Pebble Transducers with Unary Output

Gaëtan Douéneau-Tabot
IRIF, Université de Paris, France

Abstract

Bojańczyk recently initiated an intensive study of deterministic pebble transducers, which are two-way automata that can drop marks (named "pebbles") on their input word, and produce an output word. They describe functions from words to words. Two natural restrictions of this definition have been investigated: marble transducers by Douéneau-Tabot et al., and comparison-free pebble transducers (that we rename here "blind transducers") by Nguyên et al.

Here, we study the decidability of membership problems between the classes of functions computed by pebble, marble and blind transducers that produce a unary output. First, we show that pebble and marble transducers have the same expressive power when the outputs are unary (which is false over non-unary outputs). Then, we characterize 1-pebble transducers with unary output that describe a function computable by a blind transducer, and show that the membership problem is decidable. These results can be interpreted in terms of automated simplification of programs.

2012 ACM Subject Classification Theory of computation → Transducers

Keywords and phrases polyregular functions, pebble transducers, marble transducers, streaming string transducers, factorization forests

Digital Object Identifier 10.4230/LIPIcs.MFCS.2021.40

Related Version *Full Version*: `https://arxiv.org/abs/2104.14019`

Funding *Gaëtan Douéneau-Tabot*: Funded by the French Ministry of Defence (DGA IP).

Acknowledgements The author is grateful to Olivier Carton for discussing about this work. He also thanks the reviewers for their helpful comments and remarks.

1 Introduction

Regular languages can be described by several models such as deterministic, non-deterministic, or two-way (the reading head can move in two directions) finite automata [12]. A natural extension consists in adding an output mechanism to finite automata. Such machines, called *transducers*, describe functions from words to words (or relations when non-deterministic) and provide a natural way to model simple programs that produce outputs. The particular model of a *two-way transducer* consists in a two-way automaton enhanced with an output function. It describes the class of *regular functions* which has been intensively studied for its fundamental properties: closure under composition [5], logical characterization by monadic second-order transductions [7], decidable equivalence problem [9], etc.

Pebble transducers and their variants. The model of k-pebble transducer can be defined as an inductive extension of two-way transducers. A 0-pebble transducer is just a two-way transducer. For $k \geq 1$, a k-pebble transducer $\mathcal{T}$ is a two-way transducer that, when in a given configuration, can "call" an *external function* $\mathfrak{f}$, computed by some $(k-1)$-pebble transducer. $\mathcal{T}$ gives as argument to $\mathfrak{f}$ its input word together with a mark, named "pebble", on the position from which the call was performed, and uses the output of $\mathfrak{f}$ within its own output.

The behavior of a 1-pebble transducer is depicted in Figure 1. Intuitively, a k-pebble transducer is some recursive program whose recursion depth is at most $k+1$. Equivalently, it can be seen as an iterative algorithm with "two-way for-loops", such that the maximal depth of nested loops is $k+1$. A k-pebble transducer can only produce an output whose length is polynomial in its input's length, more precisely $\mathcal{O}(n^{k+1})$ when n is the input's length (this is intuitive from the "nested loops" point of view). The functions computed by a k-pebble transducer for some $k \geq 0$ are thus called *polyregular functions* [3]. Several properties of polyregular functions have been investigated: closure under composition [3], logical characterization by monadic second-order interpretations [4], etc. The equivalence problem (given two machines, do they compute the same function?) is however still open.

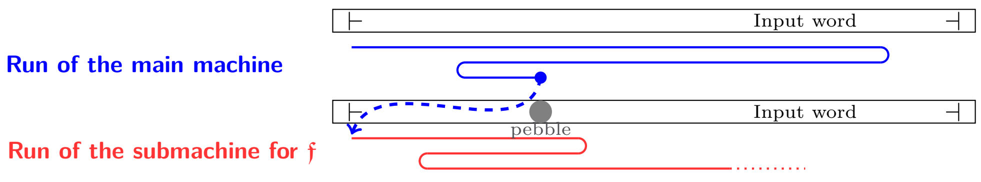


Figure 1 Behavior of a 1-pebble transducer.

Recently, two natural restrictions of pebble transducers have been introduced. First, the *k-marble transducers* of [6] only give as argument to their external function the prefix of the input word which ends in the calling position (see Figure 4). Second, the *k-blind transducers*[1] of [11] give the whole input word, but no pebble on the calling position (see Figure 3). The classes of functions they compute are strict subclasses of polyregular functions [6, 11].

Class membership problems. These various models of transducers raise several membership problems: given a function computed by a machine of model X, can it be computed by some machine of model Y? When Y is a restriction of X, this problem reformulates as a program optimization question: given a "complex" algorithm in a class X, can we build an equivalent "simpler" one in class Y? Thus it is of a foremost interest in practice.

Given a function f computed by an ℓ-pebble transducer, one can ask whether it is computable by a k-pebble transducer for a given $k < \ell$. The problem is decidable [10], and it turns out that a necessary and sufficient condition for this membership is that $|f(w)| = \mathcal{O}(|w|^{k+1})$. Using the "nested loops" interpretation of pebble transducers, it means that an output of size $\mathcal{O}(|w|^{k+1})$ can always be produced with at most $k{+}1$ nested loops. Similar results have been obtained in [6] and [11] for their variants, with the same conclusion: an output of size $\mathcal{O}(|w|^{k+1})$ can always be produced with depth at most $k{+}1$.

Contributions. In this paper, we study a different membership problem: can a function given by a k-pebble transducer be computed by a k-marble or k-blind transducer? It turns out to be a more difficult question, since there is no intuitive and machine-independent candidate for a membership condition (such as the size of the output). In general, membership problems for transducers are difficult, since contrary to regular languages, there is no "canonical" object known to represent a regular function. Hence, there can be several seemingly unrelated manners to produce the same function, and moving from one to another can be technical.

We focus on *transducers whose output alphabet is unary*, and our proof techniques are new. The first main result is that (when the outputs are unary) k-pebble transducers and k-marble transducers compute the same functions (one direction is obvious since k-marble is a restriction of k-pebble). The transformation is effective, but the way of producing the output must sometimes be completely modified (the transformation modifies the *origin semantics*, in the sense of [2]), which creates an additional difficulty. The correspondence fails as soon as the output is not over a unary alphabet, as detailed in Example 1.

[1] The original terminology of [11] is *comparison-free pebble transducers*, but we strongly believe that the term "blind" is more adapted, since there are no pebbles in this model.

▶ **Example 1** ([6, 11]). The partial function $\{a,b\}^* \to \{a,b\}^*, a^m b^n \mapsto (b^n a)^m$ can be computed by a 1-pebble transducer, but not by a k-marble for any $k \geq 0$.

Since the equivalence problem is decidable for marble transducers, it follows from our result that it is also decidable for pebble transducers with unary output.

As a second main result, we show how to decide (when the outputs are unary) whether a function given by 1-pebble ($\equiv$ 1-marble) transducer can be computed by a 1-blind transducer, or more generally by a k-blind transducer for some $k \geq 0$. The technical proof also gives a syntactical characterization of 1-marble transducers whose function verify this property: it describes a kind of "symmetry" in the production of the machine on its input. Furthermore, the conversion is effective when possible, but once more the manner of producing the output can be strongly modified. Our techniques heavily rely on the theory of *factorization forests*; this is, to our knowledge, the first time this notion is used for membership problems of transducers, and we believe this approach to be fruitful.

Our results are summarized in red in Figure 2. We also give some examples of functions (their outputs are non-negative integers, since we identify $\{a\}^*$ with $\mathbb{N}$).

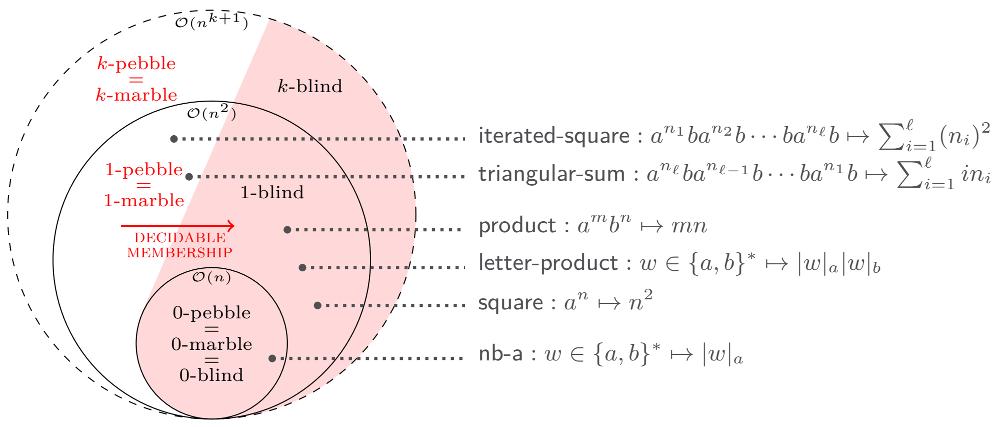

$\mathcal{O}(n^{k+1})$
k-pebble = k-marble
k-blind
$\mathcal{O}(n^2)$
1-pebble = 1-marble
1-blind
DECIDABLE MEMBERSHIP
$\mathcal{O}(n)$
0-pebble = 0-marble = 0-blind
iterated-square : $a^{n_1}ba^{n_2}b\cdots ba^{n_\ell}b \mapsto \sum_{i=1}^{\ell}(n_i)^2$
triangular-sum : $a^{n_\ell}ba^{n_{\ell-1}}b\cdots ba^{n_1}b \mapsto \sum_{i=1}^{\ell} in_i$
product : $a^m b^n \mapsto mn$
letter-product : $w \in \{a,b\}^* \mapsto |w|_a|w|_b$
square : $a^n \mapsto n^2$
nb-a : $w \in \{a,b\}^* \mapsto |w|_a$

Figure 2 Classes of functions with unary output and results of this paper.

