{this formula says when a pair of positions } (x_1, y_1) \\ \text{is before a pair } (x_2, y_2) \text{ in lexicographic order}}}.$$

In this particular example, the formulas defining the labels and ordering are quantifier free, but in general we will allow monadic second-order logic MSO.

**Imperative programs, see Section 1.** The function $f$ is computed by the following program:

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

This program is an example of a for-transducer, which is an imperative program that can loop over positions in the input string, and ask for their labels, but which is not allowed to do any non-trivial arithmetic on loop positions, such as asking if $x = 2y$. In this sense, a for-transducer is similar to a formula of first-order logic, except that, unlike quantifiers in a formula, the for loops have an order of execution that is used to define the output string.

**Functional programs, see Section 4.** Yet another way to compute the function $f$ is the following Haskell program:

```
\w -> concat (map (\x-> w) w)
```

`concat`: operation that inputs a string of strings and concatenates them into a single string

`(map (\x-> w) w)`: string of strings that is obtained from `w` by replacing each letter with `w`

The appropriate fragment of Haskell has higher-order functions, and some basic string manipulating functions, such as `map` or `concat`, but beyond these primitives it has no recursion.

□

Here is the plan for the rest of the paper:

**Sections 1–5** We describe five equivalent models that describe the class of polyregular functions.

**Section 6** We discuss the growth rate of polyregular functions, including the special case of polyregular functions of linear growth, which turn out to be exactly the usual regular functions of Engelfriet and Hoogeboom.

**Section 6** We discuss subclasses of the polyregular functions, including a variant that uses first-order logic instead of MSO, as well as variants that are defined by constraining the pebbles in a pebble transducer.

**Section 8** We discuss the equivalence problem, i.e. the decision problem of determining if two devices define

[1] When talking about about linear or polynomial transducers, the quantity measured is the output size. Transducers are typically declarative formalisms, and therefore the "running time" depends on the evaluation strategy, somewhat similar to functional programs. As a rule of thumb, for most finite-state models evaluation is highly efficient, e.g. after linear preprocessing of the input string, one can have constant time random access to positions in the output string.

the same string-to-string function. Decidability of this problem is open for polyregular functions.

*Acknowledgement.* The author was supported by Polish National Agency for Academic Exchange, as well as the Leverhulme Trust. Also, the author would like to thank Bartek Klin and Rafał Stefański for helpful comments on draft versions.

# 1 FOR-TRANSDUCERS

We begin our discussion of polyregular functions with *for-transducers*, a model that was introduced in [5, Section 3]. Roughly speaking, a for-transducer is an imperative program, which uses for loops to range over positions in the input string. We already saw one example of a for-transducer in Example 1. That example used first-to-last for loops, but a for-transducer can also use last-to-first loops, as in the following example.

**Example 2.** The last-to-first order can be used to reverse the input string, as in the following program:

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

□

The two remaining features of for-transducers that have not been discussed yet are Boolean variables and equality tests on positions.

**Example 3.** Let us consider a slightly extended version of the squaring function from Example 1. This function inputs a string over alphabet $\{a, b\}$, and produces, for each prefix, a copy of the input string with that prefix underlined, and the copies are separated by |, as in the following example:

$$abaab \;\mapsto\; |\underline{a}baab|\underline{ab}aab|\underline{aba}ab|\underline{abaa}b|\underline{abaab}|$$

This function is implemented by the following program, which has two nested loops and uses a Boolean variable `prefix`.

```
output |
for x in first..last
  prefix := true
  for y in first..last
    if prefix then
      if a(y) then output a else output b
    else
      if a(y) then output a else output b
    if x = y then prefix := false
  output |
```

The variant of squaring that is used in this example will turn out to be fundamental for the polyregular functions, see Theorem 3.2, and therefore we will call it *squaring* from now on. □

More formally, a for-transducer is constructed as follows. There are two kinds of variables: position variables that range over positions in the input string, and Boolean variables that range over {true, false}. There are two kinds of loops

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

Such a loop creates a position variable `x`, which ranges over positions of the input string in one of the two specified orders. The atomic statements are

$$\underbrace{\texttt{output a}}_{\substack{\text{append a letter}\\ \text{to the output string}}} \qquad \underbrace{\texttt{X := true} \qquad \texttt{X := false}}_{\text{assign a value to a Boolean variable}}.$$

It is important that there are no assignments to position variables, i.e. the position variables are read-only. A sequential composition `I;J` of two for-transducers is also a for-transducer, and there is a conditional `if`, where the condition is a Boolean combination of tests of the following kinds:

$$\underbrace{\texttt{X}}_{\substack{\text{value of a}\\ \text{Boolean variable}}} \qquad \underbrace{\texttt{x = y}}_{\substack{\text{equality on}\\ \text{positions}}} \qquad \underbrace{\texttt{a(x)}}_{\substack{\text{label of}\\ \text{position}}}$$

The equality and label tests refer to the input string, because this is where the position variables live. We will not write variable declarations in the programs, with the types being implicit: the variables bound by the for loops are position variables, and the remaining ones are Booleans. The Booleans are initialized to false.

This completes the syntax of for-transducers. The semantics of a for-transducer is a string-to-string function which is defined in the expected way. The output size of a for-transducer is polynomial – if it has at most $k$ nested loops, then the output size will be $O(n^k)$. This bound is not tight: we will show in Section 6 that there are for-transducers of quadratic output size that need three (or more) nested loops.

**Example 4.** An order test $x \leq y$ can be implemented by running a for loop of first-to-last kind, and checking which of the variables $x$ or $y$ is seen first in the loop, with the result remembered in a Boolean variable. Similarly, we can implement a successor test $x = y + 1$. As we will see later, we can implement any test $\varphi(x_1, \ldots, x_n)$ that can be defined using a formula of monadic second-order logic. □

## 1.1 Regularity and closure under composition

We prove two fundamental results about for-transducers: (a) in the string-to-Boolean case they coincide with the regular languages; and (b) they are closed under compositions. Both of these results were originally proved for a model with pebbles that will be described in the next section, but the

proofs using for-transducers, as presented below, are a bit simpler.

*String-to-Boolean.* The main purpose of for-transducers is to define string-to-string functions, but a for-transducer can also be used to define a string-to-Boolean function (i.e. a language). This is done by eliminating the output instructions, and having a designated Boolean variable whose value at the end of execution is used as the output. As shown in the following theorem, in the string-to-Boolean case, we get exactly the regular languages, which underscores the idea that for-transducers are an extension of finite automata.

THEOREM 1.1. *The string-to-Boolean functions computed by for-transducers are exactly the regular languages.*

**Proof**
Clearly, every regular language is recognized by a string-to-Boolean for-transducer. This is because a for-transducer can simulate the run of a DFA, by storing the state in the Boolean variables, and using a single first-to-last for loop to process the movement of the one-way head in the DFA.

Consider the converse inclusion in the theorem, which says that every string-to-Boolean for-transducer computes a regular language. The proof uses a simplified version of an argument from [21, Theorem 3.2] about pebble automata. In the proof of this theorem, and also later on, it will be convenient to use a normal form, as explained in the following lemma. (The lemma works for both string-to-string and string-to-Boolean for-transducers.)

LEMMA 1.2. *For every for-transducer there is an equivalent one in the following form (called* prenex normal form*):*

```
for x_1 in τ_1 ∈ {first-to-last, last-to-first}
  for x_2 in τ_2 ∈ {first-to-last, last-to-first}
    ...
      for x_k in τ_k ∈ {first-to-last, last-to-first}
        body
epilogue
```

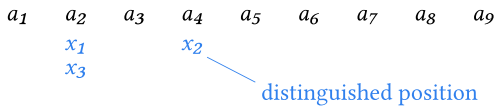
*where the subprograms* `body` *and* `epilogue` *do not use any for loops. Also,* `body` *produces at most one output letter in each loop iteration.*

**Proof**
The same kind of argument as when converting first-order formulas to prenex normal form. The epilogue is only needed for inputs of length at most one. For inputs with at least two letters; we can simulate sequential composition `I;J` using a for loop, with the first loop iteration running `I`, the second loop iteration running `J`, and the remaining iterations unused. To remember if we are in the first or second loop iteration, we use Boolean variables. □

Thanks to the lemma, it remains to show that if a string-to-Boolean for-transducer is in prenex normal form, then the corresponding language is regular. Suppose that the for-transducer has $k$ nested for loops. Let $Q$ be the finite set of valuations of the Boolean variables, i.e. all functions from Boolean variables to Booleans. Consider an input string $w$ together with $\ell \in \{0, \ldots, k\}$ distinguished positions $x_1, \ldots, x_\ell$. The corresponding *Boolean variable update* is defined to be the following function of type $Q \to Q$:

- **Input.** A valuation of the Boolean variables;
- **Output.** The valuation of Boolean variables that would arise after running the $k - \ell$ innermost for loops in the for-transducer, assuming that the Boolean variables are initialized to the input valuation, and the position variables of the $\ell$ outermost for loops are initialized to distinguished positions.

We will prove in Lemma 1.3 below that the Boolean variable updates can be described using a regular languages. In the lemma, we talk about regularity for strings with distinguished positions, as in the following example:

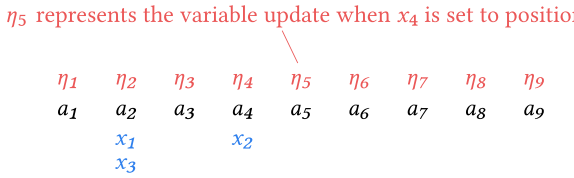



To talk about regularity for strings with (a fixed number of) distinguished positions, we represent them as strings without distinguished positions, and then appeal to the usual notion of regularity. A string with $\ell$ distinguished positions is represented as a string without distinguished positions over an alphabet consisting of $2^\ell$ copies of the original alphabet, with each copy indicating which of the distinguished positions are on the current position.

LEMMA 1.3. *For every $\ell \in \{0, \ldots, k\}$ and Boolean variable update $\eta : Q \to Q$, the following language of strings with distinguished positions is regular*

$$\{w, x_1, \ldots, x_\ell \mid w, x_1, \ldots, x_\ell \text{ has Boolean variable update } \eta\}.$$

**Proof**
Induction on $k - \ell$. The induction basis of $\ell = k$ is straightforward, since the Boolean variable update is obtained by running a for-transducer without any loops. Consider now the induction step, where the claim is assumed to be true for $\ell + 1$ and we want to prove it for $\ell$. For a string $w$ with distinguished positions $x_1, \ldots, x_\ell$, its Boolean variable update is obtained as follows. For every choice of the distinguished position $x_{\ell+1}$, we get a corresponding variable update, with one update for each position in the input string. Here is a picture:



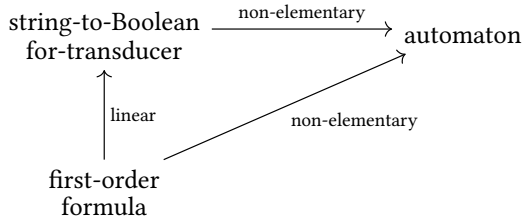

The automaton guesses these updates nondeterministically, and then: (a) checks, using the induction assumption, that they were guessed correctly; and (b) computes the composition of the updates, in first-to-last or last-to-first order, depending on the kind of loop that binds variable $x_{\ell+1}$. The check in item (a) requires applying a subset construction to the automaton from the induction assumption, since we need to run this automaton in parallel for all possible choices of distinguished position. □

To complete the proof of Theorem 1.1, apply the above lemma for $\ell = 0$, yielding for every Boolean variable update a corresponding regular language of strings without distinguished positions. The language in the conclusion of the theorem is obtained by taking the union of these regular languages, corresponding to those Boolean variable updates which map the all-false valuation (which is how the Boolean variables are initialized) to a valuation that has true on the designated output variable. □

**Example 5.** In this example, we show how string-to-Boolean for-transducers can easily simulate first-order logic. First-order logic can be used to define properties of strings, as illustrated in the following example, which gives a formula that defines the language $\{a, b\}^* a$:

$$\underbrace{\forall x\ \exists y\ y \geq x \wedge a(y)}_{\text{for every position } x\text{, there is a later position } y \text{ with label } a} .$$

It is easy to see that for every first-order formula, there is an equivalent string-to-Boolean for-transducer, which simulates quantifiers using loops. This construction is simple and direct, in particular the code of the for-transducer is linear in the size of the formula. A corollary is that the complexity of the construction in Theorem 1.1 is non-elementary (i.e. higher than any tower of exponentials of fixed height), if we want the regular language to be represented by an automaton. This is because the translation from first-order logic to automata is non-elementary [27, p. 150 (i)]. Here is a summary:




We can also convert a formula of monadic second-order logic, see Section 5, into a for-transducer, by passing through automata, but this incurs a non-elementary blowup (which seems to be necessary). □

*Composition.* We now show that string-to-string functions computed by for-transducers are closed under composition. This result is a variant of composition closure for pebble transducers [16, Corollary 7]. The construction is rather straightforward, in contrast to more involved constructions that are needed to compose linear size devices such as two-way transducers [11, Theorem 2] or streaming string transducers [7, Section 13.2]. The reason for the straightforward construction is that for-transducers can freely call sub-procedures that execute nested loops.

THEOREM 1.4. *String-to-string functions computed by for-transducers are closed under composition.*

**Proof**

The construction is easiest to see if we assume that both composed for-transducers are in the prenex normal form Lemma 1.2, as follows:

**First for-transducer**

```
for x_1 in τ_1 ∈ {first-to-last, last-to-first}
  for x_2 in τ_2 ∈ {first-to-last, last-to-first}
    ...
      for x_k in τ_k ∈ {first-to-last, last-to-first}
        body1(x_1, ..., x_k)
epilogue1
```

**Second for-transducer**

```
for y_1 in σ_1 ∈ {first-to-last, last-to-first}
  for y_2 in σ_2 ∈ {first-to-last, last-to-first}
    ...
      for y_ℓ in σ_ℓ ∈ {first-to-last, last-to-first}
        body2(y_1, ..., y_ℓ)
epilogue2
```

To simplify the notation, we assume below that there is no epilogue in the first for-transducer, i.e. `epilogue1` is empty; dealing with this epilogue is a simple exercise.

