Transducers

Mikołaj Bojańczyk

September 7, 2026

Preface

This book started out as a collection of lecture notes for a course on transducers, taught at the University of Warsaw in 2024 and 2026, but it is meant to evolve into a more comprehensive resource on transducers. I would recommend visiting the online version of the book, which links to a Lean formalisation of the entire text. I hope that the presence of such a formalisation will allow others to build on results presented here without fear of errors. Also, I hope that the Lean formalisation can free me to adopt a readable and informal writing style, with pictures and examples, since doubts can be resolved conclusively by reading the linked formalisation.

Acknowledgements. I would like to thank the students of the course for their feedback, and Antek Puch for his support in selecting exercises. I would also like to thank Rafał Stefański and Anshul Goyal for helping start the Lean formalisation project. Ultimately, the formalisation was done using the AI assistant Aristotle. (I confess that I have done little more than spot checks to see if the formalisation crresponds to the text, but I hope that convenient web interface with direct links to Lean will ensure that any confabulations are exposed.) I have also used AI, mainly Claude, to: autocomplete when writing, find errors, write draft solutions for the more routine exercises (roughly one in ten of the solutions had a first draft written by AI), and implement the design of the web version. The web version uses reflowtex by Radek Piórkowski, which allows displaying TeX on web pages.

Lean. All theorems are proved in Lean without any assumptions. The same is true for all exercise solutions, with two exceptions: two solutions rely on external assumptions (Ehrenfeucht-Fraïsse games and the Parikh Theorem). I will try to fix these issues over time. Hopefully some of them will be resolved by improvements in Mathlib that I expect to see soon.

Future work. This book is a work in progress. My plans for the immediate future include the following, mainly concerning the polyregular functions:

- regular list functions and their polyregular extension;
- MSO interpretations as the logical version of polyregular functions;
- the λ-calculus version of polyregular functions;
- growth rates of polyregular functions.

Further down the line, I would like to extend from strings to trees and graphs.

Contents

Introduction

Unlike the study of regular languages, which can be viewed as functions with Boolean outputs

$$L : A^* \to \mathsf{Bool} = \{\text{yes}, \text{no}\},$$

this book is about string-to-string functions

$$f : A^* \to B^*,$$

where A and B are some finite alphabets. The functions should be computed by models that are finite-state in some way; such models will be called transducers. There will be four main transducer models treated in this book, which are ordered in a "transducer ladder" in increasing order of expressive power as follows:

$$\left.\begin{array}{c} \text{Part A. Mealy machines} \\ \cap \\ \text{Part B. Rational functions} \\ \cap \\ \text{Part C. Regular functions} \\ \cap \\ \text{Part D. Polyregular functions} \end{array}\right\} \text{ the transducer ladder}$$

Before delving into the details of these models, we begin by explaining some general properties that we would like our transducer models to have.

Continuity. The transducer ladder does not include Turing machines, or pushdown automata, suitably generalised to compute string-to-string functions. Why?

The reason is that, in our understanding of "finite-state", the regular languages are finite-state, but pushdown automata and Turing machines are not. This is because we want finite-state models to correspond to regular languages. While the distinction between regular languages and non-regular ones is well understood, the same distinction for string-to-string functions is less clear. Although one of the transducer models in the transducer ladder is called the "regular" functions, this might well be only a historical artefact, and cogent arguments can be made to justify the name "regularity" for any of the other models in the transducer ladder, and also for other models that are not in the ladder. Summing up, we do not really know what is the right analogue for regular languages in the case of string-to-string functions.

Nevertheless, there are some models that we can reject out of hand, because they violate a criterion that we call *continuity*. This criterion requires a function to be compatible with regularity in the string-to-Boolean case, as formalised in the following definition.

Definition .0.1 (Continuous string-to-string functions)**.** *A function*

$$f : A^* \to B^*$$

is called continuous *if for every regular language $L \subseteq B^*$ over the output alphabet, the inverse image $f^{-1}(L)$ is also regular.*

For example, if we take a string-to-string function that is computed by a Turing machine, or by a pushdown automaton with some kind of output mechanism, then the function will typically not be continuous. One of the salient properties of continuity is that for functions with two possible outputs "yes" and "no", continuity is the same as regularity in the usual string-to-Boolean sense (this observation is valid when we see the outputs as one-letter words, and also as multi-letter words).

The name continuity is loosely inspired by the topological notion of continuity, in which the inverse image of an open set is open. This is not a perfect analogy, since open sets in topology should be closed under possibly infinite unions, while regular languages are only closed under finite unions. A more in-depth discussion of this analogy can be found in the exercises, where we show that Definition .0.1 is the same as uniform continuity with respect to a certain distance on strings.

As we will see, all functions in the transducer ladder are continuous. Unfortunately, continuity on its own is not enough to guarantee a reasonable transducer model, and it should be treated as a necessary but not sufficient condition for having a good transducer model. In fact, as the following example shows, there are uncountably many continuous functions.

Example 1. Consider an output alphabet B and any sequence of words

$$w_1, w_2, \ldots \in B^*$$

which has the following *Cauchy property*: every regular language $L \subseteq B^*$ gives the same answer on all but finitely many words in the sequence. By a simple extraction process, one can show that every infinite sequence contains a subsequence with the Cauchy property. A more explicit example of a Cauchy sequence is the sequence of words obtained by repeating an input letter a factorial number of times

$$b^{1!}, b^{2!}, b^{3!}, \ldots \qquad \text{where } b \in B.$$

Let us prove that this sequence has the Cauchy property. Consider some regular language, and a corresponding deterministic finite automaton (DFA) that recognises this language. (We assume that the reader is familiar with DFA.) When reading a string that uses only letter b, the run of the automaton looks like a lasso, with some prefix of length k followed by some loop of length ℓ. Therefore, if $n > k$, the acceptance of an input string b^n depends only on the remainder $n - k \mod \ell$. If n is furthermore a sufficiently large factorial – and hence divisible by ℓ – then this remainder is always the same, namely $\ell - k \mod \ell$. Hence, from some point on the automaton will give the same answer to all strings in the sequence, thus proving that the sequence is Cauchy.

Fix a sequence with the Cauchy property, and consider any function

$$f : A^* \to B^*$$

where all outputs are in the sequence, and furthermore each output is used for finitely many inputs. For every regular language $L \subseteq B^*$, the language will give the same result on all but finitely many words in the Cauchy sequence, and therefore the inverse image $f^{-1}(L)$ will be either finite or cofinite. Since languages that are finite or

cofinite are regular, it follows that the inverse image will necessarily be regular, thus proving that the function is continuous. Since a Cauchy sequence can be constructed in uncountably many ways, and the function f can also be constructed in uncountably many ways, there are uncountably many continuous functions of this form. □

Despite the issues raised in the above example, continuity will be a good – if not perfect – criterion for checking if a transducer model is reasonable. There are, however, other criteria.

Composition. Another desirable property of transducer models is closure under composition, i.e. if we have two functions

$$A^* \xrightarrow{f} B^* \xrightarrow{g} C^*$$

that can be defined in some transducer model, then the same should be true for their composition. All transducer models in this book will have this property. Arguably, continuity itself can be seen as a special case of composition, since taking the inverse image is the same as composing with a string-to-Boolean function:

$$A^* \xrightarrow{f} B^* \xrightarrow{L} \mathsf{Bool}.$$

One of the advantages of composition is that functions can also be decomposed, i.e. presented as compositions of simpler functions. This will be a fundamental theme in this book, starting with the Krohn-Rhodes decomposition of Mealy machines, and continuing with similar decomposition theorems for models that are higher up in the transducer ladder.

Decidable algorithmic problems. The above conditions are necessary, but not sufficient, for a good transducer model. For example, we could take the class of all continuous functions: this class is easily seen to be closed under composition, but it contains many pathological examples, as we have seen in Example 1. An important further condition is that the model should be thoroughly decidable, similarly to finite automata. There are many algorithmic problems to consider, including:

- **evaluation:** given a transducer and an input string, compute the output string;
- **computing inverse images:** given a transducer and a regular property of output strings, compute the inverse image;
- **composition:** given two transducers, compute a transducer for their composition;
- **equivalence:** given two transducers, decide if they compute the same function.

The second and third problems can be seen as effective versions of continuity and closure under composition. For all transducer models in this book, all the above problems will be decidable, with one exception: we do not know if equivalence is decidable for polyregular functions.

Is this all? The properties that we have discussed above are not exhaustive. For all we know, there could be infinitely many transducer models that satisfy all of these properties. In this book, we try to give complete characterisations of some of the models in the transducer ladder, in a way that is machine independent. However, as we will see, this gets harder and harder as the models become more expressive.

Exercises

Exercise .0.1. Show continuity for the reversal function $a_1 \cdots a_n \mapsto a_n \cdots a_1$.

Solution. Given a deterministic automaton for the output language, we can construct a nondeterministic automaton for the inverse image, by reversing the arrows in the automaton, and swapping the initial and accepting states.

Here is an alternative solution, which uses monoids instead of automata, and which will extend more easily to the subsequent exercises. It is well known that a language $L \subseteq A^*$ is regular if and only if there is some finite monoid M and a monoid homomorphism

$$A^* \xrightarrow{h} M$$

such that the language L is an inverse image $h^{-1}(F)$ for some accepting set $F \subseteq M$. Under this characterisation, reversal is straightforward: we simply consider a monoid with the reverse operation, and the corresponding homomorphism, with the same accepting set.

Exercise .0.2. Show continuity for the duplication[1] function $w \mapsto ww$.

Solution. We use the monoid approach that was discussed in the solution to Exercise .0.1. Consider a regular language $L \subseteq A^*$. To see that its inverse image, under duplication, is regular, we use the same monoid homomorphism as for L, but we change the accepting set to be the set of elements m such that $mm \in F$. This way, the inverse image of the new accepting set is exactly the inverse image of L under duplication.

Exercise .0.3. Show continuity for the squaring[2] function $w \mapsto w^{|w|}$.

Solution. Suppose that the original language L is recognised by a monoid homomorphism

$$A^* \xrightarrow{h} M.$$

[1] We use the name duplication for the function because the output length is double the input length. This clashes with the notation $ww = w^2$ which seems to suggest squaring. The reason for the clash is that string concatenation is traditionally written in multiplicative notation, while additive notation is used for string lengths. For us, the more important one will be string length. This notation clash would be resolved by additve notation for strings, which is done for example in Haskell, where the concatenation operator is called ++. Nevertheless, we stay with the standard multiplicative notation for string concatenation.

[2] Similarly to the duplication function, the notation $w^{|w|}$ suggest exponentiation, but we call it squaring because of the output length.

The inverse image in the statement of the exercise is the language

$$\{\ w \in A^* \mid\ (h(w))^{|w|} \text{ belongs to the accepting set of the original language }\}$$

To recognise this inverse image, we use a deterministic finite automaton where the state space is

$$M \times M^M,$$

subject to the following invariant: after reading an input string w, the first coordinate is the image $h(w)$; and the second coordinate the function $m \mapsto m^{|w|}$. To maintain the invariant, the state is updated as follows:

$$(m, \varphi) \xmapsto{a} (m \cdot h(a), \lambda n \mapsto \varphi(n) \cdot n).$$

The accepting set is

$$\{\ (m, \varphi) \mid\ \varphi(m) \text{ belongs to the original accepting set }\}.$$

Exercise .0.4. Show continuity for the factorial function $w \mapsto w^{|w|!}$.

Solution. We use a similar construction as in the monoid solution to the previous exercise. Let h and g be as in that solution. We define a deterministic finite automaton for the inverse image, where the state space is

$$M \times M^M \times M^M.$$

The first two coordinates are updated as in the previous exercise, while the third coordinate is updated so that it stores the function $m \mapsto m^{|w|!}$. This is implemented by taking the previous value of the third component, and multiplying it coordinatewise with the new value of the second coordinate. The accepting set is defined similarly to the previous exercise, using the third coordinate instead of the second one.

Exercise .0.5. Let $g : \mathbb{N} \to \mathbb{N}$ be any non-decreasing function, possiby non-computable. Show that the function $w \mapsto w^{g(|w|)!}$ is continuous.

Solution. The main observation is the following claim, which says that factorial powers in a monoid must necessarily stabilise.

Claim .0.2. *For every finite monoid M and every $m \in M$, the following sequence of monoid elements is eventually constant:*

$$m^{1!}, m^{2!}, m^{3!}, \ldots$$

Let us write $m^! \in M$ for the stable value of the sequence in the above claim. By the above claim, if we take a monoid homomorphism h into a monoid M and we replace the accepting set F by

$$\{\ m \in M \mid\ m^! \in F\ \},$$

then the new regular language will agree with the inverse image of the original language up to a finite number of exceptions. Since regular languages are closed under finite differences, the inverse image will be regular, and the function is continuous.

Exercise .0.6. Consider a string-to-string function with finitely many output values. Show that this function is continuous if and only if the inverse image of each output value is regular.

Solution. The "only if" direction is immediate from the definition of continuity, since the inverse image of a singleton set is the inverse image of a regular language. For the "if" direction, we use the fact that every regular language can be expressed as a finite union of singleton sets, and that regular languages are closed under finite unions. Therefore, if the inverse image of each output value is regular, then the inverse image of any regular language will also be regular.

Exercise .0.7. Show that the assumption on finitely many output values in Exercise .0.6 is necessary.

Solution. Otherwise, every injective string-to-string function would be continuous, since images of singletons would also be singletons and hence regular. Clearly not all injective functions are continuous, since every function can be made injective by adding a copy of the input string to the output, after some separator symbol.

Exercise .0.8. Consider a function which has an empty output for inputs of even length, and otherwise outputs the (one-letter string consisting of) the letter in the middle of the input string. Show that this function is not continuous when the alphabet has at least two letters.

Solution. For a letter a in the alphabet, the inverse image of the regular language $\{a\}$ is the set of strings of odd length whose middle letter is a. This language is not regular, since it is not recognised by any finite automaton, which can be proved using the pumping lemma.

