Algebraic Recognition of Regular Functions

Mikolaj Bojańczyk, Lê Thành Dũng Nguyễn

Algebraic Recognition of Regular Functions

Mikołaj Bojańczyk
Institute of Informatics, University of Warsaw, Poland

Lê Thành Dũng (Tito) Nguyễn
Laboratoire de l'informatique du parallélisme (LIP), École normale supérieure de Lyon, France

Abstract

We consider regular string-to-string functions, i.e. functions that are recognized by copyless streaming string transducers, or any of their equivalent models, such as deterministic two-way automata. We give yet another characterization, which is very succinct: finiteness-preserving functors from the category of semigroups to itself, together with a certain output function that is a natural transformation.

2012 ACM Subject Classification Theory of computation → Transducers

Keywords and phrases string transducers, semigroups, category theory

Category Full version with appendix of an ICALP 2023 Track B paper

Related Version *Conference proceedings*: https://doi.org/10.4230/LIPIcs.ICALP.2023.117

Funding *Lê Thành Dũng (Tito) Nguyễn*: Supported by the LABEX MILYON (ANR-10-LABX-0070) of Université de Lyon, within the program "Investissements d'Avenir" (ANR-11-IDEX-0007) operated by the French National Research Agency (ANR).

Acknowledgements We thank the reviewers for helpful suggestions on the presentation of the paper.

1 Introduction

This paper is about the regular string-to-string functions (see e.g. [17]). This is a fundamental class of functions; it is one of the standard generalizations of regular languages to produce string outputs (instead of merely accepting or rejecting inputs), covering examples such as

string reversal: $123 \mapsto 321$ duplication: $123 \mapsto 123123$

It has many equivalent descriptions, including deterministic two-way automata [22, Note 4], copyless streaming string transducers (SST) [1, Section 3] (or the earlier and very similar single-use restricted macro tree transducers [14, Section 5]), MSO transductions [13, Theorem 13], combinators [4, Section 2], a functional programming language [8, Section 6], λ-calculus with linear types [15, Theorem 3] (see also [19, Claim 6.2] and [18, Theorem 1.2.3]), decompositions *à la* Krohn–Rhodes [10, Theorem 18, item 4], etc.

The number of equivalent characterizations clearly indicates that the class of regular functions is important and worth studying. However, from a mathematical point of view, a disappointing phenomenon is that each of the known descriptions uses syntax that is more complicated than one could wish for.[1] These complications are perhaps minor annoyances, and the corresponding models are undeniably useful. Nevertheless, it would be desirable to have a model with a short and abstract definition, similar to the definition of recognizability of regular languages by finite semigroups.

[1] For example, the definition of a two-way automaton requires a discussion of endmarkers and what happens when the automaton loops. In an MSO transduction, an unwiedly copying mechanism is necessary. In an SST, one needs to be careful about bounding the copies among registers. The calculi of [4, 8] both have a long list of primitives. Similar remarks apply to the other formalisms.

This paper proposes such an abstract model. We prove that the regular string-to-string functions are exactly those that can be obtained by composing two functions

$$\Sigma^* \xrightarrow{\text{some semigroup homomorphism}} \mathsf{F}(\Gamma^*) \xrightarrow{\text{out}_{\Gamma^*}} \Gamma^*$$

where F is a *functor* from the category of semigroups to itself that maps finite semigroups to finite semigroups, and the output function out_{Γ^*} – not necessarily a homomorphism – is part of a family $\text{out}_A \colon \mathsf{F}A \to A$ that is natural in the semigroup A.

We use the name transducer semigroup for the model implicit in this description, i.e. a semigroup-to-semigroup functor F together with a natural transformation for producing outputs. One of the surprising features of this model is the fact that linear growth of the output size, which is one of the salient properties of the regular string-to-string functions, does not seem to be a trivial consequence of the definition.

2 Transducer semigroups and warm-up theorems

In this section, we define the model that is introduced in this paper, namely transducer semigroups. The purpose of this model is to recognize *string-to-string* functions, which are defined to be functions of type $\Sigma^* \to \Gamma^*$, for some finite alphabets Σ and Γ. Some results will work in the slightly more general case where the domain or codomain is a more general semigroup, but we focus on the string-to-string case for the sake of concreteness.

The model is defined using terminology based on category theory. However, we do not assume that the reader has a background in category theory, beyond the two most basic notions of category and functor. Recall that a *category* consists of objects with morphisms between them, such that the morphisms can be composed and each object has an identity morphism to itself. In this paper, we will be working mainly with two categories:

Sets. The objects are sets, the morphisms are functions between them.

Semigroups. The objects are semigroups, the morphisms are semigroup homomorphisms.

To transform categories, we use functors. Recall that a *functor* between two categories consists of two maps: one that assigns to each object A in the source category an object in the target category, and another one that assigns to each morphism $f \colon A \to B$ a morphism $\mathsf{F}f \colon \mathsf{F}A \to \mathsf{F}B$. These maps need to be consistent with composition of morphisms, and the identity must go to the identity.

▶ **Example 2.1.** The *forgetful functor* from the category of semigroups to the category of sets maps a semigroup to its underlying set, and a semigroup homomorphism to the corresponding function on sets. It is an example of a semigroup-to-set functor.

▶ **Example 2.2.** These constructions can be seen as semigroup-to-semigroup functors:

Tuples. This functor maps a semigroup A to its square $A \times A$, with the semigroup operation defined coordinate-wise. The functor extends to morphisms in the expected way. This functor also makes sense for higher powers, including infinite powers, such as A^ω.

Opposite. This functor maps a semigroup A to the semigroup where the underlying set is the same, but multiplication is reversed, i.e. the product of a and b in the new semigroup is the product b and a in the old semigroup. Morphisms are not changed by the functor: they retain the homomorphism property despite the change in the multiplication operation.

Lists. This functor maps a semigroup A to the free monoid A^* that consists of lists of elements of A equipped with concatenation. (When A is finite, it can be regarded as an alphabet, in which case we shall also call these lists "strings" or "words".) On morphisms, the functor is defined element-wise (or letter-wise). A similar construction would make sense as a set-to-semigroup functor.

Non-empty lists. A variant of the previous example, which sends a semigroup A to the free semigroup A^+ that consists of non-empty lists of elements in A.

Powerset. This (covariant) powerset functor maps a semigroup A to the powerset semigroup $\mathsf{P}A$, whose underlying set is the family of all subsets of A, endowed with the operation

$$(A_1, A_2) \quad \mapsto \quad \{a_1 a_2 \mid a_1 \in A_1 \text{ and } a_2 \in A_2\}.$$

We now present the central definition of this paper.

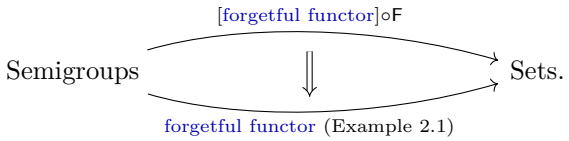
▶ **Definition 2.3** (Transducer semigroup). *A* transducer semigroup *consists of*

- *a semigroup-to-semigroup functor* F*,*
- *together with an* output mechanism *which associates to each semigroup* A *a function of type* $\mathsf{F}A \to A$ *called the* output function for A*,*

such that for every homomorphism $h\colon A \to B$*, the diagram below commutes:*

$$\begin{array}{ccc} \mathsf{F}A & \xrightarrow{\mathsf{F}h} & \mathsf{F}B \\ \downarrow{\scriptstyle \text{output function for } A} & & \downarrow{\scriptstyle \text{output function for } B} \\ A & \xrightarrow[h]{} & B \end{array}$$

In the language of category theory, a *natural transformation* between two semigroup-to-set functors G and K is a family of functions $f_A\colon \mathsf{G}A \to \mathsf{K}A$ such that $f_B \circ \mathsf{G}h = \mathsf{K}h \circ f_A$ for every semigroup homomorphism $h\colon A \to B$. So in Definition 2.3, the diagram says that the output mechanism is a natural transformation of type



Note that in a transducer semigroup, the output functions *are not necessarily homomorphisms*, which is why the forgetful semigroup-to-set functor appears above. This is important in view of the purpose of transducer semigroups, which is to define functions between semigroups, as explained in the following definition; asking out_B to be a homomorphism would severly restrict the functions that can be recognized.

▶ **Definition 2.4.** *We say that a function* $f\colon A \to B$ *between semigroups, not necessarily a homomorphism, is* recognized *by a transducer semigroup* $(\mathsf{F}, \mathrm{out})$ *if it can be decomposed as*

$$A \xrightarrow{h} \mathsf{F}B \xrightarrow{\mathrm{out}_B} B \qquad \text{for some semigroup homomorphism } h.$$

The definition discusses functions between arbitrary semigroups, with no assumption on F, but we will mainly care about the special case – treated in Section 3 – where:

- the function is string-to-string[2] ($f\colon \Sigma^* \to \Gamma^*$), i.e. both the input and output semigroups are finitely generated free monoids;
- the functor F is *finiteness-preserving*, i.e. it maps finite semigroups to finite semigroups.

This special case will correspond to the regular string functions. Some minor results that do not assume F is finiteness-preserving are presented in Section 2.2: we characterize all functions (Theorem 2.10) and "recognizability reflecting" string-to-string functions (Theorem 2.13).

[2] Although this case involves monoids, which are a special case of semigroups, the use of a semigroup homomorphism that is not necessarily a monoid homomorphism is required to recognize functions such that $f(\varepsilon) \neq \varepsilon$. Furthermore, it will be useful in the proofs to work in the category of semigroups, rather than the category of monoids.

2.1 Examples and intuitions

▶ **Example 2.5.** Consider the transducer semigroup in which the functor is the identity, and the output mechanism is also the identity. The functions of type $A \to B$ that are recognized by this transducer semigroup are exactly the semigroup homomorphisms from A to B.

▶ **Example 2.6.** Consider the transducer semigroup in which

- the functor is the "opposite semigroup" functor from Example 2.2;
- the output function maps $a \in \mathsf{F}A$, seen as an element in A, to $aa \in A$.

The functions of type $A \to B$ that are recognized by this transducer semigroup are exactly those of the form $a \mapsto h(a)h(a)$ where $h\colon A \to B$ is some "anti-homomorphism", i.e. satisfies $h(a_1a_2) = h(a_2)h(a_1)$ for all $a_1, a_2 \in A$. In particular, if h is the string reversal function $\texttt{rev}$ on the free monoid Σ^*, which is also a semigroup, then we get the "reverse then duplicate" function that maps a string w over the alphabet Σ to $\texttt{rev}(w) \cdot \texttt{rev}(w)$.

