Algebraic Recognition of Regular Functions

Anonymous author
Anonymous affiliation

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-to-string transducers, semigroups, natural transformations

1 Introduction

This paper is about the regular string-to-string functions. This is a fundamental class of functions, which covers examples such as the string reversal function $123 \mapsto 321$ or duplication $123 \mapsto 123123$. Similarly to the class of regular languages, the class of regular functions has many equivalent descriptions, including deterministic two-way automata [17, Note 4], copyless streaming string transducers (SST) [1, Section 3] (or the earlier and very similar single0use restricted macro tree transducers [11, Section 5]), MSO transductions [10, Theorem 13], combinators [3, Section 2], a functional programming language [7, Section 6], λ-calculus with linear types [12, Theorem 3] (see also [14, Claim 6.2] and [13, Theorem 1.2.3]), decompositions *à la* Krohn–Rhodes [8, Theorem 18, item 4], etc.

The number of equivalent descriptions clearly indicates that, similarly to the class of regular languages, 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. 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 a streaming string transducer, one needs to be careful about bounding the copies among registers, and there are some delicate questions regarding lookahead. Each of the combinator calculi has a long list of combinators. Similar remarks apply to the other calculi. 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. Such a model would give further evidence in favour of the accepted notion of regularity for string-to-string functions, and answer questions for the other models such as "why not allow this or that feature to two-way automata?", "why not allow copying for streaming string transducers?" or "why not add this or that combinator?".

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

$$\Sigma^* \xrightarrow{h} \mathsf{F}(\Gamma^*) \xrightarrow{\mathrm{out}_{\Gamma^*}} \Gamma^*,$$

where F is a functor from the category of semigroups to itself that maps finite semigroups to finite semigroups, h is a semigroup homomorphism, and the output function out_{Γ^*} is natural in the sense of natural transformations. 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 features of the regular string-to-string functions, is not explicitly included in the model, but it is a provable consequence of it.

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. Some results will work in the slightly more general case where the input or output is a semigroup that is not necessarily a finitely generated free monoid, 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 $f : A \to B$ a morphism $\mathsf{F}f : \mathsf{F}A \to \mathsf{F}B$. These maps need to be consistent with composition of morphisms, and the identity must go to the identity. An example of a functor is the *forgetful functor* from the category of semigroups to the category of sets, which maps a semigroup to its underlying set, and a semigroup homomorphism to the corresponding function on sets. The forgetful functor is an example of a semigroup-to-set functor, which goes from the category of semigroups to the category of sets. The following example discusses semigroup-to-semigroup functors.

▶ **Example 2.1.** A semigroup-to-semigroup can be seen as a semigroup construction. Here are sompe examples.

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.

Reverse. 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.

Non-empty lists. This functor maps a semigroup A to the free semigroup A^+ that consists of non-empty lists (or strings) over the alphabet A equipped with concatenation. On morphisms, the functor is defined element-wise (or letter-wise). A similar construction would make sense as a set-to-semigroup functor.

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

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

Variants of the powerset functor require the subsets to be nonempty, or finite, or both.

We now present the central definition of this paper.

▶ **Definition 2.2** (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*. The output function does not need to be a semigroup homomorphism. The output mechanism is required to be* natural*, which means that the diagram*

$$\begin{tikzcd}
\mathsf{F}A \arrow[r, "\mathsf{F}h"] \arrow[d, "\textit{output function for } A"'] & \mathsf{F}B \arrow[d, "\textit{output function for } B"] \\
A \arrow[r, "h"'] & B
\end{tikzcd}$$

commutes for every semigroup homomorphism $h : A \to B$.

In the language of category theory, the naturality condition from the above definition says that the output mechanism is a natural transformation of type

$$\begin{tikzcd}
\text{Semigroups} \arrow[r, bend left, "\text{apply } \mathsf{F} \text{ and return underlying set}"] \arrow[r, bend right, "\text{return underlying set}"'] \arrow[r, phantom, "\Downarrow"] & \text{Sets.}
\end{tikzcd}$$

We are mainly interested in the special case of transducer semigroups where the functor F is *finiteness-preserving*, i.e. it maps finite semigroups to finite semigroups. This special case will correspond to the regular string-to-string functions. Some minor results about the general case, when F is not necessarily finiteness-preserving, are presented in Section 2.1.

The purpose of transducer semigroups is to define functions between semigroups, as explained in the following definition.

▶ **Definition 2.3.** *We say that a function $f : A \to B$ between semigroups, not necessarily a homomorphism, is* recognized *by a transducer semigroup if it can be decomposed as*

$$A \xrightarrow{\quad h \quad} \mathsf{F}B \xrightarrow{\textit{output function for } B} B \qquad \textit{for some semigroup homomorphism } h.$$

The definition discusses functions between arbitrary semigroups, but we will mainly care about string-to-string functions $f : \Sigma^* \to \Gamma^*$, i.e. the special case when both the input and output semigroups are finitely generated free monoids. Although the case that we care about involves monoids, which are a special case of semigroups, it will be useful in the proofs to define the model in terms of semigroups.