Our goal is to write a for-transducer for the composed function, in which the second for-transducer is executed on the output of the first for-transducer. In this composition, the variables $y_1, \ldots, y_\ell$ of the second for-transducer range over outputs of the first one, which are in one-to-one correspondence with tuples $(x_1, \ldots, x_k)$ where the body of the first transducer produces an output letter. Therefore, each loop in the second transducer can be replaced by $k$ loops, giving a for-transducer with $\ell \times k$ variables which looks like this (when multiplying kinds of loops, we assume that

first-to-last is represented by the number 1 and last-to-first is represented by the number −1):

**Composed for-transducer**

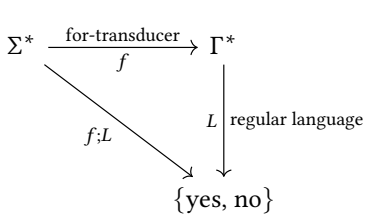
```
for z_11 in σ_1 · τ_1
  for z_12 in σ_1 · τ_2
    ...
      for z_1k in σ_1 · τ_k
        for z_21 in σ_2 · τ_1
          for z_22 in σ_2 · τ_2
            ...
              for z_ℓk in σ_ℓ · τ_k
                body(z_11, ..., z_ℓk)
epilogue2
```

The idea is that the new body is obtained from the body of the second for-transducer, by viewing each position variable $y_i$ as a tuple $(z_{i1}, \ldots, z_{ik})$. The details of the construction are left to the reader. One of these details is that we only care about iterations where all of the variables $y_1, \ldots, y_\ell$ are defined, which means that for every $i \in \{1, \ldots, \ell\}$, the body of the first for-transducer produces an output letter in the iteration corresponding to the valuation $(z_{i1}, \ldots, z_{ik})$. □

We end our discussion of for-transducers with the following corollary of Theorems 1.1 and 1.4, which shows that a variant of the halting problem is decidable in the case of for-transducers.

Corollary 1.5. *Given a for-transducer and a regular property of output strings, one can decide if all outputs of the for-transducer satisfy the regular property.*

**Proof**
We are given a for-transducer $f$ and a regular property $L$ and we want to know if the composition described by the diagonal arrow in the following picture gives only "yes" outputs:



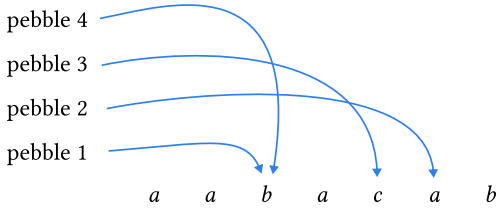

By Theorem 1.1, the regular language is a string-to-Boolean for-transducer. By Theorem 1.4, the diagonal arrow $f; L$ is a for-transducer as a composition of two for-transducers. By Theorem 1.1 again, the diagonal arrow is a regular language. Since all constructions are effective, we can decide the problem by testing the regular language $f; L$ for universality. □

By the remarks in Example 5, the computational complexity of the problem in the above corollary is non-elementary.

## 2 PEBBLE TRANSDUCERS

We now describe the second model for polyregular functions, which is called *pebble transducers*. Historically, this is the original model: pebble transducers were introduced by Milo et al. in [28, Section 3.1], as a transducer variant of pebble automata with "yes" and "no" outputs that were introduced by Globerman and Harel [21, Definition 4.1]. Milo et al. also generalized from strings to trees, but from the point of view of this paper, the string case originally considered by Globerman and Harel is more important: Engelfriet and Maneth [16, 20] observed that in the string-to-string case, pebble transducers have polynomial size outputs and are closed under composition (neither of which is true in the tree-to-tree case). Therefore, in this paper, we discuss only the string-to-string version of pebble transducers.

The idea behind a pebble transducer is that it is an automaton that uses pebbles to mark positions in the input word, as in the following picture:




Importantly, the pebbles are organized in a stack, with pebble 1 at the bottom of the stack. The stack discipline is what ensures that the model is finite-state [21, Theorem 4.2]; without the stack discipline the model would be the same as multi-head automata, which correspond to logarithmic space [24, Corollary 3.5]. The stack can be empty, but its height is bounded by a constant $k$ that is given in the pebble automaton. (There are models with unbounded numbers of pebbles [18].) We use the name *head* for the topmost pebble on the stack[2]. Apart from the pebble stack, the automaton has a state from a finite set. To update its configuration, the pebble transducer looks at its state, and the answers to the following *questions*, where $i, j \in \{1, \ldots, k\}$ range over pebble numbers:

(1) is the input string empty?
(2) is the $i$-th pebble on a position with label $a$?
(3) is the $i$-th pebble to the left of the $j$-th pebble?
(4) is the $i$-th pebble on the first/last input position?

The questions have "no" answers if they refer to pebbles that are not in the stack, which can happen if the stack has non-maximal height. Based on the state and the answers

[2]Like [28] but unlike [20, 21], we count the head as just another pebble. Therefore, a transducer with $k$ pebbles in our notation becomes, in the notation of [20, 21], a transducer with $k-1$ pebbles and a head. The motivation for our choice is that a transducer with $k$ pebbles in our notation will have output size $O(n^k)$.

to these questions, the pebble transducer deterministically chooses a new state and an *action* from the following set:

(1) output letter $a$;
(2) move the head one position to the left/right;
(3) push the first input position onto the pebble stack;
(4) pop the topmost pebble from the pebble stack;
(5) terminate.

Formally speaking, a *configuration* of the pebble transducer is defined to be an input string, together with a state and a stack $x_1, \ldots, x_\ell$ of positions in the input string whose height $\ell$ is in $\{0, \ldots, k\}$. The transducer begins in the configuration which has an empty stack, and a designated initial state. Next, it keeps on updating its configuration according to the above transition rules. The resulting sequence of configurations is called the *run* of the pebble transducer, and it stops when the "terminate" action is executed. The *output string* is defined to be the sequence of output letters that are produced by output actions during the run, in the order that they were executed. The output string is undefined if one of the following errors occurs: (a) the run never terminates; or (b) the head moves out of the input string; or (c) the empty stack is popped; or (d) the push action is executed when the stack has maximal height $k$. We will only care about pebble transducers which define total functions, which means that for every input string none of the errors (a) – (d) occur.

We use the name $k$-pebble transducer for a pebble transducer where the bound on the stack height is $k$. In the case of $k = 1$, we recover deterministic two-way automata with output, a model that was introduced by Shepherdson in 1959 [34, Note 4], and which is one of the descriptions of the class of *regular string-to-string functions* that was mentioned in the introduction and will be discussed later in this paper.

## 2.1 Equivalence with for-transducers

We now prove that pebble transducers compute the same string-to-string functions as for-transducers.

Theorem 2.1. *Pebble transducers and for-transducers compute the same string-to-string functions.*

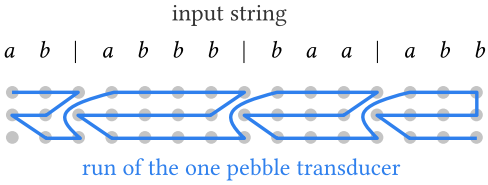
One direction is clear: a for-transducer can easily be simulated by a pebble transducer, with one pebble for each nested for loop, and with the state used to store the values of the Boolean variables and the currently executed instruction.

The interesting direction is simulating a pebble transducer with a for-transducer. The difficulty is that a pebble transducer can move its head both ways, while a loop in a for-transducer has a fixed direction, which is either first-to-last or last-to-first. Therefore, the converting a pebble transducer into a for-transducer can be seen as a sort of "untwisting", to use the terminology of [4]. Furthermore, even when simulating a one pebble transducer we might need more than one nested loop, as explained in the following example.

**Example 6.** Consider the *map reverse* function, which inputs several strings over alphabet $\{a, b\}$ separator by a symbol $|$, and reverses each of the strings, as illustrated in the following example:

$$ab|abbb|baa|a \quad \mapsto \quad ba|bbba|aab|a.$$

This function can be computed by a one pebble transducer, where the head movement illustrated as follows:




The map reverse function can also be easily computed by a for-transducer, which has two nested loops: an outer loop of first-to-last kind which looks for the separators, and an inner loop of last-to-first kind which produces the reversed strings.

We now argue that the nested loops are necessary, i.e. the map reverse function cannot be computed by a for-transducer that has no nested for loops (but it can use several for loops, as long as they are not nested in each other). This is proved in two steps: (a) every for-transducer without nested for loops is equivalent to a *sweeping two-way transducer*, i.e. a version of a one pebble transducer where the head can change directions only at the beginning and end of the input string; and (b) a sweeping two-way transducer cannot compute the map reverse function. Both of these steps are left to the reader; for the second step we refer to the characterization of sweeping two-way transducers in [3, Theorem 10]. □

As the above example illustrates, the simulation of a pebble transducer with a for-transducer will require some kind of non-trivial simulation of the two-way movement of the head in a pebble transducer. The rest of this section is devoted to presenting such a simulation. Fix a pebble transducer, with a stack bound $k$.

*Reachability oracles.* We begin with a sub-procedure, called the *reachability oracle*, which does not produce any output, but only tells us when two configurations of the pebble transducer can be connected by a run. We will only care about special kinds of runs: for $\ell \in \{0, \ldots, k\}$, an $\ell$-run is defined to be a run where both the source and target configurations have height exactly $\ell$, and all other configurations used in the run have height at least $\ell$. Here, the height of a configuration is the height of its pebble stack. This means that pebble $\ell$ is never popped, although it can be moved left and right,

and in particular the source and target configurations in such a run must agree on pebbles $< \ell$. The following lemma shows that a for-program with Boolean outputs can compute reachability for such runs.

Lemma 2.2 (Reachability oracle). *For every $\ell \in \{1, \ldots, k\}$ there is a for-transducer which inputs two configurations of height $\ell$ that have a common input string, call them* source *and* target*, and answers if they can be connected by an $\ell$-run.*

Before proving the lemma, we explain how the input and output are represented. As usual in a for-transducer, we get the input string, but we also get two states (stored in Boolean variables) as well as $\ell+1$ distinguished input positions (stored in position variables)

$$\underbrace{x_1, \ldots, x_{\ell-1}}_{\substack{\text{common part of the stack} \\ \text{in the source and} \\ \text{target configurations}}} \quad \underbrace{x_\ell}_{\substack{\text{head in} \\ \text{source}}} \quad \underbrace{x'_\ell}_{\substack{\text{head in} \\ \text{source}}}.$$

As in a string-to-Boolean transducer, the Boolean return value is stored in a distinguished output Boolean variable.

**Proof**

We use the classical Shepherdson algorithm for simulating a two-way automaton in a one-way pass [34], extended with recursive calls used to describe $(\ell+1)$-runs that might appear inside an $\ell$-run. The lemma is proved by induction on $\ell$, starting with $k$ and progressing down to 1. Suppose that we want to prove the lemma for $\ell$, and we are either at the induction basis of $\ell = k$, or we have already proved the lemma for $\ell + 1$.

In the proof, we use the following notation. For a state $q$ and a position $x$ in the input string, we write $\langle q, x\rangle$ for the configuration of height $\ell$ that is obtained from the source (or, equivalently, the target) configuration by changing the state to $q$ and the head position to $x$. Define the *Shepherdson profile* of a position $x$ to be the information about which pairs of configurations in the set

$$\{\text{source}, \text{target}\} \cup \{\langle q, x\rangle \mid q \text{ is a state}\}$$

can be connected by an $\ell$-run in which all configurations have pebble $\ell$ pointing to a position in the prefix $\{1, \ldots, x\}$. Note that the $\ell$-runs are allowed to place pebbles outside this prefix, as long as these pebbles are not pebble $\ell$. Our ultimate goal is to compute the Shepherdson profile of the last position $x$; in this case the Shepherdson profile tells us if there is an $\ell$-run from the source to target without any restrictions on the position of pebble $\ell$. The point of the Shepherdson profile is that it can be computed in a single first-to-last pass ranging over positions $x$, with recursive procedure calls that ask about runs of height $\ell + 1$. This is because the Shepherdson profile of a position $x$ is uniquely determined by the Shepherdson profile of the previous position $x - 1$ (if it exists), and the information about which pairs of configurations from the set

$$\{\langle q, x\rangle, \langle q, x-1\rangle \mid q \text{ is a state}\}$$

can be connected by an $\ell$-run that is *atomic*, which means that all of its configurations, except for the first and last one, have height $> \ell$. The answers to the questions about atomic $\ell$-runs can be obtained by a recursive call using the induction assumption, since an atomic $\ell$-run is either a single transition, or it is a push transition, followed by an $(\ell+1)$-run, followed by a pop transition. Therefore, the Shepherdson profile of the last position can be computed using a first-to-last loop over positions $x$ in the input string, which uses Boolean variables to store the most recently computed Shepherdson profile, and uses recursive calls for Shepherdson profiles corresponding to height $\ell + 1$. □

**Remark:** The code of the for-transducer in the above lemma is polynomial in the code of the pebble transducer; also it only uses first-to-last loops. The polynomial size in this remark assumes that we use a syntactic variant of for-transducers where sub-procedures that are used multiple times need only be defined once, instead of being inlined in each place of the code where they are used. This syntactic variant corresponds to the code of the for-transducer being represented as a directed acyclic graph, instead of a tree.

*Producing the output.* We now show the main part of the construction, where a for-transducer produces the output of a pebble transducer. This is described in the following lemma.