▶ **Example 2.7.** We present here a transducer semigroup that recognizes the squaring function $w \in \Sigma^* \mapsto w^{|w|} \in \Sigma^*$ (illustrated by $\texttt{123} \mapsto \texttt{123123123}$) for any alphabet Σ.

The functor maps A to $A \times \mathbb{N}$, with the semigroup structure defined componentwise ($\mathbb{N} = \{0, 1, \dots\}$ is equipped with addition), and making the morphisms act on the left component. The output mechanism $A \times \mathbb{N} \to A$ is defined below:

$$\text{for } n \geq 1,\ (a, n) \mapsto a^n \qquad \underbrace{(a, 0) \mapsto a}_{\text{we handle this case separately because } a^0 \text{ does not make sense in an arbitrary semigroup}}$$

▶ **Example 2.8.** Our last example function is

$$w \in \{a, b, c\}^* \quad \mapsto \quad (\text{longest } c\text{-free suffix of } w) \cdot (\text{longest } c\text{-free prefix of } w) \in \{a, b\}^*$$

This can be recognized using the functor $\mathsf{F}A = A + A^2$, equipped with a suitable semigroup operation that makes the following map $h\colon \{a, b, c\}^* \to \mathsf{F}(\{a, b\}^*)$ a homomorphism:

$$h(w) = w \text{ for } w \in \{a, b\}^* \qquad h(uc \dots cv) = (u, v) \text{ for } u, v \in \{a, b\}^*$$

The output mechanism of the transducer semigroup sends any element $a \in A$ – seen as belonging to the left summand of $A + A^2$ – to aa, and (b, c) to cb.

▶ Remark 2.9. Consider a transducer semigroup with functor F. For a semigroup S, we may often think of an element of $\mathsf{F}S$ as a data structure that contains elements of S (such as a pair or a list, cf. Example 2.2). Then naturality of the output mechanism expresses that, being defined "uniformly in S", it cannot "look inside"[3] those elements of S (but it can combine them using the semigroup operation). In other words, the control flow may depend only on the part that is "independent of S" – and the condition that F is finiteness preserving (satisfied by all our examples except Example 2.7) somehow means that this part is finite.

2.2 Two simple characterizations

All functions. Our first theorem concerns transducer semigroups without any restrictions.

▶ **Theorem 2.10.** *Every function between semigroups is recognized by a transducer semigroup.*

[3] This is similar to the "generic" or "polymorphic" function definitions supported by many statically typed programming languages. The corresponding notion in type theory is *parametric polymorphism*, and it is closely related to naturality, see the introduction to [16].

Proof. We prove a slightly stronger result: for any semigroup A, there exists a transducer semigroup that recognizes all functions from A to other semigroups. The functor is

$$\mathsf{F}B = A \times (\text{set of all functions of type } A \to B, \text{ not necessarily recognizable}).$$

The semigroup operation in $\mathsf{F}B$ is defined as follows: on the first coordinate, we inherit the semigroup operation from A, while on the second coordinate, we use the trivial *left zero* semigroup structure, in which the product of two functions is simply the first one (this is a trivial way of equipping every set with a semigroup structure). The functor is defined on morphisms as in the tuple construction from Example 2.2: the first coordinate, corresponding to A, is not changed, and the second coordinate, corresponding to the set of functions, is transformed coordinate-wise, when viewed as a tuple indexed by A. This is easily seen to be a functor. The output mechanism, which is easily seen to be natural, is function application i.e. $(a, f) \mapsto f(a)$. Every function $f\colon A \to B$ is recognized by this transducer semigroup, with the appropriate homomorphism is $a \in A \mapsto (a, f)$. ◀

Recognizability reflecting functions. We now characterize functions with the property that inverse images of recognizable languages are also recognizable. We use a slightly more general setup, where instead of languages we use functions into arbitrary sets (languages can be seen as the case of functions into $\{\text{yes}, \text{no}\}$).

▶ **Definition 2.11.** *We say that a function from a semigroup A to some set X is* recognizable *if it factors through some semigroup homomorphism from A to a* finite *semigroup.*

In the rest of the paper, we shall sometimes speak of recognizable functions with infinite codomain, but note that the range of a recognizable function is always finite.

A function $f\colon B \to A$ between semigroups is called *recognizability reflecting* if for every recognizable function $g\colon A \to X$, the composition $g \circ f$ is recognizable.

▶ **Example 2.12.** Consider the semigroup $(\mathbb{N}, +)$ of natural numbers with addition, which is isomorphic to the free monoid a^*. In this semigroup, the recognizable functions are ultimately periodic colourings of numbers. A corollary is that every recognizable function gives the same answer to all factorials $\{1!, 2!, \ldots\}$, with finitely many exceptions. Take any function $f\colon \mathbb{N} \to \mathbb{N}$ such that (a) every output number arises from at most finitely many input numbers; (b) every output number is a factorial. The composition of f with any recognizable function will give the same answer to all numbers with finitely many exceptions, thus being also recognizable.

In the above example, a function with conditions (a) and (b) can be chosen in uncountably many ways, even if we require that it has linear growth. Therefore, there are too many recognizability reflecting functions (even just from $\{a\}^*$ to itself) to allow a machine model, or some other effective syntax. The following result gives a non-effective syntax.

▶ **Theorem 2.13.** *The following conditions are equivalent for a string-to-string function:*

1. *it is recognizability reflecting.*
2. *it is recognized by a transducer semigroup such that for every finite semigroup A, the corresponding output function of type $\mathsf{F}A \to A$ is recognizable.*

▶ **Example 2.14.** For any finite semigroup A, the map $(a, n) \in A \times (\mathbb{N} \setminus \{0\}) \mapsto a^n \in A$ is recognizable. This is because, since $a^{|A|!}$ is idempotent for every $a \in A$, this map factors through a homomorphism into the semigroup $A \times ((\mathbb{N} \setminus \{0\})/\sim)$ where

$$n \sim m \quad \iff \quad n = m \quad \vee \quad \big(n, m \geq |A|! \ \wedge \ n \equiv m \bmod |A|!\big)$$

is a congruence of finite index. Extending this slightly to handle the case $(a, 0) \mapsto a$, one can show that the output mechanism out_A of Example 2.7 is recognizable whenever A is finite. Therefore, the squaring function is recognizability reflecting.

3 The regular functions

The two straightforward constructions in Theorems 2.10 and 2.13 amount to little more than symbol pushing. In this section, we present a more substantial characterization, which is the main result of this paper. This characterization concerns finiteness-preserving functors. This is a strengthening of the condition from Theorem 2.13: if the functor F in a transducer semigroup is finiteness-preserving, then for every finite semigroup A, the output function $\mathsf{F}A \to A$ will be recognizable, since all functions from a finite semigroup are trivially recognizable. However, the condition is strictly stronger, as witnessed by Example 2.7, which is recognizability reflecting (cf. Example 2.14) but not finiteness preserving. As we will see, the stronger condition will characterize exactly the regular string-to-string functions.

The following counterexample illustrates the non-trivial interaction between naturality of the output mechanism and the requirement that the functor is finiteness preserving.

▶ **Example 3.1.** Consider the powerset functor P from Example 2.2. It is finiteness preserving, since the powerset of a finite semigroup is also finite. One could imagine that using powersets, one could construct a transducer semigroup that recognizes functions that are not regular, e.g. because they have exponential growth (unlike regular functions, which have linear growth). It turns out that this is impossible, because there is no possible output mechanism, i.e. no natural transformation of type $\mathsf{P}A \to A$, as we explain below.

The issue is that the naturality condition disallows choosing elements from a subset. To see why, consider a semigroup A with two elements, with the trivial left zero semigroup structure. For this semigroup, the output mechanism of type $\mathsf{P}A \to A$ would need to choose some element $a \in A$ when given as input the full set $A \in \mathsf{P}A$. However, none of the two choices is right, because swapping the two elements of A is an automorphism of the semigroup A, which maps the full set to itself, but does not map any element to itself.

We now state the main theorem of this paper.

▶ **Theorem 3.2.** *The following conditions are equivalent for every string-to-string function:*

1. *it is a regular string-to-string function;*
2. *it is recognized by a transducer semigroup in which the functor is finiteness preserving;*
3. *it is recognized by a transducer semigroup in which the functor* F maps the singleton semigroup 1 to a finite semigroup*:* $|\mathsf{F}1| < \infty$.

(Note that (2) ⇒ (3) is immediate.) Here is the plan for the rest of this section:
Section 3.1 gives a formal definition of regular functions
Section 3.2 proves the easy implication in the theorem, namely (1) ⇒ (2)
Section 3.3 proves the hard implication in the theorem, namely (3) ⇒ (1)

Before continuing, we remark on one advantage of the characterization in item (2), namely a straightforward proof of closure under composition. In contrast, for some (but not all) existing models defining regular string-to-string functions, composition requires a non-trivial construction – examples include two-way transducers [11, Theorem 2] or copyless streaming string transducers [2, Theorem 1].

▶ **Proposition 3.3.** *Functions recognized by finiteness-preserving transducer semigroups are closed under composition.*

Proof. This is because finiteness-preserving functors are closed under composition, natural families of output functions are also closed under composition, and naturality means by definition that the output functions "commute" in a suitable sense with functors.

More precisely, consider the following diagram:

$$\begin{array}{ccccc} \Sigma^* & \xrightarrow{h} & \mathsf{F}(\Gamma^*) & \xrightarrow{\mathrm{out}_{\Gamma^*}} & \Gamma^* \\ & & \downarrow{\scriptstyle \mathsf{F}h'} & & \downarrow{\scriptstyle h'} \\ & & \mathsf{FF}'(\Pi^*) & \xrightarrow{\mathrm{out}_{\mathsf{F}'(\Pi^*)}} \mathsf{F}'(\Pi^*) & \xrightarrow{\mathrm{out}'_{(\Pi^*)}} \Pi^* \end{array}$$

where the square commutes because the output mechanism is natural. The upper path describes the composition of two functions recognized by the transducer semigroups $(\mathsf{F}, \mathrm{out})$ and $(\mathsf{F}', \mathrm{out}')$, while the lower path describes a function recognized by $(\mathsf{FF}', \mathrm{out}_{\mathsf{F}'(-)} \circ \mathrm{out}')$. ◀

3.1 Defininition of streaming string transducers

In this section, we formally describe the regular functions, using a model based on streaming string transducers (SST). This model, like our proof of Theorem 3.2, covers a slightly more general case, namely string-to-semigroup functions instead of only string-to-string functions. These are functions of type $\Sigma^* \to A$ where Σ is a finite alphabet and A is an arbitrary semigroup. The purpose of this generalization is to make notation more transparent, since the fact that the output semigroup consists of strings will not play any role in our proof.