Exercise .0.9. For words over an alphabet A, define the distance between two strings w, w' to be zero if they are equal, and otherwise to be

$$\frac{1}{\text{minimal number of states in a DFA that accepts } w \text{ but not } w'}.$$

Show that this is indeed a distance.

Solution. The distance is symmetric and zero if and only if the two strings are equal. The only thing to check is the triangle inequality. We will show a stronger inequality, where $\max$ is used instead of $+$ (which is called an ultrametric):

$$d(w_1, w_3) \leq \max(d(w_1, w_2), d(w_2, w_3)).$$

By unfolding the reciprocals in the definition, this is the same as

$$s(w_1, w_3) \geq \min(s(w_1, w_2), s(w_2, w_3)),$$

where s is the minimal number of states in a DFA that distinguishes two words. To see why the inequality holds, we observe that if a DFA distinguishes w_1 from w_3, then the same DFA must also distinguish w_1 from w_2 or w_2 from w_3. Observe that this proof does not use any properties of automata, and we would also get a metric if used any class of distinguishers, such as Turing machines or pushdown automata.

Exercise .0.10. Show that all string-to-string functions are continuous with respect to the distance from Exercise .0.9.

Solution. Continuity means that for every input string $w \in A^*$ and every $\varepsilon > 0$, there is some $\delta > 0$ such that for all input strings at distance at most δ from w, the output is at distance at most ϵ from w. This is true, because there is a DFA whose language is the singleton $\{w\}$, and the distance from w to any other string is at least $\delta = 1/n$, where n is the number of states in the DFA. All input strings at distance $< \delta$ will be mapped to exactly $f(w)$, because w is the only such string.

Here is a topological description of the same argument. As we explained above, for sufficiently small δ, the δ-neighbourhood of w is just the singleton $\{w\}$. This means that all singletons are open sets, and therefore all sets are open. Hence the topology is discrete, and all functions are continuous [3].

Exercise .0.11. Show that the continuous functions, in the sense of Definition .0.1, are exactly the uniformly continuous functions with respect to the above metric.

Solution. Uniform continuity means that for every $\varepsilon > 0$, there is some $\delta > 0$ such that for all input strings w, w' at distance at most δ, the outputs are at distance at most ϵ. This means that for every $n \in \mathbb{N}$ (think of $1/\varepsilon$) there is some $m \in \mathbb{N}$ (think of $1/\delta$) such that if input strings cannot be distinguished by automata with at most m states, then the corresponding output strings cannot be distinguished by automata with at most n states. This is the same as continuity in the sense of Definition .0.1, because there are finitely many automata with a given number of states (and a given alphabet).

[3] It is worth pointing out that the issues go away if we consider a completion of the space, i.e. we extend A^* to include limits of Cauchy sequences. The resulting topological space is called the space of *profinite strings*, and it is compact. For more information, see Jean-Éric Pin, *Mathematical Foundations of Automata Theory*, 2025, Chapter X.

Part A: Mealy machines

In this part, we study the simplest transducer model, namely Mealy machines. Mealy machines are barely different from deterministic finite automata, and in fact the early literature on automata theory did not even distinguish between the two. For example, in the original published reference for the Myhill-Nerode Theorem, the formalism used is a Mealy machine, see [Ner58, Lemma 2]. This trend did not last – in one of the foundational papers on automata theory, Rabin and Scott comment that they want to simplify previous automata models so that only "yes" or "no" outputs are possible, see [RS59, p.114]. (Although Scott seemed to show some remorse, see [Sco67, Sentence 3 in Section 5].)

From this perspective, Mealy machines are a good starting point for the study of transducers, since many of the results that we care about – such as continuity, composition and equivalence – are easy to prove and barely different from analogus results about automata as acceptors. There is one exception though, which is the Krohn-Rhodes decomposition theorem from Section A.2: of the various decomposition theorems that we will see in this book, this is possibly the most difficult one.

A.1 Mealy machines

A Mealy machine is a deterministic finite automaton in which transitions are labelled by output letters. Also, there is no accepting set of states, since the purpose of the machine is to produce an output string instead of accepting or rejecting. Here is a formal definition.

Definition A.1.1 (Mealy machine)**.** *A* Mealy machine *is a tuple consisting of*

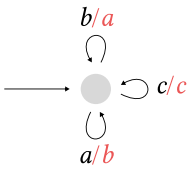
$$\underbrace{A}_{\substack{\text{input}\\ \text{alphabet}}} \quad \underbrace{B}_{\substack{\text{output}\\ \text{alphabet}}} \quad \underbrace{Q}_{\substack{\text{state}\\ \text{space}}} \quad \underbrace{q_0 \in Q}_{\substack{\text{initial}\\ \text{state}}} \quad \underbrace{\delta : Q \times A \to Q \times B}_{\substack{\text{transition}\\ \text{function}}}.$$

The semantics of the machine is a function

$$f : A^* \to B^*,$$

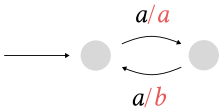
which is defined as follows. We consider the deterministic finite automaton which is obtained from the Mealy machine by ignoring the output letters. We run this automaton on the input string, and then we label each position by the output letter of the corresponding transition. This function is a *letter-to-letter* function, which means that the input and output have the same length.

Example 2. [Letter-to-letter homomorphism] We begin with the most trivial case of a Mealy machine, in which there is one state only. Here is a picture of such a machine, using the convention that transitions are labelled by pairs (input letter / output letter):



This particular machine swaps a with b, and leaves c unchanged. More generally, a one-state Mealy machine is the same as a *letter-to-letter homomorphism*, i.e. we apply some fixed letter-to-letter function independently to each input letter. □

Example 3. [Alternating a and b] In this example, we have a two-state Mealy machine whose input alphabet is $\{a\}$ and whose output alphabet is $\{a, b\}$. Here is a picture of this machine:



This machine outputs letters a and b in alternation, as in the following picture:

a	a	a	a	a	a	a	a	a	a	input
a	b	a	b	a	b	a	b	a	b	output

□

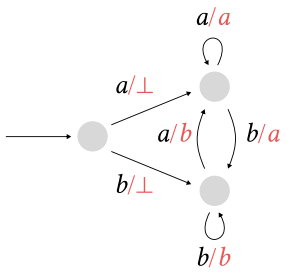
Example 4. [Delay] For an input alphabet A, consider the *delay function*, which shifts the input letters by one step to the right, as in the following picture:

b	c	a	b	c	b	a	a	c	b	input
$\perp$	b	c	a	b	c	b	a	a	c	output

The output alphabet, call it $A+1$, is the same as the input alphabet, except that it has an extra letter for the undefined output in the first position. The Mealy machine has states $A+1$, i.e. the same as the input alphabet. The initial state is the extra letter, which stands for the undefined output in the first position. The transition function is

$$\delta(q, a) = (a, q),$$

which means that the input letter is stored in the state, and the output is the previous state. Here is a picture of this machine, in the case when the input alphabet is $\{a, b\}$:



□

A.1.1 Equivalence, compositions and continuity

In this section, we establish that Mealy machines satisfy the three criteria that were mentioned in the introduction: they are continuous, closed under composition, and have decidable equivalence.

Decidable equivalence We begin with the equivalence problem, where we want to know if two Mealy machines compute the same function.

Theorem A.1.2. *The equivalence problem for Mealy machines is decidable.*

Proof. This is an easy reduction to the equivalence problem for regular languages, which is decidable. For a function

$$f : A^* \to B^*$$

computed by a Mealy machine, and an output letter $b \in B$, consider the language

$$\underbrace{\{ w \in A^* \mid \text{ last output letter of } w \text{ is } b \}}_{\text{call this the "ends with } b\text{" language}}$$

This is easily seen to be a regular language; the corresponding automaton is the same as the one in the Mealy machine, except that it remembers in its state whether the last output letter was b. (If we could use an extended variant of automata, in which the accepting set is a subset of transitions and not states, then we would not even need to modify the automaton in any way.) Since the function f is letter-to-letter, it is completely determined by the family of its "ends with b" languages. Therefore, two Mealy machines f and g are equivalent if and only if the corresponding "ends with b" languages are the same for every output letter b. □

The above proof is trivial because there is a perfect alignment between input and output letters. For more advanced transducer models that will be studied later in the book, there will not be such an alignment, which will make the equivalence problem more difficult. This issue would arise already in the mild extension of Mealy machines, in which transitions produce not letters, but output strings of arbitrary (and possibly empty) length. As we will see in the next chapter, the equivalence problem will remain decidable for this extension, but the proof will be more complicated.

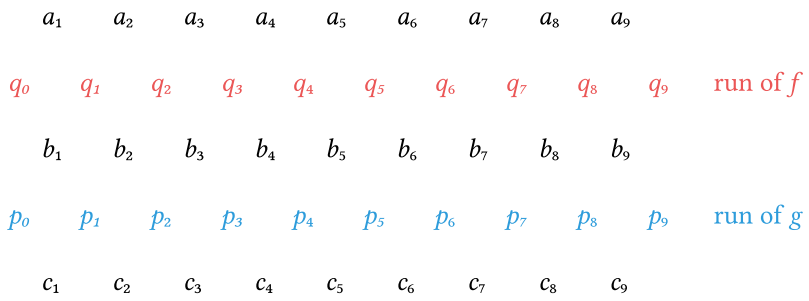
Compositions. We now establish closure under composition for Mealy machines. In this book, we use a dot to represent left-to-right composition of functions, i.e. $f \cdot g$ is the function where we first apply f, and then we apply g to the output of f. To denote the opposite order of composition, we will write $f \circ g$, which is the same as $g \cdot f$. In order to avoid confusion, we try to avoid the $\circ$ notation altogether.

Theorem A.1.3. *Mealy machines are closed under composition, i.e. if*

$$A^* \xrightarrow{f} B^* \xrightarrow{g} C^*$$

are Mealy machines, then so is their composition $f \cdot g$.

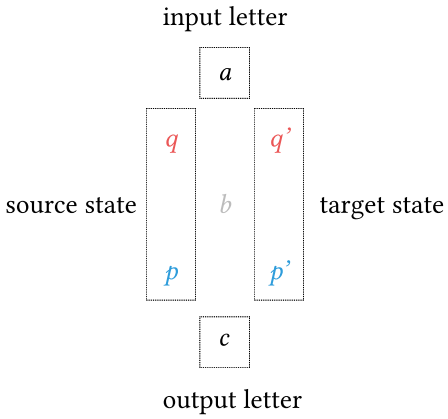
Proof. The proof is a straightforward product construction. Consider the composition $f \cdot g$ applied to some input string, with the runs of the two transducers shown in the following picture:



As the state space of the composition $f \cdot g$ we use the product

$$(\text{state space of } f) \times (\text{state space of } g).$$

The initial state is the pair of initial states. The transition function is easily understood by looking at the following diagram:



If we know the pair of source states (q, p) and the input letter a, then we can compute the intermediate letter b, and then use it to compute the pair of target states (q', p') and the output letter c. Hence, one can define a transition function in the product automaton which gives the desired composition. □

Continuity. We finish this section by establishing continuity for Mealy machines.

Theorem A.1.4. *Mealy machines are continuous.*

Proof. This can easily be proved using a direct product construction similar to Theorem A.1.3, but we prefer an indirect proof, which leverages the similarities of composition and continuity that were explained in the introduction. For a language $L \subseteq B^*$, define its *letter-to-letter lifting* to be the function

$$\text{lift}(L) : B^* \to \{0, 1\}^*,$$

where each position is labelled by 0 or 1, depending on whether the corresponding prefix of the input string belongs to the language. It is easy to see that the letter-to-letter lifting gives a bijective correspondence between regular languages and Mealy machines with output alphabet $\{0, 1\}$. In terms of this correspondence, inverse images become compositions, i.e.

$$\text{lift}(f^{-1}(L)) = f \cdot \text{lift}(L).$$

Therefore, continuity for Mealy machines becomes a corollary of closure under composition from Theorem A.1.3. □

Exercises

Exercise A.1.1. Give an example of a function that is continuous, letter-to-letter (i.e. input length is the same as output length), and is not computed by a Mealy machine.

Solution. The reverse function from Exercise .0.1.

Exercise A.1.2. Show that a composition of n Mealy machines, each one with at most n states, might require a number of states that is exponential in n.

Solution. Consider the first ℓ prime numbers $p_1 < \cdots < p_\ell$ and a composition

$$(2^0)^* \xrightarrow{f_1} (2^1)^* \xrightarrow{f_2} (2^2)^* \xrightarrow{f_3} \cdots \xrightarrow{f_\ell} (2^\ell)^*$$

in which the i-th machine copies the input string and extends each position with one bit that says if the position is divisible by p_i. The i-th machine has p_i states. For the composition, the shortest input string that produces a letter with all bits equal to 1 has length $p_1 \cdots p_\ell$, which is exponential in ℓ and p_ℓ.

Exercise A.1.3. We say that a Mealy machine

$$f : A^* \to A^*$$

is *invertible* if there is some Mealy machine

$$f^{-1} : A^* \to A^*$$

such that the composition $f \cdot f^{-1}$ is the identity function. Show that one can decide if a given Mealy machine is invertible.

Solution. We assume that all states in the Mealy machine are reachable. The criterion is (*): for every state q, the following function must be a bijection:

$$a \in A \mapsto \text{output letter of the transition with source } q \text{ and input letter } a.$$

Clearly if the Mealy machine is invertible, then (*) must hold, since a violation of (*) results in two input strings giving the same output string. Let us now prove the converse implication: if (*) holds then the machine is invertible. To do this, we can reconstruct the input string from the output string in a step by step manner, by keeping track of the current state. The new machine has the same states as the original one. Finally, it is easy to see that (*) can be checked in polynomial time.

Exercise A.1.4. Consider the group of invertible Mealy machines of type $A^* \to A^*$, with the group operation being composition. If we take a finite set of generators, is the generated subgroup necessarily finite?

Solution. No. Even a stronger result is true: if we take a single invertible Mealy machine, the group generated by it (i.e. the group of powers of the machine) is infinite. Consider a Mealy machine over a binary alphabet, which inputs a binary string of length n, thinks of it as a number in $\{0, \ldots, 2^n\}$ encoded in binary, and outputs the binary representation of the result of adding 1 modulo 2^n. This machine is invertible, and the group generated by it is infinite.