Outline. We first recall in Section 2 the definitions of k-pebble, k-marble and k-blind transducers, simplified for the case of unary outputs. In Section 3, we define the notions of k-pebble, k-marble and k-blind bimachines, and show their equivalence with the transducer models. Bimachines are easier to handle in the proofs, due to the fact that they avoid two-way moves. In Section 4, we show that k-marble and k-pebble transducers are equivalent. Finally, we solve in Section 5 the class membership problem from 1-pebble to 1-blind. Due to space constraints, several proofs are sketched in the main paper, and we focus on the most significant lemmas and characterizations.

2 Preliminaries

$\mathbb{N}$ is the set of nonnegative integers. If $0 \leq i \leq j$, the set $[i{:}j]$ denotes $\{i, i+1, \dots, j\} \subseteq \mathbb{N}$ (empty if $j < i$). Capital letters A, B denote finite sets of letters (alphabets). The empty word is denoted ε. If $w \in A^*$, let $|w| \in \mathbb{N}$ be its length, and for $1 \leq i \leq |w|$ let $w[i]$ be its i-th letter. If $I = \{i_1 < \cdots < i_\ell\} \subseteq \{1, \dots, |w|\}$, let $w[I] := w[i_1]\cdots w[i_\ell]$. If $a \in A$, we denote by $|w|_a$ the number of letters a occurring in w. Given $A = \{a, \dots\}$, let $\underline{A} := \{\underline{a}, \dots\}$ be a disjoint copy of A. For $1 \leq i \leq |w|$, we define $w{\uparrow}i := w[1{:}i{-}1]\underline{w[i]}w[i{+}1{:}|w|]$ as "w in which position i is underlined". We assume that the reader is familiar with the basics of automata theory, in particular the notion of two-way deterministic automaton.

Two-way transducers. A deterministic two-way transducer is a deterministic two-way automaton enhanced with the ability to produce outputs along its run. The class of functions described by these machines is known as *regular functions* [5, 7].

▶ **Definition 2.** *A* (deterministic) two-way transducer $(A, B, Q, q_0, F, \delta, \lambda)$ *is:*

- *an input alphabet A and an output alphabet B;*
- *a finite set of states Q with an initial state $q_0 \in Q$ and a set $F \subseteq Q$ of final states;*
- *a (partial) transition function $\delta : Q \times (A \uplus \{\vdash, \dashv\}) \to Q \times \{\triangleleft, \triangleright\}$;*
- *a (partial) output function $\lambda : Q \times (A \uplus \{\vdash, \dashv\}) \to B^*$ with same domain as δ.*

When given as input a word $w \in A^*$, the two-way transducer disposes of a read-only input tape containing $\vdash w \dashv$. The marks $\vdash$ and $\dashv$ are used to detect the borders of the tape, by convention we denote them as positions 0 and $|w|{+}1$ of w. Formally, a *configuration* over $\vdash w \dashv$ is a tuple (q, i) where $q \in Q$ is the current state and $0 \leq i \leq |w|{+}1$ is the position of the reading head. The *transition relation* $\to$ is defined as follows. Given a configuration (q, i), let $(q', \star) := \delta(q, w[i])$. Then $(q, i) \to (q', i')$ whenever either $\star = \triangleleft$ and $i' = i - 1$ (move left), or $\star = \triangleright$ and $i' = i + 1$ (move right), with $0 \leq i' \leq |w|{+}1$. A *run* is a sequence of configurations $(q_1, i_1) \to \cdots \to (q_n, i_n)$. Accepting runs are those that begin in $(q_0, 0)$ and end in a configuration of the form $(q, |w|{+}1)$ with $q \in F$ (and it never visits such a configuration before). The function $f : A^* \to B^*$ computed by the machine is defined as follows. Let $w \in A^*$, if there exists an accepting run on $\vdash w \dashv$, then $f(w)$ is the concatenation of the $\lambda(q, w[i])$ along this unique run on $\vdash w \dashv$. To make f a total function, we let $f(w) := \varepsilon$ if there is no accepting run (the language of words having an accepting run in a two-way transducer is regular [12], hence the domain does not matter).

▶ **Example 3.** reverse : $A^* \to A^*, abac \mapsto caba$ can be computed by a two-way transducer.

From now on, the output alphabet of the machines will always be a singleton. Up to identifying $\{a\}^*$ and $\mathbb{N}$, we assume that $\lambda : Q \times (A \uplus \{\vdash, \dashv\}) \to \mathbb{N}$ and $f : A^* \to \mathbb{N}$.

External functions. We now extend the notion of output function λ: it will not give directly an integer, but performs a call to an *external function* which returns an integer. For pebbles, the output of the external functions depends on the input word and the current position.

▶ **Definition 4.** *A* two-way transducer with external pebble functions $(A, Q, q_0, F, \delta, \mathfrak{F}, \lambda)$ *is:*

- *an input alphabet A;*
- *a finite set of states Q with an initial state $q_0 \in Q$ and a set $F \subseteq Q$ of final states;*
- *a (partial) transition function $\delta : Q \times (A \uplus \{\vdash, \dashv\}) \to Q \times \{\triangleleft, \triangleright\}$;*
- *a finite set $\mathfrak{F}$ of external functions $\mathfrak{f} : (A \uplus \underline{A})^* \to \mathbb{N}$;*
- *a (partial) output function $\lambda : Q \times (A \uplus \{\vdash, \dashv\}) \to \mathfrak{F}$ with same domain as δ.*

Configurations (q, i) and runs of two-way transducers with external functions are defined as for classical two-way transducers. The function $f : A^* \to \mathbb{N}$ computed by the machine is defined as follows. Let $w \in A^*$ such that there exists an accepting run on $\vdash w \dashv$. If $\lambda(q, w[i]) = \mathfrak{f} \in \mathfrak{F}$, we let $\nu(q, i) := \mathfrak{f}(w{\uparrow}i)$, that is the result of $\mathfrak{f}$ applied to w marked in i. Finally, $f(w)$ is defined as the sum of the $\nu(q, i)$ along this unique accetping run on $\vdash w \dashv$. We similarly set $f(w) = 0$ if there is no accepting run.

▶ Remark 5. If the external functions are constant, we exactly have a two-way transducer.

▶ **Example 6.** Let $a, b \in A$, $\mathfrak{f}_b : w \in (A \uplus \underline{A})^* \mapsto |w|_b$ and $\mathfrak{f}_0 : w \in (A \uplus \underline{A})^* \mapsto 0$. The two-way transducer with external pebble functions, which makes a single pass on its input and calls $\mathfrak{f}_b$ if reading a and $\mathfrak{f}_0$ otherwise, computes letter-product : $w \in A \mapsto |w|_a|w|_b$.

We define two other models. Their definition is nearly the same, except that the external functions of $\mathfrak{F}$ have type $A^* \to \mathbb{N}$ and $\nu(q,i)$ is defined in a slightly different way:

- in a *two-way transducer with external blind functions*, we define $\nu(q,i) := \mathfrak{f}(w)$. The external function is applied to w without marking the current position;
- in a *two-way transducer with external marble functions*, we define $\nu(q,i) := \mathfrak{f}(w[1{:}i])$. The external function is applied to the prefix of w stopping at the current position.

Pebble, blind and marble transducers. We now describe the transducer models using the formalism of external functions. These are not the original definitions from [3, 6, 11] (we are closer to the nested transducers of [10]), but the correspondence is straightforward, as soon as we know that pebble automata can only recognize regular languages.

▶ **Definition 7.** *For $k \geq 0$, a* k-pebble *(resp.* k-blind, k-marble*)* transducer *is:*

- *if $k = 0$, a two-way transducer;*
- *if $k \geq 1$, a two-way transducer with external pebble (resp. blind, marble) functions that are computed by $(k-1)$-pebble (resp. $(k-1)$-blind, $(k-1)$-marble) transducers.*

The intuitive behavior of a 1-pebble transducer is depicted in Figure 1 in Introduction. We draw in Figure 3 the behavior of a 1-blind transducer, which is the same except that the calling position is not marked for the machine computing the external function.

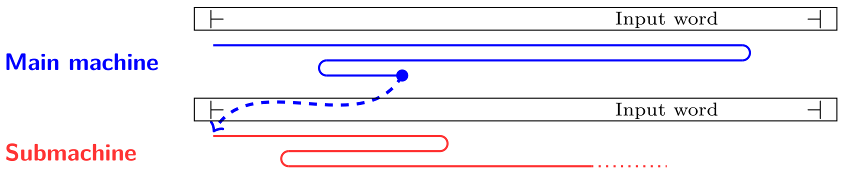


Figure 3 Behavior of a 1-blind transducer.