Lemma 2.3. *For every $\ell \in \{1, \ldots, k\}$ there is a for-transducer which inputs an $\ell$-run, represented by its source and target configurations, and produces the output of this $\ell$-run.*

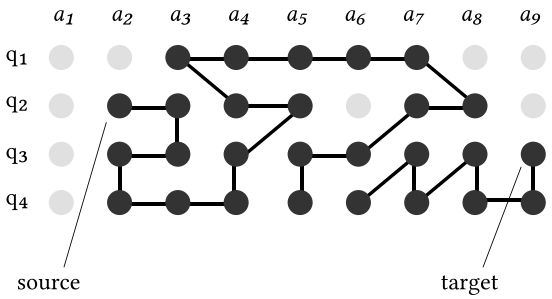
Before proving the lemma, we use it to conclude the proof of Theorem 2.1. Without loss of generality, we can assume that if the input string is nonempty, then the pebble transducer begins by pushing a pebble onto the initially empty stack, never pops that pebble, and then terminates in a stack of height one. For such a pebble transducer, the output can be computed by a for-transducer by using the lemma applied to $\ell = 1$.

**Proof**

We begin by clarifying what we mean by the output of an $\ell$-run. We think of an output letter as being produced in a transition that connects two consecutive configurations. The output of a run is defined to be all output letters that are produced between any two consecutive configurations inside the run; this output does not include any letter that would be produced when applying the transition function to the target configuration in the run. This definition avoids

double counting of output letters when decomposing a run into two parts.

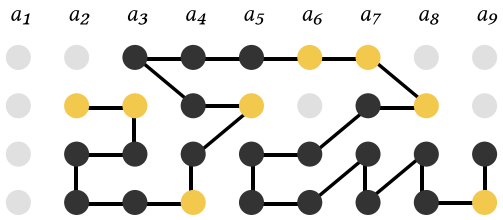
The condition that the run from source to target is indeed an $\ell$-run, i.e. it uses only configurations of height $\ell$, can be checked using the reachability oracle from Lemma 2.2. Therefore, from now on we assume that this condition is satisfied. Using the reachability oracle, we can find all configurations of level exactly $\ell$ that are used in the run, these configurations can be visualized as in the following picture, which shows only the head position (columns) and the state (rows):




In the above picture, a line connects two configurations of height $\ell$, but it represents a sub-run which can use many configurations of height $> \ell$. We will produce the output by induction on a parameter of the run which is called its *width*: the width of an $\ell$-run is the maximal number of times a position can have the head in a configuration of height exactly $\ell$. The maximal width is the number of states.

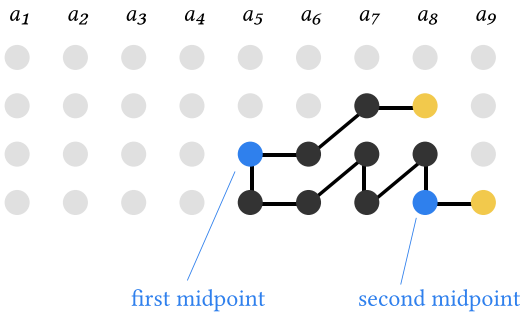
It remains to compute the outputs of $\ell$-runs that have width $m$, assuming that we already know how to compute the outputs of $\ell$-runs that have width strictly smaller than $m$. Suppose that we are given an $\ell$-run, represented by its source and target configurations. Define the *leftmost position* in an $\ell$-run to be the leftmost position that is used by pebble $\ell$ in the run, without taking into account the other pebbles. Similarly, we define the rightmost position. We present the construction under the assumption that the head position in the source is either leftmost or rightmost. The reduction to the case when this assumption is satisfied is an easy exercise that is left to the reader.

Assume by symmetry that the head in the source is the leftmost position of the $\ell$-run. Define the *pioneer configuration at position $x$* to be the first configuration of height exactly $\ell$ in the $\ell$-run where the head is at position $x$. Here is a picture with pioneer configurations highlighted in yellow:




Define the *step* of a position $x$ to be the $\ell$-run that begins in the pioneer configuration at position $x$ and continues until the pioneer configuration at the next position $x + 1$ (or until the end of the original $\ell$-run, if $x$ is the rightmost position). We show below how to produce the output of each step, assuming that the position $x$ is given. This will complete the proof, since the entire output is obtained by doing a first-to-last loop ranging over all positions which have a defined pioneer, and producing for each such position the output of its step.

To produce the output of the step of position $x$, we distinguish in this step two configurations of height $\ell$: the *first midpoint* is first time when pebble $\ell$ is placed in the leftmost position of the step, while the *second midpoint* is the last time in the step when pebble $\ell$ is placed in position $x$. These two configurations are depicted in the following picture:



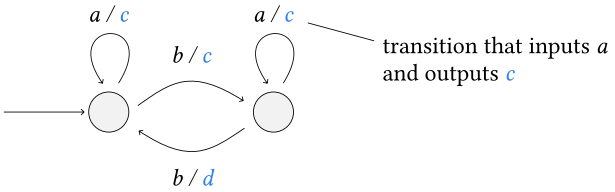

This way, the step of $x$ can be split into three parts: (1) from the beginning until the first midpoint; (2) from the first midpoint to the second midpoint; and (3) from the second midpoint until the end. The first two parts have strictly smaller width, because each position of pebble $\ell$ is used by both parts; therefore their outputs can be produced by induction assumption on smaller width. The last part has only two configurations of height $\ell$, and therefore its output can be produced by induction assumption on the height. □

**Remark:** Unlike the code for the reachability oracle, the code of the for-transducer in Lemma 2.3 is not polynomial, even if we use a directed acyclic graph representation of the code in a for-transducer. In fact, the code is exponential in the number of states, because of the induction on widths in the proof.

## 3 PRIME FUNCTIONS

So far we have two equivalent models: for-transducers and pebble transducers. Define the *polyregular functions* to be the string-to-string functions that are computed by these models. The models have features such as variables/pebbles and loops, which makes it easy to program them. In some contexts, it is useful to have a minimal set of features, e.g. when we want to prove that every polyregular function has some property. In this section, we give such a minimal model.

*Krohn-Rhodes for Mealy machines.* The inspiration for the minimal model is the Krohn-Rhodes Theorem [25, p. 454], which shows that each Mealy machine can be obtained by composing certain prime Mealy machines. Recall that a Mealy machine is defined to be a DFA without accepting states, where every transition is labelled by exactly one output letter, as in the following picture:




When composing Mealy machines, we mean composing the string-to-string functions that they compute. The prime machines are defined in terms of their state transformations, i.e. state-to-state functions that arise from input strings.

THEOREM 3.1 (KROHN-RHODES THEOREM). *Every Mealy machine is a composition of finitely many Mealy machines which satisfy one of the two following conditions:*

- reversible*: every state transformation is a permutation*[3]*;*
- flip-flop*: there are two states and every state transformation is a constant function or the identity.*

A Mealy machine that satisfies one of the two conditions in the theorem is called *prime*. A typical application of the theorem is to prove that all prime Mealy machines have some property, and that this property is preserved under composition; thus concluding that all Mealy machines have this property.

**Example 7.** Hopcroft and Ullman [23, Theorem 5] prove that two-way automata with output are closed under pre-composition with Mealy machines. This means that for every functions

$$\Sigma^* \xrightarrow[\text{Mealy machine}]{f} \Gamma^* \xrightarrow[\substack{\text{two-way automaton}\\ \text{with output}}]{g} \Delta*$$

their composition $f;g$ is also computed by a two-way automaton with output. The proof of Hopcroft and Ullman uses a smart trick, which can be avoided if we use the Krohn-Rhodes Theorem. Indeed, by the Krohn-Rhodes Theorem, it is enough to show that two-way automata with output are closed under pre-composition with the prime Mealy machines, in which case a straightforward construction can be applied. □

*Prime polyregular functions.* We now present a Krohn-Rhodes decomposition for polyregular functions. We use all of the primes for Mealy machines, since Mealy machines are a special case of polyregular functions. Beyond that, we add string-to-string homomorphisms, which are used essentially to erase letters from the input, as well as two prime functions for reversing strings and squaring the input length.

THEOREM 3.2. *The polyregular functions are the least class of string-to-string functions which is closed under composition, and which contains the following* prime *functions:*

(1) *the prime Mealy machines*
(2) *homomorphisms, i.e. string-to-string functions satisfying*

$$f(w_1 w_2) = f(w_1) f(w_2)$$

(3) *the squaring and map reverse functions that were defined in Examples 3 and 6, and are illustrated below:*

$$\underbrace{abaab \;\mapsto\; |\underline{a}baab|\underline{ab}aab|\underline{aba}ab|\underline{abaa}b|\underline{abaab}|}_{\textit{squaring}}$$

$$\underbrace{ab|aab|ba|abbb \;\mapsto\; ba|baa|ba|bbba}_{\textit{map reverse}}$$

**Proof**

In this proof, we call a string-to-string function *derivable* if it is a finite composition of the prime functions in the theorem. The theorem says that the polyregular functions are exactly the derivable ones. One inclusion is easy: each of the prime functions is computed by a for-transducer, and since for-transducers are closed under composition, it follows that all derivable functions are compute by for-transducers, i.e. polyregular. It remains to prove that all polyregular functions are derivable.

We begin with a basic toolkit of operations that will be useful later on.

CLAIM 3.3. *All of the following functions are derivable:*

(1) *Mealy machines;*
(2) *an extension of Mealy machines, where transitions are labelled by possibly empty strings (not just single letters);*

[3] In Section 7.1, we will care about languages and functions that can be defined in first-order logic, see Example 5. The Krohn-Rhodes Theorem, in its full version, also says that reversible machines with more than one state are not needed for Mealy machines which are first-order definable in the following sense: for every state $q$, there is a first-order formula that is true in exactly the input strings which lead to state $q$.

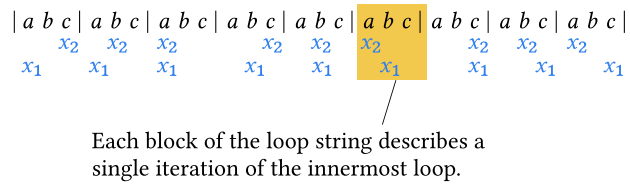
(3) *the extensions of squaring and map reverse which use any finite alphabet, instead of just two letters* $\{a, b\}$.

**Proof**
Item 1, about Mealy machines, is the Krohn-Rhodes Theorem. For item 2, about extended Mealy machines, we use homomorphisms: we first use a (non-extended) Mealy machine to label each input position with the corresponding transition of the extended Mealy machine, and then we use a homomorphism to replace each transition by its output string. Item 3, about squaring and map reverse for non-binary alphabets, is proved by encoding larger alphabets using a binary alphabet; such encoding and decoding can be done using the extended Mealy machines from item 2. □

We now show that every polyregular function is derivable. Suppose that we have a polyregular function, given by a for-transducer. By Lemma 4.2, we can assume that it is in prenex normal form:

```
for x1 in τ1
  for x2 in τ2
    ...
      for xk in τk
        body
epilogue
```

We will assume that the epilogue is empty; it is not hard to extend the construction to deal with the epilogue since it produces a constant number of letters that depend on regular properties of the input string. For an input string, define the corresponding *loop string* to be the string which describes all possible ways of evaluating the position variables, listed in the order that they are visited by the nested for loops, as illustrated in the following picture which considers the case of $k = 2$:




In the picture above, the input is *abc*, the external loop for $x_1$ is first-to-last, and internal loop for $x_2$ is last-to-first.

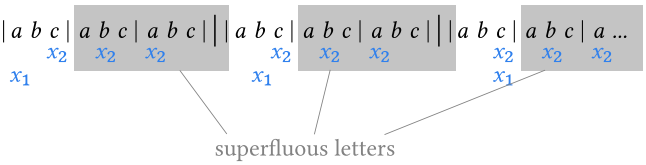
Claim 3.4. *A derivable function can produce the loop string.*

**Proof**
Consider first the case when there is only one variable in the for-transducer. If this variable has kind first-to-last, then there is nothing to do: the loop string is almost literally the output of the string function, except for a small recoding that can be implemented using an extended Mealy machine as in item 2 of Claim 3.3. If this variable has kind last-to-first, then we use a last-to-first variant of squaring

$$abaab \mapsto |abaab|abaab|abaab|abaab|abaab|$$

which can be derived from the usual squaring by composing reverse, squaring, and then map reverse.

Consider now the case when there are are at least two variables. We only show the construction for two variables, but the case of more variables is done in the same way. Suppose that there are two variables, say $x_1$ is first-to-last and $x_2$ is last-to-first. If we apply squaring twice (with the second squaring having last-to-first type), then we get a result that is almost the loop string, except that there are some superfluous letters as in the following picture:




To get rid of the superfluous letters, we use an extended Mealy machine. In fact – we need two passes, a first-to-last pass which marks the superfluous blocks (those that have just one variable), followed by a last-to-first pass which removes the superfluous blocks. The reason why we need to passes is that in order to mark a block as superfluous, we need to read it, and place a mark at the end of it; the second last-to-first pass deletes the blocks with the marks. Extended Mealy machines that process their input in last-to-first order can be obtained using reversing and the usual extended Mealy machines. □

Once we are given the loop string, the output of the for-transducer can be produced by an extended Mealy machine. This machine processes the loop string from left to right, using its state to store the values of the Boolean variables in the for-transducer, and produces the appropriate output at the end of each block of the loop string. □

## 4 FUNCTIONAL PROGRAMS

In this section, we present a description of the polyregular functions which uses functional programming. We begin with some examples in Haskell syntax, but a more formal description later in this section will use the simply typed $\lambda$-calculus.