Exercise A.1.5. Give an algorithm which inputs a Mealy machine $f : A^* \to B^*$ and decides if the following is bounded by a polynomial in n:

$$n \in \mathbb{N} \quad \mapsto \quad |\{\, f(w) \mid \ w \in A^* \text{ has length at most } n \,\}|.$$

Solution. The set of possible outputs of a Mealy machine is a regular language, which can be computed. (The corresponding automaton guesses the input string.) Therefore, the exercise reduces to checking if a given regular language has polynomial growth. This is a classical result, which can be proved by looking at the strongly connected components of the automaton. The neccessary and sufficient criterion for exponential growth is that: (*) there are two cycles around the same state which have different input strings of the same length. If one can find two such cycles, then the language has exponential growth. Let us prove the converse implication. Let us now prove that if (*) fails, then the growth is polynomial. Consider the strongly connected components of the automaton, and let us look at the directed acyclic graph of these components. If (*) fails, then every path in this directed acyclic graph has at most one cycle, which means that the number of paths of length n is polynomial in n.

Exercise A.1.6. We say that a regular language $L \subseteq A^*$ is *regular-complete under Mealy reductions* if for every regular language $K \subseteq B^*$, there exists a Mealy machine $f : B^* \to A^*$ such that

$$w \in K \quad \text{iff} \quad f(w) \in L \qquad \text{for every nonempty } w \in B^*.$$

(We use nonempty strings because a Mealy machine cannot change the empty string.) Show that there is a language L with this property.

Solution. The language is

$$\{\, w \in \{a, b\}^* \mid \ \text{the last letter of } w \text{ is } a \,\}.$$

The reduction outputs a as the last letter if the input string belongs to K, and b otherwise.

Exercise A.1.7. Give an algorithm, which checks if a given regular language is regular-complete under Mealy reductions. The algorithm can even be made polynomial, assuming that the input regular language is given by a deterministic automaton.

Solution. Consider the regular language L from the solution to Exercise A.1.6. A language K is regular-complete under Mealy reductions if and only if there is a reduction, as in the definition of completeness, from L to K. This in turn is easily seen to be equivalent to the following criterion: in the minimal automaton for K (in fact, in any deterministic automaton for K), there is a subset of states P with the following property:

(*) P contains the initial state and for every state $q \in P$, there are input letters a and b such that $qa, qb \in P$ but only one of them is accepting.

It remains to show that the above criterion can be decided in polynomial time. To do this, we use a game, played between two players, who are called Prover and Refuter. The game plays as follows:

1. Start in the initial state of the Mealy machine.
2. Player Prover chooses two letters a and b, such that exactly one of the states qa and qb is accepting. If there are no such letters, then Refuter wins.
3. Player Refuter chooses a letter $\sigma \in \{a, b\}$ and the state is updated to $q\sigma$. The game continues from step 2.

The game is a special case of a safety game, and hence the winner can be decided in polynomial time using a fixpoint procedure. The set of states q from which Prover has a winning strategy is exactly the set P that we are looking for.

A.2 The Krohn-Rhodes Decomposition Theorem

As we have shown in Theorem A.1.3, Mealy machines can be composed. This motivates the idea of *decomposing* Mealy machines into simpler ones. For example, if we compose the delay function from Example 4 with itself, then we get the *double delay* function, which shifts the input letters by two steps to the right, as in the following picture:

$$\begin{array}{cccccccccc} b & c & a & b & c & b & a & a & c & b \\ \bot & \bot & b & c & a & b & c & b & a & a \end{array} \quad \begin{array}{l} \text{input} \\ \text{output} \end{array}$$

Therefore, the double delay function can be decomposed into two copies of the delay function. The main result of this section is the Krohn-Rhodes Decomposition Theorem, which shows that every Mealy machine can be decomposed into a composition of certain special Mealy machines, which we call *prime Mealy machines*. Later on in this book, we will prove similar decomposition theorems for more advanced transducer models. Interestingly enough, the decomposition theorem for Mealy machines is arguably the most difficult one to prove.

Primes. We begin by defining the prime Mealy machines. These prime machines are defined in terms of the state transformations that they can perform. Here, the *state transformation* of an input letter is the function $Q \to Q$ which shows how this letter updates the state (the output of the machine is not relevant for this definition).

Definition A.2.1. *There are two kinds of* prime Mealy machines*:*

- *Reversible: all state transformations of all letters are permutations.*
- *Flip-flop: for each letter, its state transformation is either: (a) the identity transformation; or (b) a constant (i.e. all states are mapped to the same one).*

The definition of state transformations can be extended from individual letters to input strings, by composing the state transformations of the individual letters. We could modify the above definition to talk about state transformations of input strings, instead of input letters, and the result would be the same. This is because composing constants gives a constant, and composing permutations gives a permutation.

Example 5. If a Mealy machine has only one state, then it is both reversible and flip-flop, since all state transformations are the identity transformation. The machine with output $ababa\cdots$ from Example 3 is a reversible machine, since the state transformation of each letter is a permutation. The delay machine from Example 4 is a flip-flop machine, since the state transformation of each letter is a constant. □

Theorem A.2.2 (Krohn-Rhodes Theorem)**.** *Every Mealy machine f admits a decomposition*

$$f = f_1 \cdot f_2 \cdots f_n$$

where each Mealy machine $f_1, \ldots, f_n$ is either reversible or flip-flop.

In some decomposition results in mathematics, such as the decomposition of a number into its prime factors, the decomposition is unique. We make no such uniqueness claims in the above theorem. Since Mealy machines are closed under composition, it follows from the above theorem that the class of functions computed by Mealy machines is the same as the class of compositions of functions computed by prime Mealy machines. This can be written as:

$$\text{Mealy} = (\text{Reversible} \cup \text{Flip-flop})^*,$$

where the Kleene star on the right-hand side stands for closure under composition. The rest of Section A.2 is devoted to the proof of this theorem. There are two steps in the proof. First, in Section A.2.1, we show an auxiliary result about map lifting, which shows that a decomposition into primes can be applied in parallel to a list of input strings. Then, in Section A.2.2, we finish the proof.

A.2.1 Map lifting

In the first step of the proof, we introduce a construction on transducers that we call the map lifting, and which will be used many times in this lecture. The idea is to apply a transducer to a list of input strings, which are separated using some separator.

Definition A.2.3 (Map lifting)**.** *Let*

$$f : A^* \to B^*$$

be a string-to-string function. The map lifting of f, denoted by

$$\underbrace{(A+1)^*}_{\substack{\text{alphabet obtained from } A \\ \text{by adjoining a fresh letter } \#}} \xrightarrow{\mathsf{map} f} (B+1)^*,$$

is the function defined by

$$\underbrace{w_1 \# w_2 \# \cdots \# w_n}_{\substack{\text{the strings } w_1, \ldots, w_n \\ \text{do not use the separator } \#}} \quad \mapsto \quad f(w_1) \# f(w_2) \# \cdots \# f(w_n).$$

The following lemma shows that map lifting is compatible with decompositions into primes.

Lemma A.2.4. *If a Mealy machine decomposes into prime Mealy machines, then the same is true for its map lifting.*

Proof. Map lifting commutes with composition of functions, i.e.

$$\mathsf{map}(f \cdot g) = (\mathsf{map} f) \cdot (\mathsf{map} g).$$

Therefore, it is enough to show the lemma in the case when f is a prime Mealy machine. If f is a flip-flop, there is nothing to do: we reset its state to the initial state

whenever the separator symbol is read. The interesting case is when f is reversible. The difficulty is that resetting the state would break reversibility.

Consider an input string $w \in (A+1)^*$ that may contain the separator symbol. We consider two state transformations:

δ_w this is the state transformation in the automaton which corresponds to the map lifting, i.e. the separator symbol resets the state to the initial one;

γ_w this is the state transformation in a variant of the above automaton, in which the separator symbol does nothing, i.e. it does not change the state.

The first state transformation is the one that we care about, but the underlying automaton is not reversible, since it involves resetting the state. The second state transformation seems less useful, but it arises from a reversible automaton. The two state transformations coincide for strings that do not use the separator symbol, and hence we have the following cancellation law:

$$\delta_w = (\underbrace{\gamma_v}_{\substack{\text{a permutation, and}\\ \text{hence can be inverted}}})^{-1} \cdot \gamma_{vw}, \qquad \text{where } w \text{ does not contain the separator symbol.} \tag{1}$$

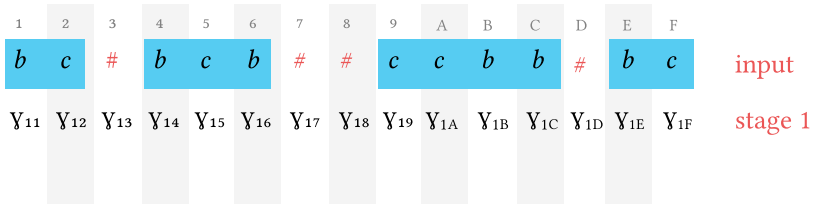
We will apply the cancellation law for strings of the form vw where v is the prefix of vw that ends with the last occurrence of the separator symbol, and w is the suffix that follows it. (In this case, we call v the $\#$-prefix of vw.) Using the cancellation law, we will decompose the map lifting of f into prime Mealy machines. The idea is that for each prefix of the input string, we will compute the two state transformations that appear on the right-hand side of the cancellation law. This is done in two stages.

1. In the first stage, we label each prefix w of the input string with the state transformation γ_w, i.e. the one in which the separator symbol is ignored. This is done using a Mealy machine of type

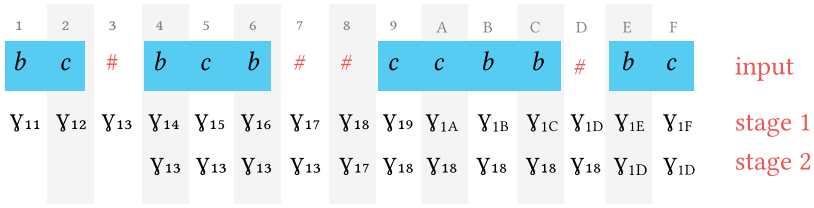
   $$(A+1)^* \to (\text{permutations of } Q)^*,$$

   where Q is the state space of the original Mealy machine. After reading an input word w, this machine stores γ_w in its state, and its last output letter is also γ_w. This machine is reversible, since the original one was reversible and the only new state transformation is the identity transformation.

   Here is a picture of the first stage. In the picture, we use hexadecimal for indexing positions (so that we can have more single digit indexes) and we write δ_{ij} for the state transformation which corresponds to the infix of the input string that begins in the i-th position and ends in the j-th position, including both endpoints:



2. We now proceed to the second stage of the construction, in which we compute the state transformation for the #-prefixes of the input string, as in the following picture:



   This is done using a flip-flop machine which is a variant of the delay machine from Example 4. The states of this machine are permutations of Q, plus an extra initial state for an undefined value. When we read a position with a separator symbol, we store the output of the first stage in the state. For other input positions, the state is unchanged. The output is always the incoming state.

3. By applying the cancellation law to the outputs of the first and second stages, we can get for each input position i the state transformation δ_{1i}. Using reversibility and the i-th input letter, we can also get the state transformation $\delta_{1(i-1)}$. (Unless the i-th letter is the separator, in which case we simply copy the separator to the output.) From this state transformation, we can get the state just before reading the i-th position for the map lifting, and from this we can get the output of the map lifting. All of these operations use only the current position and its labels (in the input string and the first two stages), and hence they can be implemented using a one-state Mealy machine.

Since each of the three stages above is a prime Mealy machine, we get the conclusion of the lemma. Although this is not important for the statement of the lemma, we observe that the third stage can be merged with the second stage, since prime Mealy machines are closed under post-composition with letter-to-letter homomorphisms. Therefore, the map lifting of f is a composition of two prime Mealy machines, a reversible one followed by a flip-flop. □

A.2.2 State transformations

In the second step of the proof, we will use the map lifting from the previous section to conclude the proof of the Krohn-Rhodes Theorem. As was the case for the map lifting,

we will be mainly interested in the state transformations of the Mealy machine. The relevant information will thus be:

$$\underbrace{A}_{\substack{\text{input}\\\text{alphabet}}} \quad \underbrace{Q}_{\substack{\text{state}\\\text{space}}} \quad \underbrace{\{\delta_a : Q \to Q\}_{a \in A}}_{\text{state transformations}}. \tag{2}$$

We use the name *pre-automaton* for the above; it is the same as a DFA without designated initial and accepting states. For a pre-automaton, define its *state transformation transducer* to be the Mealy machine of type

$$A^* \to (Q \to Q)^*,$$

in which the n-th letter of the output string is the state transformation for the first n input letters. The main lemma in the proof of the Krohn-Rhodes Theorem is the following.

Lemma A.2.5. *For every pre-automaton, its state transformation transducer is a composition of prime Mealy machines.*

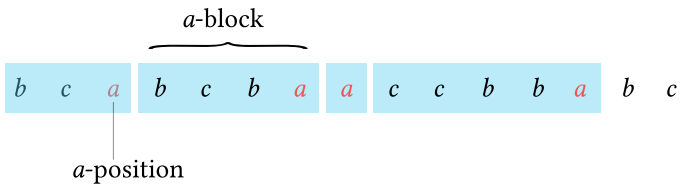
Before proving the lemma, we use it to conclude the proof of the Krohn-Rhodes Theorem. Take a Mealy machine, and consider its underlying pre-automaton. By applying the lemma, we get a composition of prime Mealy machines, which computes for each input position the state of the underlying pre-automaton after reading this position. To get the output of the Mealy machine, we can use a delay machine as in Example 4 to get the state from the previous position, and then we can get the output letter using a letter-to-letter homomorphism that is based on the transition function. It remains to prove the lemma.

Proof of Lemma A.2.5. We prove the lemma by induction on two parameters of the pre-automaton $\mathcal{A}$: (a) the size of the state space Q; and (b) the size of the input alphabet A. These parameters are ordered lexicographically, so that we first compare the size of the state space, and if there is a tie, then we compare the size of the input alphabet. In particular, we can make progress in the induction by decreasing the number of states but increasing the size of the input alphabet.