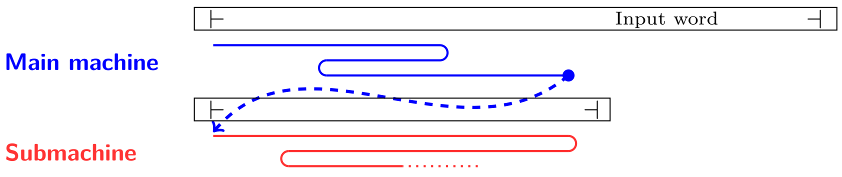


Figure 4 Behavior of a 1-marble transducer.

▶ **Example 8.** By restricting the functions $\mathfrak{f}_b$ and $\mathfrak{f}_0$ of Example 6 to A^*, we see that letter-product : $w \in A^* \mapsto |w|_a|w|_b$ can be computed by a 1-blind transducer.

The intuitive behavior of a 1-marble transducer is depicted in Figure 4.

▶ **Example 9.** The function letter-product : $w \in A^* \mapsto |w|_a|w|_b$ can be computed by a 1-marble transducer as follows. Assume that $a \neq b$, let $\mathfrak{f}_a : w \mapsto |w|_a$ and $\mathfrak{f}_b : w \mapsto |w|_b$. The machine calls $\mathfrak{f}_b$ when reading a and $\mathfrak{f}_a$ when reading b. This way, each a is "counted" $|w|_b$ times (from the call of $\mathfrak{f}_b$ starting in this a which computes all the b before it, plus each time it is seen in a call of $\mathfrak{f}_a$ starting from some b after this a).

The strategy for computing letter-product is really different between Examples 8 and 9. This illustrates the difficulty to obtain a “canonical” form for a transduction.

3 From two-way transducers to bimachines

Since we consider a commutative output monoid, the order in which the production is performed does not matter. It is thus tempting to simplify a two-way transducer in a one-way machine which visits each position only once. This is exactly what we do with bimachines, with the subtlety that they are able to check regular properties of the prefix (resp. suffix) starting (resp. ending) in the current position. From now on, we consider only total functions of type $A^+ \to \mathbb{N}$ (the output on ε can be treated separately and does not matter).

▶ **Definition 10.** *A* bimachine with external pebble functions $(A, M, \mu, \mathfrak{F}, \lambda)$ *consists of:*

- *an input alphabet A;*
- *a morphism into a finite monoid $\mu : A^* \to M$;*
- *a finite set $\mathfrak{F}$ of external functions $\mathfrak{f} : (A \uplus \underline{A})^+ \to \mathbb{N}$;*
- *a total output function $\lambda : M \times A \times M \to \mathfrak{F}$.*

Given $1 \leq i \leq |w|$ a position of $w \in A^*$, let $\mathfrak{f}_i := \lambda(\mu(w[1{:}i{-}1]), w[i], \mu(w[i{+}1{:}|w|])) \in \mathfrak{F}$. The bimachine defines a function $f : A^+ \to \mathbb{N}$ as follows:

$$f(w) := \sum_{1 \leq i \leq |w|} \mathfrak{f}_i(w{\uparrow}i).$$

As before, we can define *bimachines with external blind* (resp. *marble*) functions (in this case we have $\mathfrak{f}_i : A^+ \to \mathbb{N}$). We then let:

$$f(w) := \sum_{1 \leq i \leq |w|} \mathfrak{f}_i(w) \left(\text{resp. } f(w) := \sum_{1 \leq i \leq |w|} \mathfrak{f}_i(w[1{:}i]) \right).$$

As for two-way transducers, we define bimachines (without external functions) by setting $\lambda : M \times A \times M \to \mathbb{N}$. Equivalently, it corresponds to bimachines with external constant functions $\mathfrak{f} : w \mapsto n$. Going further, we define k-pebble bimachines by induction.

▶ **Definition 11.** *For $k \geq 0$, a k-*pebble *(resp. k-*blind*, k-*marble*)* bimachine *is:*

- *if $k = 0$, a bimachine (without external functions);*
- *if $k \geq 1$, a bimachine with external pebble (resp. blind, marble) functions which are computed by $(k{-}1)$-pebble (resp $(k{-}1)$-blind, $(k{-}1)$-marble) bimachines.*

▶ **Example 12.** The function triangular-sum : $a^{n_\ell}ba^{n_{\ell-1}}b \cdots ba^{n_1}b \mapsto \sum_{i=1}^{\ell} in_i$ can be computed by a 1-marble bimachine. It uses the singleton monoid $M = \{1_M\}$ and the morphism $\mu : a, b \mapsto 1_M$ in all its bimachines. The output function of the main bimachine is defined by $\lambda(1_M, a, 1_M) := \mathfrak{f}_a$ and $\lambda(1_M, b, 1_M) := \mathfrak{f}_b$. For computing $\mathfrak{f}_a : w \mapsto 0$ we use output $\lambda_{\mathfrak{f}_a}(1_M, a, 1_M) := 0$ and $\lambda_{\mathfrak{f}_a}(1_M, b, 1_M) := 0$, and for computing $\mathfrak{f}_b : w \mapsto |w|_a$ we use output $\lambda_{\mathfrak{f}_b}(1_M, a, 1_M) := 1$ and $\lambda_{\mathfrak{f}_b}(1_M, b, 1_M) := 0$.

Standard proof techniques allow to relate bimachines and transducers.

▶ **Proposition 13.** *k-pebble (resp. k-blind, k-marble) bimachines and k-pebble (resp. k-blind, k-marble) transducers compute the same functions, and both conversions are effective.*

Proof sketch. Both directions are treated by induction. From bimachines to transducers, we show that a bimachine with external pebble functions can be transformed in an equivalent two-way transducer with the same external pebble functions (we use a *lookaround* [7] to simulate μ). From transducers to bimachines, the induction step shows that a two-way transducer with external pebble functions can be transformed in an equivalent bimachine with external pebble functions, by adapting the classical reduction from two-way to one-way automata [12]. However the new external functions can be linear combinations of the former ones, since we produce "all at once" the results of several visits in a position. We only need to use a finite number of combinations, since in its accepting runs, a two-way transducer can only visit each position a bounded number of times. ◀

4 Equivalence between k-pebble and k-marble transducers

The main goal of this section is to show equivalence between k-pebble and k-marble transducers, over unary outputs. We shall use another model which is equivalent to marble transducers [6]: a *streaming string transducer (with unary output)*, which consists in a deterministic automaton with a finite set $\mathfrak{X}$ of registers that store integers. At each letter read, the values of the registers are updated by doing a linear combination of their former values, whose coefficients depend on the current state of the automaton. In our definition we focus on the registers and forget about the states, which corresponds to a weighted automaton over the semiring $(\mathbb{N}, +, \times)$ (it is shown in [6] that both models are equivalent). The update is represented by a matrix from $\mathbb{N}^{\mathfrak{X} \times \mathfrak{X}}$, which is chosen depending on the letter read.

▶ **Definition 14.** *A* streaming string transducer *(SST) $\mathcal{T} = (A, \mathfrak{X}, I, T, F)$ is:*

- *an input alphabet A and a finite set $\mathfrak{X}$ of registers;*
- *an initial row vector $I \in \mathbb{N}^{\mathfrak{X}}$;*
- *a register update function $T : A \to \mathbb{N}^{\mathfrak{X} \times \mathfrak{X}}$;*
- *an output column vector $F \in \mathbb{N}^{\mathfrak{X}}$.*

T can be extended as a monoid morphism from A^* to $(\mathbb{N}^{\mathfrak{X} \times \mathfrak{X}}, \times)$. Given $w \in A^*$, the vector $IT(w)$ intuitively describes the values of the registers after reading w. To define the function $f : A^* \to \mathbb{N}$ computed by $\mathcal{T}$, we combine these values by the output vector:

$$f(w) := IT(w)F.$$

▶ **Example 15.** The function triangular-sum : $a^{n_\ell} b a^{n_{\ell-1}} b \cdots b a^{n_1} b \mapsto \sum_{i=1}^{\ell} i n_i$ can be computed by an SST. We use two registers x, y and allow constants in the updates for more readability: x is initialized to 0 and updated $x \leftarrow x + 1$ on a and $x \leftarrow x$ on b, and y is initialized to 0 and updated $y \leftarrow y$ on a and $y \leftarrow y + x$ on b. Finally we output y.

We are now ready to state the main results of this section.

▶ **Theorem 16.** *Given a k-pebble bimachine, one can build an equivalent SST.*

The proof is done by induction on $k \geq 0$. Consider a bimachine whose external functions are computed by $(k-1)$-pebble bimachines. By hypothesis, we can compute these functions by SSTs. The induction step is shown by Lemma 17, which uses new proof techniques.