**Example 8.** In Example 1, we already saw one example of a Haskell program, which implemented a version of the squaring function from Theorem 3.2, except that it did not have the underlines that indicate loop positions. To get these underlines, we will use another atomic function, apart from `map` and `concat`, which is called `split`. The `split` function maps a list to all possible decompositions into two parts,

with the second part nonempty, as explained in the following example:

$$[1,2,3]$$
$$\downarrow$$
$$[([1,2],[3]),([1],[2,3]),([],[1,2,3])].$$

Using the `split` function, the squaring function from Example 3 is computed as follows:

```
\x -> concat (map (\(a,b) ->
   concat [map left a, map right b]) (split x))
```

where `left` and `right` are the two co-projections

$$\texttt{left}, \texttt{right} : \{a,b\} \to \underbrace{\{a,b\}}_{\text{underlined}} + \underbrace{\{a,b\}}_{\text{not underlined}}.$$

For readability, we ignored the separator symbols | in the above code; these can be recovered with a minor modification. □

**Example 9.** Another operation that can be implemented using split from the previous example is list reverse. To the input list, say $[1,2,3]$, we apply the split function, yielding a list of pairs of lists like this:

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

Next, we use map to project each pair of lists to its second coordinate, yielding a list of lists like this:

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

The general idea, is that we now want to apply the head function to each list. However, in our programming language we will only have total functions, and therefore we will use the following list version of the head operation, which uses the empty list to simulate an error value:

```
\x -> if null x then [] else [head x].
```

Applying this function, call it `listhead`, to the list of lists, and then concatenating all results, we get the list reverse. Putting all of this into a single line of code, the reverse function is equal to

```
\x -> concat (map (\(a,b)-> listhead b) (split x)).
```

□

## 4.1 Polyregular $\lambda$-terms

We now give a formal description of the functional programming language that was illustrated in the above examples. This description uses a variant of the simply typed $\lambda$-calculus, which will be called the *polyregular $\lambda$-terms*. It is important that the system does not have any iteration or recursion mechanisms, such as the Y-combinator. Since polyregular $\lambda$-terms are the only kind of $\lambda$-terms that are considered in this paper, we will simply say $\lambda$-term when meaning polyregular $\lambda$-terms.

*Types.* We begin by describing the types of $\lambda$-terms. The atomic type is called 1, and it represents a set that has only one element. More complicated types are generated using the following type constructors:

$$\underbrace{\tau \times \sigma}_{\text{product}} \quad \underbrace{\tau \times \sigma}_{\substack{\text{co-product, i.e.} \\ \text{disjoint union}}} \quad \underbrace{\tau^*}_{\text{lists}} \quad \underbrace{\tau \to \sigma}_{\text{functions}}$$

This type system covers all finite sets, which are viewed as co-products of several copies of 1, e.g. a two element set is viewed as the type 1 + 1.

*The $\lambda$-terms.* We use $\lambda$-terms in Church style, i.e. each $\lambda$-term comes together with a type; this also covers the case of variables. The type is written using coloured superscripts as in $M^\tau$. The $\lambda$-terms are constructed as follows by induction. In the induction base, we use variables, with every variable coming together with a type. Apart from variables, the other kind of atomic $\lambda$-term is any of the atomic functions described in Figure 1. From the atomic $\lambda$-terms, further ones are constructed using the following term constructors:

$$\underbrace{(M^{\tau\to\sigma} N^\tau)^\sigma}_{\text{application}} \quad \underbrace{(\lambda x^\tau . M^\sigma)^{\tau\to\sigma}}_{\lambda\text{-abstraction}} \quad \underbrace{(M^\tau, N^\sigma)^{\tau\times\sigma}}_{\text{pairing}}$$

$$\underbrace{(\texttt{left}M^\tau)^{\tau+\sigma} \ (\texttt{right}M^\sigma)^{\tau+\sigma}}_{\text{co-projections}} \quad \underbrace{[M_1^\tau, \ldots, M_n^\tau]^{\tau^*}}_{\text{lists}} \quad \underbrace{1^1}_{\substack{\text{unique} \\ \text{element of} \\ \text{type 1}}}$$

This completes the description of the syntax of the programming language, i.e. the definition of the set of $\lambda$-terms. We consider two $\lambda$-terms to be equal if one can be obtained from the other by renaming bound variables, i.e. $\lambda$-terms are identified modulo $\alpha$-conversion.

The semantics of polyregular $\lambda$-terms is defined using reduction rules, which are presented in Figure 2. A polyregular $\lambda$-term is said to be in *normal form* if no further reductions can be applied to it. Because our calculus is a variant of the simply typed $\lambda$-calculus, every polyregular $\lambda$-term reduces in a finite number of steps to some polyregular $\lambda$-term in normal form, and the normal form is unique in the sense that it does not depend on the order in which the reductions were applied.

## 4.2 Equivalence to polyregular functions

The purpose of polyregular $\lambda$-terms is to compute polyregular string-to-string functions. What does this mean exactly? As mentioned before, every finite set $\Sigma$ can be viewed as a type, and therefore also the set of strings $\Sigma^*$ can be viewed

$\texttt{first}^{\tau\times\sigma\to\tau}$

$\texttt{second}^{\tau\times\sigma\to\sigma}$

$\texttt{cases}^{(\tau\to\rho)\to(\sigma\to\rho)\to(\tau+\sigma)\to\rho}$

$\texttt{headtail}^{\tau^*\to 1+\tau\times\tau^*}$

$\texttt{concat}^{\tau^{**}\to\tau^*}$

$\texttt{map}^{(\tau\to\sigma)\to\tau^*\to\sigma^*}$

$\texttt{split}^{\tau^*\to(\tau^*\times\tau^*)^*}$

$\texttt{mult}_G^{G^*\to G}$

**Figure 1: Atomic functions. For all types $\tau, \sigma, \rho$ and for all finite groups $G$, we have the atomic functions described above. The last function works as follows: for every finite type (i.e. every type defined using only $1$ and $+$) and every multiplication operation on it that defines a group, we have a function for the group multiplication; with multiplication viewed as a function from lists of group elements to group elements.**

as a type. A single string $w \in \Sigma^*$ can be viewed as a $\lambda$-term, which is a list of letters. Consider a $\lambda$-term without free variables, of type $\Sigma^* \to \Gamma^*$. The string-to-string function that it computes is defined to be the function that maps a string $w \in \Sigma^*$ to the string that is obtained by taking the application $Mw$ and then computing the normal form. The following theorem shows that the polyregular $\lambda$-terms of string-to-string type define exactly the polyregular functions.

Theorem 4.1. *A string-to-string function is polyregular if and only if it is computed by a polyregular $\lambda$-term.*

**Proof**

When going from polyregular functions to $\lambda$-terms, we use the definition of polyregular functions in terms of compositions of primes from Theorem 3.2. For the other direction, we use for-transducers.

*From prime functions to $\lambda$-terms.* Since functions computed by $\lambda$-terms are clearly closed under composition, and since all polyregular functions can be decomposed into polyregular primes by Theorem 3.2, it is enough to show that the polyregular primes can be computed by $\lambda$-terms.

- *Homomorphisms.* A homomorphism is implemented by first replacing each input letter by a corresponding list using `map`, and then using `concat` to flatten the resulting list of lists.
- *Squaring.* This function is implemented in Example 8.
- *Map reverse.* We begin by reversing lists without separators. This function was implemented in Example 9; the only missing part is that we need to implement the `listhead` function used in Example 9 in terms of our atomic `headtail`, which is done as follows:

$$\lambda x.\texttt{cases}\ (\lambda y.[])\ (\lambda y.[\texttt{fst}\ y])\ (\texttt{headtail}\ x).$$

We now turn to map reverse, where the input is a string of strings, represented as a string using the separator symbol |. We need to process this representation, by using each occurrence of the separator to split the list, as in the following function:

$$[a, b, |, a, a, b, |, |, a] \ \mapsto\ [[a, b], [|], [a, a, b], [|, |], [a]].$$

The implementation of this function is left to the reader; the idea is to apply split twice to the input list so that we can inspect all of its infixes.

- *Prime Mealy machines.* Consider a function

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

that is computed by a prime Mealy machine with states $Q$. We begin with a $\lambda$-term that computes the *last state function* of the machine, i.e. the function which maps an input string to the state of the Mealy machine after reading it.

If the prime Mealy machine is reversible, then the last state function can easily be derived using multiplication in the group of reversible state transformations, which is one of the atomic $\lambda$-terms. Let us now derive the last state function for a prime Mealy machine of flip-flop kind, which means that for each input letter the corresponding state transformation is constant or the identity. Let us represent the identity state transformation as the empty list [], and a constant state transformation as a one element list $[q]$ where $q$ is the constant state. Under this representation, to compose a list of state transformations, we simply need to concatenate this list, reverse it, and then take the head. All of these operations can be implemented using $\lambda$-terms, thus computing the last state function.

Once we have the last state function, the output of the Mealy machine is produced as follows. Using split, we compute a function of type

$$\Sigma^* \to (\Sigma^* \times \Sigma)^*$$

which decorates each input position with the prefix that leads up to – but not including – that position. Next, using map and the last state function, we can replace each such prefix by its corresponding state, thus yielding a list of type $(Q \times \Sigma)^*$. Finally, each pair $Q\times\Sigma$ is replaced by the corresponding output letter, by consulting the output function of the Mealy machine.

$$
\begin{array}{rcll}
(\lambda x.M)\ N & \to & M[x := N] & \text{assuming } x \text{ is not free in } N \\
\texttt{fst}\ (M, N) & \to & M & \\
\texttt{snd}\ (M, N) & \to & N & \\
\texttt{cases}\ M\ N\ (\texttt{Left}\ K) & \to & M\ K & \\
\texttt{cases}\ M\ N\ (\texttt{Right}\ K) & \to & N\ K & \\
\texttt{headtail}\ [] & \to & \texttt{Left}\ 1 & \\
\texttt{headtail}\ [M_1, \ldots, M_n] & \to & \texttt{Right}\ (M_1, [M_2, \ldots, M_n]) & \text{assuming } n \geq 1 \\
\texttt{concat}[[M_1, \ldots, M_{n_1}], [M_{n_1+1}, \ldots, M_{n_2}], \ldots, [M_{n_{\ell-1}+1}, \ldots, M_{n_\ell}]] & \to & [M_1, \ldots, M_{n_\ell}] & \\
\texttt{map}\ M\ [N_1, \ldots, N_n] & \to & [M\ N_1, \ldots, M\ N_n] & \\
\texttt{split}\ [M_1, \ldots, M_n] & \to & [([M_1, \ldots, M_{n-1}], [M_n]), \ldots, ([], [M_1, \ldots, M_n])] &
\end{array}
$$

**Figure 2: Reduction rules. (For readability, we omit the Church style type superscripts.)**

*From $\lambda$-terms to for-transducers.* We now move to the more interesting direction in the theorem: from $\lambda$-terms to polyregular functions. As our representation of polyregular functions, we use for-transducers.

We will show that a for-transducer can compute the normal form of a $\lambda$-term, assuming certain bounds on the resources used by this term. To compute the normal form of a $\lambda$-term, we use a certain reduction strategy. The idea is that the reduction strategy is parallel, i.e. if the $\lambda$-term is a list, then the $\lambda$-terms in the list will be reduced independently in a single reduction step. This way it will be possible to get to the normal form in a constant number of steps, even for lists of unbounded length.

We now explain our approach more precisely. Define a *redex* in a $\lambda$-term to be a subterm to which a reduction rule can be applied. For example, the following $\lambda$-term (without type annotation for brevity) has four redexes:

$$
\overbrace{\texttt{split}\ [\underbrace{\texttt{concat}\ [[1,1],[]]}_{\text{inner redex 1}},\ \underbrace{\texttt{map}\ (\lambda x.x)\ [1,1]}_{\text{inner redex 2}},\ \underbrace{\texttt{fst}([],1)}_{\text{inner redex 3}}]}^{\text{outer redex}}.
$$

Reducing a redex can create new redexes, e.g. reducing inner redex 2 in the above example will create two terms of the form $(\lambda x.x)\ 1$, which are themselves redexes. Define a *reduction strategy* to be a function which inputs a $\lambda$-term and returns a set of redexes in the $\lambda$-term. The set might contain more than one redex, but its redexes must be independent in the sense that the corresponding sub-terms do not overlap. An example of a parallel reduction strategy is to return all innermost redexes, i.e. redexes in which the corresponding sub-terms do not have any smaller redexes. For a $\lambda$-term, define the result of *applying one step* of a reduction strategy to be the term that is obtained by reducing, in parallel, every redex that is returned by the reduction strategy. We will be interested in applying a constant number of steps in order to achieve normal form.

To achieve normal form in a constant number of steps, we will assume certain resource bounds on the $\lambda$-term. Let us begin by naming these resources:

- *Types.* Define the types used by a $\lambda$-term to be the types that appear in the $\lambda$-term and its sub-terms.
- *Variables.* Define the variables used by a $\lambda$-term to be all variables – free or bound – that appear in it.
- *Height.* Define the height of a $\lambda$-term to be the maximal length of a root-to-leaf path in its syntax tree.

Note that $\lambda$-terms of bounded height can have unbounded size; this is thanks to the list constructor which allows the syntax tree to have unbounded branching. If we fix a finite set of allowed variables and types, then a $\lambda$-term can be represented as a string over a finite alphabet, since there are finitely many atomic terms and term constructors. When talking about for-transducers which apply reduction strategies, we assume that the input and output are given in such a string representation.

We now state the main result about computing normal forms using for-transducers.

Lemma 4.2. *There is a reduction strategy, such that for every resource bounds*

$$
\underbrace{X}_{\text{set of variables}} \qquad \underbrace{\Theta}_{\text{set of types}} \qquad \underbrace{k}_{\text{bound on height}}
$$

*the following conditions hold:*

(1) *There is a for-transducer which inputs a $\lambda$-term subject to the resource bounds, and outputs the result of applying one step of the reduction strategy. Both the input and output are represented as strings.*

(2) *There is a bound $\ell \in \{0, 1, \ldots\}$ such that for every $\lambda$-term subject to the resource bounds, applying the reduction strategy $\ell$ times leads to a $\lambda$-term in normal form.*

**Proof** (Sketch)
The first item holds for every reduction strategy that can be defined by a for-transducer in the following sense: there is a for-transducer which inputs a $\lambda$-term and an infix (represented by its first and last positions), and answers if this infix is one of the redexes selected by the reduction strategy. If the reduction strategy is definable in this sense, then a for-transducer implementing one step can be easily written, by replacing each selected redex with its reduction using a straightforward analysis of the reduction rules in Figure 2.