Induction basis. The induction basis is when there is one state and one input letter. We can easily cover a generalisation of the induction basis, namely when the pre-automaton is reversible, i.e. all state transformations are permutations (which trivially holds when there is one state only). In this case, there is nothing to do, since the natural Mealy machine for state transformations is already reversible.

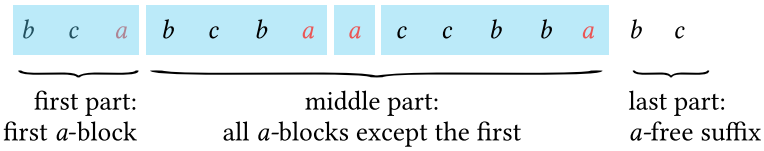
Induction step. We are left with the induction step, under the assumption that the pre-automaton is not reversible. This means that there is some input letter $a \in A$ with a state transformation that is not a permutation. This means that the image of this state transformation is some proper subset $P \subset Q$ of the state space. Fix this letter a for the rest of the proof.

In the proof, we will be interested in a-positions, i.e. positions in the input string that are labelled with the letter a, and a-blocks, i.e. pieces of the input string that go from one a-position to the next one, including the latter a-position but not the former one. Here is a picture:



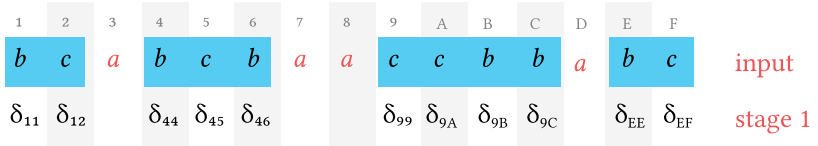
The a-blocks are disjoint and partition the input string, except for some suffix which does not use the letter a. The general idea in the induction step is that an a-block is subject to the assumption on a smaller input alphabet, since a appears only once, while the a-blocks can be seen as individual letters for a pre-automaton with a smaller set of states. This will be made precise below.

To compute the state transformation, we will be interested in a decomposition of the input word into three parts, as depicted in the following picture:



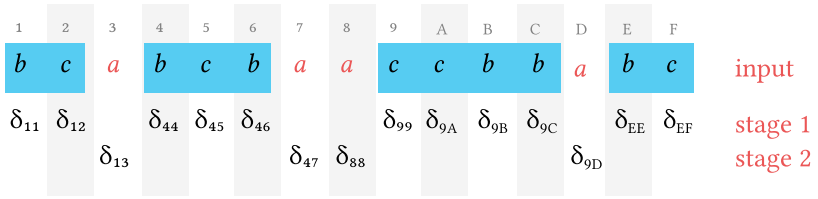
We refer to this decomposition as the *tripartite decomposition*. We now describe a sequential composition of prime Mealy machines, in which all state transformations in the tripartite decomposition are computed. This decomposition proceeds in five stages. We assume that in each stage, we have access to the outputs of the previous stages, since a prime Mealy machine can always be adapted so that it includes a copy of the input string in its output.

1. In the first stage, we compute the state transformation for the a-free suffixes, i.e. the last of the three parts in the tripartite decomposition. Here is a picture of this stage, using the same conventions as in the proof of Lemma A.2.4:



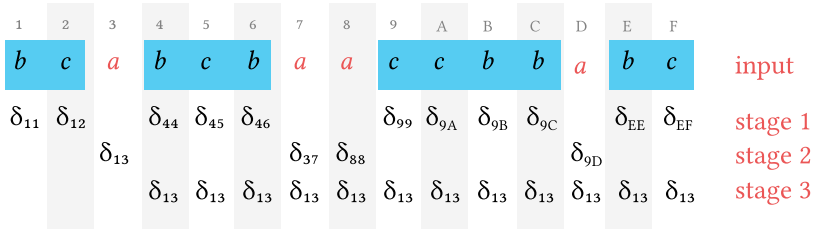
   This stage is implemented by removing the letter a from the input alphabet in the pre-automaton, applying the induction assumption for a smaller input alphabet, and then applying map lifting to the result by Lemma A.2.4.

2. In the second stage, we compute the state transformations of the individual a-blocks, and we store them in the a-positions that end them, as in the following picture:



To implement this, we first use a delay machine as in Example 4, which is a flip-flop machine, to shift the outputs of the first stage by one letter to the right. Next, for positions that are not a-positions, we use a blank letter as the output, and for a-positions we take the value from the preceding position and post-compose it with the state transformation of a. This stage is a flip-flop machine followed by a letter-to-letter homomorphism, and hence it is a flip-flop machine.

3. In the third stage, we compute the first part of its tripartite decomposition, i.e. the first a-block. To do this, we use a flip-flop machine to copy the state transformation in the first a-block to the positions that are not in the first a-block, as in the following picture:



This way, each position outside the first a-block has access to the state transformation in the first a-block.

4. After the first three stages, we have access to the state transformations of the first and last parts of the tripartite decomposition. We are left with the middle part, which is the union of all a-blocks except for the first one. To compute this, we will use the induction assumption on the smaller set of states $P \subset Q$ which is the image of the state transformation of a. Consider a pre-automaton with states P as follows. The input alphabet is the alphabet used to store the original input word together with the results of the preceding stages, while the transition function is defined as follows:

   - If the position is an a-position that is not the first one (which can be detected by looking at the input letter and the third stage), then apply the state transformation stored in the second stage.

- Otherwise, do not change the state.

We can apply the induction assumption to this pre-automaton, since it has a smaller set of states. The state transformation of this new automaton is the state transformation of the original automaton for the middle part of the tripartite decomposition.

5. In the stages so far, we have computed the state transformations for all three parts in the tripartite decomposition. These can be combined, using a letter-to-letter homomorphism, to get the state transformation for the entire prefix, as desired in the lemma.

□

A.2.3 Aperiodic Mealy machines

In the Krohn-Rhodes Theorem, we have two kinds of prime Mealy machines, reversible and flip-flop. What about functions that decompose into only one kind of primes? In other words, what are the classes

$$\text{(Reversible)}^* \quad \text{and} \quad \text{(Flip-flop)}^*?$$

For the first class, there is not much to say, since reversible Mealy machines are closed under composition, and hence the first class is just the class of reversible Mealy machines.

Lemma A.2.6. *Reversible Mealy machines are closed under composition.*

A much more interesting class is the second one, namely compositions of flip-flop machines. To capture this class, we will use the following notion of aperiodicity, which says that the machine cannot have periodic behaviour. Aperiodic machines will be studied in more detail in Section C.4.4, in light of the connection between aperiodicity and logic.

Definition A.2.7 (Aperiodic)**.** *A length preserving string-to-string function is called* aperiodic *if for all input strings u, v, w, with uvw nonempty, there is some output letter b such that*

$$b = \textit{last letter of } f(uv^n w) \qquad \textit{for all sufficiently large } n.$$

In the definition above, we only talk about the last letter, but aperiodicity also implies that the last k letters of $f(uv^n w)$ are eventually fixed for every k, by the arbitrary choice of w.

Example 6. [Aperiodic and periodic functions] Consider the function

$$f : \{a\}^* \to \{0,1\}^*$$

where the last letter says if the string strictly before it is nonempty:

$$aaaaaaaaa \mapsto 011111111.$$

This function is clearly aperiodic, since the last letter of any sufficiently long input string will always be 1. On the other hand, if we change the function so that it indicates the parity of the length, as in the following example:

$$aaaaaaaaa \mapsto 010101010,$$

then the aperiodicity condition will be violated by repeating the letter a. In this example, we did not need the words u and w in $uv^n w$. To see that they are necessary, one can consider the function defined by

$$\text{last letter of } f(w) = \begin{cases} \text{parity of } n & \text{if } w = \#a^n\# \text{ for some } n \\ \bot & \text{otherwise.} \end{cases}$$

□

Theorem A.2.8. *Consider a string-to-string function f that is computed by a Mealy machine. Then f is aperiodic if and only if it is computed by a composition of flip-flop Mealy machines. Moreover, this property can be decided, given a Mealy machine that computes f.*

Proof. We will begin with the implication

$$\text{composition of flip-flop} \implies \text{aperiodic.}$$

Then, before proving the converse implication, we will prove the decidability part of the theorem, as it will introduce notions that will be useful in the converse implication.

From composition of flip-flop to aperiodicity. To see that a composition of flip-flops is aperiodic, we will show that flip-flops are aperiodic, and that aperiodicity is preserved under composition. The first part is easy, since the last letter of $f(uv^n w)$ depends only on: (1) the last letter of w (or v if w is empty); and (2) the last letter in uvw that has a constant state transformation. Neither of these values depends on n. To see that aperiodicity is preserved under composition, it will be more convenient to use the following equivalent characterisation of aperiodicity.

Claim A.2.9. *A function f computed by a Mealy machine is aperiodic if and only if for all input strings u, v, w there are output strings x, y, z and a number k such that*

$$f(uv^{n+k}w) = xy^n z \quad \textit{for all } n > 0.$$

Proof. The right-to-left direction is easy, since the last letter of $xy^n z$ does not depend on n. For the left-to-right direction, we use the remark after the definition of aperiodicity, which says that the last k letters of the output are eventually fixed. If we apply this to outputs of the form $f(uv^n)$ and the last $|v|$ letters, we see that from some point on, the output ends with a repetition of some fixed string y of length $|v|$. If we apply this to outputs of the form $f(uv^n w)$ and the last $|w|$ letters, we get the part z. □

The condition in the above claim is easily seen to be compatible with composition, since the number k for the composition $f \cdot g$ can be taken to be the sum of the corresponding numbers for f and g. This completes the proof of the implication from composition of flip-flop to aperiodicity.

Decidability. As announced at the beginning of the proof of the theorem, before proving the converse implication, we prove the decidability part. To check if function f is aperiodic, we will check if the corresponding Mealy machine satisfies a certain aperiodicity condition on the state transformations, which will be described in Lemma A.2.11. However, in order for the latter check to be meaningful, we will need to use a machine that avoids unnecessary states. Since Mealy machines are very close to regular languages, we can use a minimisation procedure that is similar to the classical Myhill-Nerode minimisation theorem. Define a *derivative* of a length preserving string-to-string function f to be any function of the form

$$f(w\_) \quad \stackrel{\text{def}}{=} \quad v \mapsto f(wv) \text{ with the first } |w| \text{ letters of the output removed.}$$

where w is some input string.

Lemma A.2.10 (Myhill-Nerode for Mealy machines)**.** *A string-to-string function is computed by a Mealy machine if and only if: (1) it has finitely many derivatives; (2) it is letter-to-letter; (3) the n-th output letter depends only on the first n input letters.*

Proof. The left-to-right direction is easy: the derivative $f(w\_)$ is uniquely determined by the state of the Mealy machine after reading w, and hence can be chosen in only finitely many ways. Let us now prove the right-to-left direction. The corresponding Mealy machine has states for the derivatives, with the initial state being the derivative corresponding to the empty string. The transition relation is defined by

$$f(w\_) \xrightarrow{a/\text{last letter of } f(wa)} f(wa\_).$$

This transition function is well-defined, since the output letter and target states depend only on the source state $f(w\_)$ and not the particular choice of w. Using a simple induction on the length of w, one can show that after reading an input string w, this machine is in state $f(w\_)$ and the output that it has produced so far is $f(w)$. □

The machine constructed in the above proof is called the *minimal machine* for the function f. The minimal machine can be easily constructed from any other Mealy machine, by computing which states define the same derivative, and merging them. To complete the proof of the theorem, it remains to show that aperiodicity can be decided based on the minimal machine.

Lemma A.2.11. *A function computed by a Mealy machine is aperiodic if and only if its minimal Mealy machine satisfies the following condition: (*) for every state transformation $\delta : Q \to Q$ that arises from some input string, the sequence of state transformations $\delta^1, \delta^2, \ldots$ eventually stabilises on a single state transformation.*

Proof. The right-to-left direction is straightforward: if a Mealy machine (minimal or not) satisfies (*), then the function that it computes is aperiodic. Let us now prove the left-to-right direction. By Lemma A.2.10, we can assume that the Mealy machine is minimal, and toward a contradiction suppose that this minimal machine does not satisfy (*). Then the input string v that gives rise to the state transformation δ can be used to produce a violation of aperiodicity. Indeed, suppose that the sequence $\delta^1, \delta^2, \ldots$

does not stabilise, and thus it sees two state transformations $\delta^i \neq \delta^j$ infinitely often. By definition of minimiality, there must be some strings u and v such that the last letters of $f(uv^i w)$ and $f(uv^j w)$ are different, for some w. This contradicts aperiodicity. □

The above lemma gives a decision procedure for checking if a function computed by a Mealy machine is aperiodic. We first compute all state transformations that arise from input strings, and for each one we check if its powers induce a non-trivial cycle. To compute the set of all state transformations, we start with the state transformations of individual letters, and we keep on adding state transformations by composing them, until we can add no new state transformation. This procedure terminates, since there are only finitely many possible state transformations. This completes the decidability part of the theorem.

From aperiodicity to composition of flip-flops. We are left with the implication

$$\text{aperiodic} \implies \text{composition of flip-flop.}$$

Consider a function f that is computed by a Mealy machine and is aperiodic. By Lemma A.2.11, we can assume that the Mealy machine satisfies condition (*) from the lemma. We now go through the proof of the Krohn-Rhodes Theorem for this particular machine, and we observe that in the induction process, all machines will continue to satisfy condition (*). This is because in the induction step, every new machine will only use state transformations that arise from the original machine, and these state transformations satisfy condition (*). In particular, a non-flip-flop Mealy machine with more than one state will never arise, since such a machine would violate condition (*). Therefore, all machines in the decomposition will be flip-flops, as desired.

□

Exercises

Exercise A.2.1. Consider the function which replaces all letters in the input string by the first input letter, as in

$$a_1 \cdots a_n \mapsto a_1^n.$$

Decompose this function into flip-flops.