▶ **Lemma 17.** *Given a bimachine with external pebble functions computed by SSTs, one can build an equivalent SST (with no external functions).*

Proof idea. Let $\mathcal{T}$ be the SST computing an external function $\mathfrak{f}$. On input $w \in A^+$, the bimachine calls $\mathfrak{f}$ on several positions $1 \leq i_1 < \cdots < i_\ell \leq |w|$, which induces executions of $\mathcal{T}$ on $w{\uparrow}i_1, \dots, w{\uparrow}i_\ell$. These executions are very similar: they only differ when reading the marked letter. Thus we build an SST which computes "simultaneously" all these executions, by keeping track of the sum of the values of the registers of $\mathcal{T}$ along them. ◀

As a consequence of Theorem 16, we obtain equivalence between pebbles and marbles over unary outputs. The result is false over non-unary output alphabets [6, 11]. We also relate these functions with those computed by SST, assuming that the output is bounded by a polynomial in the input's length.

▶ **Corollary 18.** *For all $k \geq 0$ and $f : A^* \to \mathbb{N}$, the following conditions are equivalent:*
1. *f is computable by a k-pebble transducer;*
2. *f is computable by a k-marble transducer;*
3. *f is computable by an SST and $f(w) = \mathcal{O}(|w|^{k+1})$.*

Furthermore the transformations are effective.

Proof. Clearly a k-pebble transducer can simulate a k-marble transducer, hence $2 \Rightarrow 1$. Let f be computed by a k-pebble transducer, we have $f(w) = \mathcal{O}(|w|^{k+1})$ and by Theorem 16 one can build an SST for f. Thus $1 \Rightarrow 3$. Finally $3 \Rightarrow 2$ is shown in [6]. ◀

Another important consequence is that we can decide equivalence of pebble transducers with unary output, since we can do so for marble transducers [6].

▶ **Corollary 19.** *One can decide if two pebble transducers compute the same function.*

This has been an open question since [3], and it is still open for generic output alphabets.

5 Deciding if 1-pebble is 1-blind

Since the equivalence between marbles and pebbles is established, we now compare 1-pebble (which are 1-marble) transducers with 1-blind transducers. It turns out that 1-pebble are strictly more expressive; the main goal of this section is to show Theorem 20.

▶ **Theorem 20** (Membership). *One can decide if a function given by a 1-marble (or 1-pebble) transducer can be computed by a k-blind transducer for some $k \geq 0$. If this condition holds, one can build a 1-blind transducer which computes it.*

Let us fix a function $f : A^+ \to \mathbb{N}$ described by a 1-marble bimachine $\mathcal{T} = (A, M, \mu, \mathfrak{F}, \lambda)$. For $\mathfrak{f} \in \mathfrak{F}$, let $\mathcal{T}_\mathfrak{f} := (A, M, \mu, \lambda_\mathfrak{f})$ be the bimachine which computes it. We enforce the morphism μ to be surjective (up to considering the co-restriction to its image) and the same in all machines (up to taking the product of all morphisms used). Our goal is to give a decidable condition on $\mathcal{T}$ for f to be computable by a 1-blind transducer. For this purpose, we define the notion of *bitype*. Intuitively, it describes two disjoint factors in an input word, together with a finite abstraction of their "context".

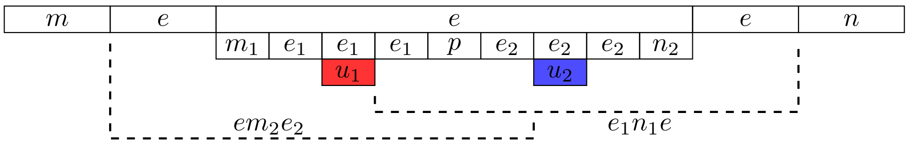
Let $\Lambda := 3|M|$ (it will be justified by Theorem 27).

▶ **Definition 21.** *A* bitype $\Phi := m\langle \mathbf{u_1} \rangle m' \langle \mathbf{u_2} \rangle m''$ *consists in $m, m', m'' \in M$, $u_1, u_2 \in A^+$.*

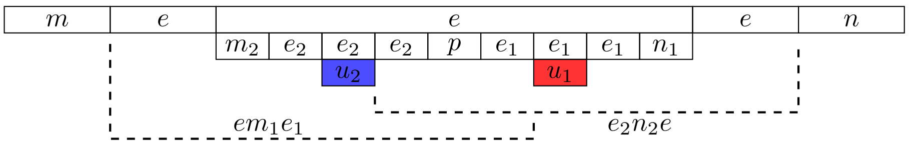
We can define "the production performed in u_1 by the calls from u_2", in Φ. For $1 \leq i \leq |u_1|$ and $1 \leq j \leq |u_2|$, let $\Phi(i,j) := \lambda_{\mathfrak{f}_j}(m\mu(u_1[1{:}i{-}1]), u_1[i], \mu(u_1[i{+}1{:}|u_1|])m'\mu(u_2[1{:}j])) \in \mathbb{N}$ where $\mathfrak{f}_j := \lambda(m\mu(u_1)m'\mu(u_2[1{:}j{-}1]), u_2[j], \mu(u_2[j{+}1{:}|u_2|])m'')$. Then we set:

$$\mathsf{prod}(m\langle \mathbf{u_1} \rangle m' \langle \mathbf{u_2} \rangle m'') := \sum_{\substack{1 \leq i \leq |u_1| \\ 1 \leq j \leq |u_2|}} \Phi(i,j) \in \mathbb{N}.$$

▶ **Definition 22.** *The* 1*-marble bimachine* $\mathcal{T}$ *is* symmetrical *whenever* $\forall m, n, m_1, n_1, m_2, n_2 \in M$ *and* $u_1, u_2 \in A^+$ *such that* $|u_1|, |u_2| \leq 2^\Lambda$, $e_1{:=}\mu(u_1)$, $e_2{:=}\mu(u_2)$ *and* $e{:=}m_1e_1n_1{=}m_2e_2n_2$ *are idempotents, there exists* $K \geq 0$ *such that* $\forall p \in M$*:*
- *if* $m_1e_1pe_2n_2 = e$, $em_1e_1pe_2 = em_2e_2$ *and* $e_1pe_2n_2e = e_1n_1e$, *then* $\mathsf{prod}(mem_1e_1\langle\mathbf{u_1}\rangle e_1pe_2\langle\mathbf{u_2}\rangle e_2n_2en) = K$*;*
- *if* $m_2e_2pe_1n_1 = e$, $em_2e_2pe_1 = em_1e_1$ *and* $e_2pe_1n_1e = e_2n_2e$, *then* $\mathsf{prod}(mem_2e_2\langle\mathbf{u_2}\rangle e_2pe_1\langle\mathbf{u_1}\rangle e_1n_1en) = K$*.*



(a) Bitype $mem_1e_1\langle\mathbf{u_1}\rangle e_1pe_2\langle\mathbf{u_2}\rangle e_2n_2en$.



(b) Bitype $mem_2e_2\langle\mathbf{u_2}\rangle e_2pe_1\langle\mathbf{u_1}\rangle e_1n_1en$

Figure 5 The bitypes used to define a symmetrical 1-marble bimachine.

Symmetry means that, under some idempotent conditions, $\mathsf{prod}(m\langle\mathbf{u_1}\rangle m'\langle\mathbf{u_2}\rangle m'')$ only depends on $m, m'', m'\mu(u_2)m''$ and $m\mu(u_1)m'$, that are the "contexts" of u_1 and u_2, but not on the element m' which separates them. The same holds if we swap u_1 and u_2. The bitypes considered in Definition 22 are depicted in Figure 5, together with the equations they satisfy.

Symmetry is the decidable condition we are looking for, as shown in Theorem 23. Recall that f is the function computed by the 1-marble bimachine $\mathcal{T}$.

▶ **Theorem 23** (Characterization)**.** *The following conditions are equivalent:*
1. f *is computable by a* k*-blind transducer for some* $k \geq 0$*;*
2. f *is computable by a* 1*-blind transducer;*
3. $\mathcal{T}$ *is symmetrical.*

Theorem 20 follows from Theorem 23, since it suffices to check whether the machine is symmetrical, which can be decided by ranging over all monoid elements (including idempotents) and words of length at most 2^Λ.

▶ **Example 24.** Let us show that the bimachine of Example 12 computing $\mathsf{triangular\text{-}sum}$ is not symmetrical. Let $u_1 := a$, $u_2 := b$, $m, n, m_1, n_1, m_2, n_2, p, e = 1_M$, $e_1{:=}\mu(u_1) = 1_M$ and $e_2{:=}\mu(u_2) = 1_M$. Then $\mathsf{prod}(mem_1e_1\langle\mathbf{u_1}\rangle e_1pe_2\langle\mathbf{u_2}\rangle e_2n_2en) = \mathsf{prod}(1_M\langle\mathbf{a}\rangle 1_M\langle\mathbf{b}\rangle 1_M) = 1$ and $\mathsf{prod}(mem_2e_2\langle\mathbf{u_2}\rangle e_2pe_1\langle\mathbf{u_1}\rangle e_1n_1en) = \mathsf{prod}(1_M\langle\mathbf{b}\rangle 1_M\langle\mathbf{a}\rangle 1_M) = 0$. Furthermore the equations of Definition 22 hold, thus $\mathsf{triangular\text{-}sum}$ is not computable by a k-blind bimachine.