▶ **Example 2.4.** 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 2.5.** Consider the transducer semigroup in which the functor is the identity, and the output function for A is $a \in A \mapsto aa \in A$. (This output function is not a semigroup homomorphism.) 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 is some homomorphism. In particular, if h is the identity on the monoid Σ^*, which is also a semigroup, then we get the duplicating function on strings over the alphabet Σ.

▶ **Example 2.6.** Consider the reversing functor from Example 2.1. Define the output mechanism to be the identity. Using this transducer semigroup, we can recognize the string reversal function.

▶ **Example 2.7.** Consider the functor $A \mapsto A^+$, as in Example 2.1, and an output function

$$[a_1, \ldots, a_n] \in A^+ \mapsto \underbrace{(a_1 \cdots a_n) \cdots (a_1 \cdots a_n)}_{n \text{ times}} \in A.$$

This transducer semigroup recognizes the squaring function $w \in \Sigma^+ \mapsto w^{|w|} \in \Sigma^*$ for nonempty strings, which is illustrated in the following example: `123` $\mapsto$ `123123123`.

2.1 Two simple characterizations

We begin with two simple theorems, which are about transducer semigroups where the functor is not necessarily finiteness-preserving. These results describe two classes of string-to-string functions: all functions (Theorem 2.8) and functions that reflect recognizability (Theorem 2.10). The main contribution of this paper, presented in Section 3, characterizes the regular functions using finiteness-preserving functors.

All functions. The first theorem shows that, without any further restrictions, transducer semigropus can recognize all functions.

▶ **Theorem 2.8.** *Every string-to-string function is recognized by a transducer semigroup.*

Proof. We prove a slightly stronger result, namely that every function between two semigroups A and B is recognized by a transducer semigroup. For a semigroup A, we define 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 2.1: 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 $f : A \to B$ 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 finite sets (languages can be seen as the special case of functions into a set with two elements "yes" and "no"). We say that a function from a possibly infinite semigroup A to some finite set X is *recognizable* if it factors through some semigroup homomorphism from A to some finite semigroup. A function $f : B \to A$ between semigroups, not necessarily a semigroup homomorphism, is called *recognizability reflecting* if for every recognizable function $g : A \to X$, the composition $g \circ f$ is recognizable.

▶ **Example 2.9** (Factorials). 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 $\{1!, 2!, \dots\}$, with finitely many exceptions. Take any function $f : \mathbb{N} \to \mathbb{N}$ 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. A function with conditions (a) and (b) can be chosen in uncountably many ways, even if we require that it has linear growth.

In light of the above example, there are too many recognizability reflecting functions to allow a machin model, or some other effective syntax. A (non-effective) syntax is given in the following theorem, which is proved the same way as Theorem 2.8.

▶ **Theorem 2.10.** *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 C, the corresponding output function of type $\mathsf{F}A \to A$ is recognizable.*

3 The regular functions

The two straightforward constructions in Theorems 2.8 and 2.10 amount to little more than symbol pushing. In this section, we present a more substantial characterization, which is the main result of this paper. In this characterization, we use functors that are finiteness-preserving. This is a strengthening of the condition from Theorem 2.10: if F is finiteness-preserving, then for every finite semigroup C, 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 but not finiteness preserving. As we will see, the stronger condition will characterize exactly the regular string-to-string functions.

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

▶ **Example 3.1.** Consider the powerset functor $\mathsf{P}A$ from Example 2.1. This is a finiteness-preserving functor, 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.*

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) $\Rightarrow$ (2)

Section 3.3 proves the hard implication in the theorem, namely (1) $\Leftarrow$ (2)

Before continuing, we remark on one advantage of the characterization, namely a straightforward proof of closure under composition. It is easy to see that functions recognized by finiteness-preserving transducer semigroups are closed under composition. This is because finiteness-preserving functors are closed under composition, natural output functions are also closed under composition, and natural output functions commute with functors. In contrast, for some (but not all) models defining regular string-to-string functions, closure under composition requires a non-trivial construction, examples of such models include two-way transducers [9, Theorem 2] or copyless SST [2, Theorem 1].

3.1 Defininition of streaming string transducers

In this section, we formally describe the regular functions, using a model based on streaming string transducers. 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 is a minor variation on streaming string transducers, which use 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

$$\underbrace{R \to A}_{\text{the set of } \textit{register valuations}} \qquad \underbrace{R \to (A + R)^+}_{\text{the set of } \textit{register updates}}.$$

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

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

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 can be applied to a register valuation, giving a new register valuation.