Solution. The natural machine for this function is not a flip-flop. Its states are $A+1$, and they store the first input letter, with the extra state used before the first letter has been read. The state transformation of a letter a maps the extra state to a and leaves all other states unchanged, which is neither the identity nor a constant. The remedy is to first find out which position is the first one, using a separate machine.

In the first machine, the states are $\{0, 1\}$, the initial state is 0, and every letter has the constant state transformation with value 1. The output is the state before the transition, and therefore the first position is labelled with 0, while all remaining positions are labelled with 1. This machine is a flip-flop and, as usual, we assume that it also copies the input letter to its output.

In the second machine, the states are $A + 1$ as in the natural machine described above, but this time the state transformation of a letter depends on the label that was added by the first machine: a letter labelled 0, i.e. a first letter a, has the constant state transformation with value a, while a letter labelled 1 has the identity state transformation. This machine is a flip-flop as well, and if its output is the state after the transition, then this output is the first letter of the input string in every position, as required.

Exercise A.2.2. Show that the function from the previous example cannot be decomposed into reversible machines only.

Solution. We use the same argument as for the delay function. We assume that the input alphabet has at least two letters, since otherwise the function is the identity, which is reversible. By Lemma A.2.6, a composition of reversible machines is again a reversible machine, and therefore it is enough to rule out a single reversible machine. Suppose that some reversible machine computes the function, and let a and b be two different input letters. The state transformation of a is a permutation of a finite set, and hence it has finite order, i.e. there is some $k \geq 1$ such that reading a^k leads from the initial state back to the initial state. Consequently, the machine is in the initial state just before reading the last letter of both input strings

$$b \qquad \text{and} \qquad a^k b,$$

and therefore it produces the same last output letter for both of them. This is a contradiction, since the function outputs b for the first input string and a for the second one.

Exercise A.2.3. Show that every flip-flop machine can be obtained as a sequential composition of two-state flip-flop machines.

Solution. In a flip-flop machine, the state after reading an input string is determined by the last letter which has a constant state transformation: the state is the value of that constant, or the initial state if there is no such letter. In particular, the state can be tracked one bit at a time.

Fix an injective encoding of the states as bit vectors

$$\text{enc} : Q \to \{0, 1\}^k \qquad \text{where } k \text{ is logarithmic in the number of states,}$$

and consider a composition of k machines, where the j-th machine copies its input to the output, extending each position with one extra bit, namely the j-th bit of the encoding of the state that the original machine had *before* reading this position. The j-th machine stores this bit in its state, and hence it has two states. It is a flip-flop

machine, since a letter whose state transformation is the identity leaves the bit unchanged, while a letter whose state transformation is a constant sets the bit to the corresponding bit of that constant.

After all k machines have been applied, each position is labelled by its original letter together with the state of the original machine before this position. This pair determines the output letter of the original machine, and therefore the construction is finished by a letter-to-letter homomorphism, i.e. a Mealy machine with one state. (Such a machine can also be seen as a two-state flip-flop machine in which every letter has the identity state transformation, so that the second state is never visited.)

Exercise A.2.4. Show that the delay function from Example 4 is not a composition of reversible Mealy machines.

Solution. By Lemma A.2.6, a composition of reversible Mealy machines is again a reversible Mealy machine, and therefore it is enough to show that the delay function is not computed by a single reversible machine. Suppose that it is, and let a be some input letter. The state transformation of a is a permutation of a finite set, and hence it has finite order, i.e. there is some $k \geq 1$ such that reading a^k leads from the initial state back to the initial state. Consequently, the machine is in the initial state just before reading the last letter of both input strings

$$a \qquad \text{and} \qquad a^k a,$$

and therefore it produces the same last output letter for both of them. This is not what the delay function does: its last output letter is the undefined letter for the first input string, and the letter a for the second one.

Exercise A.2.5. Show that the function from Example 3 is not a composition of flip-flop Mealy machines.

Solution. By Theorem A.2.8, the compositions of flip-flop machines are exactly the aperiodic functions, and hence it is enough to observe that this function is not aperiodic. Take u and w to be empty and $v = a$ in Definition A.2.7. The last letter of the output for the input string a^n is a or b, depending on the parity of n, and hence it does not stabilise for large n. This is the same argument as the one for the parity function in Example 6.

Exercise A.2.6. Is every invertible Mealy machine, as defined in Exercise A.1.3, necessarily reversible? The other way round?

Solution. Neither implication holds. The reason is that the two notions constrain different parts of the transition function: invertibility is about the output letters, while reversibility is about the target states.

For a machine that is invertible but not reversible, consider the machine over the alphabet $\{a, b\}$ which stores the previous input letter in its state, and whose output is: the current input letter, if the previous letter was a or there was no previous letter; and the current input letter with a and b swapped otherwise. Every state transformation is a constant, and hence this is a flip-flop machine, which is not reversible,

since a constant transformation on a state space with more than one state is not a permutation. It is invertible, because in each state the two outgoing transitions have different output letters, which is the criterion from the solution to Exercise A.1.3.

For a machine that is reversible but not invertible, consider the machine with one state which sends every input letter to the same output letter. Its only state transformation is the identity, and hence it is reversible, but the function that it computes is not injective, and therefore it has no inverse.

Exercise A.2.7. Show that if $f : A^* \to B^*$ is continuous, then the same is true for its map lifting.

Solution. Let $\mathcal{B}$ be an automaton which recognises a regular language over the output alphabet of the map lifting; we want to show that the inverse image of this language is regular. Consider an input string

$$w_1 \# \cdots \# w_n.$$

Its image under the map lifting is accepted by $\mathcal{B}$ if and only if there are states p_1, q_1, $\ldots$, p_n, q_n of $\mathcal{B}$ such that:

1. p_1 is initial and q_n is accepting;
2. for every i, the automaton can go from p_i to q_i while reading $f(w_i)$;
3. for every $i < n$, the automaton can go from q_i to p_{i+1} while reading the separator.

An automaton over the input alphabet can guess these states and check the three conditions. The first and third conditions speak about individual positions, and hence they are easily checked. The second condition is an inverse image, under f, of the language recognised by $\mathcal{B}$ with initial state p_i and accepting state q_i; this inverse image is regular thanks to the continuity assumption on f. The same argument is used later in the book, see Lemma C.1.3.

A.3 References

Mealy machines were introduced in [Mea55, Section 2.1]. A related model is Moore machines [Moo56, p.133], in which the n-th output letter depends on the first $n-1$ input letters.

The Krohn-Rhodes Theorem was originally proved in [KR65, p. 454]. Let us remark on the differences between the original statement and the present text. In the original, semigroup terminology is used, while here we use the language of Mealy machines. The original also has an extra kind of product – parallel composition – instead of using only sequential composition as in the present text. Finally, the original statement says that the groups in the decomposition must have been present in the original machine, which is also true here but not stated explicitly (however, we see parts of this in the results on counter-free machine, which talk about machines that have no groups at all). The proof in this book is inspired by a proof about LTL from [Wil99, p.35].

Part B: Rational functions

In this part, we move one step up the transducer ladder, to the rational functions. A Mealy machine produces its output deterministically in a left-to-right manner, and each input letter generates exactly one output letter. The rational functions lift both of these restrictions. There are intermediate models, which lift one restriction but not the other; however, we do not discuss them here in detail, in the interest of not proliferating transducer models.

Before defining the rational functions, in Section B.1 we take a detour, and define a more general class, which is called the rational relations. These are binary relations and not functions, because the underlying model is non-deterministic, and the same input string might generate multiple outputs, even infinitely many. Many results about rational functions are easily proved in the more general setup of relations, and so it is instructive to begin with the relational case. Nevertheless, there is a price to pay: the equivalence problem for rational relations is undecidable.

Next, in Section B.2, we define the rational functions as those rational relations which are semantically functional, i.e. each input produces exactly one output. Although the definition uses a nondeterministic model, we also identify an equivalent model that is deterministic, which is called bimachines, although the determinism is not of the left-to-right kind that we are used to from Mealy machines. Finally, we show how restricting to functions allows us to recover decidability of equivalence.

B.1 Rational relations

We begin with the rational relations, which are a nondeterministic version of the rational functions, in which the underlying model is an NFA equipped with an output mechanism. One of the advantages of this model is that it does not treat the input and output differently.

Definition B.1.1 (NFA with output)**.** *A* nondeterministic automaton with output *consists of the following ingredients:*

- *input and output alphabets A and B;*
- *a finite set of states Q;*
- *distinguished initial and final subsets $I, F \subseteq Q$;*
- *a transition relation, which is a finite set*

$$\delta \subseteq Q \times A^* \times B^* \times Q.$$

In other words, such an automaton is a directed graph, where the vertices are states, and each edge is labelled by a pair of input/output words. A run of this automaton is a path that begins in an initial state and ends in a final state. The input string of the run is obtained by concatenating the input strings on the transitions, and likewise we define the output string. The semantics of the automaton is the relation

$$R \subseteq A^* \times B^*,$$

which consists of pairs of input/output strings, ranging over accepting runs of the automaton. The relations that arise this way are called rational, as stated in the following definition.

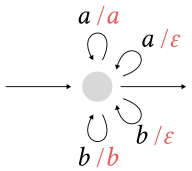
Definition B.1.2 (Rational relation)**.** *A* rational relation *is any relation that can be defined by a nondeterministic automaton with output.*

Every Mealy machine is a special case of a rational relation, since it corresponds to a nondeterministic automaton with output in which: (a) every transition is labelled by a single input letter and a single output letter; (b) the automaton is deterministic; and (c) every state is accepting. Here are some other examples of rational relations, which are not Mealy machines.

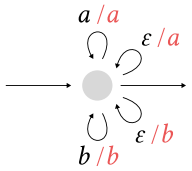
Example 7. [Subword relation] For every alphabet A, the subword relation

$$\{\ (w, v) \in A^* \times A^* \mid\ v \text{ can be obtained from } w \text{ by deleting some letters }\}$$

is a rational relation. Here is a picture of the automaton when $A = \{a, b\}$:

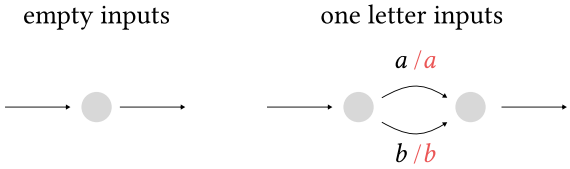


An important property of rational relations is that they are input/output symmetric. Therefore, the inverse of the subword relation, i.e. when letters can be added, is also rational, with the corresponding automaton looking like this:



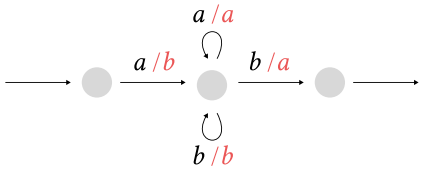
In particular, thanks to transitions that have ε on the input part, a rational relation can have infinitely many outputs for a single input. □

Example 8. [Swap first and last] Here is an example of a rational relation which is a function, i.e. every input string has exactly one output string: the output string is obtained from the input string by swapping the first and last letters (if the input string has at most one letter, it is unchanged). To implement this function, we use a union of six nondeterministic automata. (In the pictures below, we use an alphabet with two letters.) The first two automata deal with inputs that have at most one letter:



The remaining four automata deal with the possible combinations of (first letter, last letter), with one of them being:

inputs that start with f and end with b



This function is not computed by a Mealy machine, even though it is letter-to-letter. The reason is that for a Mealy machine, the first output letter depends only on the first input letter. □

Why the name rational? We finish this section by explaining the naming convention. The idea is that we can try to identify the "regular" languages in any monoid, and not just the free monoid A^*. To formalise regularity in an abstract monoid, two

ideas come to mind, which are based on (two-sided) Myhill-Nerode equivalence and regular expressions, respectively, as explained in the following definition.

Definition B.1.3. *[Rational and recognisable subsets of a monoid]*

- ***The recognisable subsets.*** *A subset L of a monoid M is called* recognisable *if it is a union of finitely many equivalence classes for some equivalence relation $\sim$ on M that has finite index and is a congruence, i.e.*

$$m_1 \sim m_1' \wedge m_2 \sim m_2' \quad \Rightarrow \quad m_1 \cdot m_2 \sim m_1' \cdot m_2'.$$

- ***The rational subsets.*** *A subset L of a monoid M is called* rational *if it can be obtained from finite subsets of M by applying a finite number of times the following operations: union, product, and Kleene star*

$$L^* \quad \stackrel{def}{=} \quad \bigcup_{n \in \{0,1,\ldots\}} L^n.$$

Often the second class is called the *regular expressions*, but the French algebraic tradition uses the name rational, which is inspired by similar terminology for power series. In the free monoid A^*, the two notions coincide, are exactly the rational subsets from the above definition, which can be proved by converting an NFA with output into a regular expression. The recognisable subsets of $A^* \times B^*$ are much weaker; these are relations that can be recognised by separately testing some regular properties of the input and output strings, as explained in Exercise B.1.6.

B.1.1 Composition and continuity

We now show that rational relations enjoy some of the important properties that we have identified in the introduction, namely closure under composition and continuity.

Theorem B.1.4. *Rational relations are closed under relational composition, i.e. if*

$$R \subseteq A^* \times B^* \qquad \textit{and} \qquad S \subseteq B^* \times C^*$$

are rational relations, then the same is true for the composition

$$R \cdot S = \{\ (u,v) \in A^* \times C^* \mid\ uRw \textit{ and } wSv \textit{ for some } w \in B^*\ \}$$

Proof. This is essentially the same product construction that we used for Mealy machines in Theorem A.1.3. By possibly splitting transitions into sequences of transitions, we can assume that in the automata for R and S, each transition produces at most one letter of input or output. Once this has been done, we synchronise the two automata using a product construction, in which two transitions

$$q \xrightarrow{u,w} q' \qquad\qquad p \xrightarrow{w,v} p'$$

in the automata for R and S respectively are combined into a new transition

$$(q, p) \xrightarrow{u,v} (q', p')$$

in the automaton for the composition. (To properly synchronise the two automata, we also assume that each one has an transition with empty input and output around each state.) The rest of the construction is as usual in a product construction, i.e. initial states are pairs that are initial on both coordinates, and likewise for final states. □

We now show that rational relations are continuous. Since we are talking about relations and not functions, the notion of continuity from Definition .0.1 needs to be adapted, by taking inverse images under relations.