The model uses *registers* to store elements of the output semigroup. We begin by describing notation for registers and their updates. Suppose that R is a finite set of register names, and A is a semigroup called the *output semigroup*. We consider two sets

$$\textit{register valuations}\colon (R \to A) \qquad \textit{register updates}\colon (R \to (A + R)^+)$$

Below we show two examples of register updates, presented as assignments, using two registers X, Y and the semigroup $A = a^*$. (The right-hand sides are the values in $(A + R)^+$.)

$$\underbrace{\begin{array}{lcl} X & := & aYaXaaa \\ Y & := & XaaXaa \end{array}}_{\text{copyful}} \qquad \underbrace{\begin{array}{lcl} X & := & aaYaaXaaa \\ Y & := & aaa \end{array}}_{\text{copyless}}$$

The crucial property is being *copyless* – a register update is called copyless if every register name appears in at most one right-hand side of the update, and in that right-hand side it appears at most once. The main operation on these sets is *application*: a register update u can be applied to a register valuation v, giving a new register valuation vu.

In our model of streaming string tranducers, the registers will be updated by a stream of register updates that is produced by a rational function, defined as follows. Intuitively speaking, a rational function corresponds to an automaton that produces one output letter for each input position, with the output letter depending on regular properties of the input position within the input string. More formally:

▶ **Definition 3.4.** *A rational function of type $\Sigma^* \to X^*$ – where Σ is a finite alphabet but X*

can be any set – is a length-preserving[4] *function with the following property: for some family*[5]

$$f_a \colon \underbrace{\Sigma^* \times \Sigma^*}_{\text{equipped with componentwise multiplication}} \to \Gamma \quad \text{for } a \in \Sigma \text{ of recognizable functions,}$$

for every input $a_1 \dots a_n$ *and* $i \in \{1, \dots, n\}$*, the* i*-th output letter is* $f_{a_i}(a_1 \dots a_{i-1},\ a_{i+1} \dots a_n)$.

Note that the range of a rational function with codomain X^* may contain only finitely many "letters" from X, so it can always be seen as a string function over finite alphabets.

Having defined register updates and rational functions, we are ready to introduce the machine model used in this paper as the reference definition of *regular functions*.

▶ **Definition 3.5.** *The syntax of a* streaming string transducer *(*SST*) is given by:*

- *A finite* input alphabet Σ *and an* output semigroup A.
- *A finite set* R *of* register names. *All register valuations and updates below use* R *and* A.
- *A designated* initial register valuation*, and a* final output pattern *in* R^+ *(that does not need to be copyless, though adding this restriction would not affect the expressive power).*
- *An* update oracle*, which is a rational function of type* $\Sigma^* \to (\text{copyless register updates})^*$.

The semantics of the SST is a function of type $\Sigma^* \to A$ defined as follows. When given an input string, the SST begins in the designated initial register valuation. Next, it applies all updates produced by the update oracle, in left-to-right order. Finally, the output of the SST is obtained by combining the last register values according to the final output pattern.

▶ **Example 3.6.** We define an SST that computes the function of Example 2.8. It has two registers X and Y, whose initial valuation is $X = Y = \varepsilon$, and the final output pattern is YX. The update associated to an input letter $\ell \in \{a, b, c\}$ at position i is:

- if the position i is part of the longest c-free prefix, then $X := X\ell$, otherwise $X := X$;
- if the position i is part of the longest c-free suffix, then $Y := Y\ell$, otherwise $Y := Y$.

This sequence of updates can be produced by a rational function generated by a family of functions $(f_\ell)_{\ell \in \{a,b,c\}}$ that are recognized by $\mathbb{B}^2$, where $\mathbb{B}$ is the monoid of booleans with conjunction (rephrase the conditions as "there is no c to the left (resp. right) of i").

In a rational function, the label of the i-th output position is allowed to depend on letters of the input string that are on both sides of the i-th input position; this corresponds to regular lookaround in a streaming string transducer. Therefore, the model described above is easily seen to be equivalent to copyless SSTs with regular lookaround, which are one of the equivalent models defining the regular string-to-string functions, see [3, Section IV.C].

3.2 From a regular function to a transducer semigroup

Having defined the transducer model, we prove the easy implication in Theorem 3.2. It is apparent from Definition 3.5 that every regular function can be decomposed as a rational function followed by a function computed by a streaming string transducer whose i-th register update depends only on the i-th input letter – let us call that a *local* SST. Thanks to closure under composition (Proposition 3.3), we only need to handle these two special cases: we show that finiteness-preserving transducer semigroups recognize

- all rational functions in Section 3.2.1;
- and all local streaming string transducers in Section 3.2.2.

[4] Often in the literature, rational functions are not required to be length-preserving, see e.g. [21, p. 525], but in this paper, we only need the length-preserving case.

[5] The family $(f_a)_{a \in \Sigma}$ is very close to what is called an *(Eilenberg) bimachine* in the literature.

3.2.1 Recognizing rational functions by transducer semigroups

Consider a rational function, generated by the family $(f_a)_{a\in\Sigma}$ of recognizable functions of type $\Sigma^* \times \Sigma^* \to \Gamma$. By definition of recognizability, each f_a decomposes into

$$\Sigma^* \times \Sigma^* \xrightarrow{h_a} B_a \xrightarrow{g_a} \Gamma \qquad \text{where } h_a \text{ is a semigroup homomorphism and } B_a \text{ is finite.}$$

One can check that every f_a then factors through a monoid morphism to the finite monoid

$$\prod_{a\in\Sigma} h_a(\Sigma^* \times \Sigma^*)$$

Thus, without loss of generality, we may assume for the rest of the proof that all of the above semigroups B_a are equal to a common finite monoid B and that each semigroup homomorphism h_a is in fact a monoid morphism.

For any semigroup A, we let[6] $\mathsf{F}A = B \times (B \to A) \times B$, endowed with the following semigroup operation:

$$(\ell_1, \varphi_1, r_1) \cdot (\ell_2, \varphi_2, r_2) = \Big(\ell_1\ell_2,\ \big(b \mapsto \varphi_1(br_2) \cdot \varphi_2(\ell_1 b)\big),\ r_1 r_2\Big).$$

The construction F is extended to morphisms by considering $B \to A$ as the set of B-indexed tuples (cf. Example 2.2) of elements of A. To get a transducer semigroup, we take the output mechanism to be $(\ell, \varphi, r) \mapsto \varphi(e)$ where $e \in B$ is the neutral element.

Our rational function is then recognized by the unique monoid homomorphism of type $\Sigma^* \to \mathsf{F}(\Gamma^*)$ (indeed, F preserves monoids) which maps $a \in \Sigma$ to $\big(h_a(a, \varepsilon), g_a, h_a(\varepsilon, a)\big)$.

3.2.2 From a local SST to a transducer semigroup

Suppose now that a string-to-semigroup function $f\colon \Sigma^* \to A$ is computed by some local streaming string transducer. In the proof below, when referring to register valuations and register updates, we refer to those that use the registers and output semigroup of the fixed transducer. We say that a register update is in *normal form* if, in every right-hand side, one cannot find two consecutive letters from the semigroup A. Any register update can be *normalized*, i.e. converted into one that is in normal form, by using the semigroup operation to merge consecutive elements of the output semigroup in the right-hand sides. Here is an example, which uses three registers X, Y, Z and the semigroup $A = (\{0, 1\}, \cdot)$:

$$\underbrace{\begin{array}{lcl} X & := & 01Y1111X111 \\ Y & := & 01011 \end{array}}_{\text{not in normal form}} \xrightarrow{\text{normalization}} \underbrace{\begin{array}{lcl} X & := & 0Y1X1 \\ Y & := & 0 \end{array}}_{\text{in normal form}}$$

The register updates before and after normalization act in the same way on register valuations. If an update is copyless and in normal form, then the combined length of all right-hand sides is at most three times the number of registers. Therefore, if a semigroup A is finite, then the set of copyless register updates in normal form, call it $\mathsf{U}A$, is also finite. (However, there are infinitely many copyful register updates even when A is finite.) This set $\mathsf{U}A$ can be equipped with a composition operation

$$u_1, u_2 \in \mathsf{U}A \quad \mapsto \quad u_1 u_2 \in \mathsf{U}A,$$

[6] A construction similar in spirit to the classical *two-sided semidirect product* [20, §6].

which is defined in the same way as applying a register update to a register valuation, except that we normalize at the end. This composition operation is associative, and compatible with applying register updates to register valuations, in the sense that $(vu_1)u_2 = v(u_1u_2)$ holds for every valuation v and all updates u_1 and u_2. Therefore, $A \mapsto \mathsf{U}A$ is a finiteness-preserving semigroup-to-semigroup functor (with the natural extension to morphisms, where the homomorphism is applied to every semigroup element in a right-hand side).

The functor U described above is almost but not quite the functor that will be used in the transducer semigroup that we will define to prove the easy implication in Theorem 3.2. That functor F will also take into account the initial register valuation:

$$\mathsf{F}A = \mathsf{U}A \times \underbrace{(R \to A)}_{\text{endowed with the trivial left zero semigroup structure}} \quad \text{with componentwise multiplication \& action on morphisms}$$

Given $(u, v) \in \mathsf{F}A$, the output mechanism in the transducer semigroup applies the register update u to the register valuation v, and then multiplies together the register values given by the resulting valuation vu according to the final output pattern. Using this, we can recognize f via the homomorphism that sends each input letter to:

- the register update that this letter determines (our SST being local) in the first component;
- the designated initial register valuation in the second component.

3.3 From a transducer semigroup to a regular function

We now turn to the difficult implication (3) $\Rightarrow$ (1) in Theorem 3.2. The proof is presented in a way which, if sometimes slightly verbose, makes it easier to see how it can be adapted to other algebraic structures instead of semigroups (such as forest algebras, cf. Section 4).