For the second item, we need a reduction strategy that is defined by a for-transducer (as required in the first item), and which normalizes in a constant number of steps. This is done by taking any reasonable reduction strategy, say from a proof of weak normalization in the simply typed $\lambda$-calculus, and then following what happens when lists of terms are reduced in parallel. See [5] for details. □

By the above lemma, and by closure of for-transducers under composing functions, it follows that for every resource bounds as in the lemma, there is a for-transducer which inputs a $\lambda$-term subject to the resource bounds, and outputs its normal form. We use this transducer to compute the function

$$w \qquad \mapsto \qquad \text{normal form of } Mw,$$

where $M$ is a fixed $\lambda$-term defining a string-to-string function, and $w$ is the input string, thus proving that for-transducers can simulate functions computed by $\lambda$-terms. □

**Remark:** Converting a $\lambda$-term into normal form can incur a non-elementary blowup. However, a corollary of the proof of the above theorem is that if a $\lambda$-term $M$ of string-to-string type is fixed, and we apply it to some string $w$ (viewed as $\lambda$-term in normal form), then the size of the output string is polynomial in $w$. The degree of this polynomial and its coefficients do depend on $M$, and this dependence is non-elementary, but they are constant once $M$ is fixed.

# 5 A LOGICAL MODEL

We now present the last model for polyregular functions, which is based on monadic second-order logic MSO. This models extends to polyregular functions the correspondence of regular languages with MSO that was originally established by Büchi, Elgot and Trakhtenbrot, see [35] for an introduction to this topic. The starting point for this model is two kinds of logically defined transformations that input and output structures such as strings, trees or graphs:

**First-order interpretations.** [22, Section 5.3] Elements of the output structure are $k$-tuples of elements in the input structure, of some fixed dimension $k \in \{1, 2, \ldots\}$, and the relations of the output structure are defined using first-order logic in the input structure.

**MSO transductions.** [13, Section 7.1.1] Like first-order interpretations, except that instead of first-order logic we are allowed to use monadic second-order logic MSO, at the price of restricting the dimension to $k = 1$.

The logical model for polyregular functions is the common generalization of both kinds of transformations above, which we call *MSO interpretations*: we allow both MSO and arbitrary dimension. At first glance, this may seem to be an unusual model. Almost by design, first-order interpretations are closed under composition [22, p. 218], and the same is true for MSO transductions [13, Theorem 7.14]. In both cases, the formulas in the composed function are obtained by substituting formulas in other formulas. In the case of MSO transductions, it is important that elements of the output model are single elements of the input model, i.e. the dimension is $k = 1$. The reason is that if we allowed dimension $k > 1$, then the natural way of composing two MSO transformations would require quantifying over sets of $k$-tuples. For this reason, MSO interpretations of dimension $k > 1$ are mainly ignored in the literature; see Example 10 below for an illustration of what can go wrong. As it turns out, however, MSO interpretations *do* make sense for string-to-string transformations, and closure under composition does hold; although the construction is much more complicated than a simple substitution of formulas.

## 5.1 Formal definition of MSO interpretations.

We begin with a formal definition of the MSO interpretations. We assume that the reader is familiar with the basic notions of model theory. We use the following terminology. A *vocabulary* is defined to be a set of relation names, each one with an associated arity in $\{1, 2, \ldots\}$. We do not use functions or constants; although this is not essential. A *structure* over a given vocabulary consists of a set, called the *universe* of the structure, together with an interpretation which maps each relation name in the vocabulary to a relation on the universe of matching arity. To define properties of structures over a fixed vocabulary, we use monadic second-order logic MSO, see [35, Section 2.3], and we assume that the reader is familiar with this logic.

*Definition of MSO interpretations.* An *MSO interpretation* consists of the following ingredients:

(1) Two vocabularies, called the *input vocabulary* and *output vocabulary*.

(2) A finite set $I$ of *components*. Each component $i \in I$ has an associated dimension $k_i \in \{0, 1, \ldots\}$, and an associated MSO formula which is called the *universe formula of component i*. This formula is over the input vocabulary, and it has $k_i$ free variables of element type and no free variables of set type. If the input structure has universe $A$, then the universe of the output structure is the disjoint union

$$\coprod_{i \in I} \{\bar{a} \in A^{k_i} \mid \bar{a} \text{ satisfies the universe formula of } i\}.$$

We use the following notation: an element of the above disjoint union is denoted as $i(\bar{a})$, where $i$ is a component and $\bar{a}$ is the tuple of input elements of suitable dimension.

(3) For each relation name $R$ in the output vocabulary, say of arity $n$, and every components $i_1, \ldots, i_n \in I$, there is a *relation formula*, which is an MSO formula over the input vocabulary whose number of free variables (as in the universe formula, all free variables have element type) is the sum of dimensions of the components $i_1, \ldots, i_n$. These formulas are used to define the relation $R$ in the output structure: the output structure satisfies

$$R(i_1(\bar{a}_1), \ldots, i_n(\bar{a}_n))$$

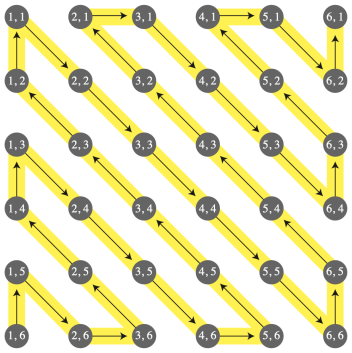
if and only if the corresponding relation formula selects the tuple $\bar{a}_1 \cdots \bar{a}_n$ in the input structure.

A simplified version of the above definition would only allow one component; having several components is a feature of minor importance which is useful in edge cases such as input structures with zero or one element in the universe. First-order interpretations are the special case of the above definition when there is a single component and all formulas use first-order logic. MSO transductions are the special case when all components have dimension one. In the above definition, we allow components of dimension zero; such components can be used to create output elements even if the input structure has an empty universe, which is allowed.

*The string-to-string case.* In the definition above, we can apply an MSO interpretation to any kind of logical structure, e.g. a graph or tree. However, we will only be interested in the case where the structures represent strings. We use the standard approach of modelling strings as structures, as in Example 1 from the introduction; see also [35, Section 2.1] for more context. In this approach, a string is viewed as a logical structure, which has unary relations for the labels and a binary relation for the linear order on positions. An alternative would be to eliminate the order, and have a binary successor relation. The difference between these two alternatives is unimportant when defining languages in MSO, because order and successor can be defined in terms of each other using MSO. However, the difference becomes crucial when defining functions with string outputs, as explained in the following example. The rough idea is that it is harder to define a linear order on tuples than it is to define a successor relation. The harder variant turns out to be the right choice.

**Example 10.** We already saw one example of a string-to-string MSO interpretation in Example 1, using the representation of strings that has a linear order. In this example, we explain how problems arise if we use the logical representation of strings where successor is used instead of order. The issue is that in the *input string*, we can define the order from the successor and vice versa, but this is no longer true in the *output string*. As we will see below, there is a way of ordering pairs of positions such that the corresponding successor relation is definable in MSO, but the order itself is not. If we use this ordering as the output string, then the output string will be definable in terms of successor, but not in terms of order.

Define the *diagonal ordering* on $\{1, \ldots, n\}^2$ to be the total ordering that is described in the following picture for $n = 6$




The successor relation of the diagonal ordering can be defined by an MSO formula

$$\varphi(x_1, y_1, x_2, y_2).$$

This is because the successor of $(x_1, y_1)$ is obtained by adding some offset in $\{-1, 0, 1\}^2$, such that the offset depends only on the parities of $x_1$ and $y_1$ and whether or not they are the first/last positions. On the other hand, the diagonal ordering itself cannot be defined in MSO. Indeed, one can show that if we could define the diagonal ordering in MSO, then we could also define the relation

$$y_1 - x_1 = y_2 - x_2,$$

which cannot be done in MSO (because it would allow to define an addition operation on positions). The opposite implication is always true: if we can define an ordering on tuples in MSO, then we can also define in MSO the corresponding successor relation.

We have just shown that there are strictly more string-to-string functions that can be defined by MSO interpretations,

if we decide to model strings as structures with successor instead of order. But maybe this is good? It is not: the larger class is not closed under composition, even with string-to-Boolean functions, which can be proved using the diagonal ordering. Also, many decision problems about the larger class are undecidable, such as the problem described in Corollary 1.5. □

Motivated by Example 10, when talking about string-to-string functions that are MSO interpretations, we use the representation of strings as total orders, i.e. a string is represented as a (finite) total order with unary predicates for the labels.

## 5.2 Equivalence with polyregular functions

We now state the main result about MSO interpretations, which is that in the string-to-string case, they describe exactly the polyregular functions. In particular, they are closed under composition, because polyregular functions are closed under composition. There is no known direct proof of closure under composition which does not pass through equivalence with other models.

THEOREM 5.1. *[9, Theorem 7] The string-to-string functions defined by MSO interpretations (with strings viewed as ordered structures) are exactly the polyregular ones.*

The translation from MSO interpretations to for-transducers is highly non-trivial, and we do not attempt a proof sketch here. We only discuss the easy direction, from for-transducers to MSO interpretations.

**Proof** (Easy direction only)
We give two proofs. Each proof has an advantage: the first proof inputs a pebble transducer and outputs an MSO interpretation of matching dimension, and the second proof inputs a for-transducer and outputs an MSO interpretation with a special property, namely that the order on tuples is lexicographic.

*First proof.* In the first proof, we assume that the polyregular function is given as a $k$-pebble transducer.

LEMMA 5.2. *For every states $p, q$ of the pebble transducer and every stack heights $\ell, m \in \{0, 1, \ldots, k\}$ there is an MSO formula*

$$\varphi(x_1, \ldots, x_\ell, y_1, \ldots, y_m)$$

*which holds of and only if the pebble transducer has a run that begins in the configuration with state $p$ with pebble stack $x_1, \ldots, x_\ell$, and ends in the configuration with state $q$ with pebble stack $y_1, \ldots, y_m$.*

**Proof**
Thanks to the reachability oracle from Lemma 2.2, and the equivalence of for-transducers with regular languages from Theorem 1.1, we have such a formula for the special case which describes $\ell$-runs, i.e. runs where the source and target configurations have the same height $\ell = m$, and pebble $\ell$ is never popped. Every other run can be decomposed into a constant number of such runs:

$$\rho = \underbrace{\rho_1}_{\ell_1\text{-run}} \cdots \underbrace{\rho_n}_{\ell_n\text{-run}} \qquad \text{with } n \leq 2k$$

by looking at the first/last configurations which reach a given stack height. The configurations that connect these runs can be quantified by the formula. □

Using the above lemma, we define an MSO interpretation as follows. Its components are pairs of the form (state, stack height), with the dimension of a component being its stack height. The idea is that an element from component $(q, \ell)$ describes a configuration with state $q$ and stack height $\ell$. The universe formulas select configurations that are reachable from the initial configuration and produce an output letter, while the order formula tells us which configuration appears first in the computation; all of this can be done in MSO thanks to Lemma 5.2. This completes the first proof. A corollary of this proof is that if a string-to-string function is computed by a $k$-pebble transducer, then it is an MSO interpretation of dimension $k$.

*Second proof.* In the second proof, we assume that the polyregular function is given as a for-transducer. In this case, the construction will yield an MSO interpretation with a special property of independent interest, namely that the order on the tuples is going to be lexicographic. We will use a variant of for-transducers that mixes for loops and MSO formulas, as expressed in the following lemma.

LEMMA 5.3. *Every for-transducer is equivalent to one of the form*

```
for x_1 in τ_1
  for x_2 in τ_2
    ...
      for x_k in τ_k
        body
epilogue
```

*where the body is a sequence of instructions of the form*

$$\texttt{if } \underbrace{\varphi(x_1, \ldots, x_k)}_{\text{MSO formula with } k \text{ free variables}} \texttt{ then output } a$$

*and the epilogue is a sequence of instructions of the form*

$$\texttt{if } \underbrace{\varphi}_{\text{MSO formula with no free variables}} \texttt{ then output } a.$$

*Furthermore, the body produces at most one letter in each loop iteration, i.e. the formulas $\varphi(x_1, \ldots, x_k)$ that appear in the body are pairwise contradictory.*

**Proof**
We first convert the for-transducer into prenex normal form using Lemma 1.2. Next, we observe that the Boolean variables can be eliminated, since their values can be recovered using MSO formulas, as explained in in Lemma 5.2. □

A for-transducer as in the conclusion of the above lemma can easily be converted into an MSO interpretation, which has one component of dimension $k$, corresponding to the body, and several components of dimension 0, corresponding to the epilogue. The resulting MSO interpretation has the following *lexicographic property*: it has one component of nonzero dimension, and on this component the order is lexicographic, with some coordinates ordered first-to-last and some coordinates ordered last-to-first. □

This completes the first part of the paper, which discusses five equivalent models for defining polyregular functions, namely:

(1) for-transducers
(2) pebble transducers
(3) compositions of prime functions
(4) polyregular $\lambda$-terms
(5) MSO interpretations.

In the remaining part of the paper, we discuss further results about polyregular functions, mainly concerning fragments that are obtained by restricting the above models in some way, e.g. by requiring at most quadratic growth.

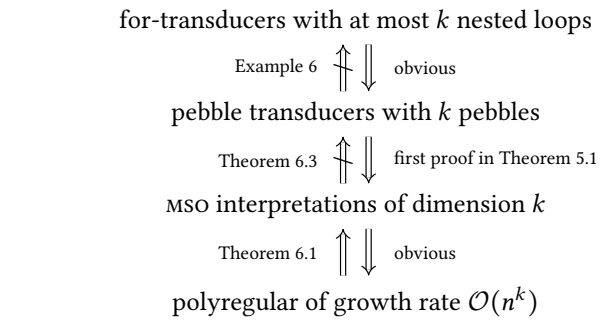
# 6 GROWTH RATES

In this section, we discuss the growth rate of polyregular functions. For a string-to-string function, define its *growth rate* to be the non-decreasing function, which maps a number $n \in \{0, 1, \ldots\}$ to the maximal length of an output string that can arise by applying the function to an input string of length at most $n$. For example, the squaring function from Example 3 has growth rate $n^2 + n + 1$, while the map reverse function from Example 6 has growth rate $n$.