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, a *rational function*, is defined to be a length-preserving[1] string-to-string function such that for some recognizable function

$$f : (\{\text{current, not current}\} \times (\text{input alphabet}))^+ \to \text{output alphabet},$$

for every input string the i-th output letter is obtained by applying the function to the string that is obtained from the input string by setting the first coordinate to "current" for the i-th position, and "not current" for the remaining positions.

In a rational function, the output of label of the i-th position is allowed to depend on letters of the input string that are to the right of the i-th input position; this corresponds to regular lookahead in a streaming string transducer.

Having defined register updates and rational functions, we are ready to define the variant of streaming string transducers used in this paper.

▶ **Definition 3.3.** *The syntax of a streaming string transducer 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 designated* final register*.*
- *An update oracle, which is a rational function of type*

$$\Sigma^* \to (\textit{copyless register updates})^*.$$

The semantics of the transducer is a function of type $\Sigma^* \to A$ defined as follows. When given an input string, the transducer 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 transducer is obtained by returning the semigroup element stored in the designated final register.

The model described above is easily seen to be equivalent to streaming string transducers with regular lookahead, which are one of the equivalent models defining the regular string-to-string functions, see [6, Section 12].

3.2 From a regular function to a transducer semigroup

Having defined the transducer model, we prove the easy implication in Theorem 3.2.

Suppose that a string-to-semigroup function $f : \Sigma^* \to A$ is computed by some 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. Here is an example, which uses three registers X, Y, Z and the semigroup $A = (\{0,1\}, \cdot)$:

$$\underbrace{\begin{array}{l} X := 01Y1111X111 \\ Y := 01011 \end{array}}_{\text{not in normal form}} \qquad \underbrace{\begin{array}{l} X := 0Y1X1 \\ Y := 0 \end{array}}_{\text{in normal form}}$$

Every 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. The register updates before and after normalization act in the same way

[1] Often in the literature, rational functions are not required to be length-preserving, see e.g. [16, p. 525], but in this paper, we only need the length-preserving case.

on register valuations. If a register 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 is finite, then the set of copyless register updates in normal form, call it $\mathsf{S}A$, is also finite. (This would not be true for copyful register updates.) The set $\mathsf{S}A$ of register updates in normal form can be equipped with a composition operation

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

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 register valuation v and register updates u_1 and u_2. Therefore, $A \mapsto \mathsf{S}A$ is a finiteness-preserving semigroup functor. (With the natural extension to morphisms, where the homomorphism is applied to every semigroup element in a right-hand side.)

The functor S described above is not 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 will also take into account the update oracle. Consider the update oracle in the streaming string transducer from the assumption of the easy implication. Since the update oracle is a rational function, there is a semigroup homomorphism

$$h : (\{\text{current, not current}\} \times (\text{input alphabet}))^* \to B,$$

into a finite semigroup such that the i-th letter produced by the update oracle depends only on the result of applying this homomorphism to the string obtained from the input in the way that was described in the definition of rational functions. Without loss of generality, we assume that B is a monoid. The functor F is defined as follows. If the input semigroup is A, then the underlying set of the output semigroup $\mathsf{F}A$ is

$$B \quad \times \quad \underbrace{(B \times B) \to \mathsf{S}A}_{\substack{\text{functions of this kind} \\ \text{are called } \textit{conditional} \\ \textit{register updates}}} \quad \times \quad \underbrace{R \to A}_{\substack{\text{register} \\ \text{valuations}}}.$$

The semigroup operation is defined as follows. On the third coordinate (in blue), we use the trivial left zero semigroup structure. On the first two coordinates (in black), the semigroup structure is defined[2] so that the product of two pairs (b_1, φ_1) and (b_2, φ_2) is the pair consisting of b_1b_2 and the function

$$(c_1, c_2) \mapsto \varphi_1(b_1, c_2b_2) \cdot \varphi_2(b_1c_1, b_2).$$

The construction F is extended to morphisms in the same way as S.

The output mechanism in the transducer semigroup is defined as follows. When given $(b, \varphi, v) \in \mathsf{F}A$, the output function returns the element of the semigroup A that is obtained as follows: (1) apply φ to the pair consisting of the neutral elements in the monoid B, yielding a register update in $\mathsf{S}A$; then (2) apply this register update to the register valuation v, yielding some new register valuation; and then (3) from the resulting register valuation, return the semigroup element stored in the distinguished output register. Checking the naturality condition is left to the reader.

Using the transducer semigroup defined above, we can recognize the function computed by our streaming string transducer.

[2] This definition coincides with the two-sided semidirect product of monoids from [15, Section 6], when applied to the monoids B and $\mathsf{S}A$.

3.3 From a transducer semigroup to a regular function

We now turn to the difficult implication (2) $\Rightarrow$ (1) in Theorem 3.2.

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 streaming string transducers, but a bit more abstract. The abstraction arises by using polynomial functors instead of registers, as described below.