3.3.1 Polynomial functors and functorial streaming string transducers

The assumption of the implication uses an abstract model (transducer semigroups), while the conclusion uses a concrete operational model (streaming string transducers). To bridge the gap, we use an intermediate model, similar to SSTs, but a bit more abstract. The abstraction arises by using polynomial functors instead of registers, as described below.

▶ **Definition 3.7.** *By* polynomial functor*, we mean a semigroup-to-set functor of the form*

$$A \quad \mapsto \quad \coprod_{q \in Q} A^{\textit{dimension of } q},$$

where Q is some possibly infinite set, whose elements are called components*, with each component having an associated* dimension *in $\mathbb{N}$. The symbol $\coprod$ stands for disjoint union of sets. This functor does not take into account the semigroup structure of the input semigroup, since the output is seen only as a set. On morphisms, the functor works in the expected way, i.e. coordinate-wise.*

A *finite polynomial functor* is a polynomial functor with finitely many components – for example, $A \mapsto A^2 + A^2 + A$. The notion of finite polynomial functor can be seen as a mild generalization of the construction which maps a semigroup A to the set A^R of register valuations for some fixed finite set R of register names. In the generalization, we allow a variable number of registers, depending on some finite information (the component).

Having defined a more abstract notion of "register valuations", namely finite polynomial functors, we now define a more abstract notion of "register updates". The first condition for such updates is that they do not look inside the register contents; this condition is captured by naturality (as discussed in Remark 2.9).

▶ **Example 3.8.** Consider the polynomial functors (where 1 represents the singleton set A^0)

$$\mathsf{F}A = A^* = 1 + A^1 + A^2 + \cdots \qquad \mathsf{G}A = A + 1.$$

An example of a natural transformation between these two functors is the function which maps a nonempty list in A^* to the product of its elements, and which maps the empty list to the unique element of 1. A non-example is the function that maps a list $[a_1, \ldots, a_n] \in A^*$ to the leftmost element a_i that is an idempotent in the semigroup, and returns 1 if such an element does not exist. The reason why the non-example is not natural is that a semigroup homomorphism can map a non-idempotent to an idempotent.

Apart from naturality, we will want our register updates to be copyless. For the purposes of the following definition, let us call a tuple of numbers in $\mathbb{N}^k$ a "sub-unit" if it belongs to $\{0,1\}^k$ and at most one coordinate is equal to 1 – or, as an edge case, if $k = 0$. For a polynomial functor F, a sub-unit of $\mathsf{F}\mathbb{N} = \sum_q \mathbb{N}^{\dim(q)}$ is a sub-unit of any of the $\mathbb{N}^{\dim(q)}$.

▶ **Definition 3.9** (Copyless natural transformation)**.** *A natural transformation between two polynomial functors* F *and* G *is called* copyless *if when instantiated to the semigroup*[7] $(\mathbb{N}, +)$*, the corresponding function of type* $\mathsf{F}\mathbb{N} \to \mathsf{G}\mathbb{N}$ *maps sub-units to sub-units.*

It will be convenient to speak of *natural functions* $f \colon \mathsf{F}A \to \mathsf{G}A$, where F and G are semigroup-to-set functors and A is a fixed semigroup, to refer to functions that can be extended to natural transformations $(f_B \colon \mathsf{F}B \to \mathsf{G}B)_{B \text{ semigroup}}$, with $f = f_A$. *Copyless natural functions* between instantiations of polynomial functors are defined analogously.

Having defined functions that are natural and copyless, we now describe the more abstract model of SSTs used in our proof. The main difference is that instead of register valuations and updates given by some finite set of register names, we have two abstract polynomial functors, one of them finite polynomial, together with an explicitly given application function. We also allow the computation to be initialized and finalized in a more liberal way, that may depend on a regular property of the input.

▶ **Definition 3.10.** *The syntax of a* functorial streaming string transducer *is given by:*

- *A finite* input alphabet Σ *and an* output semigroup A*.*
- *A finite polynomial functor* R*, called the* register functor*, and a (not necessarily finite) polynomial functor* U *called the* update functor*.*
- *A copyless natural function of type* $\mathsf{R}A \times \mathsf{U}A \to \mathsf{R}A$*, called* application*.*
- *An* initial function $\Sigma^* \to \mathsf{R}A$ *which is recognizable (and therefore has finite range).*
- *A polynomial* final data functor K*, a* final data function $\Sigma^* \to \mathsf{K}A$ *which is recognizable, and a* final output function *of type* $\mathsf{R}A \times \mathsf{K}A \to A$ *which is a natural function (but not necessarily copyless).*
- *An* update oracle*, which is a rational function of type* $\Sigma^* \to (\mathsf{U}A)^*$*.*

Analogously to Definition 3.5, the functorial SST computes the function $\Sigma^* \to A$ obtained by the following composition, where the first map bundles together the initial function, the update oracle and the final data function:

$$\Sigma^* \to \mathsf{R}A \times (\mathsf{U}A)^* \times \mathsf{K}A \xrightarrow{(\textit{apply} \text{ updates successively}) \times \mathrm{id}_{\mathsf{K}A}} \mathsf{R}A \times \mathsf{K}A \xrightarrow{\text{final output function}} A$$

In the appendix, we prove that this model is no more expressive than usual copyless SSTs.

[7] The choice of the semigroup $(\mathbb{N}, +)$ in the Definition 3.9 is not particularly important. For example, the same notion of copylessness would arise if instead of $(\mathbb{N}, +)$, we used the semigroup $\{0, 1, 2\}$ with addition up to threshold 2. In the appendix, we present a more syntactic characterization of copyless natural transformations as part of our proof of Lemma 3.11.

▶ **Lemma 3.11.** *Definitions 3.5 and 3.10 characterize the same string-to-semigroup functions.*

3.3.2 Coproducts and views

Apart from the more abstract transducer model from Definition 3.10, the other ingredient used in the proof of the hard implication in Theorem 3.2 will be coproducts of semigroups, and some basic operations on them, as described in this section.

The *coproduct*[8] of two semigroups A and B, denoted by $A \oplus B$, is the semigroup whose elements are nonempty words over an alphabet that is the disjoint union of A and B, restricted to words that are *alternating* in the sense that two consecutive letters cannot belong to the same semigroup. The semigroup operation is defined in the expected way. We draw elements of a coproduct using coloured boxes, with the following picture showing the product operation in the coproduct of two copies, red and blue, of the semigroup $\{a, b\}^+$:

$$(\boxed{aba} \cdot \boxed{b} \cdot \boxed{b} \cdot \boxed{aa}) \cdot (\boxed{abba} \cdot \boxed{aabb}) = \boxed{aba} \cdot \boxed{b} \cdot \boxed{b} \cdot \boxed{aaabba} \cdot \boxed{aabb}.$$

A coproduct can involve more than two semigroups; in the pictures this would correspond to more colours, subject to the condition that consecutive boxes have different colours.

▶ Remark 3.12. The copyless register updates $u : R \to (A + R)^+$ of ordinary SSTs that are in normal form (cf. Section 3.2) can be seen as maps $R \to A \oplus \bigoplus_{X \in R} \{X\}^+$.

We write 1 for the semigroup that has one element. This semigroup is unique up to isomorphism and it is a *terminal object* in the category of semigroups, which means that it admits a unique homomorphism from every other semigroup A. This unique homomorphism will be denoted by $! : A \to 1$. (It has no connection with the factorial function on numbers.)

Consider the semigroup-to-set functors defined by (the underlying set of) a coproduct of several copies of their argument with several copies of 1, such as $A \mapsto A \oplus A \oplus A \oplus 1 \oplus 1$. In our proof, it will be useful to see them as polynomial functors, even though strictly speaking they are not defined as sums of products. This identification is allowed by the following observation (stated for $A \oplus 1$ for convenience, but the same idea applies in general).

▶ **Proposition 3.13.** *There is a family of bijections, natural in the semigroup A, between*

$$A \oplus 1 \quad \text{and} \quad \coprod_{q \in 1 \oplus 1} A^{\text{dimension of } q},$$

where the dimension of q is the number of times that the first copy of 1 appears in q.

Idea. Given $x \in A \oplus 1$, we apply $! : A \to 1$ to the elements of A in x to determine the component q of the polynomial functor that contains the image of x by the left-to-right bijection. This operation, a special case of what is called the shape below, forgets those elements of A appearing in x, so we record them in a tuple living in $A^{\dim(q)}$. For example, $\boxed{aba} \cdot \boxed{1} \cdot \boxed{aa} \cdot \boxed{1}$ is sent to the tuple (aba, aa) in the component $\boxed{1} \cdot \boxed{1} \cdot \boxed{1} \cdot \boxed{1}$. ◀

The crucial property of semigroups that will be used in our proof is Lemma 3.14 below, which says that an element of a coproduct can be reconstructed based on certain partial information. This information is described using the following operations.

[8] The name *coproduct* is used because of the following universal property: if $f : A \to C$ and $g : B \to C$ are two semigroup homomorphisms, then there is a unique homomorphism $A \oplus B \to C$ that coincides with f (resp. g) on the subsemigroup consisting of words with a single letter from A (resp. B).