Lemma 25 shows $1 \Rightarrow 3$ in Theorem 23. It allows to show that some function cannot be computed by a k-marble transducer. Its proof is technical; a coarse intuition is that a 1-blind bimachine which makes a production on u_1 when called from u_2 cannot see the monoid element m' between u_1 and u_2 (since u_2 is not marked, its position is "forgotten").

▶ **Lemma 25.** *If* f *is computable by a* k*-blind bimachine, then* $\mathcal{T}$ *is symmetrical.*

Since $2 \Rightarrow 1$ in Theorem 23 is obvious, it remains to show that if $\mathcal{T}$ is symmetrical, then f is effectively computable by a 1-blind bimachine. This is the goal of the two following subsections. The main tool for the proof is the notion of factorization forest: using Lemma 36, it allows us to compute the function f without directly referring to a machine.

5.1 Factorization forests

Recall that $\mu : A^+ \to M$ is a fixed monoid morphism. A *factorization forest* [1] of $w \in A^+$ is an unranked tree structure which decomposes w following the image of its factors by μ.

▶ **Definition 26** ([13, 1]). *A* factorization (forest) *of $w \in A^+$ is a tree defined as follows:*
- *if $w = a \in A$, it is a leaf a;*
- *if $|w| \geq 2$, then $(\mathcal{F}_1)\cdots(\mathcal{F}_n)$ is a factorization of w if each $\mathcal{F}_i$ is a factorization of some $w_i \in A^+$ such that $w = w_1 \cdots w_n$, and either:*
  - *$n = 2$: the root is a* binary node*;*
  - *or $n \geq 3$ and $\mu(w_1) = \cdots = \mu(w_n)$ is an idempotent: the root is an* idempotent node.

The set of factorizations over w is denoted $\mathsf{Fact}(w)$. Recall that $\Lambda = 3|M|$.

▶ **Theorem 27** ([13, 1]). *For all $w \in A^+$, there is $\mathcal{F} \in \mathsf{Fact}(w)$ of height at most Λ.*

Let $\widehat{A} := A \uplus \{(,)\}$. We have defined $\mathsf{Fact}(w)$ as a set of tree structures, but we can assume that $\mathsf{Fact}(w) \subseteq \widehat{A}^+$. Indeed, in Definition 26, a factorization of w can also be seen as "the word w with parentheses". There exists a rational function which computes factorizations, under this formalism. We reformulate this statement in Proposition 28 using a two-way transducer (which, exceptionally in this paper, has a non-unary output alphabet $\widehat{A}$).

▶ **Proposition 28** (Folklore). *One can build a two-way transducer which computes a function $A^+ \to \widehat{A}^+, w \mapsto \mathcal{F} \in \mathsf{Fact}(w)$ for some $\mathcal{F}$ of height at most Λ.*

We denote by $\mathsf{Nodes}(\mathcal{F})$ the set of (idempotent or binary) nodes of $\mathcal{F}$. In order to simplify the statements, we identify a node with the subtree rooted in this node. Thus $\mathsf{Nodes}(\mathcal{F})$ can also be seen as the set of subtrees of $\mathcal{F}$, and $\mathcal{F} \in \mathsf{Nodes}(\mathcal{F})$. We shall use the standard tree vocabulary of "height" (a leaf is a tree of height 1), "parent node", "descendant" and "ancestor" (defined in a non-strict way: a node is itself one of its ancestors), "branch", etc.

▶ **Example 29.** Let $A = \{a, b, c\}$, $M{=}\{1_M, 2_M, 3_M\}$ with $2_M^2{=}1_M$, 3_M absorbing, $\mu(a){:=}2_M$ and $\mu(b){:=}\mu(c){:=}3_M$. Then $\mathcal{F} := (aa)(bc(a(cbbcb))b) \in \mathsf{Fact}(aabcaccbcbcb)$ (we dropped the parens around single letters for more readability) is depicted in Figure 6. Idempotent nodes are drawn using a horizontal line.

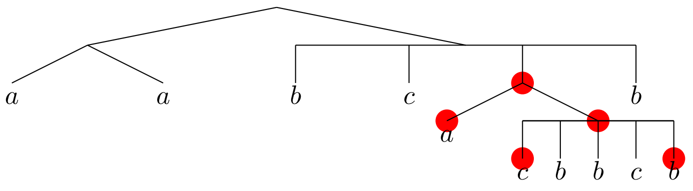


Figure 6 The factorization $(aa)(bc(a(cbbcb))b)$ of $aabcacbbcbb$.

We define $\mathsf{Iterable\text{-}nodes}(\mathcal{F}) \subseteq \mathsf{Nodes}(\mathcal{F})$ as the set of nodes which are the middle child of an idempotent node. Intuitively, such nodes can be copied without modifying their "context".

▶ **Definition 30.** *Let $\mathcal{F} \in \mathsf{Fact}(w)$, we define the set of iterable nodes of $\mathcal{F}$ by induction:*

- *if $\mathcal{F} = a \in A$ is a leaf,* $\mathsf{Iterable\text{-}nodes}(\mathcal{F}) := \varnothing$*;*
- *if $\mathcal{F} = (\mathcal{F}_1) \cdots (\mathcal{F}_n)$ is a binary or idempotent node, then:*

$$\mathsf{Iterable\text{-}nodes}(\mathcal{F}) := \{\mathcal{F}_i : 2 \leq i \leq n{-}1\} \biguplus_{1 \leq i \leq n} \mathsf{Iterable\text{-}nodes}(\mathcal{F}_i).$$

On the contrary, we now define sets of nodes which cannot be duplicated individually.

▶ **Definition 31.** *Let $\mathcal{F} \in \mathsf{Fact}(w)$, we define the* dependency *of $\mathcal{F}$ as follows:*

- *if $\mathcal{F} = a \in A$ is a leaf, then $\mathsf{Dep}(\mathcal{F}) := \{a\}$;*
- *if $\mathcal{F} = (\mathcal{F}_1) \cdots (\mathcal{F}_n)$ is binary or idempotent, then $\mathsf{Dep}(\mathcal{F}) := \{\mathcal{F}\} \cup \mathsf{Dep}(\mathcal{F}_1) \cup \mathsf{Dep}(\mathcal{F}_n)$.*

Intuitively, $\mathsf{Dep}(\mathcal{F}) \subseteq \mathsf{Nodes}(\mathcal{F})$ contains all the nodes of $\mathcal{F}$ except those which are descendant of a middle child. If $\mathcal{I} \in \mathsf{Nodes}(\mathcal{F})$, we consider $\mathsf{Dep}(\mathcal{I}) \subseteq \mathsf{Nodes}(\mathcal{I})$ as a subset of $\mathsf{Nodes}(\mathcal{F})$. We then define the frontier of $\mathcal{I}$, denoted $\mathsf{Fr}_{\mathcal{F}}(\mathcal{I}) \subseteq \{1, \dots, |w|\}$ as the set of positions of w which belong to $\mathsf{Dep}(\mathcal{I})$ (when seen as leaves of $\mathcal{F}$).

▶ **Example 32.** In Figure 6, the top-most red node $\mathcal{I}$ is iterable. Furthermore $\mathsf{Dep}(\mathcal{I})$ is the set of red nodes, $\mathsf{Fr}_{\mathcal{F}}(\mathcal{I}) = \{5, 6, 10\}$ and $w[\mathsf{Fr}_{\mathcal{F}}(\mathcal{I})] = acb$.

The relationship between iterable nodes and dependencies is detailed below. We denote by $\mathsf{Part}(\mathcal{F}) := \mathsf{Iterable\text{-}nodes}(\mathcal{F}) \uplus \{\mathcal{F}\}$, the set of iterable nodes plus the root.

▶ **Lemma 33.** *Let $\mathcal{F} \in \mathsf{Fact}(w)$, then $\{\mathsf{Dep}(\mathcal{I}) : \mathcal{I} \in \mathsf{Part}(\mathcal{F})\}$ is a partition of $\mathsf{Nodes}(\mathcal{F})$; and $\{\mathsf{Fr}_{\mathcal{F}}(\mathcal{I}) : \mathcal{I} \in \mathsf{Part}(\mathcal{F})\}$ is a partition of $\{1, \dots, |w|\}$.*

We define $\mathsf{prod}(i, j)$ in w as "the production performed in i when called from j".

▶ **Definition 34.** *Let $w \in A^+$ and $1 \leq i \leq j \leq |w|$ two positions of w. We define $\mathsf{prod}(i, j) \in \mathbb{N}$ as $\lambda_{\mathfrak{f}_j}(\mu(w[1{:}i{-}1]), w[i], \mu(w[i{+}1{:}j]))$, where $\mathfrak{f}_j := \lambda(\mu(w[1{:}j{-}1]), w[j], \mu(w[j{+}1{:}|w|]))$.*

We extend this definition to pairs of nodes: given $\mathcal{I}, \mathcal{J} \in \mathsf{Nodes}(\mathcal{F})$, we define $\mathsf{prod}(\mathcal{I}, \mathcal{J})$ "the sum of all productions performed in the frontier of $\mathcal{I}$, when called from the frontier of $\mathcal{J}$" as follows (we have to ensure that the calling positions are "on the right").