Define a *polynomial functor* to be a semigroup-to-set functor of the form

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

where Q is some possibly infinite set, called the *components*, with each component having an associated *dimension* in $\{0, 1, \ldots\}$. 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 one with finitely many components. For example, $A \mapsto A^2 + A^2 + A$ is a finite polynomial functor. A 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 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 described in the following definition.

▶ **Definition 3.4** (Natural functions). *Let* F *and* G *be polynomial functors, let* A *be a semigroup. A function*[3] $f : \mathsf{F}A \to \mathsf{G}A$ *is called* natural *if it can be extended to natural transformation of type* $\mathsf{F} \Rightarrow \mathsf{G}$*. This means that there is a family of functions, with one function* $f_A : \mathsf{F}A \to \mathsf{G}A$ *for each semigroup* A*, such that* $f = f_A$*, and the the diagram*

$$\begin{tikzcd} \mathsf{F}A \arrow[r, "\mathsf{F}h"] \arrow[d, "f_A"'] & \mathsf{F}B \arrow[d, "f_B"] \\ \mathsf{G}A \arrow[r, "h"] & \mathsf{G}B \end{tikzcd}$$

commutes for every semigroup homomorphism h*.*

▶ **Example 3.5.** Consider the polynomial functors

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

where 1 represents the singleton set A^0. 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

[3] This function is not necessarily a semigroup homomorphism. In fact, it would not even make sense call it a homomorphism, since the functors F and G produce sets and not semigroups.

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.

▶ **Definition 3.6** (Copyless natural function). *A natural function $f : \mathsf{F}A \to \mathsf{G}A$ is called* copyless *if it arises from some natural transformation with the following property: when instantiated to the semigroup*[4] $(\mathbb{N}, +)$*, the corresponding function of type* $\mathsf{F}\mathbb{N} \to \mathsf{G}\mathbb{N}$ *does not increase the norm. Here, the norm of an element in a polynomial functor* $\mathsf{F}\mathbb{N}$ *or* $\mathsf{G}\mathbb{N}$ *is defined to be the sum of numbers that appear in it.*

Having defined functions that are natural and copyless, we now describe the more abstract model of streaming string transducers that is be 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 finite polynomial functors, together with an explicitly given application function. Another minor difference is that we allow the model to define partial functions; this will be useful in the proof.

▶ **Definition 3.7.** *The syntax of a functorial streaming string transducer is given by:*

- *A finite* input alphabet Σ *and an* output semigroup A*.*
- *Two finite polynomial functors* R *and* U*, called the* register *and* update *functors, together with a function of type* $\mathsf{R}A \times \mathsf{U}A \to \mathsf{R}A$*, called* appliction*, which is natural and copyless.*
- *A distinguished* initial register valuation *in* $\mathsf{R}A$*.*
- *A* final function *of type* $\mathsf{R}A \to A + 1$*, which is natural and copyless.*
- *An* update oracle*, which is a rational function of type* $\Sigma^* \to (\mathsf{U}A)^*$*.*

The semantics of the transducer is a partial function of type $\Sigma^* \to A$ defined as follows. As in Definition 3.3, for every input string we use the initial register valuation, the application function and the update oracle to define a sequence of register valuations in $\mathsf{F}A$. We then apply the final function to the last register valuation, yielding a result in $A + 1$. If this result is in the 1 part, then the output of the transducer is undefined, otherwise the output of the transducer is the semigroup element stored in the A part. We will care about transducers that compute total functions, which corresponds to the property that for every input string, the last register valuation is in the A part of $A + 1$.

▶ **Lemma 3.8.** *The models defined in Definitions 3.3 and 3.7 define the same (total) string-to-semigroup functions.*

Coproducts and views. Apart from the more abstract transducer model from Definition 3.7, 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.

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.

[4] The choice of the semigroup $(\mathbb{N}, +)$ in the Definition 3.6 is not particularly important. For example, the same notion of copylessness would arise if instead of $(\mathbb{N}, +)$, we used the semigroup $\{0, 1\}$ with addition up to threshold 1 (i.e. the only way to get zero is to add two zeros). In the appendix, we present a more syntactic characterization of copyless natural transformations, which will be used later on when proving equivalence with streaming string transducers.

The *coproduct* 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 $\{a, b\}^+$:

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

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.

The polynomial functors that we use in our proof will arise using coproducts with the singleton semigroup 1. Consider the semigroup-to-set functor $A \mapsto A \oplus 1$, which maps a semigroup to the underlying set of its coproduct with the singleton semigroup. Although not defined as a polynomial functor, this functor is isomorphic to a polynomial functor. This is because for every semigroup A there is a bijective correspondence between the sets

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

where the dimension of q is defined to be the number of times that the first copy of 1 appears in q. Furthermore, this bijection is natural, and thus we can speak of $A \oplus 1$ as being a polynomial functor. This remark applies to similar constructions, which involve a coproduct of several copies of A with several copies of 1, such as $A \oplus A \oplus A \oplus 1 \oplus 1$.