1. **Merging**. Consider a coproduct $A_1 \oplus \cdots \oplus A_n$, such that the same semigroup A appears on all coordinates from a subset $I \subseteq \{1, \ldots, n\}$, and possibly on other coordinates as well. Define *merging the parts from I* to be the function of type

$$A_1 \oplus \cdots \oplus A_n \to A \oplus \bigoplus_{i \notin I} A_i$$

that is defined in the expected way, and explained in the following picture. In the picture, merging is applied to a coproduct of three copies of the semigroup $\{a, b\}^+$, indicated using colours red, black and blue, and the merged coordinates are red and blue:

$$\boxed{aba} \cdot \boxed{b} \cdot \boxed{aa} \cdot \boxed{b} \cdot \boxed{aa} \cdot \boxed{abba} \cdot \boxed{b} \quad \mapsto \quad \underbrace{\boxed{abab} \cdot \boxed{aa} \cdot \boxed{baaabba}}_{\text{the merge of red and blue is drawn in violet}} \cdot \boxed{b}.$$

2. **Shape.** Define the *shape operation* to be the function of type

$$A_1 \oplus \cdots \oplus A_n \to 1 \oplus \cdots \oplus 1$$

obtained by applying ! on every coordinate. The shape says how many alternating blocks there are, and which semigroups they come from, as explained in the following picture:

$$\boxed{aba} \cdot \boxed{b} \cdot \boxed{aa} \cdot \boxed{b} \cdot \boxed{aa} \cdot \boxed{abba} \cdot \boxed{b} \quad \mapsto \quad \boxed{1} \cdot \boxed{1} \cdot \boxed{1} \cdot \boxed{1} \cdot \boxed{1} \cdot \boxed{1} \cdot \boxed{1}.$$

3. **Views.** The final operation is the i-th *view*

$$A_1 \oplus \cdots \oplus A_n \to 1 \oplus A_i.$$

This operation applies ! to all coordinates other than i, and then it merges all those coordinates. Here is a picture, in which we take the view of the blue coordinate:

$$\boxed{aba} \cdot \boxed{b} \cdot \boxed{aa} \cdot \boxed{b} \cdot \boxed{aa} \cdot \boxed{abba} \cdot \boxed{b} \quad \mapsto \quad \boxed{aba} \cdot \boxed{1} \cdot \boxed{aa} \cdot \boxed{1}.$$

The key observation is that an element of a coproduct is fully determined from its shape and views, as stated in the following lemma. It seems to contain the essential property of semigroups that makes the construction work. We expect our theorem to also be true for other algebraic structures for which the lemma is true; however, the lemma seems to fail in certain settings. Concrete examples will be discussed in the conclusion (Section 4).

▶ **Lemma 3.14.** *Let $A_1, \ldots, A_n$ be semigroups. The deconstruction function of type*

$$A_1 \oplus \cdots \oplus A_n \longrightarrow (1 \oplus A_1) \times \cdots \times (1 \oplus A_n) \times (1 \oplus \cdots \oplus 1),$$

which is obtained by combining the views for all $i \in \{1, \ldots, n\}$ and the shape, is injective.

We prove this by exhibiting an explicit partial left inverse: a *reconstruction* function of type

$$(1 \oplus A_1) \times \cdots \times (1 \oplus A_n) \times (1 \oplus \cdots \oplus 1) \longrightarrow (A_1 \oplus \cdots \oplus A_n) + 1$$

such that deconstruction followed by reconstruction maps every element of $A_1 \oplus \cdots \oplus A_n$ to itself. The idea is to start with the shape and replace the entries from 1 with the elements appearing in the views in the right order. Rather than a formal definition, we illustrate this on an example (in the 3 views, we omit the boxes around the 1s to avoid visual cluttering):

$$\begin{array}{ccccccccccccc} \boxed{aba} & & 1 & & \boxed{aa} & & 1 & & & & & & \\ 1 & \boxed{b} & 1 & \boxed{b} & 1 & \boxed{abba} & 1 & & & & & & \\ & 1 & & \boxed{aa} & & 1 & & \boxed{b} & & & & & \\ \boxed{1} & \boxed{1} & \boxed{1} & \boxed{1} & \boxed{1} & \boxed{1} & \boxed{1} & & & & & & \end{array} \quad \mapsto \quad \boxed{aba} \cdot \boxed{b} \cdot \boxed{aa} \cdot \boxed{b} \cdot \boxed{aa} \cdot \boxed{abba} \cdot \boxed{b}$$

Besides proving Lemma 3.14, this reconstruction function also enjoys the following property, which can be seen from the definition and Proposition 3.13.

▶ **Proposition 3.15.** *When each A_i is either A or 1, reconstruction can be seen as a copyless natural function between polynomial functors in A.*

3.3.3 Factorized output

Now, consider some transducer semigroup, with the functor being F, and fix a string-to-semigroup function $f\colon \Sigma^* \to A$ that decomposes as some homomorphism $h\colon \Sigma^* \to \mathsf{F}A$ followed by the output function of type $\mathsf{F}A \to A$.

For semigroups $A_1, \ldots, A_n$, define the *vectorial output function* to be the function of type

$$\mathsf{F}A_1 \times \cdots \times \mathsf{F}A_n \longrightarrow A_1 \oplus \cdots \oplus A_n$$

that is obtained by the composition of three functions described below (where *co-projection* is the function $A_i \to A_1 \oplus \cdots \oplus A_n$ that outputs a singleton list containing its input):

$$\begin{array}{c} \mathsf{F}A_1 \times \cdots \times \mathsf{F}A_n \\ \downarrow \mathsf{F}(\text{co-projection}) \times \cdots \times \mathsf{F}(\text{co-projection}) \\ \mathsf{F}(A_1 \oplus \cdots \oplus A_n) \times \cdots \times \mathsf{F}(A_1 \oplus \cdots \oplus A_n) \\ \downarrow \text{semigroup operation} \\ \mathsf{F}(A_1 \oplus \cdots \oplus A_n) \\ \downarrow \text{output mechanism for } A_1 \oplus \cdots \oplus A_n \\ A_1 \oplus \cdots \oplus A_n. \end{array}$$

To illustrate the definitions in this section, we use a *running example* with the transducer semigroup for the "reverse then duplicate" function from Example 2.6. The functor F sends a semigroup A to the opposite semigroup (cf. Example 2.2), and the output mechanism is $a \mapsto aa$. Our example function on $\{a, b\}^*$ is obtained by composing the string reversal homomorphism $\{a, b\}^* \to \mathsf{F}(\{a, b\}^*)$ with the output function. Here is an example of the vectorial output function (for now, the homomorphism plays no role):

$$(1, abbb) \in \mathsf{F}1 \times \mathsf{F}(\{a, b\}^*) \qquad \mapsto \qquad \boxed{abbb}\,\boxed{1}\,\boxed{abbb}\,\boxed{1} \in 1 \oplus \{a, b\}^*.$$

The vectorial output function is natural in all of its arguments, which means that for all semigroup homomorphisms $h_1, \ldots, h_n$, the diagram below commutes:

$$\begin{array}{ccc} \mathsf{F}A_1 \times \cdots \times \mathsf{F}A_n & \xrightarrow{\text{vectorial output function}} & A_1 \oplus \cdots \oplus A_n \\ \downarrow \mathsf{F}h_1 \times \cdots \times \mathsf{F}h_n & & \downarrow h_1 \oplus \cdots \oplus h_n \\ \mathsf{F}B_1 \times \cdots \times \mathsf{F}B_n & \xrightarrow{\text{vectorial output function}} & B_1 \oplus \cdots \oplus B_n \end{array}$$

This is because each of the three steps in the definition of the vectorial output function is itself a natural transformation, and natural transformations compose. Naturality of the first two steps is easy to check, while for the last step we use the assumption that the (non-vectorial) output function is natural.

Let us return to our function $f = \mathrm{out}_A \circ h$ recognized by our transducer semigroup $(\mathsf{F}, \mathrm{out})$. For strings $w_1, \ldots, w_n \in \Sigma^*$, define the corresponding *factorized output* to be the

result of first applying the semigroup homomorphism $h : \Sigma^* \to \mathsf{F}A$ to all the strings, and then applying the vectorial output function; we denote it by

$$\langle w_1 | \cdots | w_n \rangle \in \underbrace{A \oplus \cdots \oplus A}_{n \text{ times}},$$

Here is the factorized output illustrated on our running example (we use colours to distinguish which of the three parts of the input is used):

$$\langle abb | \varepsilon | baaba \rangle = \boxed{abaab}\ \boxed{\varepsilon}\ \boxed{bba}\ \boxed{abaab}\ \boxed{\varepsilon}\ \boxed{bba} \in \{a,b\}^* \oplus \{a,b\}^* \oplus \{a,b\}^*.$$

As we can see above, when the output semigroup is a free monoid, the factorized output morally tells us "which part of the output string comes from which part in the input string".

▶ Remark 3.16. This is similar to the idea of *origin semantics* [5] of regular functions (see also [17, Section 5]). Indeed, our definition of factorized output is inspired by a similar tool of the same name that appears in the study of origin semantics [5, Section 2].

We also use a similar notation but with some input strings underlined, e.g. the input could be $\langle \underline{abb} | \varepsilon | baaba \rangle$ with an underline for the first red part. In the underlined case, before applying the vectorial output function, we apply h to the non-underlined strings and $(\mathsf{F}! \circ h) \colon \Sigma^* \to \mathsf{F}1$ to the underlined strings. In our running example, we have

$$\langle \underline{abb} | \varepsilon | baaba \rangle = \boxed{abaab}\ \boxed{\varepsilon}\ \boxed{1}\ \boxed{abaab}\ \boxed{\varepsilon}\ \boxed{1} \in 1 \oplus \{a,b\}^* \oplus \{a,b\}^*.$$

3.3.4 Proof of (3) ⇒ (1) in Theorem 3.2

We have now collected all necessary ingredients to prove this hard direction of the equivalence. Therefore, our goal is now to show that the function $f \colon \Sigma^* \to A$ that we have previously fixed is computed by some functorial streaming string transducer as in Definition 3.10, *assuming that* $\mathsf{F}1$ *is finite*.

The idea is that we want the functorial SST to maintain the following *invariant*:

> after processing the first i letters in an input string $a_1 \cdots a_n$,
> the register valuation is equal to the factorized output $\langle a_1 \cdots a_i | \underline{a_{i+1} \cdots a_n} \rangle$.

This way, after processing all input letters, the last valuation $\langle a_1 \cdots a_n | \underline{\varepsilon} \rangle$ is very close to the output; indeed, if we see A as a 1-ary coproduct, then $f(a_1 \dots a_n) = \langle a_1 \dots a_n \rangle \in A$.

The naive choice for the register functor is then $\mathsf{R}' : A \mapsto A \oplus 1$, since $\langle w | \underline{v} \rangle \in A \oplus 1$ for all $w, v \in \Sigma^*$ by definition. However, while R' can be seen as a polynomial semigroup-to-set functor, whose set of components is $1 \oplus 1$ (cf. Proposition 3.13), it is not *finite* polynomial (the set $1 \oplus 1$ is infinite). That said, we have by naturality of vectorial output:[9]

[9] Let us give some details. From Proposition 3.13, we see that the component in $1 \oplus 1$ of an element in $A \oplus 1$ is obtained by applying $! \oplus 1$. So it suffices to show that

$$\langle \underline{w} | \underline{v} \rangle = (! \oplus 1)(\langle w | \underline{v} \rangle)$$

Expanding the definitions, our goal can be rewritten as

$$\text{vectorial output of } (\mathsf{F}! \circ h(w),\ \mathsf{F}! \circ h(v)) = (! \oplus 1)\big(\text{vectorial output of } (h(w),\ \mathsf{F}! \circ h(v))\big)$$

This is a direct consequence of the naturality of the vectorial output function, that can be expressed as follows: for every semigroup homomorphism $g \colon A \to B$,

$$(\text{vectorial output for } B) \circ (\mathsf{F}g \times \mathrm{id}_1) = (g \oplus 1) \circ \text{vectorial output for } A$$

(take $g = !$ and apply both sides of the equality to $(h(w), \mathsf{F}! \circ h(v))$ to get the desired Claim 3.17).