As we have mentioned several times before, polyregular functions have polynomial growth rates. Also, for some of the models defining polyregular functions, there is a natural parameter that bounds the degree of the polynomial in the growth rate, as explained in the following table:

| model | parameter |
|---|---|
| for-transducer | nesting depth of for loops |
| pebble transducer | number of pebbles |
| MSO interpretation | dimension |

The purpose of this section is to inspect the role of the parameters in the above table. For the two descriptions of polyregular functions that do not appear in the above table, namely prime functions and the $\lambda$-terms, the growth rate is less visible in the syntax, and we do not discuss those models in this section. The results of this section are summarized in the following diagram, with arrows representing inclusions of classes of string-to-string functions:




Only two of the arrows in the above diagram have not been shown yet, namely the second and third upward arrows. These are discussed in the rest of this section, which is based on the unpublished manuscript [6].

## 6.1 Growth rates and MSO dimension

Theorem 6.1 below concerns the third upward arrow in the diagram: it shows that the dimension of an MSO interpretation corresponds exactly to the degree of the polynomial in its growth rate. The theorem also rules out intermediate growth rates such as $\sqrt{n}$ or $n \cdot \log n$.

THEOREM 6.1. *If $k \in \{0, 1, \ldots\}$ is the minimal dimension of an MSO interpretation which defines a string-to-string function, then the function has growth rate $\Theta(n^k)$, i.e. the growth rate is both upper- and lower-bounded by a polynomial of degree $k$. Furthermore, $k$ can be computed given an MSO interpretation of possibly sub-optimal dimension $\ell \geq k$.*

Before discussing the proof of the theorem, let us examine its consequences in the important special case of linear growth.

**Example 11.** Consider the special case of Theorem 6.1 where $k = 1$. In this case we have the following equalities (the equalities mention a model, called streaming string transducers, which will be described in Section 7.2.1, together with a proof of the last equality):

polyregular functions of linear growth

$\Big\|$ Theorem 6.1 for $k = 1$

MSO interpretations of dimension $k = 1$
which are the same as MSO transductions

$\Big\|$ [20, Theorem 13]

deterministic two-way automata with output
which are the same as one pebble transducers

$\Big\|$ [1, Theorems 1, 2, 3]

streaming string transducers [1, Section 3]

The class of string-to-string functions that is described by the last three models in the diagram above is called the *regular string-to-string functions*. The diagram says that the regular functions are exactly the polyregular functions of linear growth. As mentioned in the introduction, in the context of polyregular functions, it would probably be more appropriate to use the name *linear regular string-to-string functions* for the functions in the above diagram. Since Theorem 6.1 allows us to compute $k$, it also follows that one can decide if a polyregular function is in fact linear regular. □

The proof of Theorem 6.1 is based on the following lemma, proved in [6, Lemma 2.4], which examines the growth rate of MSO formulas that define properties of strings with distinguished positions.

LEMMA 6.2 (SEED LEMMA). *Let $\varphi(x_1, \ldots, x_\ell)$ be an MSO formula, which defines a property of strings (over some fixed input alphabet) with $\ell$ distinguished positions. One can compute logically equivalent formula of the form*

$$\coprod_{i \in I} \varphi_i(x_1, \ldots, x_\ell),$$

*where $\coprod$ refers to disjoint union (for every input string, every $\ell$-tuple of positions is selected by at most one $\varphi_i$), and for each $i \in I$ there is a subset of variables $X_i \subseteq \{x_1, \ldots, x_\ell\}$ with the following properties:*

1. *If two $k$-tuples in the same input string agree on variables from $X_i$, then $\varphi_i$ can select at most one of them*[4].
2. *The growth rate of $\varphi_i$ is $\Theta(n^{|X_i|})$, where the growth rate is defined to be the maximal number of tuples that can be selected in a single input string of length at most $n$.*

We do not prove the Seed Lemma. Its proof is based on an analysis of pumping patterns. Once we have the Seed Lemma, Theorem 6.1 can easily be deduced, as explained below.

**Proof** (of Theorem 6.1)
Consider a polyregular function, given as an MSO interpretation. Without loss of generality we assume that the MSO interpretation has only one component of nonzero dimension $\ell$, see the comments at the end of the proof in Theorem 5.1. To simplify notation, we assume that there is simply one component, the proof can easily be extended to handle the components of dimension zero. And apply the Seed Lemma to the universe formula $\varphi(x_1, \ldots, x_\ell)$ of the unique component, yielding an equivalent disjunction:

$$\coprod_{i \in I} \varphi_i(x_1, \ldots, x_\ell),$$

along with subsets of variables $\{X_i\}_i$. Consider an input string $w$. The universe of the output string under the MSO interpretation is equal to the disjoint union

$$\coprod_{i \in I} \{\bar{a} \in \text{positions}^\ell \mid w \models \varphi_i(\bar{a})\}.$$

By condition 1 in the Seed Lemma, the above set is in bijective correspondence with the set

$$\coprod_{i \in I} \{\bar{a} \in \text{positions}^{|X_i|} \mid w \models \varphi_i'(\bar{a})\},$$

where $\varphi_i'$ is obtained from $\varphi_i$ by quantifying existentially the variables that are not in $X_i$. Furthermore, this bijective correspondence is definable in MSO, since an MSO formula can extend a tuple of dimension $|X_i|$ to the unique tuple of dimension $k$ that satisfies $\varphi_i$. Based on this observation, we can define a new MSO interpretation, which has a component for each $i \in I$, of dimension $|X_i|$ and universe formula $\varphi_i'$. The remaining structure of the output string, namely the order on positions and their labels, is defines using the bijective correspondence discussed bove and the MSO formulas from the old interpretation.

By item 2 in the Seed Lemma, the growth rate of the new MSO interpretation is $\Theta(n^k)$ where $k$ is the maximal size of sets $\{X_i\}_i$, and the dimension of the new interpretation is also the same $k$. □

## 6.2 Growth rate and pebbles

In Theorem 6.1, we have established that MSO interpretations of dimension $k$ define exactly the polyregular functions of growth $O(n^k)$. Is there a similar correspondence in the case of $k$-pebble transducers, or in the case of for-transducers with $k$ nested loops? We have already seen in Example 6 a negative answer to the second question: there are functions of linear growth that need at least two nested loops. In this section we present a negative answer to the first question: the number of pebbles in a pebble transducer does not match the growth rate[5]. As we have seen in Example 11, the growth rate does match the number of pebbles for $k = 1$, i.e. pebble transducers with one pebble are the same as the polyregular

[4] This is the same as being a key in database terminology.

[5] In [26], it was claimed that the growth rate matches the number of pebbles. Unfortunately, that proof contains an error.

functions of linear growth[6]. However, for $k = 2$ there is no longer a match, as stated in the following theorem, which uses a function that was suggested by Gäetan Douénau-Tabot.

Theorem 6.3. *The following function, call it* inner squaring, *is polyregular and has quadratic growth, but it is not recognized by any pebble transducer with at most two pebbles:*

$$w_1|\cdots|w_n \quad \mapsto \quad (w_1)^n|\cdots|(w_n)^n \qquad w_1,\ldots,w_n \in \{a,b\}^*.$$

The counterexample from the above theorem can even be strengthened: for every $k \in \{2,3,\ldots\}$ there is a quadratic polyregular function which needs at least $k$ pebbles [6, Theorem 3.1]. In other words, no fixed number of pebbles is enough to cover all quadratic polyregular functions.

**Proof** (Sketch)
The function clearly has quadratic growth. It is also recognized by a pebble transducer with three pebbles, even a for-transducer with nesting depth three, which looks is explained in the following pseudo-code:

```
input: w_1|···|w_n
for x in {1,...,n}
  for y in {1,...,n}
    for z in positions of w_x
      output z
```

It remains to show that three pebbles are indeed necessary.

We begin with a simpler variant of the lower bound, which uses an infinite alphabet, and explains the main idea. Consider an infinite alphabet $\mathbb{A}$, whose letters will be called *atoms*, and the following function of type $\mathbb{A}^* \to \mathbb{A}^*$

$$\underbrace{a_1 \cdots a_n \quad \mapsto \quad (a_1)^n \cdots (a_n)^n.}_{\text{call this function inner squaring with atoms}}$$

In other words, each input letter from the infinite alphabet is copied as many times as the length of the string. This function behaves similarly to the function from the theorem, except that the atoms play the role of blocks in $\{a,b\}^*$. We will show that the inner squaring with atoms cannot be computed by the variant of pebble transducers, which is obtained from the original variant by adding a new action "output the letter that is under the head". We use the name *pebble transducer with atoms* for this variant. In principle, the transducer could use tests about specific atoms that are hard-coded into its syntax, and actions that output such atoms, but these features will be useless for inputs where the hard-coded atoms do not appear. The model is not allowed to do equality tests on atoms, and it does not have any registers.

Inner squaring with atoms can be recognized using three pebbles in the atom model, using the same idea as explained at the beginning of this proof. The advantage of the atom model is that the lower bound is easy to prove.

Lemma 6.4. *Inner squaring with atoms cannot be computed by a pebble transducer with atoms that has at most two pebbles.*

**Proof**
Fix a pebble transducer with atoms that has at most two pebbles. We will show that it cannot compute inner squaring with atoms. Consider an input string $a_1 \cdots a_n$ which consists of distinct atoms. In the proof below, we say that a quantity is *bounded* if it has an upper bound that depends only on the fixed transducer and not on the input string.

Consider a configuration $c$ of the pebble transducer with atoms that has height one, i.e. there is only one pebble in the stack. A *child* of this configuration is defined to be any configuration of height two that appears between $c$ and the next configuration of height one. Output atoms can only be produced when the head points to them, and the head cannot visit a position twice in the same state without entering an infinite loop. Since we assume that no atom appears twice in the input, head positions and input atoms are in one-to-one correspondence, and therefore it follows that for every atom, the number of children of $c$ that output this atom is bounded by some constant $N$ that depends only on the transducer. Since the output string consists of each atom repeated $n$ times, it follows that if $n > N$, then for every configuration $c$ of height one, at most two different atoms are output in its children, and each of these atoms is output at most $N$ times. Summing up, each configuration of height one produces only bounded size output in its children. Since the number of configurations of height one is linear in $n$, it follows that the total output size must also be linear in $n$; but this cannot be since the output size is $n^2$. □

This completes the lower bound of three pebbles in the case of an infinite alphabet. To prove the theorem in its original form, one uses a non-trivial reduction from the finite alphabet to the infinite alphabet [6, Theorem 3.14]. We do not present this reduction here; it examines all possible ways in which a pebble transducer could "cheat" when atoms are coded as strings in $\{a,b\}^*$, and it shows that each of these ways of cheating must necessarily fail. □

# 7 SUBCLASSES OF POLYREGULAR FUNCTIONS

In this section we discuss subclasses of the polyregular functions.

## 7.1 First-order polyregular functions

In the case of MSO interpretations, there is a natural restriction, namely *first-order interpretations*, which uses first-order

[6] As pointed out by Lê Thành Dung Nguyên, this result, i.e. the special case of Theorem 6.1 for $k = 1$, can be deduced from [19, Theorem 43].

logic instead of MSO. It turns out that the first-order restriction can be imposed on the other models as well, as described in the following theorem.

THEOREM 7.1. *The following models describe the same string-to-string functions:*

1. *for-transducers in the normal form from Lemma 5.3, where all formulas use first-order logic instead of MSO;*
2. *pebble transducers such that all reachability queries described in Lemma 5.2 can be defined in first-order logic;*
3. *compositions of polyregular primes which do not use reversible Mealy machines that have more than one state;*
4. *polyregular $\lambda$-terms which do not use the atomic functions for group multiplication;*
5. *first-order interpretations.*

**Proof** (Sketch)
The hard implication is to go from a first-order interpretation to any of the remaining models. In fact, this hard implication is the core of the proof in [9], where the MSO case is deduced from the first-order case. We do not describe this hard implication, only the remaining implications.

- $2 \Rightarrow 1$ Same proof as in Theorem 2.1, with the reachability oracles taken from the assumption on the pebble transducer having reachability definable in first-order logic.
- $1 \Rightarrow 3$ Same proof as in Theorem 3.2. If the input function is first-order definable, then reversible Mealy machines are not needed in the Krohn-Rhodes decomposition, see Footnote 3.
- $3 \Rightarrow 4$ Same proof as in Theorem 4.1. In that proof, the translation from prime functions to $\lambda$-terms needs the group functions only for reversible Mealy machines.
- $4 \Rightarrow 5$ Same proof as in Theorem 4.1. In that proof, the translation from $\lambda$-terms to for-transducers would also work with first-order interpretations as the target language, assuming that the $\lambda$-term does not use group multiplication.

The hard implication that was discussed at the beginning of this proof is not needed to close the loop for proving equivalence of conditions $\{2, 3, 4, 5\}$. One can, for example, prove the implication $4 \Rightarrow 2$ in the same way as the implication $4 \Rightarrow 5$. □

We use the name *first-order polyregular* for the class of string-to-string functions that is described in the above theorem. The following result shows that, up to preprocessing by a Mealy machine, the first-order polyregular functions are equivalent to the general polyregular functions.

THEOREM 7.2. *Every polyregular function can be decomposed as a Mealy machine followed by a first-order polyregular function.*

**Proof**
We give two proofs of this theorem.

*First proof.* The first proof uses interpretations. It is based on the following reduction of MSO to first-order logic [12, Theorem 2]: for every MSO formula $\varphi(x_1, \ldots, x_n)$ there is a Mealy machine $f$ and a first-order formula $\psi(x_1, \ldots, x_n)$ such that every string with $k$ distinguished positions satisfies

$$w \models \varphi(x_1, \ldots, x_n) \quad \text{iff} \quad f(w) \models \psi(x_1, \ldots, x_n).$$

The alphabets used in the formulas $\varphi$ and $\psi$ are not necessarily the same; the translation between these two alphabets is handled by the Mealy machine. Applying this result to all MSO formulas that are used in an MSO interpretation, and combining the Mealy machines into a single one, we see that every MSO interpretation can be decomposed as a Mealy machine followed by a first-order interpretation.