The crucial property of semigroups that will be used in our proof is described in Lemma 3.9 below, which says that a coproduct can be reconstructed based on certain partial information. This partial information is described using the following operations on coproducts.

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:

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

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 can be reconstructed from its shape and views, as stated in the following lemma.

▶ **Lemma 3.9.** *Let $A_1, \ldots, A_n$ be semigroups. The function of type*

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

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

Proof. The input can be reconstructed from the output as follows. Start with the shape, and replace the entries from 1 with the semigroup elements used in the views. ◀

This lemma seems to contain the essential property of semigroups that makes the construction work. Our theorem will also be true for other algebraic structures for which the lemma is true, such as forest algebras. However, the lemma seems to fail for certain algebraic structures, such as groups, even if we allow 1 to be replaced by some fixed finite group. Another example where the lemma seems to fail is the monad of weighted sums of words (i.e. this monad corresponds to weighted automata).

Proof of (2) ⇒ (1) **in Theorem 3.2.** We have now collected all necessary ingredients to prove the hard implication Theorem 3.2. Consider some transducer semigroup, with the functor being F, and fix a string-to-semigroup function $f : \Sigma^* \to A$ that decomposes as some homomorphism $h : \Sigma^* \to \mathsf{F}A$ followed by the output function of type $\mathsf{F}A \to A$. We will show that this function is computed by some functorial streaming string transducer as in Definition 3.7.

The main idea behind the proof is that, using coproducts, we will be able to identify the origin semantics [4] of the function f which means that we will now which parts of the input string are responsible for which parts of the output semigroup. This will be done using coproducts, as described below.

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

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

that is obtained by composing the three functions described below

$$\begin{tikzcd}
\mathsf{F}A_1 \times \cdots \times \mathsf{F}A_n \arrow[d, "\mathsf{F}(\text{co-projection}) \times \cdots \times \mathsf{F}(\text{co-projection})"] \\
\mathsf{F}(A_1 \oplus \cdots \oplus A_n) \times \cdots \times \mathsf{F}(A_1 \oplus \cdots \oplus A_n) \arrow[d, "\text{semigroup operation}"] \\
\mathsf{F}(A_1 \oplus \cdots \oplus A_n) \arrow[d, "\text{output mechamism for } A_1 \oplus \cdots \oplus A_n"] \\
A_1 \oplus \cdots \oplus A_n.
\end{tikzcd}$$

To illustrate the definitions in this section, we use a running example with the transducer semigroup from Example 2.5 for the duplicating functions. In this transducer semigroup, the functor is the identity $\mathsf{F}A = A$, and the output mechanism is $a \mapsto aa$. The duplicating function on $\{a,b\}^*$ is obtained by composing the identity homomorphism on $\{a,b\}^* = \mathsf{F}\{a,b\}^*$ with the output function. Here is an example of the vectorial output function, applied to the semigroups $A_1 = 1$ and $A_2 = \{a,b\}^*$:

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

The vectorial output function is natural in all of its arguments, which means that

$$\begin{tikzcd}
\mathsf{F}A_1 \times \cdots \times \mathsf{F}A_n \arrow[r, "\text{vectorial output function}"] \arrow[d, "\mathsf{F}h_1 \times \cdots \times \mathsf{F}h_n"'] & A_1 \oplus \cdots \oplus A_n \arrow[d, "h_1 \oplus \cdots \oplus h_n"] \\
\mathsf{F}B_1 \times \cdots \times \mathsf{F}B_n \arrow[r, "\text{vectorial output function}"] & B_1 \oplus \cdots \oplus B_n
\end{tikzcd}$$

commutes for every semigroup homomorphisms $h_1, \ldots, h_n$. 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 string-to-semigroup function $f : \Sigma^* \to A$ in the assumption of the hard implication from Theorem 3.2. Using the vectorial output mechanism, we will be able to track the origins in the output of the function f, with respect to some partition of the input string into several nonempty parts. For strings $w_1, \ldots, w_n \in \Sigma^*$, define the corresponding *factorized output*, denoted by

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

to be the result of first applying the semigroup homomorphism $h : \Sigma^* \to \mathsf{F}A$ to all the strings, then applying the factorized output function, and finally removing the elements of the output co-product that correspond to input coordinates $i \in \{1, \ldots, n\}$ in which the string w_i was the empty string ε. Here is the factorized output illustrated in our running example:

$$\langle abbbbb | \varepsilon | bbabaaa \rangle = \boxed{abbbbb}\ \boxed{bbabaaa}\ \boxed{abbbbb}\ \boxed{bbabaaa} \in \{a,b\}^+ \oplus \{a,b\}^* \oplus \{a,b\}^*.$$

Here, we use colours to distinguish which of the three parts of the input is used; the empty middle part has black colour which is not used in the output. As the above example shows, the factorized output tells us which parts of the output string come from which of the three parts in the input string.