▷ Claim 3.17. The component for $\langle w|\underline{v}\rangle \in \mathsf{R}'A$ is $\langle \underline{w}|\underline{v}\rangle \in 1 \oplus 1$.

This index is determined by definition by the values of $(\mathsf{F}! \circ h)\colon \Sigma^* \to \mathsf{F}1$ on w and v, where $h\colon \Sigma^* \to \mathsf{F}A$ is the homomorphism used to recognize f. *Since* $\mathsf{F}1$ *is finite,* the $\langle w|\underline{v}\rangle$ for w, v ranging over Σ^* live in finitely many components. We take our register functor $\mathsf{R}A \subset \mathsf{R}'A$ to be the finite polynomial functor consisting of these "useful" components, plus the unique component that does not use A (it will serve as a "*null value*").

To design the register updates, the key is the following lemma. It shall be proved later using the machinery of views on coproducts that we have introduced for this very purpose.

▶ **Lemma 3.18.** *There are two copyless natural functions*

$$\delta\colon (A \oplus 1) \times (1 \oplus A \oplus 1) \to (A \oplus 1) + 1 \qquad \kappa\colon (A \oplus 1) \times (1 \oplus A) \to A + 1$$

such that, for every pair of strings $w, v \in \Sigma^*$ *and every letter* $a \in \Sigma$*,*

$$\langle wa|\underline{v}\rangle = \delta(\langle w|\underline{av}\rangle, \langle \underline{w}|a|\underline{v}\rangle) \qquad f(w) = \langle w\rangle = \kappa(\langle w|\underline{\varepsilon}\rangle, \langle \underline{w}|\varepsilon\rangle)$$

Again, to make "copyless natural" meaningful in this context, we invoke Proposition 3.13 to see δ and κ as functions between polynomial functors in A.

This leads us to use the update functor $\mathsf{U} : A \mapsto 1 \oplus A \oplus 1$ and to define the application of updates to registers, of type $\mathsf{R}A \times \mathsf{U}A \to \mathsf{R}A$, to be δ followed by the map $(A \oplus 1) + 1 \to \mathsf{R}A$ which sends the components of $A \oplus 1$ that are in $\mathsf{R}A$ to themselves, and everything else to the "null value". As an direct consequence of the lemma, the desired invariant holds using

- the initial function $w \mapsto \langle \varepsilon|\underline{w}\rangle$,
- and the update oracle $a_1 \dots a_n \mapsto \langle \underline{\varepsilon}|a_1|\underline{a_2 \dots a_n}\rangle \dots \langle \underline{a_1 \dots a_{n-1}}|a_n|\underline{\varepsilon}\rangle$.

To fit Definition 3.10, we have to check that the initial function is recognizable and that the update oracle is a rational function; by definition, the latter amounts to saying that for any $a \in \Sigma$, the function $(w, v) \in (\Sigma^*)^2 \mapsto \langle \underline{w}|a|\underline{v}\rangle$ is recognizable. According to the definition of factorized output, the initial function factors through the semigroup homomorphism $\mathsf{F}! \circ h$, *whose codomain* $\mathsf{F}1$ *is finite*; therefore, the initial function is recognizable. The other recognizability condition holds for a similar reason.

To finish building our functorial streaming string transducer, we use the function κ from Lemma 3.18. Thanks to our invariant and to the equation concerning κ, it is immediate that the following choices lead to a functorial SST that indeed computes f. We take:

- the final data functor $\mathsf{K} : A \mapsto (1 \oplus A) \times A$,
- the final data function $w \in \Sigma^* \mapsto (\langle \underline{w}|\varepsilon\rangle$, some arbitrary fixed value in $A)$ – once again, it is recognizable because $\mathsf{F}1$ is finite,
- and the final output function $\mathsf{R}A \times \mathsf{K}A \to A$ that proceeds as follows: first, it applies κ to get some value in $(A + 1) \times A$; if the left half of the pair is in A, it returns it; otherwise, it returns the right half.

This being done, let us discharge our only remaining subgoal.

Proof of Lemma 3.18. We cover here the part concerning δ; for κ, the arguments are similar and a bit simpler. We use the following claim, which is proved using mechanical diagram chasing (as detailed in the appendix). Recall that the merging, shape and view operations were introduced just before Lemma 3.14.

▷ Claim 3.19. $\langle wa|\underline{v}\rangle$ is obtained from $\langle w|a|\underline{v}\rangle$ by merging the first two parts in $A \oplus A \oplus 1$.

The above claim shows that the factorized output $\langle wa|\underline{v}\rangle$ is obtained from $\langle w|a|\underline{v}\rangle$ by a copyless natural function. In turn, $\langle w|a|\underline{v}\rangle$ is the image by the reconstruction function – which is copyless natural (Proposition 3.15) – of the following four items (the equalities below are proved similarly to Claims 3.17 and 3.19):

1. First view of $\langle w|a|\underline{v}\rangle$, which is equal to $\langle w|\underline{av}\rangle$ – this is the first argument which is passed, in the lemma statement, to the function δ that we want to define.
2. Second view of $\langle w|a|\underline{v}\rangle$, which is obtained by merging the first and third parts in $\langle \underline{w}|a|\underline{v}\rangle$.
3. Third view of $\langle w|a|\underline{v}\rangle$, which is equal to $\langle \underline{wa}|\underline{v}\rangle$.
4. Shape of $\langle w|a|\underline{v}\rangle$, which is equal to $\langle \underline{w}|\underline{a}|\underline{v}\rangle$.

To complete the proof, it remains to justify that the last three items above can be collectively obtained from the second argument given to δ, namely $\langle \underline{w}|a|\underline{v}\rangle$, by applying some copyless natural function. Each item is obtained separately by applying a natural function. Furthermore, the second item is obtained in a copyless way, while the last two items do not use A at all, and therefore they are obtained in a copyless way for trivial reasons, even when combined with the second item. ◀

4 Conclusions

In this paper, we have exhibited a concise algebraic characterization of the regular string-to-string functions, in the style of the definition of regular languages using recognizability by finite semigroups. To perform this extension from languages to functions, we have relied on the basic concepts of category theory: categories, functors, natural transformations.

It should be noted that our use of categories is quite different in spirit from many of the works that take a categorical perspective on automata-theoretic results – see for instance [12], whose introduction points to many further references. In such works, the correspondence between concrete automata models and their rephrasing as suitable (co)algebras or functors tends to be straightforward, with the technical focus lying elsewhere (typically, in generalizing constructions such as determinisation or minimisation). On the contrary, we define a truly new transducer model whose equivalence with the preexisting copyless streaming string transducers requires a non-trivial proof.

An advantage of our characterization of the regular string functions is that, as one would expect from an abstract result, it lends itself to generalizations.

Semigroup-to-semigroup functions. The notion of recognition by a finiteness-preserving transducer semigroup makes sense for functions between arbitrary semigroups. Furthermore, such functions are closed under composition (the proof of Proposition 3.3 works as it is). To check their robustness, it would be desirable to have a more concrete, machine-like model capturing the same function class; possibly a variant of streaming string transducers where the underlying finite automaton is morally "replaced" by a finite semigroup.

More string functions. Another direction is characterizing other classes of string-to-string functions, such as the rational functions or the polyregular functions [7]. In this paper, we have discovered that, somewhat mysteriously, combining two conditions – naturality and preserving finiteness – characterizes exactly the regular functions, which have linear growth. Perhaps there is some way of tweaking the definitions to describe, say, some class with polynomial growth. For instance, the squaring function (Example 2.7) seems to be recognized by a mixed-variance functor $A \mapsto (A \to A) \times A$ with a dinatural output mechanism.

Functions on other free algebras. The definition of a transducer semigroup can applied to other algebras, and not just semigroups. This may be done by taking some monad T and considering functions that can be decomposed, for some endofunctor F of the category of Eilenberg-Moore algebras for the monad T and some natural transformation out, as

$$\mathsf{T}\Sigma \xrightarrow{\text{some } \mathsf{T}\text{-algebra homomorphism}} \mathsf{F}\mathsf{T}\Gamma \xrightarrow{\mathrm{out}_{\mathsf{T}\Gamma}} \mathsf{T}\Gamma.$$

An example of this approach is forest algebras [6, Section 5], which are algebras for describing trees. Preliminary work shows that, in the case of forest algebras, the suitable version of Theorem 3.2 also holds, i.e. the finiteness-preserving functors lead to a characterization of the standard notion of regular tree-to-tree functions, namely MSO transductions (see [14, 9]). We believe that these results apply even further, namely for graphs of bounded treewidth, modeled using suitable monads [6, Section 6]. The crucial property is that Lemma 3.14, about reconstructing a coproduct from its views, holds for other monads than just the nonempty list monad for semigroups. Unfortunately, this lemma fails for some monads, such as the monad of formal linear combinations of strings that corresponds to weighted automata. In the future, we intend to conduct a more systematic investigation of the extent to which the characterizations from this paper can be generalized to other algebraic structures.

References

1 Rajeev Alur and Pavol Černý. Expressiveness of streaming string transducers. In Kamal Lodaya and Meena Mahajan, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2010, December 15-18, 2010, Chennai, India*, volume 8 of *LIPIcs*, pages 1–12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2010. doi:10.4230/LIPIcs.FSTTCS.2010.1.

2 Rajeev Alur, Taylor Dohmen, and Ashutosh Trivedi. Composing copyless streaming string transducers, 2022. arXiv:2209.05448.

3 Rajeev Alur, Emmanuel Filiot, and Ashutosh Trivedi. Regular transformations of infinite strings. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, pages 65–74. IEEE Computer Society, 2012. doi:10.1109/LICS.2012.18.

4 Rajeev Alur, Adam Freilich, and Mukund Raghothaman. Regular combinators for string transformations. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS) - CSL-LICS '14*, pages 1–10, Vienna, Austria, 2014. ACM Press. doi:10.1145/2603088.2603151.