*Second proof.* The second proof uses prime decompositions, and it proves a slight strengthening of the theorem, where the Mealy machine in the conclusion is reversible. This strengthening can be written as follows

$$\underbrace{\text{polyregular}}_{\substack{\text{all polyregular}\\ \text{functions}}} = \underbrace{\text{reversible}}_{\substack{\text{reversible}\\ \text{Mealy}\\ \text{machines}}}; \underbrace{\text{fo-polyregular}}_{\substack{\text{polyregular functions}\\ \text{that are first-order}\\ \text{definable}}}. \qquad (1)$$

The non-trivial inclusion in the above equality is left-to-right. To prove it, we use the prime decomposition result from Theorem 3.2. This result can be stated as

$$\text{polyregular} = \underbrace{\text{prime}^*}_{\text{compositions of prime polyregular functions}}. \qquad (1)$$

In light of the above, to prove the left-to-right inclusion in (1), it is enough to prove the inclusion

$$\text{prime}; \text{reversible} \subseteq \text{reversible}; \text{fo-polyregular}. \qquad (2)$$

Indeed, by repeatedly applying the above inclusion, we get

$$\text{prime}^*; \text{reversible} \subseteq \text{reversible}; \text{fo-polyregular}^*.$$

which proves the left-to-right inclusion in (1).

It remains to prove (2). This is shown by inspecting all possible prime machines. In each case, the idea is to first label the input string by state transformations of the reversible machine; this can be done in a reversible way and it gives enough information to recover the output string. The details are left to the reader. □

## 7.2 Layered streaming string transducers

The second subclass of polyregular functions that we discuss here is a generalization of the streaming string transducer model of Alur and Černý [1]. This is a model that processes its input in a single first-to-last pass, and it is is equivalent

to the regular functions of linear growth that were discussed in Example 11. There is no known streaming model that computes all polyregular functions, but there is one that computes some of them. The purpose of this section is to describe that model.

*7.2.1 Streaming string transducers.* We begin with the original model of streaming string transducers, which describes functions of linear growth, and then explain how it can be generalized to get polynomial growth.

The idea behind a streaming string transducer is that it processes the input only once, but it uses registers to store parts of the output string. There is an important restriction on the way that the registers are updated: a register cannot be copied. If copying would be allowed (the model with copying is called *copyful streaming string transducers*), then one could produce exponential outputs by duplicating a register in each step.

**Example 12.** Consider the *map reverse* function from Example 6. We can implement this function using two registers `X` and `Y`. Register `X` stores the reverse of the currently processed block, while `Y` stores the output for all previous blocks. Initially, both registers are empty. The transducer processes the input in a single first-to-last pass. When it reads an input letter $a$, it prepends $a$ to register `X` and leaves `Y` unchanged, as explained in the following register update

```
X := aX
Y := Y
```

A similar update is executed when reading input $b$. This way the reverse of the present block is stored in `X`, while `Y` is unchanged. Once a block ends, i.e. the separator symbol | is read, then register `X` is erased and its value is appended to `Y`, as in the following register update

```
X := ε
Y := YX
```

Once the transducer finishes reading the input string, the output is recovered as `XY`. □

Here is a more formal definition of the model of streaming string transducers [1, Section 3]. Suppose that $\Gamma$ is an output alphabet and $R = \{X_1, \ldots, X_n\}$ is a set of *register names*. A *register update* is defined to be simultaneous assignment

$$\begin{aligned} X_1 &:= w_1 \\ &\cdots \\ X_n &:= w_n \end{aligned}$$

where each right hand side $w_i$ is a possibly empty string whose letters are either register names or output letters. We require the register update to be *copyless* in the following sense: each register name appears at most once in the string $w_1 \cdots w_n$ that is obtained by concatenating all right hand sides in the register update. The syntax of a streaming string transducer consists of the following ingredients

$$\underbrace{Q}_{\text{states}} \quad \underbrace{q_0 \in Q}_{\substack{\text{initial}\\ \text{state}}} \quad \underbrace{\Sigma}_{\substack{\text{input}\\ \text{alphabet}}} \quad \underbrace{\Gamma}_{\substack{\text{output}\\ \text{alphabet}}} \quad \underbrace{R,}_{\text{registers}} \quad \underbrace{r_0 \in Q}_{\substack{\text{final}\\ \text{register}}}$$

together with a transition function

$$\delta : Q \times (\Sigma + \{\dashv\}) \to Q \times (\text{copyless register updates}).$$

The semantics of the transducer is a string-to-string function which is defined as follows. Initially, every register stores the empty string. The transition function says how the state and register contents are updated upon reading each input letter, or the end-of-string marker $\dashv$. The transducer reads the input string (extended with an end-marker $\dashv$) in a single first-to-last pass (hence the name "streaming"). The output string is then defined to be the contents of the designated final register at the end of this pass. The purpose of the copyless restriction is to ensure that the output string has linear size: in each step the combined length of the strings stored in the registers can grow only by a constant.

The main result about streaming string transducers is that they are expressively complete for the regular functions of linear growth.

Theorem 7.3. *[1, Theorems 1, 2, 3] Streaming string transducers recognize the same string-to-string functions as 1-pebble transducers, i.e. two-way automata with output.*

**Proof**
Although this result is classical, we present below a self-contained proof, which also leverages the power of the Krohn-Rhodes theorem.

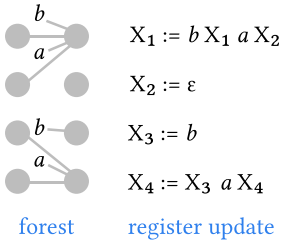
*From streaming to two-way.* A function recognized by a streaming string string transducer can be decomposed as

$$\begin{array}{l} \Sigma^* \\ \quad\Big\downarrow \text{append end-marker} \\ \Sigma^*\dashv \xrightarrow[\text{compute register updates}]{} \Delta^* \xrightarrow[\text{evaluate register updates}]{} \Gamma^*, \end{array}$$

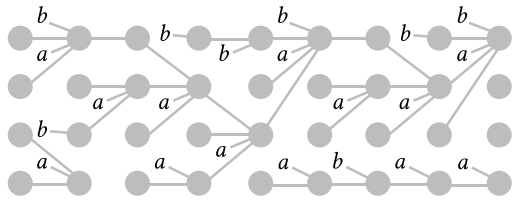
where $\Delta$ is the finite set of register updates that is used in the syntax of the transducer, the first horizontal arrow maps each input position to the corresponding register update, while the second horizontal arrow evaluates a sequence of register updates and returns the contents of the designated register at the end. The first horizontal arrow is a Mealy machine, since to compute the register updates one only needs to keep track of the state of the transducer. Since two-way automata with output are closed under pre-composition with Mealy machines, see Example 7, and also under adding an end-marker, it remains to show that the second horizontal arrow

can be computed by a two-way automaton with output. This is argued below.

A single register update can be seen as a forest of height one, as explained in the following picture:

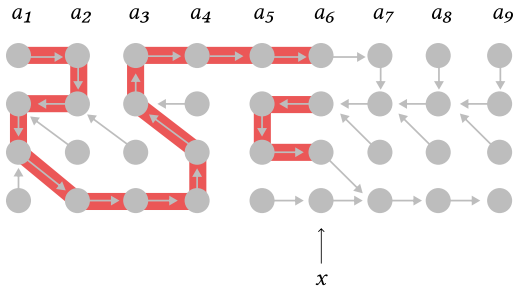



The picture is indeed a forest, thanks to the copyless restriction; without this restriction the picture would a bi-partite graph. A sequence of $n$ register updates can also be seen as a forest of height at most $n$, as in the following picture:



A circle in the picture represents the value of a register (rows) in an input position (columns). The output corresponding to a circle is computed by traversing the corresponding tree using depth-first search, which is something that can be implemented by a two-way automaton.

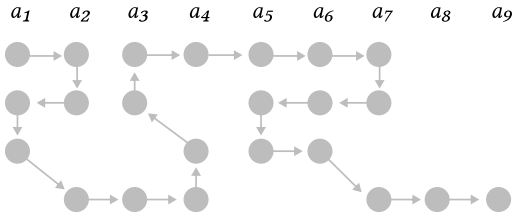
*From two-way to streaming.* Suppose that we want to simulate a two-way automaton with output using a streaming string transducer. The basic idea is to implement the Shepherdson profile from the proof of Lemma 2.2, while storing output strings in the registers. More precisely, after it has read the prefix of the input that ends in a position $x$, the streaming string transducer uses its registers to store the outputs of runs that: (a) stay in this prefix; and (b) begin in the initial configuration or a configuration with position $x$; and (c) end in the terminating configuration or a configuration with position $x$. Here is a picture:




The number of such runs is at most quadratic in the number of states, and thus fixed once the two-way automaton is fixed. Therefore the outputs of these runs can be stored in a fixed number of registers. Furthermore, the updates are compositional, i.e. if we know the outputs corresponding to position $x$, then we can apply a register update to get the outputs corresponding to the next position $x + 1$. The difficulty, however, is that this update is not copyless. The problem is that in the transition graph of the two-way automaton, there might be two configurations with the same successor, as in the following picture:

In such a situation, the registers corresponding to both configurations on the right will need to appeal to the same register on the left, thus violating the copyless restriction.

The solution to this problem is to restrict the configuration graph to the reachable configurations, as in the following picture




This will solve the problem of common successors, since a reachable configuration has at most one incoming transition; otherwise the transducer would loop. Therefore, the function computed by the two-way automaton can be represented as a composition of two functions: (a) compute the graph that represents the reachable configurations; followed by (b) compute the output string. As we have argued above, (b) can be computed by a copyless streaming string transducer using Shepherdson profiles. The string-to-string function in (a) can be computed, also using the Shepherdson construction, by a composition of two Mealy machines: (a1) a usual Mealy machine that runs first-to-last, followed by (a2) a Mealy machine that runs last-to-first. It remains to show that streaming string transducers are closed under pre-compositions with Mealy machines of both kinds, i.e. first-to-last and last-to-first. For first-to-last Mealy machines, there is nothing to do, since their states can be integrated into the state of the streaming string transducer. The remaining part is last-to-first:

Lemma 7.4. *Functions computed by streaming string transducers are closed under pre-composition with last-to-first Mealy machines.*

Another way of stating the lemma is that one can add regular lookahead to streaming string transducers without affecting the expressive power of the model. The lemma can

be proved directly, see e.g. [7, Section 13.2], or as a corollary of the Krohn-Rhodes Theorem: similarly to Example 7, it is enough to show pre-composition under (the last-to-first variants of) prime Mealy machines, which is a straightforward exercise. □

*7.2.2 Layered streaming string transducers.* Having defined streaming string transducers, we are now ready to explain how they copyless restriction can be relaxed to obtain polynomial growth instead of just linear growth.

How can we guarantee that the output has polynomial growth, say quadratic growth? One idea is to group the registers into two layers, linear and quadratic. The linear registers can only depend on linear ones, and the dependency must be copyless. The quadratic registers can depend on both linear and quadratic registers, and the dependency on quadratic (but not linear) registers must be copyless. This idea is illustrated in the following example.

**Example 13.** Suppose that we want to map a string to all of its prefixes, as in the following example:

$$abc \mapsto aababc.$$

A natural implementation of this function is a register transducer, which has a register `X` storing the current prefix, and a register `Y` storing the output. When reading an input letter $a$, the registers are updated as follows:

```
X := Xx
Y := YX
```

This register update is not copyless, since register `X` appears twice in the right hand sides. However, it satisfies a more relaxed layered condition, if we think of `X` as being linear and `Y` as being quadratic. □

The idea from the above example is formalized as follows. For $k \in \{1, 2, \ldots\}$, a $k$-layered streaming string transducer is defined in the same way as a streaming string transducer, except for the following differences. First, each register is assigned a layer in $\{1, \ldots, k\}$. Second, instead of being copyless, each register update must be *layered copyless* in the following sense: for every layer $\ell$, concatenating the right hand sides of all registers of layer $\ell$ gives a string where: (a) only output letters and registers from layers $\{1, \ldots, \ell\}$ can appear; and (b) each register of layer $\ell$ appears at most once. The point of this restriction is that if the input length is $n$, then a register on layer $\ell$ will store a string whose length is $O(n^\ell)$. In particular, the function computed by such a transducer will have polynomial growth. Furthermore, in the special case of $k = 1$, we recover the original notion of streaming string transducers.

The following theorem shows, in particular, that the layered streaming string transducers correspond to a special case of the polyregular functions.

Theorem 7.5. *[15, Corollary 18] Let $k \in \{1, 2 \ldots\}$. The following models describe the same string-to-string functions*

(1) *$k$-layered streaming string transducers;*
(2) *$k$-pebble transducers that satisfy have the following monotonicity property:*
   (*) *For every reachable configuration, say with a stack of height $\ell \in \{1, \ldots, k\}$, the pebbles are ordered in the input string in a non-decreasing way*
   $$\underbrace{x_\ell}_{\substack{\text{head, i.e. the}\\ \text{top of the stack}}} \leq \cdots \leq \underbrace{x_1}_{\substack{\text{bottom of}\\ \text{stack}}}.$$
(3) *copyful streaming string transducer of growth rate $O(n^k)$*

We do not present the proof here. The equivalence of the first two items uses an adaptation of the correspondence of two-way automata and streaming string transducers from Theorem 7.3. The second item in the theorem demonstrates that the class is a subclass of the polyregular functions. It is a proper subclass. For example, the class of functions from the above theorem does not cover the squaring function from Example 3, and even the following weaker function

$$a^n|w \quad \mapsto \quad w^n \qquad \text{where } w \in \{a, b\}^*,$$

see [15, Claim 37]. Also, layered streaming string transducers are not closed under composition (even if we allow more layers).