We also use a similar notation but with some input strings underlined, e.g. the input could be $\langle \underline{abbbbb} | \varepsilon | bbabaaa \rangle$ with an underline for the first red part. In the underlined case, before applying the vectorial output function, we use h for the non-underlined strings we apply h, and

$$\Sigma^+ \xrightarrow{h} \mathsf{F}A \xrightarrow{\mathsf{F}!} \mathsf{F}1.$$

for the underlined strings. (As before, the empty input strings are removed from the output.) In our running example, we have

$$\langle \underline{abbbbb} | \varepsilon | bbabaaa \rangle = \boxed{1}\ \boxed{bbabaaa}\ \boxed{1}\ \boxed{bbabaaa}.$$

The following lemma uses the ingredients described in this section to define what is essentially the register update function in a streaming string transducer recognizing our function f. As discussed earlier in this section, we consider functors such as $A \oplus 1$ and $1 \oplus A \oplus 1$ as polynomial semigroup-to-set functors, which enables us to talk about natural and copyless functions that operate on them.

▶ **Lemma 3.10.** *There is a copyless natural function*

$$\delta : (A \oplus 1) \times (1 \oplus A \oplus 1) \to A \oplus 1$$

such that every strings $w, v \in \Sigma^*$ and letter $a \in \Sigma$, one obtains $\langle wa|\underline{v}\rangle$ by applying δ to the pair consisting of $\langle w|\underline{av}\rangle$ and $\langle \underline{w}|a|\underline{v}\rangle$.

Using the above lemma, one can easily define a functorial streaming string transducer that computes function f, thus completing the proof of Theorem 3.2. The idea is that the transducer maintains the following invariant: after processing the first i letters in an input string $a_1 \cdots a_n$, the configuration of the transducer is the factorized output $\langle a_1 \cdots a_i|\underline{a_{i+1} \cdots a_n}\rangle$. After processing all input letters, the configuration stores the output string. The details are in the appendix. It remains to prove the lemma, which is done using the operations on coproducts described earlier in this section.

Proof of Lemma 3.10. We use the following claim, which is proved using naturality of the output mechanism.

▷ Claim 3.11. $\langle wa|\underline{v}\rangle$ is obtained from $\langle w|a|\underline{v}\rangle$ by merging the first two parts.

Since merging the first two parts is a copyless natural function, 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. To complete the proof of the lemma, we will show that latter value $\langle w|a|\underline{v}\rangle$ can also be obtained by applying some copyless natural function to the pair consisting of $\langle wa|\underline{v}\rangle$ and $\langle \underline{w}|a|\underline{v}\rangle$. This will be done using (an extension of) Lemma 3.9. Consider the function of type

$$A \oplus A \oplus 1 \to \underbrace{(1 \oplus A)}_{\substack{\text{first}\\ \text{view}}} \times \underbrace{(1 \oplus A)}_{\substack{\text{second}\\ \text{view}}} \times \underbrace{(1 \oplus 1)}_{\substack{\text{third}\\ \text{view}}} \times \underbrace{(1 \oplus 1 \oplus 1)}_{\text{shape}},$$

which is the injective function from Lemma 3.9 in the special case of the coproduct $A \oplus A \oplus 1$. We use the name *deconstruction* for this function. By the same proof as in Lemma 3.9, this function is not only injective, but it also has a one-sided inverse, i.e a function of type

$$(1 \oplus A) \times (1 \oplus A) \times (1 \oplus 1) \times (1 \oplus 1 \oplus 1) \to A \oplus A \oplus 1,$$

which we call *reconstruction*, such that deconstruction followed by reconstruction is the identity on $A \oplus A \oplus 1$. Furthermore, reconstruction is natural and copyless.

By the above observations, one can obtain the factorized output $\langle w|a|\underline{v}\rangle$ by applying reconstruction to the following four items (the equalities below are proved using Claim 3.11):

1. First view of $\langle w|a|\underline{v}\rangle$, which is equal to $\langle w|\underline{av}\rangle$.
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 of the lemma, it remains to justify that the last three items can be obtained from $\langle \underline{w}|a|\underline{v}\rangle$ 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

Another advantage of the model is that, as one would expect from an abstract result, it lends itself naturally to generalizations.

The definition of a transducer semigroup can applied to other algebras, and not just semigroups, by taking some monad T, and considering functions that can be decomposed as

$$\mathsf{T}\Sigma \xrightarrow{h} \mathsf{F}\mathsf{T}\Gamma \xrightarrow{\mathrm{out}_{\mathsf{T}\Gamma}} \mathsf{T}\Gamma,$$

for some endofunctor F in the category of Eilenberg-Moore algebras for the monad T, some algebra homomorphism h, and some natural transformation out. An example of this approach is forest algebras [5, 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 transduction. We believe that these results apply even further, namely for graphs of bounded treewidth, modeled using suitable monads [5, Section 6]. The crucial property is that Lemma 3.9, about reconstructing a co-product from its views, holds for other monads than just the list monad for semigroups. Unfortunately, this lemma fails for some monads, such as the monad of 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.