5 Mikołaj Bojańczyk. Transducers with origin information. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II*, volume 8573 of *Lecture Notes in Computer Science*, pages 26–37. Springer, 2014. doi:10.1007/978-3-662-43951-7_3.

6 Mikołaj Bojańczyk. Languages recognised by finite semigroups, and their generalisations to objects such as trees and graphs, with an emphasis on definability in monadic second-order logic, 2020. arXiv:2008.11635.

7 Mikołaj Bojańczyk. Transducers of polynomial growth. In Christel Baier and Dana Fisman, editors, *LICS '22: 37th Annual ACM/IEEE Symposium on Logic in Computer Science, Haifa, Israel, August 2 - 5, 2022*, pages 1:1–1:27. ACM, 2022. doi:10.1145/3531130.3533326.

8 Mikołaj Bojańczyk, Laure Daviaud, and Shankara Narayanan Krishna. Regular and First-Order List Functions. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science - LICS '18*, pages 125–134, Oxford, United Kingdom, 2018. ACM Press. doi:10.1145/3209108.3209163.

9 Mikołaj Bojańczyk and Amina Doumane. First-order tree-to-tree functions, 2020. Corrected version with erratum of a LICS 2020 paper. arXiv:2002.09307v2.

10 Mikołaj Bojańczyk and Rafał Stefański. Single-use automata and transducers for infinite alphabets. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, volume 168 of *LIPIcs*, pages 113:1–113:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ICALP.2020.113.

11 Michal Chytil and Vojtech Jákl. Serial composition of 2-way finite-state transducers and simple programs on strings. In Arto Salomaa and Magnus Steinby, editors, *Automata, Languages and Programming, Fourth Colloquium, University of Turku, Finland, July 18-22, 1977, Proceedings*, volume 52 of *Lecture Notes in Computer Science*, pages 135–147. Springer, 1977. doi:10.1007/3-540-08342-1_11.

12 Thomas Colcombet and Daniela Petrişan. Automata Minimization: a Functorial Approach. *Logical Methods in Computer Science*, 16(1), March 2020. doi:10.23638/LMCS-16(1:32)2020.

13 Joost Engelfriet and Hendrik Jan Hoogeboom. MSO definable string transductions and two-way finite-state transducers. *ACM Transactions on Computational Logic*, 2(2):216–254, April 2001. doi:10.1145/371316.371512.

14 Joost Engelfriet and Sebastian Maneth. Macro Tree Transducers, Attribute Grammars, and MSO Definable Tree Translations. *Information and Computation*, 154(1):34–91, October 1999. doi:10.1006/inco.1999.2807.

15 Paul Gallot, Aurélien Lemay, and Sylvain Salvati. Linear high-order deterministic tree transducers with regular look-ahead. In Javier Esparza and Daniel Král', editors, *45th International Symposium on Mathematical Foundations of Computer Science, MFCS 2020, August 24-28, 2020, Prague, Czech Republic*, volume 170 of *LIPIcs*, pages 38:1–38:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.MFCS.2020.38.

16 Claudio Hermida, Uday S. Reddy, and Edmund P. Robinson. Logical Relations and Parametricity – A Reynolds Programme for Category Theory and Programming Languages. In John Power and Cai Wingfield, editors, *Proceedings of the Workshop on Algebra, Coalgebra and Topology, WACT 2013, Bath, UK, March 1, 2013*, volume 303 of *Electronic Notes in Theoretical Computer Science*, pages 149–180. Elsevier, 2013. doi:10.1016/j.entcs.2014.02.008.

17 Anca Muscholl and Gabriele Puppis. The Many Facets of String Transducers. In Rolf Niedermeier and Christophe Paul, editors, *36th International Symposium on Theoretical Aspects of Computer Science (STACS 2019)*, volume 126 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:21. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. doi:10.4230/LIPIcs.STACS.2019.2.

18 Lê Thành Dũng Nguyễn. *Implicit automata in linear logic and categorical transducer theory*. PhD thesis, Université Paris XIII (Sorbonne Paris Nord), December 2021. URL: https://theses.hal.science/tel-04132636.

19 Lê Thành Dũng Nguyễn and Cécilia Pradic. Implicit automata in typed λ-calculi I: aperiodicity in a non-commutative logic. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, volume 168 of *LIPIcs*, pages 135:1–135:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ICALP.2020.135.

20 John Rhodes and Bret Tilson. The kernel of monoid morphisms. *Journal of Pure and Applied Algebra*, 62(3):227–268, 1989. doi:10.1016/0022-4049(89)90137-0.

21 Jacques Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, 2009. Translated by Reuben Thomas. doi:10.1017/CBO9781139195218.

22 John C. Shepherdson. The reduction of two-way automata to one-way automata. *IBM Journal of Research and Development*, 3(2):198–200, April 1959. doi:10.1147/rd.32.0198.

A Proof of Theorem 2.13

We prove a slight strengthening of the theorem, which concerns not only string-to-string functions, but also functions $f\colon A \to B$ between semigroups, not necessarily homomorphisms, such that the target semigroup B is finitely generated. (The free monoid of strings Σ^* is generated as a semigroup by $\{\varepsilon\} \cup \Sigma$, which is a finite set since we always assume that our strings are over a finite alphabet.)

(1) ⇒ (2). We use a similar construction as in the proof of Theorem 2.10. Let $f\colon A \to B$ be recognizability reflecting. Define a functor as follows:

$$\mathsf{F}C = A \times (\text{all semigroup homomorphisms of type } B \to C).$$

Similarly to Theorem 2.10, the semigroup operation on the first coordinate of $\mathsf{F}C$ is inherited from A, and on the second coordinate we use the left zero semigroup structure, where the product of g and h is g. On morphisms, the functor is defined as in the proof of Theorem 2.10.

The output mechanism is $(a, g) \mapsto g(f(a))$ – think of it as function application with f inserted as an interface. The transducer semigroup thus defined recognizes the function f via the homomorphism $a \in A \mapsto (a, \mathrm{id})$.

We now argue that the output mechanism for a finite semigroup C is a recognizable function. To do so, we show that the inverse image of every $c \in C$ is a recognizable subset of $\mathsf{F}C$. This inverse image is

$$\bigcup_{g\colon B \to C \text{ is a semigroup homomorphism}} \{(a, g) \mid g(f(a)) = c\}.$$

Each of the sets in this union is recognizable, by the assumption that f is recognizability reflecting. By the assumptions that B is finitely generated and that C is finite, there are finitely semigroup homomorphisms $B \to C$, and therefore the union is finite. This implies that the inverse image of c is recognizable, as a finite union of recognizable subsets of $\mathsf{F}C$.

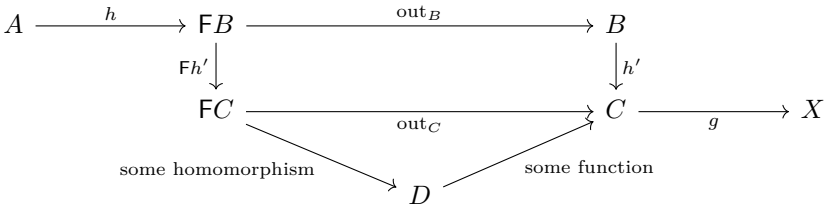
(1) ⇐ (2). Take a function $f\colon A \to B$ that satisfies (2), i.e. it is a composition

$$A \xrightarrow{h} \mathsf{F}B \xrightarrow{\mathrm{out}_B} B$$

where h is some homomorphism. We want to show that f is recognizability reflecting. To prove this, let us consider some recognizable function from the output semigroup

$$B \xrightarrow{h'} C \xrightarrow{g} X$$

where C is a finite semigroup and h' is a homomorphism. We want to show that its precomposition f is also recognizable. Consider the following diagram.



The triangle, with D *finite*, describes the assumption that the output function out_C is recognizable when C is finite. The upper path from A to X describes the precomposition by f. The rectangle commutes by naturality of the output mechanism, and therefore the upper path describes the same function as the lower path from A to X. The lower path is a recognizable function, since the first three arrows are homomorphisms and D is finite.

B Proof of Lemma 3.11

In this part of the appendix, we prove that the models defined in Definitions 3.5 and 3.10 define the same string-to-semigroup functions.

The easy implication is left-to-right. A streaming string transducer as in Definition 3.5 can be seen as a special case of a functorial SST as in Definition 3.10, because the sets of register valuations and register updates in normal form (cf. Section 3.2) are constructed using finite polynomial functors, and the application operation is a copyless natural function.

The rest of this section is devoted to the harder right-to-left implication, that is, to translating functorial SSTs into ordinary SSTs.

B.1 A syntactic description

The main step in the proof of this harder direction is an analysis of what can be done by copyless natural functions between polynomial functors.

We begin with *monomial functors*, i.e. polynomial functors with one component. Consider two monomial functors, say $A \mapsto A^k$ and $A \mapsto A^\ell$, for some $k, \ell \in \mathbb{N} = \{0, 1, \ldots\}$. One way of specifying a natural transformation between these two functors is to start with a function

$$\alpha \colon \{1, \ldots, \ell\} \to \{1, \ldots, k\}^+, \tag{1}$$

which we call a *syntactic description*, and to then define the natural transformation as follows. For a semigroup A, the corresponding function of type $A^k \to A^\ell$ maps a tuple $\bar{a} \in A^k$ to the tuple in A^ℓ defined by

$$\{1, \ldots, \ell\} \xrightarrow{\text{syntactic description}} \{1, \ldots, k\}^+ \xrightarrow{\text{substitute } \bar{a}} A^+ \xrightarrow{\text{semigroup operation}} A.$$

▷ Claim B.1. All natural transformations between monomial functors arise this way, i.e. they are in one-to-one correspondence with syntactic descriptions.

Proof. Concretely, the syntactic description may be recovered from the natural transformation instantiated at the free semigroup $\{1, \ldots, k\}^+$ by applying it to $(1, \ldots, k) \in (\{1, \ldots, k\}^+)^k$.