## 7.3 Comparison free polyregular functions

The last subclass of polyregular functions that we present is the *comparison-free polyregular functions* of Nguyên et al. This subclass is described in the following theorem, whose proof we do not summarize here.

Theorem 7.6. *[30, Theorem 6.1 and Corollary 5.7] The following models describe the same class of string-to-string functions:*

(1) *Pebble transducers which are not allowed to ask any question of the form*
   - *is the $i$-th pebble to the left of the $j$-th pebble?*
(2) *The least class of string-to-string functions which is closed under composition, contains all linear regular functions, and contains the following function for every input alphabet $\Sigma$:*
   $$\underbrace{a_1 \cdots a_n}_{w} \quad \mapsto \quad |a_1 w|a_2 w \cdots |a_n w$$
(3) *The least class of string-to-string functions which contains all linear regular functions, and has the following closure property: if $g : \Sigma^* \to \Delta^*$ is in the class, and*
   $$\{f_i : \Sigma^* \to \Gamma^*\}_{i \in \Delta}$$

*are all in the class, then so is*

$$w \mapsto f_{i_1}(w) \cdots f_{i_n}(w) \qquad \text{where } g(w) = i_1 \cdots i_n.$$

Furthermore, the class from the above theorem, call it the *comparison-free polyregular functions*, coincides with a certain $\lambda$-calculus with linear types [29, Theorem 7.2].

## 7.4 All sub-classes are incomparable

All of the classes discussed in this section are incomparable. Here are two functions, see [30, Theorems 8.1 and 8.3], that witness incomparability of the comparison-free class from Theorem 7.6 and the layered class from Theorem 7.5:

$$\underbrace{a^n \mapsto (a^n b)^{n+1}}_{\substack{\text{comparison-free} \\ \text{but not layered}}} \qquad \underbrace{a^n \mapsto ba^{n-1}ba^{n-2}\cdots bab}_{\substack{\text{layered but not} \\ \text{comparison-free}}}$$

Both subclasses are proper, since none of them contains the squaring function from Example 3. All of these examples are first-order definable, but if we add some parity conditions, e.g. inputs of even length are mapped to the empty string, then the new functions are no longer first-order definable. Putting these observations together, we see that there are no non-trivial relationships between polyregular functions and the three subclasses discussed in this section.

# 8 THE EQUIVALENCE PROBLEM

An important open problem for polyregular functions is decidability of the equivalence problem.

**Open problem.** Can one decide, given two polyregular functions, if they are equivalent in the sense that for every input string the two output strings are equal?

The decidable problem in Corollary 1.5 can be seen as a special case of the open problem, where only functions with Boolean outputs are considered. Therefore, the non-elementary hardness is inherited from the problem in Corollary 1.5 and this hardness holds regardless of the choice of representation, i.e. if the inputs could be given as for-transducers, pebble transducers, polyregular $\lambda$-terms, etc.

An interesting known decidable case for the equivalence problem is the string-to-number case, i.e. polyregular functions with a one letter output alphabet. (If the output alphabet has one letter, then an output string is the same thing as a natural number.)

Theorem 8.1. *[14, Corollary 19] The equivalence problem is decidable for string-to-number polyregular functions.*

**Proof**
In the proof, we use weighted automata over the semiring of natural numbers. Such an automaton consists of an input alphabet $\Sigma$, a set of states $Q$, and three weight functions of types

$$\underbrace{\delta : Q \times \Sigma \times Q \to \{0, 1, \ldots\}}_{\text{weights of transitions}} \qquad \underbrace{I, F : Q \to \{0, 1, \ldots\}}_{\text{initial and final weights}}.$$

A run is defined as usual. The *weight* of a run is defined to be the product of: the initial weight of the first state, the weights of all transitions (including repetitions), and the final weight of the last state. The semantics of a weighted automaton is the string-to-number function which maps an input string to the sum of weights of all possible runs over it.

The following lemma shows that polyregular string-to-number functions are exactly the weighted automata of polynomial growth. In the lemma, when we talk about the growth of a string-to-number function, we mean the magnitude of the output number, and not the number of bits used to represent it. In other words, we use the unary, and not binary, representation of the output number. □

Lemma 8.2. *A string-to-number function is polyregular if and only if it has polynomial growth and it is computed by a weighted automaton.*

**Proof** (of Lemma 8.2)
We begin with the right-to-left direction:

weighted automaton of polynomial growth
⇓ the models are essentially the same, see [7, Lemma 8.3]
copyful streaming string transducer with a one letter output alphabet and polynomial growth
⇓ Theorem 7.6
layered streaming string transducer
⇓ Theorem 7.5
polyregular function

□

Consider now the left-to-right direction in the the lemma. Assume that the polyregular function is given in the normal form Lemma 5.3, i.e. it has the shape

```
for x_1 in τ_1
  for x_2 in τ_2
    ...
      for x_k in τ_k
        body
epilogue
```

The epilogue is a string-to-number function, which outputs numbers of bounded size, since it has no loops. This function can easily be computed by a weighted automaton, since for each of the finitely many possible output numbers $m \in \{0, 1, \ldots\}$, the set of input strings where the epilogue outputs $m$ is a regular language.

Consider now the part of the for-transducer before the epilogue, the one which uses loops. For $\ell \in \{0, \ldots, k\}$, the innermost $k - \ell$ loops in the for-transducer can be viewed as for-program that that inputs a string with $\ell$ distinguished positions, representing the variables in the outermost loops, and which outputs a number. This for-program can be seen as a string-to-number function, call it $f_\ell$, assuming that the string with $\ell$ distinguished positions is represented as a string over $2^\ell$ disjoint copies of the input alphabet, see the comments before Lemma 1.3. By induction on $k-\ell$, we show that the function $f_\ell$ is computed by a weighted automaton. This proof is the same as for Lemma 1.3. In the induction basis of $\ell = k$, we use the same argument as for the epilogue, since the function has outputs of bounded size. In the induction step, we observe that the function $f_\ell$ is obtained from $f_{\ell+1}$ by applying the following claim.

Claim 8.3. *Consider two disjoint copies $\Sigma$ and $\underline{\Sigma}$ of a finite input alphabet. If a string-to-number function*

$$f : (\Sigma + \underline{\Sigma})^* \to \{0, 1, \ldots\}$$

*is computed by weighted automaton, then the same is true for the string-to-number function defined by*

$$a_1 \cdots a_n \in \Sigma^* \quad \mapsto \quad \sum_{i=1}^{n} f(a_1 \cdots a_{i-1} \underbrace{a_i}_{\substack{\text{the } i\text{-th input position is} \\ \text{distinguished using the second copy } \underline{\Sigma}}} a_{i+1} \cdots a_n).$$

**Proof**
The weighted automaton for the function in the conclusion of the claim uses nondeterminism to guess the distinguished position. □

The claim completes the proof of the lemma about polyregular string-to-number functions being equivalent to weighted automata of polynomial growth.

To complete the proof of Theorem 8.1 we only need to left-to-right direction in the lemma. Indeed, by this direction, equivalence for polyregular string-to-number functions reduces to equivalence for weighted automata. The latter problem is decidable, even with weights that are rational numbers. The algorithm dates back to the original paper of Schützenberger that introduces weighted automata over a field [32], see also [7, Section 8] for a proof.

## REFERENCES


[1] Rajeev Alur and Pavol Černý. 2010. Expressiveness of Streaming String Transducers. In *Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2010, Chennai, India (LIPIcs, Vol. 8)*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 1–12.

[2] Rajeev Alur, Adam Freilich, and Mukund Raghothaman. 2014. Regular combinators for string transformations. In *Computer Science Logic and Logic in Computer Science, CSL-LICS 2014, Vienna, Austria,*. ACM, 1–10.

[3] Félix Baschenis, Olivier Gauwin, Anca Muscholl, and Gabriele Puppis. 2016. Minimizing resources of sweeping and streaming string transducers. In *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*, Vol. 55. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 114–1.

[4] Félix Baschenis, Olivier Gauwin, Anca Muscholl, and Gabriele Puppis. 2017. Untwisting two-way transducers in elementary time. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 1–12.

[5] Mikołaj Bojańczyk. 2018. Polyregular Functions. *CoRR* abs/1810.08760 (2018).

[6] Mikołaj Bojańczyk. 2022. The growth rate of polyregular functions (unpublished). (May 2022).

[7] Mikołaj Bojańczyk and Wojciech Czerwiński. [n.d.]. An Automata Toolbox.

[8] Mikołaj Bojańczyk, Laure Daviaud, and Shankara Narayanan Krishna. 2018. Regular and First-Order List Functions. In *Logic in Computer Science, LICS, Oxford, UK*. ACM, 125–134.

[9] Mikolaj Bojanczyk, Sandra Kiefer, and Nathan Lhote. 2019. String-to-String Interpretations With Polynomial-Size Output. In *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*. 106:1–106:14.

[10] Arthur W Burks and Hao Wang. 1957. The Logic of Automata—Part I. *Journal of the ACM (JACM)* 4, 2 (1957), 193–218.

[11] Michal Chytil and Vojtech Jákl. 1977. Serial Composition of 2-Way Finite-State Transducers and Simple Programs on Strings. In *International Colloquium on Automata, Languages and Programming, ICALP, Turku, Finland (Lecture Notes in Computer Science, Vol. 52)*. Springer, 135–147.

[12] Thomas Colcombet. 2007. A Combinatorial Theorem for Trees. In *International Colloquium on Automata, Languages and Programming, ICALP, Wrocław, Poland (Lecture Notes in Computer Science)*. Springer, 901–912.

[13] Bruno Courcelle and Joost Engelfriet. 2012. *Graph Structure and Monadic Second-Order Logic - A Language-Theoretic Approach*. Encyclopedia of Mathematics and Its Applications, Vol. 138. Cambridge University Press.

[14] Gaëtan Douéneau-Tabot. 2021. Pebble Transducers with Unary Output. In *46th International Symposium on Mathematical Foundations of Computer Science, MFCS 2021, August 23-27, 2021, Tallinn, Estonia (LIPIcs, Vol. 202)*, Filippo Bonchi and Simon J. Puglisi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 40:1–40:17.

[15] Gaëtan Douéneau-Tabot, Emmanuel Filiot, and Paul Gastin. 2020. Register Transducers Are Marble Transducers. In *45th International Symposium on Mathematical Foundations of Computer Science (MFCS 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 170)*, Javier Esparza and Daniel Kráľ (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 29:1–29:14.

[16] Joost Engelfriet. 2015. Two-way pebble transducers for partial functions and their composition. *Acta Inf.* 52, 7-8 (2015), 559–571.

[17] Joost Engelfriet and Hendrik Jan Hoogeboom. 2001. MSO Definable String Transductions and Two-Way Finite-State Transducers. *ACM Trans. Comput. Logic* 2, 2 (2001), 216–254.

[18] Joost Engelfriet, Hendrik Jan Hoogeboom, and Bart Samwel. 2007. XML transformation by tree-walking transducers with invisible pebbles. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 63–72.

[19] Joost Engelfriet, Kazuhiro Inaba, and Sebastian Maneth. 2021. Linear-bounded composition of tree-walking tree transducers: linear size increase and complexity. *Acta Informatica* 58, 1-2 (2021), 95–152.

[20] Joost Engelfriet and Sebastian Maneth. 2002. Two-way finite state transducers with nested pebbles. In *International Symposium on Mathematical Foundations of Computer Science*. Springer, 234–244.

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

[22] Wilfrid Hodges. 1993. *Model Theory*. Cambridge University Press.

[23] J. E. Hopcroft and J. D. Ullman. 1967. An approach to a unified theory of automata. In *8th Annual Symposium on Switching and Automata Theory (SWAT 1967)*. 140–147.

[24] Oscar H Ibarra. 1971. Characterizations of some tape and time complexity classes of Turing machines in terms of multihead and auxiliary stack automata. *J. Comput. System Sci.* 5, 2 (1971), 88–117.

[25] Kenneth Krohn and John Rhodes. 1965. Algebraic theory of machines. I. Prime decomposition theorem for finite semigroups and machines. *Trans. Amer. Math. Soc.* 116 (1965), 450–450.

[26] Nathan Lhote. 2020. Pebble Minimization of Polyregular Functions. In *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*. 703–712.

[27] Albert R. Meyer. 1975. Weak monadic second order theory of succesor is not elementary-recursive. In *Logic Colloquium (Boston, Mass., 1972–1973)*. Springer, Berlin, 132–154. Lecture Notes in Math., Vol. 453.

[28] Tova Milo, Dan Suciu, and Victor Vianu. 2003. Typechecking for XML transformers. *J. Comput. Syst. Sci.* 66, 1 (2003), 66–97.

[29] Lê Thành Dung Nguyên, Camille Noûs, and Pierre Pradic. [n.d.]. Comparison-Free Polyregular Functions (Hal version 1).

[30] Lê Thành Dung Nguyên, Camille Noûs, and Pierre Pradic. 2021. Comparison-Free Polyregular Functions. In *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*. 139:1–139:20.

[31] Michael O. Rabin and Dana Scott. 1959. Finite automata and their decision problems. *IBM J. Res. Develop.* 3 (1959), 114–125.

[32] Marcel Paul Schützenberger. 1961. On the definition of a family of automata. *Information and control* 4, 2–3 (1961), 245–270.

[33] Dana Scott. 1967. Some definitional suggestions for automata theory. *J. Comput. System Sci.* 1, 2 (1967), 187–212.

[34] J. C. Shepherdson. 1959. The Reduction of Two-Way Automata to One-Way Automata. *IBM Journal of Research and Development* 3, 2 (April 1959), 198–200.

[35] Wolfgang Thomas. 1997. Languages, automata, and logic. In *Handbook of formal languages, Vol. 3*. Springer, Berlin, 389–455.

[36] Boris Avraamovich Trakhtenbrot. 1957. On operators realizable in logical nets. In *Doklady Akademii Nauk*, Vol. 112. Russian Academy of Sciences, 1005–1007.