▶ **Definition 35.** *Let $w \in A^+$, $\mathcal{F} \in \mathsf{Fact}(w)$ and $\mathcal{I}, \mathcal{J} \in \mathsf{Nodes}(\mathcal{F})$. We define:*

$$\mathsf{prod}(\mathcal{I}, \mathcal{J}) := \sum_{\substack{i \in \mathsf{Fr}_{\mathcal{F}}(\mathcal{I}) \\ j \in \mathsf{Fr}_{\mathcal{F}}(\mathcal{J}) \\ i \leq j}} \mathsf{prod}(i, j) \in \mathbb{N}.$$

If $\mathcal{I}$ is an ancestor of $\mathcal{J}$ (or the converse) then $\mathsf{Fr}_{\mathcal{F}}(\mathcal{I})$ and $\mathsf{Fr}_{\mathcal{F}}(\mathcal{J})$ are interleaved, hence we can have both $\mathsf{prod}(\mathcal{I}, \mathcal{J}) \neq 0$ and $\mathsf{prod}(\mathcal{J}, \mathcal{I}) \neq 0$. However, if $\mathcal{I}$ and $\mathcal{J}$ are not on the same branch, we have either $\mathsf{prod}(\mathcal{I}, \mathcal{J}) = 0$ or $\mathsf{prod}(\mathcal{J}, \mathcal{I}) = 0$.

Applying Lemma 33, it is not hard to compute $f(w)$ using the $\mathsf{prod}(\mathcal{I}, \mathcal{J})$.

▶ **Lemma 36.** *Let $w \in A^+$, $\mathcal{F} \in \mathsf{Fact}(w)$. Then:*

$$f(w) = \sum_{\mathcal{I}, \mathcal{J} \in \mathsf{Part}(\mathcal{F})} \mathsf{prod}(\mathcal{I}, \mathcal{J}).$$

5.2 Typology of pairs of nodes

We intend to compute (if possible) f using a 1-blind transducer. Following Lemma 36, it is enough to consider the productions performed on the pairs of nodes of a factorization. For this study, we split the pairs depending on their relative position in the tree.

Pairs separated by the frontier of the root. The frontier of the root $\mathsf{Fr}_{\mathcal{F}}(\mathcal{F})$ plays a very specific role with respect to blind transducers. Indeed, over factorizations of height at most Λ, the size of the frontier is bounded, hence it splits the word in a bounded number of distinguishable "blocks". Formally, we define the notion of *basis*.

▶ **Definition 37.** *An idempotent node is a* basis *if it belongs to the dependency of the root.*

The following result is shown by induction.

▶ **Lemma 38.** *Let $w \in A^+$ and $\mathcal{F} \in \mathsf{Fact}(w)$. Given $\mathcal{I} \in \mathsf{Iterable\text{-}nodes}(\mathcal{F})$, there exists a unique basis, denoted $\mathsf{basis}_{\mathcal{F}}(\mathcal{I})$, such that $\mathcal{I}$ is the descendant of a middle child of $\mathsf{basis}_{\mathcal{F}}(\mathcal{I})$.*

▶ **Definition 39.** *Given $w \in A^+$ and $\mathcal{F} \in \mathsf{Fact}(w)$, we define $D(\mathcal{F}) \subseteq \mathsf{Part}(\mathcal{F}) \times \mathsf{Part}(\mathcal{F})$ by:*

$$D(\mathcal{F}) := \{(\mathcal{I},\mathcal{J}) : \mathcal{I},\mathcal{J} \in \mathsf{Iterable\text{-}nodes}(\mathcal{F}) \text{ and } \mathsf{basis}_{\mathcal{F}}(\mathcal{I}) \neq \mathsf{basis}_{\mathcal{F}}(\mathcal{J})\}.$$

Intuitively $\mathsf{basis}_{\mathcal{F}}(\mathcal{I}) \neq \mathsf{basis}_{\mathcal{F}}(\mathcal{J})$ means that $\mathsf{Fr}_{\mathcal{F}}(\mathcal{I})$ and $\mathsf{Fr}_{\mathcal{F}}(\mathcal{J})$ belong to two different "blocks" of the input. Lemma 40 is shown by building a 1-blind bimachine which visits successively each basis $\mathcal{B}$, and for each iterable $\mathcal{J}$ such that $\mathsf{basis}_{\mathcal{F}}(\mathcal{J}) = \mathcal{B}$, calls a submachine which visits the $\mathcal{I}$ such that $\mathsf{basis}_{\mathcal{F}}(\mathcal{I}) \neq \mathcal{B}$ and produces $\mathsf{prod}(\mathcal{I},\mathcal{J})$. The key element for doing this operation without pebbles is that the number of bases is bounded.

▶ **Lemma 40.** *One can build a 1-blind bimachine computing:*

$$f_D : (\widehat{A})^+ \to \mathbb{N}, \mathcal{F} \mapsto \begin{cases} \displaystyle\sum_{(\mathcal{I},\mathcal{J}) \in D(\mathcal{F})} \mathsf{prod}(\mathcal{I},\mathcal{J}) & \text{if } \mathcal{F} \text{ factorization of height at most } \Lambda; \\ 0 & \text{otherwise.} \end{cases}$$

Linked pairs. Let $U(\mathcal{F}) := \mathsf{Part}(\mathcal{F}) \times \mathsf{Part}(\mathcal{F}) \smallsetminus D(\mathcal{F})$, it corresponds to the pairs of $\mathsf{Iterable\text{-}nodes}(\mathcal{F})$ which have the same basis, plus all the pairs $(\mathcal{F},\mathcal{I})$ and $(\mathcal{I},\mathcal{F})$ for $\mathcal{I} \in \mathsf{Part}(\mathcal{F})$. We now study the pairs of $U(\mathcal{F})$ which are "linked", in the sense that one node is (nearly) the ancestor of the other.

▶ **Definition 41.** *Let $w \in A^+$, $\mathcal{F} \in \mathsf{Fact}(w)$. Let $L(\mathcal{F})$ be the set of all $(\mathcal{I},\mathcal{J}) \in U(\mathcal{F})$ such that $\mathcal{I}$ (or $\mathcal{J}$) is either the ancestor of, or the right/left sibling of an ancestor of $\mathcal{J}$ (or $\mathcal{I}$).*

In particular, we have $(\mathcal{F},\mathcal{F}), (\mathcal{I},\mathcal{F}), (\mathcal{F},\mathcal{I}), (\mathcal{I},\mathcal{I}) \in L(F)$ for all $\mathcal{I} \in \mathsf{Part}(\mathcal{F})$. If $\mathcal{F}$ has height at most Λ, there are at most 3Λ nodes which are either an ancestor or the right/left sibling of an ancestor of $\mathcal{I}$. Lemma 42 follows from this boundedness.

▶ **Lemma 42.** *One can build a 0-blind bimachine computing:*

$$f_L : (\widehat{A})^+ \to \mathbb{N}, \mathcal{F} \mapsto \begin{cases} \displaystyle\sum_{(\mathcal{I},\mathcal{J}) \in L(\mathcal{F})} \mathsf{prod}(\mathcal{I},\mathcal{J}) & \text{if } \mathcal{F} \text{ factorization of height at most } \Lambda; \\ 0 & \text{otherwise.} \end{cases}$$

Independent nodes. The remaining sum is the most interesting, since it is the only case where we use the assumption that $\mathcal{T}$ to be symmetrical (and this assumption is crucial). Let $\mathcal{F} \in \mathsf{Fact}(w)$, we define the set $I(\mathcal{F}) := U(\mathcal{F}) \smallsetminus L(\mathcal{F})$. It contains the pairs $(\mathcal{I},\mathcal{J})$ of iterable nodes such that $\mathsf{basis}_{\mathcal{F}}(\mathcal{I}) = \mathsf{basis}_{\mathcal{F}}(\mathcal{J})$ (i.e. they descend from a common "big" idempotent), and $\mathcal{I}$ (or $\mathcal{J}$) is not an ancestor of $\mathcal{J}$ (or $\mathcal{I}$), nor the left or right sibling of its ancestor.