Theorem B.1.5. *If $R \subseteq A^* \times B^*$ is a rational relation and $L \subseteq B^*$ is a regular language, then the inverse image*

$$\{ w \in A^* \mid \ wRv \text{ for some } v \in L \}$$

is also a regular language.

Observe that rational relations are symmetric with respect to input and output, and therefore the above theorem also shows that forward images of regular languages under rational relations are also regular.

Proof. One proof is by repeating a product construction, similar to the one used for composition. However, we prefer to deduce continuity from composition directly, since this highlights how the two notions are related. Consider a rational relation with an empty output alphabet. There is only one possible output string, namely the empty string, and hence the relation can be described by saying which input strings can produce the unique output. All regular languages over the input alphabet arise this way, i.e. the map

$$R \quad \mapsto \quad \{ w \in A^* \mid \ wR\varepsilon \}$$

is a bijective correspondence between rational relations with an empty output alphabet and regular languages over the input alphabet.

Using the above bijective correspondence, we can prove the continuity property using closure under composition. Indeed, take some R and L as in the statement of the theorem. By Theorem B.1.4, the composition

$$R \cdot \{ (w, \varepsilon) \mid \ w \in L \}$$

is a rational relation. The corresponding language is exactly the inverse image of L under R, and hence it is regular. □

B.1.2 Undecidable equivalence

The trouble with rational relations is that nondeterminism makes the equivalence problem undecidable.

Theorem B.1.6 (Griffiths). *The equivalence problem $R = S$ is undecidable for rational relations.*

Proof. We will show undecidability already for a special case of the equivalence problem, namely the universality problem: is a rational relation equal to the full relation $A^* \times B^*$? We will do this via a reduction from the Post Correspondence Problem, which is known to be undecidable: given two string homomorphisms $g, h : A^* \to B^*$, decide if there is some nonempty input string where both homomorphisms produce the same output. (Recall that a string homomorphism is a function that applies some function of type $A \to B^*$ to each input letter. This is like the letter-to-letter homomorphisms, except that letters can also be erased, or replaced by longer strings.) The main observation is that the complement of a homomorphism can be defined by a rational relation.

Claim B.1.7. *If $h : A^* \to B^*$ is a homomorphism, then its complement defined by*

$$R = \{\ (w, v) \in A^* \times B^* \mid\ v \neq h(w)\ \}$$

is a rational relation.

Proof. The automaton guesses a prefix of the input word for which the homomorphism is correctly applied. After reading this prefix and producing the correct output, it insists on making an error. This means that for the next input letter a after the correct input prefix, it does one of the following: (i) outputs a string that is a proper prefix of $h(a)$ and then produces no more output; or (ii) outputs a string that is incomparable with $h(a)$ under the prefix relation and then produces any output. □

Using the complements from the above claim, we will recast the Post Correspondence Problem as an instance of universality for rational relations. Consider an instance of the Post Correspondence Problem[4]:

$$\exists w \in A^*\ h(w) = g(w).$$

The negation of this problem is

$$\forall w \in A^*\ \forall v \in B^*\ (w, v) \in \underbrace{(\text{complement of } g) \cup (\text{complement of } h)}_{\text{a union of two rational relations}}.$$

Since rational relations are closed under union, we have shown that the complement of the Post Correspondence Problem is the same as an instance of the universality problem for rational relations, and hence the latter problem must be undecidable. □

The above problem is one of the reasons why we avoid relations in this book. The undecidability and related complications would get only worse for the more advanced models of transducers that we will be looking at later.

[4]This problem was proved undecidable in Emil L. Post, "A Variant of a Recursively Unsolvable Problem", 1946, p.264, where it was called the "correspondence problem". A more modern proof can be found in Michael Sipser, *Introduction to the Theory of Computation*, 2012, Theorem 5.15.

Exercises

Exercise B.1.1. Show that for every rational relation $R \subseteq A^* \times B^*$, the following languages are regular:

1. its domain, i.e. input strings that produce at least one output;
2. its range, i.e. output strings that arise from at least one input;
3. input strings that produce infinitely many outputs.

Solution. For the domain, we use continuity from Theorem B.1.5: the domain is the inverse image of the regular language B^*. A more direct argument is that an automaton for the domain is obtained from the automaton with output by erasing the output part of every transition. The range is handled in the same way, using the symmetry of rational relations with respect to input and output.

The third language requires an analysis of the runs. Call a state *pumping* if it admits a cycle with empty input and nonempty output. We claim that an input string has infinitely many outputs if and only if it admits an accepting run which visits a pumping state. The right-to-left implication is clear, since the cycle in a pumping state can be repeated any number of times, each repetition making the output longer. For the left-to-right implication, consider an input string with infinitely many outputs, and for each of these outputs choose an accepting run which uses the least number of transitions. Since each transition produces a bounded amount of output, these runs use unboundedly many transitions, while the number of transitions that consume a nonempty input is at most the length of the input string. Therefore, some run uses more consecutive transitions with empty input than there are states, and hence it visits some state twice, with empty input in between. The cycle obtained this way must have nonempty output, since otherwise we could remove it, contradicting the minimal choice of the run.

Since the pumping states can be computed, the third language is recognised by the automaton which is obtained from the automaton with output by erasing the outputs, and adding to the state a bit which says if a pumping state has already been visited.

Exercise B.1.2. Show that for some rational relation $R \subseteq A^* \times B^*$, the set of inputs that produce at most one output is non-regular.

Solution. The crucial idea is to find two rational functions for which the following language is non-regular:

$$\{ w \in A^* \mid \ f(w) = g(w) \}.$$

Once we have found them, the relation is simply the union of the two functions, and the set of inputs that produce at most one output is exactly the above language, since an input produces two outputs exactly when the two functions disagree on it. A simple example of such functions is when f keeps only the a's, while g keeps only the b's and replaces them by a's; the two functions agree exactly on the strings that have as many a's as b's. (The two functions must use the same output alphabet, since otherwise they could only agree on the empty string.)

Exercise B.1.3. Show that rational relations are not closed under intersection.

Solution. Consider the two relations

$$R = \{\ (a^n, b^n c^m) \mid\ n, m \geq 0\ \} \qquad \text{and} \qquad S = \{\ (a^n, b^m c^n) \mid\ n, m \geq 0\ \}.$$

Both are rational: the automaton for R first outputs one letter b for each input letter, and then, without consuming any input, outputs any number of letters c; the automaton for S does the same with the two phases swapped. Their intersection is

$$\{\ (a^n, b^n c^n) \mid\ n \geq 0\ \},$$

which is not rational. Indeed, the image of the regular language a^* under this relation is $\{\ b^n c^n \mid\ n \geq 0\ \}$, which is not regular, while rational relations map regular languages to regular languages, as observed after Theorem B.1.5.

Exercise B.1.4. Show that it is undecidable if two rational relations have nonempty intersection.

Solution. We reduce from the Post Correspondence Problem, as in the proof of Theorem B.1.6, except that this time we do not need to complement the homomorphisms. Consider two homomorphisms $g, h : A^* \to B^*$, and view each of them as a relation, e.g.

$$\{\ (w, g(w)) \mid\ w \in A^* \text{ is nonempty }\}.$$

This relation is rational: the automaton has an initial and a final state, and for each input letter a it has a transition with input a and output $g(a)$, which goes from the initial to the final state, and also from the final state to itself. (The two states are used only to rule out the empty input string.) The intersection of the two relations obtained this way is nonempty if and only if the two homomorphisms agree on some nonempty input string, which is exactly the Post Correspondence Problem. Since the two relations used in the reduction are functions, the problem remains undecidable for rational functions.

Exercise B.1.5. Show that the following conditions are equivalent for a rational relation:

- every input string produces at most finitely many outputs;
- output lengths are bounded by an affine function of the input length.

Solution. The implication from the second condition to the first one is immediate, since there are finitely many strings of bounded length.

For the converse implication, we use the analysis of runs from Exercise B.1.1. Assume that every input string has finitely many outputs. As we have seen in that exercise, this means that no accepting run visits a state which admits a cycle with empty input and nonempty output. Consider an accepting run, and shorten it as long as possible by removing cycles with empty input. By the previous sentence, the cycles that are removed this way have empty output, and therefore removing them changes

neither the input string nor the output string. In the run that remains, every group of consecutive transitions with empty input is shorter than the number of states, since otherwise some state would repeat and we could shorten the run further. Since the transitions that consume a nonempty input are at most as many as the letters in the input string, it follows that the total number of transitions in the run is at most

$$(\text{number of states}) \cdot (\text{length of the input string} + 1).$$

Each transition produces a bounded amount of output, and hence the length of the output is bounded by an affine function of the length of the input.

Exercise B.1.6. Show that the recognisable subsets of $A^* \times B^*$ are exactly the unions of finitely many products of regular languages, i.e. unions of the form

$$K_1 \times L_1 \cup \cdots \cup K_n \times L_n$$

where each K_i is a regular language over the input alphabet and each L_i is a regular language over the output alphabet.

Solution. We use the standard reformulation of recognisability: a subset of a monoid is recognisable if and only if it is the inverse image $h^{-1}(F)$ of some subset F, under some monoid homomorphism h into a finite monoid. (Given a congruence of finite index, take the quotient monoid; conversely, the equivalence which identifies elements with the same image under h is a congruence of finite index.)

Consider first a product $K \times L$ of regular languages, which are recognised by monoid homomorphisms

$$A^* \xrightarrow{g} M \qquad \text{and} \qquad B^* \xrightarrow{k} N.$$

This product is recognised by the homomorphism into $M \times N$ which maps a pair (w, v) to $(g(w), k(v))$. For a finite union of such products, we take the product of the corresponding monoids, thus obtaining a single homomorphism which recognises all of the products at the same time, and hence also their union, since the recognisable subsets for a fixed homomorphism are closed under Boolean operations.

For the converse implication, consider a recognisable relation $R = h^{-1}(F)$, for some homomorphism h into a finite monoid M. The point is that the two coordinates are independent, because every pair factors as

$$(w, v) = (w, \varepsilon) \cdot (\varepsilon, v).$$

Applying h to this factorisation, we get $h(w, v) = g(w) \cdot k(v)$, where

$$g(w) \stackrel{\text{def}}{=} h(w, \varepsilon) \qquad \text{and} \qquad k(v) \stackrel{\text{def}}{=} h(\varepsilon, v)$$

are monoid homomorphisms from A^* and B^* into M, and hence they recognise regular languages. It follows that

$$R = \bigcup g^{-1}(m) \times k^{-1}(n),$$

where the union ranges over the finitely many pairs $(m, n) \in M \times M$ whose product mn belongs to F. This is a finite union of products of regular languages, as required.

B.2 Rational functions

After the brief detour into relations and nondeterminism, we return to the main focus of this book, which is about functions and deterministic models. As we will see, this will allow us to work around the undecidability result from Theorem B.1.6. We define rational functions semantically, as the functional fragment of the rational relations.

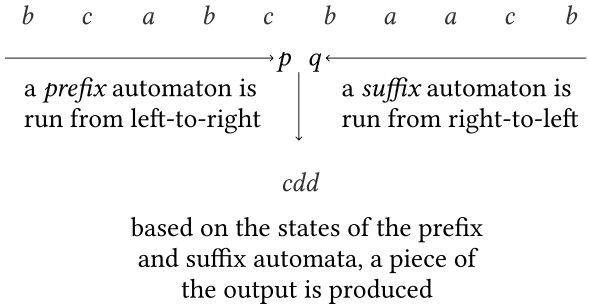
Definition B.2.1 (Rational function)**.** *A rational function is the special case of a rational relation, in which each input string is related to exactly one output string.*

An advantage of the above definition is its simplicity, but a disadvantage is that the underlying automaton model is nondeterministic, and only happens to have unique outputs (possibly arising from different runs with the same output). The disadvantage will be overcome in Section B.2.1 below, where we introduce a deterministic model, called bimachines, which defines exactly the same class of functions.

Rational functions are closed under composition and are continuous, since these properties are inherited from rational relations. As we will see in the next chapter, on weighted automata, rational functions have decidable equivalence.

B.2.1 Bimachines

A bimachine is based on two automata (called the prefix and suffix automata), which are deterministic, but which will be used in opposite directions: the prefix automaton will be used in a left-to-right way, but the suffix automaton will be used in a right-to-left way. For each gap between positions in the input string, it produces a piece of the output according to the following picture:



Here is a more formal definition.

Definition B.2.2 (Bimachine)**.** *A bimachine consists of the following ingredients:*

- *input and output alphabets A and B;*
- *two deterministic automata with input alphabet A, which are called the* prefix *and* suffix *automata (the accepting states in these automata will be irrelevant, so they can be omitted);*

- *an output function of type*

$$\textit{(states of prefix automaton)} \times \textit{(states of suffix automaton)} \to B^*$$

A bimachine computes a function of type $A^* \to B^*$, which is defined as follows. Suppose that the input string is $a_1 \cdots a_n$. For each $i \in \{0, 1, \ldots, n\}$, we consider the partition of the input word as

$$\underbrace{a_1 \cdots a_i}_{\text{prefix}} \quad \underbrace{a_{i+1} \cdots a_n}_{\text{suffix}}.$$

We run the prefix automaton on the prefix, and the suffix automaton on the reverse[5] of the suffix, yielding a pair of states. To this pair of states, we apply the output function, yielding a piece of the output string. The $n+1$ pieces obtained this way are concatenated, in increasing order of i, thus yielding the output string.

We now show that bimachines compute exactly the rational functions. Along the way, we give another characterisation of the rational functions, which uses unambiguous nondeterministic automata with output, i.e. nondeterministic automata that have exactly one accepting run for each input string.

Theorem B.2.3 (Eilenberg)**.** *The following are equivalent for a string-to-string function:*

1. *it is a rational relation which happens to be functional;*
2. *it is computed by an unambiguous nondeterministic automaton with output;*
3. *it is computed by a bimachine.*

Proof. We prove the implications in the order $3 \Rightarrow 1 \Rightarrow 2 \Rightarrow 3$.