Another direction is characterizing other classes of string-to-string functions, such as the rational functions or the polyregular functions. In this paper, we have discovered that, somehow 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, such that other classes of functions are described, e.g. linear growth is replaced by polynomial growth.

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, 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`.

4 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.

5 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. `https://www.mimuw.edu.pl/~bojan/papers/algebra-26-aug-2020.pdf`, 2020.

6 Mikołaj Bojańczyk and Wojciech Czerwiński. An automata toolbox. Lecture notes for a course at the University of Warsaw (version of February 6, 2018), 2018. URL: `https://www.mimuw.edu.pl/~bojan/paper/automata-toolbox-book`.

7 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`.

8 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`.

9 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`.

10 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`.

11 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`.

12 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`.

13 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://nguyentito.eu/thesis.pdf`.

14 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`.

15 John Rhodes and Bret Tilson. The kernel of monoid morphisms. *J. Pure Appl. Algebra*, 62(3):227–268, 1989.

16 Jacques Sakarovitch. *Elements of automata theory*. Cambridge University Press, 2009.

17 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 2.10

We prove a slight strenghening of the theorem, which concerns not only string-to-string functions, but also functions $f : A \to B$ between semigroups, not necessarily homomorphisms, such that the target semigroup B is finitely generated.

(1) $\Rightarrow$ (2). We use a similar construction as in the proof of Theorem 2.8. Let $f : A \to B$ 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 2.8, 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 2.8.

The output mechanism is function application with f inserted as an interface i.e. $(a, g) \mapsto g(f(a))$. We now argue that the output mechanism is a recognizable function. Suppose that C is finite. To prove that the output mechanism is recognizable, we show that the inverse image of every $c \in C$ is a recognizable subset of $\mathsf{F}C$. This inverse image is the union, ranging over the finitely many semigroup homomorphisms $g : B \to C$, of sets

$$\{(a, g) \mid \ g(f(a)) = c \ \}.$$

Each of these sets is recognizable, by the assumption that f is recognizability reflecting. By the assumption that B is finitely generated and C IS finite, there are finitely many choices for g, and therefore the union is finite. This implies that the inverse image of c is recognizable, as a finite union of recognizable subsets of $\mathsf{F}C$.

The transducer semigroup defined above recognizes the function f via the homomorphism $a \in A \mapsto (a, \mathrm{id})$.

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

$$\begin{tikzcd}A \arrow[r, "h"'] & \mathsf{F}B \arrow[r, "\mathrm{out}_B"'] & B\end{tikzcd}$$

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

$$\begin{tikzcd}B \arrow[r, "g"'] & C \arrow[r, "\text{accepting set}"'] & \{\text{yes, no}\}\end{tikzcd}$$

where C is a finite semigroup. We want to show that is inverse image of the language under f is also recognizable. Consider the following diagram.

$$\begin{tikzcd}
A \arrow[r, "h"] & \mathsf{F}B \arrow[rr, "\mathrm{out}_B"] \arrow[d, "\mathsf{F}g"'] & & B \arrow[d, "g"] & \\
 & \mathsf{F}C \arrow[rr, "\mathrm{out}_C"'] \arrow[dr, "\text{some homomorphism}"'] & & C \arrow[r, "\text{accepting set}"'] & \{\text{yes, no}\} \\
 & & D \arrow[ur, "\text{some function}"'] & &
\end{tikzcd}$$

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 $\{\text{yes, no}\}$ describes the inverse image under 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 $\{\text{yes, no}\}$. The lower path is a recognizable function, since the first three arrows are semigroup homomorphisms and D is finite.

B Proof of Lemma 3.8

In this part of the appendix, we prove Lemma 3.8, which says that the models defined in Definitions 3.3 and 3.7 define the same (total) string-to-semigroup functions.

The easy implication is left-to-right. A streaming streaming transducer as in Definition 3.3 can be seen as a special case of an SST as in Definition 3.7, because the sets of register valuations and register updates are constructed using finite polynomial functors, and the application operation is natural and copyless.

The rest of this section is devoted to the harder right-to-left implication.

A syntactic description. The main step in the proof of the right-to-left implication is a syntactic description of what can be done by copyless natural functions between finite polynomial functors.

We begin with *monomial functors*, i.e. polynomial functors with one component. Consider two monomial functors, say A^k and A^ℓ, 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

$$\alpha : \{1, \ldots, \ell\} \to \{1, \ldots, k\}^+, \tag{2}$$

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

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

It turns out that all natural transformation between monomial functors arise this way, i.e. they are in one-to-one correspondence with syntactic descriptions. To see this, the syntactic description is recovered by using the natural transformation for the free semigroup $A = \{1, \ldots, k\}^+$, and applying it to the tuple $(1, \ldots, k) \in A^k$. 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 3.3: a natural function between two monomial functors is copyless if and only if its syntactic description has the following property: (*) after concatenating all ℓ output strings gives a string where each letter from $\{1, \ldots, k\}$ appears at most once.