▶ **Lemma 43.** *If $\mathcal{T}$ is symmetrical, one can build a 1-blind bimachine computing:*

$$f_I : (\widehat{A})^+ \to \mathbb{N}, \mathcal{F} \mapsto \begin{cases} \sum_{(\mathcal{I},\mathcal{J}) \in I(\mathcal{F})} \mathsf{prod}(\mathcal{I},\mathcal{J}) \text{ if } \mathcal{F} \text{ factorization of height at most } \Lambda; \\ 0 \text{ otherwise.} \end{cases}$$

Proof idea. We define $\mathsf{type}_{\mathcal{F}}(\mathcal{I})$ for $\mathcal{I} \in \mathsf{Iterable\text{-}nodes}(\mathcal{F})$ as a bounded abstraction of $\mathcal{I}$ which describes the frontier and the location of $\mathcal{I}$ in $\mathcal{F}$ and in $\mathsf{basis}_{\mathcal{F}}(\mathcal{I})$. Using symmetry, we show that for $(\mathcal{I},\mathcal{J}) \in I(\mathcal{F})$, $\mathsf{prod}(\mathcal{I},\mathcal{J})$ only depends on $\mathsf{type}_{\mathcal{F}}(\mathcal{I})$ and $\mathsf{type}_{\mathcal{F}}(\mathcal{J})$, but not on their relative positions. Hence we build a 1-blind bimachine, whose main bimachine ranges over all possible $\mathcal{J}$ and computes $\mathsf{type}_{\mathcal{F}}(\mathcal{J})$, and whose submachines range over all possible $\mathcal{I}$ (a special treatment has to be done to avoid $\mathcal{I}$ such that $(\mathcal{I},\mathcal{J}) \in L(\mathcal{F})$), compute $\mathsf{type}_{\mathcal{F}}(\mathcal{I})$ and output $\mathsf{prod}(\mathcal{I},\mathcal{J})$. The submachines do not need to "see" $\mathcal{J}$. ◀

We finally show 3 ⇒ 2 in Theorem 23. Given $w \in A^+$ we first compute $\mathcal{F}$ of height at most Λ by Proposition 28. Then we use the machines from Lemmas 40, 42 and 43 and build a 1-blind transducer computing the sum of their outputs. The original function can be recovered since 1-blind transducers are closed under composition with two-way [11].

6 Conclusion and outlook

As a conclusion, we discuss future work. This paper introduces new proof techniques, in particular the use of factorization forests to study the productions of transducers. We believe that these techniques give a step towards other membership problems concerning pebble transducers. Among them, let us mention the membership problem from k-pebble to k-blind, at first over unary alphabets. Similarly, the membership from k-pebble to k-marble over non-unary alphabets is worth being studied (the answer seems to rely on combinatorial properties of the output, since unary outputs can always be produced using marbles).

References

1 Mikołaj Bojańczyk. Factorization forests. In *International Conference on Developments in Language Theory*, pages 1–17. Springer, 2009.

2 Mikołaj Bojańczyk. Transducers with origin information. In *International Colloquium on Automata, Languages, and Programming*, pages 26–37. Springer, 2014.

3 Mikolaj Bojańczyk. Polyregular functions. *arXiv preprint*, 2018. `arXiv:1810.08760`.

4 Mikolaj Bojańczyk, Sandra Kiefer, and Nathan Lhote. String-to-string interpretations with polynomial-size output. In *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019*, pages 106:1–106:14, 2019.

5 Michal P Chytil and Vojtěch Jákl. Serial composition of 2-way finite-state transducers and simple programs on strings. In *4th International Colloquium on Automata, Languages, and Programming, ICALP 1977*, pages 135–147. Springer, 1977.

6 Gaëtan Douéneau-Tabot, Emmanuel Filiot, and Paul Gastin. Register transducers are marble transducers. In *45th International Symposium on Mathematical Foundations of Computer Science, MFCS 2020, August 24–28, 2020, Prague, Czech Republic*, 2020.

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

8 Emmanuel Filiot and Pierre-Alain Reynier. Copyful streaming string transducers. In *International Workshop on Reachability Problems*, pages 75–86. Springer, 2017.

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

10 Nathan Lhote. Pebble minimization of polyregular functions. In *35th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS*. IEEE, 2020.

11 Lê Thành Dung Nguyên, Camille Noûs, and Pierre Pradic. Comparison-free polyregular functions. In *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021*, 2021.

12 John C Shepherdson. The reduction of two-way automata to one-way automata. *IBM Journal of Research and Development*, 3(2):198–200, 1959.

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

A Proof of Lemma 17

We show that given a bimachine with external pebble functions, which are computed by SSTs, one can build an equivalent SST.

A.1 SST with lookaround

We first define a variant of SST with the same expressive power. Intuitively, this model is similar to bimachines, in the sense that the register update not only depends on the current letter, but also on a finite abstraction of the prefix and suffix.

▶ **Definition 44.** *An* SST with lookaround $\mathcal{T} = (A, \mathfrak{X}, M, \mu, I, \lambda, F)$ *is:*

- *an input alphabet A and a finite set $\mathfrak{X}$ of registers;*
- *a morphism into a finite monoid $\mu : A^* \to M$;*
- *an initial row vector $I \in \mathbb{N}^{\mathfrak{X}}$;*
- *a register update function $\lambda : M \times A \times M \to \mathbb{N}^{\mathfrak{X} \times \mathfrak{X}}$;*
- *an output column vector $F \in \mathbb{N}^{\mathfrak{X}}$.*

Let us define its semantics. Intuitively, in position i of $w \in A^+$, we perform the register update $\lambda(\mu(w[1{:}i{-}1]), w[i],]\mu(w[i{+}1{:}|w|]))$. Formally, for $0 \leq i \leq |w|$, we define $\mathcal{T}^{w,i} \in \mathbb{N}^{\mathfrak{X}}$ ("the values of the registers after reading $w[1{:}i]$"[2]) as follows:

- $\mathcal{T}^{w,0} := I$;
- for $i \geq 1$, $\mathcal{T}^{w,i} := \mathcal{T}^{w,i-1} \times \lambda(\mu(w[1{:}i{-}1]), w[i],]\mu(w[i{+}1{:}|w|]))$.

To define the function $f : A^+ \to \mathbb{N}$ computed by $\mathcal{T}$, we combine the final values by the output vector:

$$f(w) := \mathcal{T}^{w,|w|} F.$$

It is known that SST with lookaround are equivalent to SST (the proof is roughly a "determinisation" procedure for eliminating the rightmost argument of λ, and an encoding of the monoid in the registers for eliminating the leftmost argument).

▶ **Lemma 45** ([8]). *Given an SST with lookaround, we can build an equivalent SST.*

Hence, it is sufficient to build an SST with lookaround.

[2] Due to the fact that λ looks "on the right", $\mathcal{T}^{w,i}$ depends on the whole w and not only on $w[1{:}i]$.

A.2 Main proof of Lemma 17

Let $\mathcal{T} = (A, M, \mu, \lambda, \mathfrak{F})$ be the bimachine with external pebble functions. Each $\mathfrak{f} \in \mathfrak{F}$ is computed by an SST $\mathcal{T}_\mathfrak{f} := (A \uplus \underline{A}, \mathfrak{X}_\mathfrak{f}, I_\mathfrak{f}, T_\mathfrak{f}, F_\mathfrak{f})$.

▶ **Example 46** (Running example)**.** Let $\mathcal{T} = (A, M, \mu, \lambda, \{\mathfrak{f}\})$ with $M = \{1_M\}$ singleton and $\lambda(1_M, a, 1_M) = \mathfrak{f}$ for $a \in A$. Let $\mathcal{T}_\mathfrak{f}$ have two registers x, y with x initialized to 1 and y initialized to 0. When reading $a \in A \uplus \underline{A}$ it performs $x \leftarrow x, y \leftarrow y + x$. Finally it outputs y.

Then $\mathfrak{f}(w) = |w|$ and $\mathcal{T}$ computes $f : w \mapsto |w|^2$.

▶ **Definition 47.** *Let $w \in A^*$ and $\mathfrak{f} \in \mathfrak{F}$. We define* $\mathsf{Calls}(w, \mathfrak{f})$ *as the set of positions of w in which $\mathfrak{f}$ is called, that is $\{1 \leq j \leq |w| : \lambda(\mu(w[1{:}j{-}1]), w[j], \mu(w[j{+}1{:}|w|]) = \mathfrak{f}\}$.*

For $1 \leq j \leq i \leq |w|$, $I_\mathfrak{f} T_\mathfrak{f}(w[1{:}i]\uparrow j)$ corresponds to the value of the registers of $\mathcal{T}_\mathfrak{f}$ in position i when the call to $\mathfrak{f}$ is performed from position j.

▷ Claim 48. For all $w \in A^*$, the following holds:

$$f(w) = \sum_{\mathfrak{f} \in \mathfrak{F}} \sum_{\substack{1 \leq j \leq |w| \\ j \in \mathsf{Calls}(w,\mathfrak{f})}} I_\mathfrak{f} T_\mathfrak{f}(w\uparrow j) F_\mathfrak{f}$$

Proof. By definition of external pebble functions we have:

$$f(w) := \sum_{1 \leq j \leq |w|} \mathfrak{f}_j(w\uparrow j)$$

where $\mathfrak{f}_j := \lambda(\mu(w[1{:}j{-}1]), w[j], \mu(w[j{+}1{:}|w|]))$ is "the external function called in j". Hence by partitioning the sum depending on the external functions it follows:

$$f(w) := \sum_{\mathfrak{f} \in \mathfrak{F}} \sum_{\substack{1 \leq j \leq |w| \\ j \in \mathsf{Calls}(w,\mathfrak{f})}} \mathfrak{f}(w\uparrow j)$$

And finally we note that $\mathfrak{f}(w\uparrow j) = I_\mathfrak{f} T_\mathfrak{f}(w\uparrow j) F_\mathfrak{f}$. ◁

Idea of the construction. Let us fix an external function $\mathfrak{f}$. Following Claim 48, we want to build an SST with lookaround $\mathcal{U}$ which computes the values of the vector:

$$\sum_{\substack{1 \leq j \leq |w| \\ j \in \mathsf{Calls}(w,\mathfrak{f})}} I_\mathfrak{f} T_\mathfrak{f}(w\uparrow j) \in \mathbb{N}^{\mathfrak{X}}.$$

For this, it will keep track when in position i of the values of:

$$\sum_{\substack{1 \leq j \leq i \\ j \in \mathsf{Calls}(w,\mathfrak{f})}} I_\mathfrak{f} T_\mathfrak{f}(w[1{:}i]\uparrow j) \in \mathbb{N}^{\mathfrak{X}}$$

and these values will be updated when going from i to $i + 1$.