From bimachines to functional. We begin with the easiest inclusion

$$\text{bimachines} \quad \subseteq \quad \text{functional rational relations.}$$

Consider some bimachine. The states of the corresponding nondeterministic automaton are pairs (states of prefix automaton, state of suffix automaton), plus one extra state that we will denote by $\dashv$. The essential idea is that the nondeterministic automaton guesses the runs of both the prefix and suffix automaton. More formally, we have transitions of the form

$$(p, q) \xrightarrow{a/w} (p', q'),$$

[5] Using the reverse of the suffix yields a pleasing symmetry in the execution of the bimachine, since both automata run in the direction of the gap. It will also be useful in the proof of Theorem B.2.3. However, running the suffix automaton in the forward direction would give the same class of functions, since the only role of the suffix automaton is to partition the suffixes into finitely many regular languages, and the reverse of a regular language is regular.

where a is an input letter, w is the output string produced by the pair (p, q) and the following are transitions of the prefix and suffix automata:

$$\underbrace{p \xrightarrow{a} p'}_{\text{in prefix automaton}} \qquad \underbrace{q' \xleftarrow{a} q}_{\text{in suffix automaton}}.$$

Since we have used the output string for (p, q) and (p', q'), the resulting automaton outputs for each input letter the output of the bimachine that corresponds to the gap on the left of the letter. This accounts for all gaps except the last one, which is where we use the extra state $\dashv$, for which we have transitions

$$(p, q) \xrightarrow{\varepsilon/w} \dashv,$$

such that w is the output on (p, q) and q is the initial state of the suffix automaton. The initial states in the automaton are pairs (p, q) where p is an initial state of the prefix automaton and q is any state of the suffix automaton. There is only one final state, namely $\dashv$. It is easy to see that the resulting automaton computes the same function as the bimachine.

From functional to unambiguous. We now prove the inclusion

$$\text{functional rational relations} \quad \subseteq \quad \text{unambiguous rational relations}.$$

In the construction, and also later in this book, it will be convenient to eliminate ε-transitions, i.e. transitions with empty input. There are two caveats to eliminating ε-transitions, which is that they are essential for two roles: (a) producing an output for the empty input string; and (b) producing infinitely many outputs for one input string. To deal with issue (a), we will discuss the empty input separately. To deal with issue (b), we allow *extended transitions*, where instead of an output string, a transition is labelled by a regular language of output strings. Subject to these two caveats, we can eliminate ε-transitions, as the following lemma shows.

Lemma B.2.4 (Elimination of ε-transitions)**.** *Every rational relation can be computed by an* NFA *with output and extended transitions, where each accepting run has the following properties:*

1. *if the input string is nonempty, then each transition inputs exactly one letter;*
2. *if the input string is empty, then the run has exactly one transition.*

Furthermore, if the rational relation is such that every input string has finitely many outputs, then extended transitions are not needed.

Proof. Without loss of generality, we can assume that: (a) there is no state that is both initial and final; and (b) every transition in the automaton consumes zero or one input letters. Both of these assumptions can be achieved by using ε-transitions. Furthermore, we assume that (c) every state appears in at least one accepting run.

Having made the above assumptions on the original automaton, we can apply the usual construction for eliminating ε-transitions. We remove all transitions from the

automaton, and replace them as follows. For each input letter a and states p and q, we create a transition of the form

$$q \xrightarrow{a/L} p,$$

where L is the set of outputs that can be produced by a run in the original automaton which starts in q, consumes the letter a and uses any number of ε-transitions, and ends in p. This set is easily seen to be a regular language. As a result, the new automaton has the same outputs for all nonempty inputs strings. Finally, in order to take care of the empty input string, we add a fresh initial state $\vdash$ and fresh final state $\dashv$, together with a transition

$$\vdash \xrightarrow{\varepsilon/L} \dashv,$$

in which L is the set of outputs that can be produced by the original automaton on an empty string.

Finally, if the original automaton had the property that every input string has finitely many outputs, then the regular languages L that label the transitions in the new automaton are finite, and therefore we can replace each transition by a finite number of transitions that produce exactly one output string. □

Using the above lemma, we show that a functional rational relation can be computed by an unambiguous automaton with output. In fact, we will show a stronger result, which we call uniformisation: if a rational relation is total (i.e. every input string produces at least one output string), then it contains an unambiguous rational relation. In the case of a functional rational relation, the totality assumption is satisfied, and the contained unambiguous rational relation must be the same function.

Lemma B.2.5. *[Uniformisation] If a rational relation is total, then it contains an unambiguous rational relation.*

Proof. We begin by eliminating ε-transitions, using the previous lemma. As a result, the automaton might use extended transitions, but this will not be an issue. We call this the original automaton, and our goal is to define a new unambiguous automaton.

Choose some arbitrary linear order on the state space of the original automaton. The new unambiguous automaton will use the run of the original automaton, which is lexicographically minimal with respect to the chosen linear order on states. This new automaton is defined as follows.

- **States.** We have the two special states $\vdash$ and $\dashv$, which will be used for the empty input, similarly to the construction in Lemma B.2.4. Apart from these two states, the new automaton has a state of the form $P \ni p$ for every subset P of states in the original automaton and every chosen element p of this subset. The idea is that P is the set of states that can reach acceptance in the future, and p is the chosen state used in the run with lexicographically minimal profile.
- **Initial states.** The special state $\vdash$ is initial. A state of the form $P \ni p$ is initial if and only if p is initial in the original automaton, and P does not contain any initial state of the original automaton that is smaller than p.

- **Final states.** The special state $\dashv$ is final. A state of the form $P \ni p$ is final if and only if P is equal to the set of accepting states of the original automaton.
- **Transitions.** For the two special states $\vdash$ and $\dashv$, there is a unique transition

$$\vdash \xrightarrow{\varepsilon/w} \dashv$$

where w is some arbitrarily chosen output that can be produced by the original automaton on an empty input. Next, we create transitions of the form

$$P \ni p \xrightarrow{a,w} P' \ni p'$$

under the following conditions:

1. P is the set of all states that can reach P' by reading input a;
2. p' is the minimal state in P' that can be reached from p by reading input a; and
3. w is some chosen output on a transition from p to p' that inputs a.

In the above, the chosen output in item 3 is unique for the automaton, which means that once the source and target states and the input letter have been fixed, it is impossible to have two different outputs.

Let us now argue that this new automaton is unambiguous. The first component of the state $P \ni p$, which is the subset P, is chosen deterministically from right-to-left. Therefore, all runs of the new automaton will agree on this component. Once the first component has been fixed, the second component is deterministic from left-to-right, and therefore all runs will also agree on this component. Finally, the output string is determined uniquely by the states and the input letter. It is also not hard to see that if the original automaton had an accepting run (which is guaranteed by the totality assumption), then the new automaton also has an accepting run, and that the output string is the same. □

From unambiguous to bimachines. The final step in the theorem is the inclusion

$$\text{unambiguous rational relations} \quad \subseteq \quad \text{bimachines.}$$

Consider an unambiguous automaton with output $\mathcal{A}$ that computes some function. We can assume without loss of generality that this automaton has the following property: every accepting run uses exactly one ε-transition (i.e. with empty input), and this is the last transition of the run. This property can be ensured by the construction in the previous step, using the proof of Lemma B.2.5. (In that construction, the automaton uses no ε-transitions on nonempty inputs, but a last ε-transition can be added separately.) We now design a bimachine that computes the same output.

The bimachine produces the output as follows. For each gap in the input string (i.e. a factorisation into prefix and suffix), the following output is produced: (a) if the gap is not the last one, i.e. the suffix begins with some input letter, then we produce the

part of the output that corresponds to the transition of the original automaton which consumes this letter; and (b) if the gap is the last one, i.e. the suffix is empty, then we produce the part of the output that corresponds to the ε-transition at the end of the run of the original automaton. It remains to define the prefix and suffix automata so that the above can be implemented. The prefix automaton computes the states that can be reached from an initial state by reading the prefix. The suffix automaton computes a little more information, namely: is the suffix empty (i.e. are we at the last gap), and if the suffix is nonempty, then:

- what is the first letter of the suffix; and
- which states can reach an accepting state via the suffix minus its first letter.

Using this information we can determine the output that should be produced at the gap, as described above. □

B.2.2 Decomposition into primes

Using bimachines, we can prove a variant of the Krohn-Rhodes decomposition theorem for rational functions. In fact, most of the decomposition work was already done in the case of Mealy machines. In the theorem, we use right-to-left Mealy machines, which are defined the same way as Mealy machines, except that the input string is processed in a right-to-left manner, with the initial state being used after the rightmost position.

Theorem B.2.6. *A string-to-string function is rational if and only if it can be obtained by composing the following functions (which we call the* prime rational functions*):*

1. *the prime Mealy machines, i.e. reversible and flip-flop Mealy machines;*
2. *the right-to-left variants of the prime Mealy machines;*
3. *string homomorphisms;*
4. *the function $w \mapsto w\#$ which appends a fresh separator symbol to the input string.*

Proof. Clearly all prime functions are rational, and rational functions are closed under composition, which gives us the implication $\Leftarrow$. Let us now prove the implication $\Rightarrow$.

By Theorem B.2.3, we can assume that f is computed by a bimachine. We first apply the function $w \mapsto w\#$. Next, we run the prefix and suffix automata on the string, and label the input string with the states of these automata. The state is stored in the position to its right, which is why we use the separator symbol $\#$ to store the state at the end of the string. The prefix and suffix automata are Mealy machines, with the suffix one being a right-to-left Mealy machine. Finally, we apply a homomorphism, which produces the desired output string. □

B.2.3 Mealy machines as a subset of the rational functions

Mealy machines are a special case of rational functions. The following theorem explains exactly which special case they are.

Theorem B.2.7. *Consider a rational function $f : A^* \to B^*$. This function is computed by a Mealy machine if and only if it satisfies the following two properties:*

- Letter-to-letter. *For every input string, the output string has the same length;*
- Determinism. *If two input strings agree on the first n input letters, then the corresponding output strings also agree on the first n output letters.*

Exercises

Exercise B.2.1. Represent the following functions as rational functions, and as bimachines: (a) if the input length is even, output it, otherwise output the empty string; (b) swap the first and last letters; (c) identity before last $\#$, after that every letter replaced by $\#$.

Solution. (a) As a rational function, we use two copies of the automaton which tracks the parity of the number of input letters that have been read so far. In the first copy, every transition copies its input letter to the output, and the accepting states are those with even parity; in the second copy, every transition has empty output, and the accepting states are those with odd parity. Putting the two copies side by side gives an unambiguous automaton, since the parity of the input length decides which copy is used. As a bimachine, we use the parity automaton for both the prefix and the suffix automata, except that the suffix automaton also stores the first letter of the suffix. The parity of the input length is the sum of the two parities, and hence it is known in every gap. The output in a gap is the first letter of the suffix if the total parity is even, and the empty string otherwise.

(b) As a rational function, this was already done in Section B.1, using a union of six automata. That union is unambiguous, since the six cases are determined by the input string. (That solution was for a binary alphabet, but a similar construction works in general.) As a bimachine, we use a prefix automaton which stores the first letter of the prefix, and a suffix automaton which stores the first and the last letter of the suffix, as well as the length of the suffix up to the threshold two. The output in a gap is defined by the following cases: the empty string, if the suffix is empty; the last letter of the suffix, if the prefix is empty; the first letter of the prefix, if the suffix has exactly one letter; and the first letter of the suffix in the remaining cases. The three letters that are used here are, respectively, the last letter of the input string, its first letter, and the letter in the position that follows the gap.

(c) We use the convention that if the input string has no $\#$, then all of its letters are replaced by $\#$; the other convention, in which the input string is returned unchanged, is handled in the same way. For an unambiguous automaton, the automaton

guesses which # is the last one: up to and including this letter, the input is copied to the output; after it, the automaton outputs # for every input letter, and rejects if it sees another #. This automaton is unambiguous, since the guess is verified. The bimachine is even simpler, and it does not need the prefix automaton at all. The observation is that a position is copied to the output if and only if the suffix which begins in this position contains a #. Therefore, it is enough that the suffix automaton stores the first letter of the suffix, together with the information about whether the suffix contains #. The output in a gap is: the empty string, if the suffix is empty; the first letter of the suffix, if the suffix contains #; and # otherwise.

Exercise B.2.2. Show that if a rational function has unbounded output size, then it has exactly linear output size in the following sense: the limit

$$\lim_{n\to\infty} \frac{\text{maximal output length on inputs of length at most } n}{n}$$

is defined and nonzero.

Solution. We use nondeterministic transducers, and we assume that every state is reachable from an initial state, and can reach an accepting state. Call the *ratio* of a run the length of its output string divided by the length of its input string. We claim that the limit in the exercise is qual to

$$\sup_{\rho} \frac{\text{length of output of } \rho}{\text{length of input of } \rho},$$

where ρ ranges over runs that are cycles (same source and target state) with nonempty input. We first observe that the supremum is reached: indeed if we take any cycle which contains nested cycles, then removing the nested cycles can only improve the ratio, unless the nested cycles had better ration. Therefore, nested cycles cannot contribute to the supremum, and thus there are finitely many candidates for the cycle in the supremum, which means that the supremum is actually a maximum. Repeating for every a cycle achieves the maximum gives a lower bound on the limit in the statement of the exercise. This ratio is also an upper bound, since it can be approximated arbitrarily closely by cycles that appear in long runs. (A more formal analysis of the upper bound would look at strongly connected components of states in the automaton).

Exercise B.2.3. Show that the limit in the previous exercise is a rational number.

Solution. As we have shown in the solution to the previous exercise, the supremal ratio is attained by a simple cycle.

Exercise B.2.4. Prove that the following functions are not rational, over an alphabet with at least two letters: (a) first half of the input string, rounded up; and (b) the duplicate function $w \mapsto ww$.