While it can be checked by hand that this works, more conceptually, it is an instance of the *Yoneda lemma*. Indeed, there is a natural bijection

$$A^k \cong A^{\{1,\ldots,k\}} \cong \{\text{semigroup homomorphisms } \{1, \ldots, k\}^+ \to A\}$$

so monomial functors are *representable*. The Yoneda lemma gives us a bijection between natural transformations $A^k \to A^\ell$ and homomorphisms $\{1, \ldots, \ell\}^+ \to \{1, \ldots, k\}^+$. ◁

The advantage of the syntactic description, which is unique, is that it allows us to define the copyless restriction in a more syntactic way, reminiscent of the definition used in Definition 3.5: a natural function between two monomial functors is copyless if and only if its syntactic description has the following property: (*) concatenating all ℓ output strings gives a string where each letter from $\{1, \ldots, k\}$ appears at most once.

To extend this definition of copylessness to a natural transformation η between general polynomial functors

$$\mathsf{F}A = \coprod_{q \in Q} A^{\dim q} \qquad \mathsf{G}A = \coprod_{p \in P} A^{\dim p},$$

we note that if $x \in \mathsf{F}A$ is in the input component q, then its image $\eta_A(x) \in \mathsf{G}A$ lives in some output component p *determined by* q *and* η *independently of* A. This can be seen from the naturality diagram below for the terminal morphism $!\colon A \to 1$:

$$\begin{array}{ccc} \mathsf{F}A & \xrightarrow{\mathsf{F}!} & \mathsf{F}1 \cong \{\text{input components}\} \\ \downarrow{\scriptstyle \eta_A} & & \downarrow{\scriptstyle \eta_1} \\ \mathsf{G}A & \xrightarrow[\mathsf{G}!]{} & \mathsf{G}1 \cong \{\text{output components}\} \end{array}$$

Thus, the restriction of η to the component q of its domain is a natural transformation of monomial functors $A^{\dim(q)} \to A^{\dim(p)}$. We then have:

▷ Claim B.2. A natural transformation between polynomial functors is copyless in the sense of Definition 3.9 if and only if for every input component, the corresponding natural transformation between monomial functors is copyless.

Proof. By considering the naturality diagrams for the homomorphisms $h\colon \{1,\dots,k\}^+ \to \mathbb{N}$ such that $h(1) + \cdots + h(k) \leq 1$. ◁

B.2 Proof of the left-to-right implication in Lemma 3.11.

Simplifying the initialization/finalization. In Definition 3.10, we allow the initial register valuation to depend in a recognizable way on the input, and similarly for some "final data" that is used by the final output function. We sketch here a translation from this model to one where the initial valuation is a constant (as in Definition 3.5), and the output is determined by a copyless natural function without auxiliary data. The idea is that the new functorial SST will simulate in parallel the old functorial SST for all possible initial valuations, and then decide after reading the whole word which of the computations to keep.

Let R be the register functor of our functorial SST, U be the update functor and K be the final data functor. Let $h\colon \Sigma^* \to S$ be a homomorphism from input words to a finite semigroup that recognizes both the initial valuation function and the final data function; that is, these functions can be decomposed respectively as $\texttt{init} \circ h$ and $\texttt{findat} \circ h$. Our simplified functorial SST will use

$$\text{the register functor } A \mapsto (\mathsf{R}A)^S \times (\mathsf{K}A)^S \times S \quad \& \quad \text{the update functor } A \mapsto (\mathsf{U}A)^S \times S$$

The initial register valuation of our simplified device is

$$\big((\texttt{init}(s))_{s\in S},\ (\texttt{findat}(s))_{s\in S},\ h(\varepsilon)\big)$$

For each input position, the corresponding update in $(\mathsf{U}A)^S \times S$ is a pair consisting of

- the constant S-indexed tuple whose components are equal to the update for this position in the original functorial SST,
- and of the image of the input letter by h.

The application of an update $((u_s)_{s\in S},\ \widehat{s})$ to a register valuation $\big((r_s)_{s\in S},\ (d_s)_{s\in S},\ \underline{s}\big)$ is

$$\big((\text{apply } u_s \text{ to } r_s)_{s\in S},\ (d_s)_{s\in S},\ \underline{s}\widehat{s}\big)$$

Finally, the final output function of the simplified functorial SST is

$$\big((r_s)_{s\in S},\ (d_s)_{s\in S},\ \underline{s}\big) \mapsto \text{apply original final input function to } r_{\underline{s}} \text{ and } d_{\underline{s}}$$

Translating a simplified functorial SST. Now, consider a simplified functorial streaming string transducer, which uses the register and update functors

$$\mathsf{R}A = \coprod_{q \in Q} A^{\dim q} \qquad \mathsf{U}A = \coprod_{p \in P} A^{\dim p}.$$

We shall convert it to an equivalent ordinary SST using the syntactic descriptions introduced in the previous subsection. For an input string $a_1 \cdots a_n$, consider the sequence of register valuations

$$v_1, \ldots, v_n \in \mathsf{R}A,$$

such that v_i arises by applying the first i register updates produced by the update oracle. Let k be the maximal dimension of the components in Q (here we use the assumption that R is a *finite* polynomial functor). Define a register valuation $w_i \in A^{k+1}$ as follows: take the register valuation used by v_i, and pad it to a tuple of length $k+1$ using some distinguished element $a_0 \in A$. In particular, since k is the maximal dimension of Q, we are guaranteed that the last coordinate with index $k+1$ stores the distinguished element a_0. We will build a streaming string transducer, as in Definition 3.5, in which the set of register names is $\{1, \ldots, k\}$.

We begin by looking at the components. Let $q_i \in Q$ be the component of the register valuation v_i. The first observation is that q_i depends only on q_{i-1} and the i-th register update. Therefore, the sequence $q_1 \cdots q_n$ can be produced by a rational function. The next observation is that, once we know the components q_{i-1} and q_i, and the register update $u_i \in \mathsf{U}A$ that would be applied in the original transducer to from v_{i-1} to v_i, then we can create a copyless register update that transforms w_{i-1} into w_i. This is done by using the syntactic descriptions of natural functions that were described above. Once we have the register valuations $w_1, \ldots, w_n$, the output of the transducer can be easily obtained.

C Proof of Claim 3.19

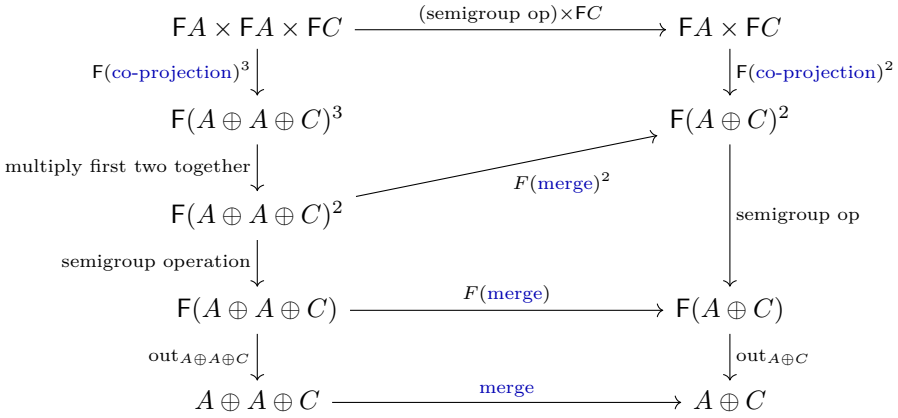
The property that we want on factorized outputs reduces immediately to the following lemma on vectorial outputs, by taking $C = 1$.

▶ **Lemma C.1.** *Let A and C be semigroups. The following diagram commutes:*

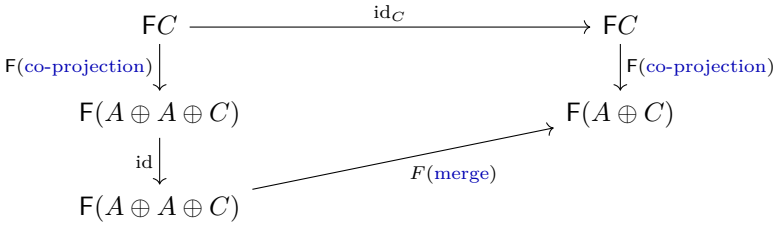
$$\begin{array}{ccc} \mathsf{F}A \times \mathsf{F}A \times \mathsf{F}C & \xrightarrow{(\textit{semigroup operation}) \times \mathsf{F}C} & \mathsf{F}A \times \mathsf{F}C \\ \Big\downarrow{\scriptstyle \textit{vectorial output}} & & \Big\downarrow{\scriptstyle \textit{vectorial output}} \\ A \oplus A \oplus C & \xrightarrow{\qquad\qquad \textit{merge} \qquad\qquad} & A \oplus C \end{array}$$

Proof. By expanding the definition of the vectorial output functions, and by using the associativity of three-fold multiplication, we see that the diagram in the lemma statement

corresponds to the outer rectangle below:



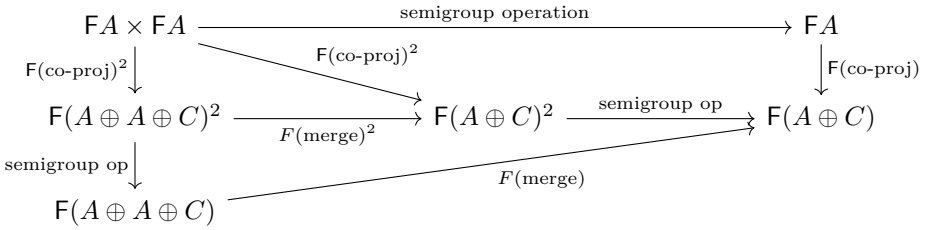
The lower rectangle commutes by naturality of the output mechanism. The middle trapeze commutes because merging is a semigroup homomorphism. Concerning the upper trapeze, it can be decomposed as the bifunctor $\times$ applied to two diagrams that we can analyze independently. We start by looking at the second of those, which is simpler:



By functoriality of F, the commutativity of this diagram reduces to

$$\text{merge} \circ (\text{co-projection of } C \text{ into } A \oplus A \oplus C) = \text{co-projection of } C \text{ into } A \oplus C$$

which is an elementary property of the coproduct $\oplus$ in any category (indeed, the generic categorical definition of merging uses a co-paring, which "cancels out" with the co-projection). There remains to check that the first of the two diagrams combined by $\times$ is also commutative; towards this purpose, we have added some parts in the middle:



The upper right trapeze commutes because F applied to the co-projection of a into $A \oplus C$ is a homomorphism. The lower triangle commutes because F(merge) is a homomorphism. Finally, the small upper left triangle can be shown to commute using the functoriality of F followed by properties of the coproduct. ◀