▶ **Example 49** (Running example)**.** We have $I_\mathfrak{f} T_\mathfrak{f}(w[1{:}i]\uparrow j)(x) = 1$ and $I_\mathfrak{f} T_\mathfrak{f}(w[1{:}i]\uparrow j)(y) = i$.

Hence $\displaystyle\sum_{\substack{1 \leq j \leq i \\ j \in \mathsf{Calls}(w,\mathfrak{f})}} I_\mathfrak{f} T_\mathfrak{f}(w[1{:}i]\uparrow j)(x) = i$ and $\displaystyle\sum_{\substack{1 \leq j \leq i \\ j \in \mathsf{Calls}(w,\mathfrak{f})}} I_\mathfrak{f} T_\mathfrak{f}(w[1{:}i]\uparrow j)(y) = i^2$.

Formal construction. Let $\mathcal{U} = (A, \mathfrak{X}, M, \mu, I, \kappa, F)$ be an SST with lookaround with:

- the set $\mathfrak{X} := \biguplus_{\mathfrak{f} \in \mathfrak{F}} \{\mathsf{Sum}_x : x \in \mathfrak{X}_\mathfrak{f}\} \uplus \{\mathsf{Old}_x : x \in \mathfrak{X}_\mathfrak{f}\}$ of registers;
- the morphism $\mu : A^* \to M$ used in $\mathcal{T}$;
- an initial column vector $I \in \mathbb{N}^{\mathfrak{X}}$ such that for all $\mathfrak{f} \in \mathfrak{F}$ and $x \in \mathfrak{X}_\mathfrak{f}$:
  - $I(\mathsf{Sum}_x) = 0$;
  - $I(\mathsf{Old}_x) = I_\mathfrak{f}(x)$;
- the update $\kappa : M \times A \times M \to \mathbb{N}^{\mathfrak{X} \times \mathfrak{X}}$ as follows. Let $(m, a, n) \in M \times A \times N$ and $\mathfrak{f} := \lambda(m, a, n)$. Then $\kappa(m, a, n)$ performs the following updates:
  - for all $\mathfrak{g} \neq \mathfrak{f}$ and $x \in \mathfrak{X}_\mathfrak{g}$:
    * $\mathsf{Sum}_x \leftarrow \sum_{y \in \mathfrak{X}_\mathfrak{g}} \alpha_y \, \mathsf{Sum}_y$;
    * $\mathsf{Old}_x \leftarrow \sum_{y \in \mathfrak{X}_\mathfrak{g}} \alpha_y \, \mathsf{Old}_y$;

    where $x \leftarrow \sum_{y \in \mathfrak{X}_\mathfrak{g}} \alpha_y y$ is the update performed by $\mathcal{T}_\mathfrak{g}$ when reading a;
  - for all $x \in \mathfrak{X}_\mathfrak{f}$:
    * $\mathsf{Sum}_x \leftarrow \sum_{y \in \mathfrak{X}_\mathfrak{f}} \alpha_y \, \mathsf{Sum}_y + \sum_{y \in \mathfrak{X}_\mathfrak{f}} \beta_y \, \mathsf{Old}_y$;
    * $\mathsf{Old}_x \leftarrow \sum_{y \in \mathfrak{X}_\mathfrak{f}} \alpha_y \, \mathsf{Old}_y$;

    where $x \leftarrow \sum_{y \in \mathfrak{X}_\mathfrak{f}} \alpha_y y$ is the update performed by $\mathcal{T}_\mathfrak{f}$ when reading a;
    and $x \leftarrow \sum_{y \in \mathfrak{X}_\mathfrak{f}} \beta_y y$ is the update performed by $\mathcal{T}_\mathfrak{f}$ when reading $\underline{a}$.
    Intuitively, the sum with the β_y corresponds to what is "added" by the new call to $\mathfrak{f}$.
- the output line vector $F \in \mathbb{N}^{\mathfrak{X}}$ such that for all $\mathfrak{f} \in \mathfrak{F}$ and $x \in \mathfrak{X}_\mathfrak{f}$:
  - $F(\mathsf{Sum}_x) = F_\mathfrak{f}(x)$;
  - $F(\mathsf{Old}_x) = 0$.

▶ **Example 50** (Running example). $\mathcal{U}$ performs the following updates:

- $\mathsf{Old}_x \leftarrow \mathsf{Old}_x$, $\mathsf{Old}_y \leftarrow \mathsf{Old}_y + \mathsf{Old}_x$;
- $\mathsf{Sum}_x \leftarrow \mathsf{Sum}_x + \mathsf{Old}_x$, $\mathsf{Sum}_y \leftarrow \mathsf{Sum}_y + \mathsf{Sum}_x + \mathsf{Old}_y + \mathsf{Old}_x$.

We can check that $\mathcal{U}^{w,i}(\mathsf{Old}_x) = 1$, $\mathcal{U}^{w,i}(\mathsf{Old}_y) = i$ and $\mathcal{U}^{w,i}(\mathsf{Sum}_x) = i$, $\mathcal{U}^{w,i}(\mathsf{Sum}_y) = i^2$.

Correctness of the construction. As the registers Old_x for $x \in \mathfrak{X}_\mathfrak{f}$ are updated following the updates of $\mathcal{T}_\mathfrak{f}$, it follows immediately that:

▷ Claim 51. Given $x \in \mathfrak{X}_\mathfrak{f}$, for all $w \in A^+$ and $1 \leq i \leq |w|$ we have:

$$\mathcal{U}^{w,i}(\mathsf{Old}_x) = I_\mathfrak{f} T_\mathfrak{f}(w[1{:}i])(x).$$

We can finally show that the registers Sum_x store the information we wanted.

▷ Claim 52. Given $x \in \mathfrak{X}_\mathfrak{f}$, for all $w \in A^+$ and $0 \leq i \leq |w|$ we have:

$$\mathcal{U}^{w,i}(\mathsf{Sum}_x) = \sum_{\substack{1 \leq j \leq i \\ j \in \mathsf{Calls}(w,\mathfrak{f})}} I_\mathfrak{f} T_\mathfrak{f}(w[1{:}i]{\uparrow}j)(x).$$

Proof. We proceed by induction on $0 \leq i \leq |w|$. For $i = 0$ both terms equal 0. For the induction step with $i \geq 1$ let $\mathfrak{f} \in \mathfrak{F}$ and $x \in \mathfrak{X}_\mathfrak{f}$.

Suppose that $\lambda(\mu(w[1{:}i{-}1]), w[i], \mu(w[i{+}1{:}|w|])) = \mathfrak{f}$ (the case when they differ is similar and even easier), then:

$$\sum_{\substack{1 \leq j \leq i \\ j \in \mathsf{Calls}(w,\mathfrak{f})}} I_\mathfrak{f} T_\mathfrak{f}(w[1{:}i]{\uparrow}j)(x) = I_\mathfrak{f} T_\mathfrak{f}(w[1{:}i]{\uparrow}i)(x) + \sum_{\substack{1 \leq j \leq i-1 \\ j \in \mathsf{Calls}(w,\mathfrak{f})}} I_\mathfrak{f} T_\mathfrak{f}(w[1{:}i]{\uparrow}j)(x). \tag{1}$$

- Let $x \leftarrow \sum_{y \in \mathfrak{X}_\mathfrak{f}} \alpha_y y$ be the update performed by $\mathcal{T}_\mathfrak{f}$ when reading a.
  Then for $j \leq i-1$, $I_\mathfrak{f} T_\mathfrak{f}(w[1{:}i]{\uparrow}j)(x) = \sum_{y \in \mathfrak{X}_\mathfrak{f}} \alpha_y \times I_\mathfrak{f} T_\mathfrak{f}(w[1{:}i{-}1]{\uparrow}j)(y)$.
- Let $x \leftarrow \sum_{y \in \mathfrak{X}_\mathfrak{f}} \beta_y y$ be the update performed by $\mathcal{T}_\mathfrak{f}$ when reading $\underline{a}$.
  Then $I_\mathfrak{f} T_\mathfrak{f}(w[1{:}i]{\uparrow}i)(x) = \sum_{y \in \mathfrak{X}_\mathfrak{f}} \beta_y \times I_\mathfrak{f} T_\mathfrak{f}(w[1{:}i{-}1])(y)$.

Hence we can rewrite Equation 1 using values in position $i{-}1$. By Claim 51 and the induction hypothesis, this sum coincides with the update in $\mathcal{U}$. $\lhd$

The fact that $\mathcal{U}$ computes f follows from the definition of F and Claim 48.