We now extend this syntactic description to copyless natural functions between general finite polynomial functors

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

which are not necessarily monomial. Such natural transformations also admit syntactic descriptions: for every input component q, there is some designated output component p, and a natural transformation $A^{\dim q} \to A^{\dim p}$. The set of possible syntactic descriptions is

$$\prod_{q \in Q} \coprod_{p \in P} \dim p \to (\dim q)^+.$$

Again, one can show that all natural transformations arise this way. The natural transformation is called copyless if for every q, the corresponding natural transformation between monomial functors is copyless.

Proof of the left-to-right implication in Lemma 3.8. Using the above syntactic descriptions, we cna complete the proof of the harder left-to-right implication in Lemma 3.8. Consider a functorial streaming string transducer as in Definition 3.7, which uses a register and update functors

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

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. 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 show a streaming string transducer, as in Definition 3.3, 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, one 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 Last step in the proof of Theorem 3.2

In this part of the appendix, we finish the proof of Theorem 3.2, by using Lemma 3.10 to define a functorial streaming string transducer to define the function $f : \Sigma^* \to A$ that is recognized by a transducer semigroup with a finiteness-preserving functor.

Using Lemma 3.10, we can design a device that recognizes our desired function $w \mapsto \langle w \rangle = f(w)$, and which is almost a functorial SST as in Definition 3.7. We say "almost", because the device will use register and update functors that are infinite polynomial functors; this construction will be later improved so that it becomes finite. The register and update functors are the (infinite) polynomial functors

$$\mathsf{R}A = 1 \oplus A \qquad \mathsf{S}A = 1 \oplus A \oplus 1.$$

As mentioned above, these are not a finite polynomial functors; we will resolve this problem shortly. Beyond that, the construction is immediate. Consider an input string $a_1 \cdots a_n$. The device begins its computatin with the initial register value

$$\langle \varepsilon | \underline{a_1 \cdots a_n} \rangle \in A \oplus 1.$$

This value does not depend on the input string, since it is always equal to the unique element of $1 \oplus A$ that does not use A. The rational function in the transducer is defined so that the i-th lettter of its output string is

$$\langle \underline{a_1 \cdots a_{i-1}} | a_i | \underline{a_{i+1} \cdots a_n} \rangle \in 1 \oplus A \oplus 1$$

We will explain shortly how these letters can be computed by a rational function. Thanks to Lemma 3.10, after applying all the register updates produced by this rational function to the initial register valuation, the register valuation at the end is

$$\langle a_1 \cdots a_n | \varepsilon \rangle \in A \oplus 1,$$

which is the same as the output when viewed as an element of $A \oplus 1$, as required in Definition 3.7 for representing the output of a partial function.

We are left with proving that the update oracle is a rational letter-to-letter function, and resolving the issue that the two functors R and S are not finite polynomial functors.

To see why the update oracle is a rational letter-to-letter function, we observe that

$$\langle \underline{a_1 \cdots a_{i-1}} | a_i | \underline{a_{i+1} \cdots a_n} \rangle \in 1 \oplus A \oplus 1$$

depends only the letter a_i, as well as the images of the words $a_1 \cdots a_{i-1}$ and $a_{i+1} \cdots a_n$ under the semigroup homomorphism obtained by composing h with $\mathsf{F}! : \mathsf{F}A \to \mathsf{F}1$. Since the target semigroup $\mathsf{F}1$ of this homomorphism is a finite, by the assumption that the functor is finiteness preserving, it follows that the update oracle is a rational letter-to-letter function.

We now explain how to turn R and S into finite polynomial functors. The key observation is that not all of $1 \oplus A$ need be used for the register values, only a small part of it, and likewise for the update functor. More formally, consider the natural bijection

$$A \oplus 1 \quad \cong \quad \coprod_{q \in 1 \oplus 1} A^{\dim q}$$

that was discussed in Section 3.3. If we apply this bijection to a factorized output $\langle w | \underline{v} \rangle \in A \oplus 1$, then the corresponding component will be $\langle \underline{w} | \underline{v} \rangle$. Since the latter depends only on $\underline{w}$ and $\underline{v}$, and these take values in the finite semigroup $\mathsf{F}1$, it follows that there are only finitely many components of $A \oplus 1$ that will be used to represent values from of the form $\langle w | \underline{v} \rangle$. Therefore, instead of using $\mathsf{R}A$ to be all of $A \oplus 1$, we can restrict it to those finitely many components, giving thus a finite polynomial functor. The same argument applies to the update functor $\mathsf{S}A$.