Solution. (a) This function is not continuous, and hence it is not rational by Theorem B.1.5. Consider the inverse image of the regular language a^*, i.e. the set of input

strings whose first half uses the letter a only. If we intersect this inverse image with the regular language a^*b^*, then we get

$$\{\, a^i b^j \mid \; j \leq i \,\},$$

which is not regular. Therefore the inverse image is not regular either.

(b) Here continuity is not violated, since duplication is continuous. Instead, we use the observation after Theorem B.1.5, which says that rational relations map regular languages to regular languages. The image of the regular language A^* under duplication is

$$\{\, ww \mid \; w \in A^* \,\},$$

which is not regular, as long as the alphabet has at least two letters. (For a one-letter alphabet, duplication is the homomorphism which maps a to aa, and hence it is rational.)

Exercise B.2.5. Show that one can decide unambiguity for a given NFA that recognises a language (not a function or relation).

Solution. We can assume that the automaton has no transitions with empty input, since these can be eliminated in the usual way. Under this assumption, the automaton is ambiguous if and only if some input string admits two accepting runs which use different transitions in some position. This is tested by a product construction. Consider the automaton whose states are triples

$$(\text{state}, \text{state}, \text{a bit}),$$

which runs two copies of the original automaton on the same input string, and uses the bit to remember if the two copies have already used different transitions. The bit is 0 in the initial states, it is set to 1 as soon as the two copies use different transitions, and it is never reset. The initial states are the pairs of initial states with bit 0, and the accepting states are the pairs of accepting states with bit 1. The original automaton is ambiguous if and only if the new automaton accepts some string, which is decided by checking if an accepting state is reachable from an initial state. Since the new automaton is only quadratically bigger than the original one, this algorithm runs in polynomial time.

Exercise B.2.6. Are the following problems about two rational functions decidable: (a) is there some input string where both outputs are equal? (b) is there some input string where both outputs have the same length?

Solution. The first problem generalises PCP and is hence undecidable. (Some care is needed with the empty input string, on which two homomorphisms always agree; this is repaired by modifying the two rational functions so that they disagree on the empty input, which is possible because a rational function treats the empty input separately.) The second problem is decidable, since it can be solved using Parikh images: the pairs

$$(|f(w)|, |g(w)|) \qquad \text{for } w \in A^*$$

form the Parikh image of a regular language, namely the language of runs of the product of the two transducers, and hence they form a semilinear set which can be computed. It remains to check if this set contains a pair with two equal coordinates.

Exercise B.2.7. Consider a rational function where the input alphabet has only one letter a. Show that the graph of the function is a finite union

$$\bigcup_{i \in I} \{\ a^{\alpha_i + \beta_i n} \mapsto x_i y_i^n z_i \mid\ \ n \in \mathbb{N}\ \}$$

where the coefficients α_i, β_i are natural numbers and strings x_i, y_i, z_i are strings over the output alphabet.

Solution. Consider a bimachine that computes the function, which exists by Theorem B.2.3. Over a one-letter input alphabet, the prefix and suffix automata are deterministic automata with a one-letter alphabet, and therefore their runs are eventually periodic: there is a threshold λ and a period π, which we can choose to be the same for both automata, such that for every $i \geq \lambda$ and for both automata, the state after reading a^i is the same as the state after reading $a^{i+\pi}$.

The input strings of length smaller than 2λ contribute finitely many pairs to the graph, and each such pair is of the required form, with the coefficient β_i equal to zero. For the remaining input strings, we group them according to the residue of their length modulo π, i.e. we consider the input strings

$$a^{\alpha + \pi m} \qquad \text{for a fixed } \alpha \text{ and } m \in \mathbb{N}.$$

The output in a gap is determined by the states of the two automata, and these are determined by the number of letters on the left and on the right of the gap. For the first λ gaps, this pair of states does not depend on m, and the same is true for the last λ gaps. For every other gap, both sides have at least λ letters, and hence the pair of states depends only on the number of letters on the left, modulo π. Therefore, increasing m by one has the following effect on the output: one more group of π consecutive gaps appears in the middle, and this group produces the same string as the other such groups. Summing up, the output for the input string $a^{\alpha+\pi m}$ is of the form $xy^m z$, where the strings x, y, z depend only on the residue α, which is the required form with $\beta_i = \pi$.

Exercise B.2.8. Show that there is a function

$$f : A^* \to B^*$$

which is not rational, but has the property that for every rational function

$$g : B^* \to \underbrace{1^*}_{\text{words over a one-letter alphabet}},$$

the composition $f \cdot g$ is rational.

Solution. Take f to be the string reversal function, which is not rational by Example 17. Let

$$g : B^* \to 1^*$$

be a rational function, and consider a bimachine that computes it, which exists by Theorem B.2.3. The output of a bimachine is the concatenation, from left to right, of the pieces that are produced in the gaps of the input string. Over a one-letter alphabet, concatenation is commutative, and therefore the output depends only on which pieces are produced, and not on the order in which they are produced.

This is what makes reversal harmless. Reversing the input string is a bijection on gaps, which swaps the prefix with the suffix. Therefore, the composition of reversal with g is computed by the bimachine that is obtained from the one for g by swapping the prefix and suffix automata, and swapping the two arguments of the output function. This new bimachine produces the same pieces as the original one, but in the opposite order, which is invisible over a one-letter alphabet.

In the following series of exercises, we will be interested in ideals of rational functions. Such an ideal is defined to be a set $\mathcal{I}$ of rational functions such that

$$\mathcal{I} = \text{Rational} \cdot \mathcal{I} \cdot \text{Rational},$$

i.e. functions that factor through the ideal must stay in the ideal.

Exercise B.2.9. Show that the following are ideals:

- functions with range of size at most k for $k \in \{1, 2, 3, \ldots\}$;
- functions where the number of outputs is $\mathcal{O}(n^k)$, where n is the input length and $k \in \{0, 1, 2, \ldots\}$ is some fixed constant.

Solution. Observe that the ideal in the second item for $k = 0$ describes the class of rational functions with finitely many outputs.

Pre-composing or post-composing with rational functions (or any functions) cannot increase the size of the range, and therefore the first item describes an ideal. For the second item, we use a similar argument, but for precomposition we observe that the output size of a rational function is at most linear in the input length, and therefore pre-composing with a rational function cannot increase the degree of the polynomial.

Exercise B.2.10. Consider an ideal where all functions have finite range. Show that the ideal is equal to one of the ideals from the first item in Exercise B.2.9, or to the ideal $\mathcal{O}(n^0)$ from the second item.

Solution. Suppose that the ideal contains a rational function f with distinct outputs

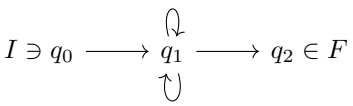
$$f(w_1), \ldots, f(w_k).$$

Every rational function g with at most k possible outputs $v_1, \ldots, v_\ell$ can be obtained as follows: first apply g with v_i replaced by w_i, then apply f, and finally replace $f(w_i)$ with v_i. The first and third steps are rational functions, and therefore g factors through f, which means that it is in the ideal.

It follows that the ideal is determined by the sizes of the ranges that appear in it. If these sizes are bounded, then the ideal is "range of size at most k", where k is the largest size that appears. Otherwise the ideal contains rational functions with arbitrarily large finite ranges, and hence, by the argument above, it contains every rational function with finite range, i.e. it is the ideal $\mathcal{O}(n^0)$.

Exercise B.2.11. Show that an ideal contains all rational functions if and only if it contains some function whose range is a regular language with super-polynomial growth. (The growth rate of a language is a function that maps an input length n to the number of strings in the language that have length at most n.)

Solution. By the analysis from Exercise A.1.5, a regular language has super-polynomial growth if and only if an automaton recognising it contains a pattern of the form



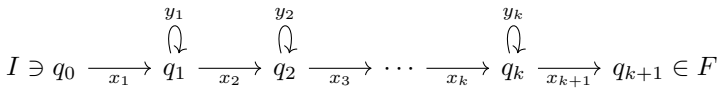
such that the two loops have different output strings of the same length. Write x and x' for the strings on the two horizontal arrows, and y_a and y_b for the strings on the two loops; recall that the latter two are different strings of the same length. For an input string $c_1 \cdots c_n$ over the alphabet $\{a, b\}$, the string

$$x\, y_{c_1} \cdots y_{c_n}\, x'$$

belongs to the range of the function, and different input strings give different strings, since the two loops have the same length. Producing these strings is easy, since this is done by the homomorphism which maps c to y_c, followed by adding x in front and x' at the end. What we need, however, is to produce them as *inputs* of the function, and not as outputs. This is where we use the Uniformisation Lemma B.2.5: the inverse relation of the function is rational, and it becomes total once we extend it with a default output for the strings outside the range, which is a regular language. This gives a rational function which chooses, for each string in the range, some input that produces it, and we pre-compose with it. Post-composing with the rational function which removes x and x', and replaces each block y_a or y_b by the corresponding letter, we recover the string $c_1 \cdots c_n$. In other words, the ideal contains the identity function on the two-letter alphabet $\{a, b\}$. From this we can get the identity function on any alphabet, and therefore all rational functions.

Exercise B.2.12. Show that if an ideal contains some function whose range has growth $\Omega(n^k)$ with $k \in \{1, 2, \ldots\}$, then it contains all functions whose range has growth $\mathcal{O}(n^k)$.

Solution. Define a k-pattern in an automaton that recognises a language (not a function or relation) to be a sequence of states and runs as in the following diagram:



such that for every $i \in \{2, \ldots, k\}$, the language $y_{i-1}^* x_i y_i^*$ is not contained in the prefixes of y_i^*. Using the analysis of loops in automata from Exercise A.1.5, we can show that the growth rate of a regular language is $\Omega(n^k)$ if and only if it contains a k-pattern. (In particular, containing a k-pattern does not depend on the choice of automaton that recognises the language.) Next, using a similar analysis as in the previous exercise, one can show that if the range of some rational function in the ideal contains a k-pattern, then the ideal contains the function

$$\{a_1, \ldots, a_k\}^* \xrightarrow{f_k} \{a_1, \ldots, a_k\}^*$$

which is the identity on sorted strings (i.e. strings from $a_1^* a_2^* \cdots a_k^*$) and outputs the empty string otherwise.

Finally, it remains to show that if an ideal contains the function f_k, then it contains all rational functions whose range has growth $\mathcal{O}(n^k)$. This can be proved by analysing the structure of strongly connected components in the automaton that computes a rational function whose range has growth $\mathcal{O}(n^k)$.

Exercise B.2.13. Show that the ideals from the previous exercise are all the ideals, i.e. all possible ideals are: "range of size at most k", "range has growth rate $\mathcal{O}(n^k)$", "range has polynomial growth", and "all rational functions", where $k \in \{0, 1, 2, \ldots\}$.

Solution. Let us begin by clarifying an edge case in the statement. When $k = 0$, then the ideal "range has size at most k" is the empty ideal (since functions have at least one output), and the ideal "range has growth rate $\mathcal{O}(n^k)$" is the ideal of functions with finitely many outputs. Indeed, by Exercise B.2.10, all ideals with functions of finite range are either of the form "range of size at most k" or "range has growth rate $\mathcal{O}(n^0)$". By Exercise B.2.11, all ideals that contain a function with super-polynomial growth are equal to the ideal of all rational functions. Finally, consider an ideal in which every function has a range of polynomial growth, and which contains at least one function with an infinite range. If there is a largest k such that some function in the ideal has range of growth $\Omega(n^k)$, then by Exercise B.2.12 the ideal is exactly the ideal of functions with growth $\mathcal{O}(n^k)$. Otherwise the ideal contains, for every k, some function of growth $\Omega(n^k)$, and hence, again by Exercise B.2.12, it is the union of the ideals $\mathcal{O}(n^k)$ over all k, i.e. the ideal of functions whose range has polynomial growth.

Exercise B.2.14. Show that it is decidable if two rational functions generate the same ideal (the ideal generated by a function is the last ideal that contains it).

Solution. By Exercise B.2.13, the ideal generated by a function is determined by its range: if the range is finite, then the ideal is "range of size at most k" where k is the size of the range; and otherwise the ideal is determined by the growth rate of the range. (The growth rate alone is not enough, since all functions with a finite range have the same growth rate.) The size of the range, and its growth rate, can both be computed by looking for the patterns from the previous exercises in the automaton for the range of the function.

Exercise B.2.15. Show that if a rational function $f : A^* \to B^*$ is surjective, then it has a one-sided inverse, i.e. a rational function $g : B^* \to A^*$ such that $g \cdot f$ is the identity on B^*.

Solution. Rational relations are symmetric with respect to input and output: swapping the input and output labels on the transitions of a nondeterministic automaton with output gives an automaton for the inverse relation. Therefore, the inverse relation

$$\{ (v, w) \in B^* \times A^* \mid \ f(w) = v \}$$

is rational, and it is total, because f is surjective. By the Uniformisation Lemma B.2.5, this relation contains a rational function $g : B^* \to A^*$. By the definition of the inverse relation, we have $f(g(v)) = v$ for every $v \in B^*$, which is the same as saying that $g \cdot f$ is the identity on B^*.

Exercise B.2.16. Show that the following problem is decidable: given a rational function f, we want to know if it is injective, i.e. different input strings are mapped to different output strings.

Solution. Consider the inverse relation of f, which is rational. It need not be total, since f need not be surjective, but its domain is the range of f, which is a regular language, and hence we can make the relation total by adding a default output for the strings outside the range. By the Uniformisation Lemma B.2.5, this relation contains a rational function g, which satisfies $f(g(v)) = v$ for every v in the range of f. The function f is injective if and only if g inverts it on the other side as well, i.e. if and only if the composition $f \cdot g$, which first applies f and then g, is the identity on A^*. Indeed, if f is injective, then $g(f(w))$ has the same image under f as w, and hence it must be equal to w; conversely, a function which has a left inverse is injective. By Theorem B.3.4, it can be decided if $f \cdot g$ is the identity.

Exercise B.2.17. Show that the following problem is undecidable: given a rational function $f : A^* \to A^*$, we want to know if it generates finitely many functions under composition, i.e. if the following set is finite:

$$\{ f^n \mid \ n \in \{0, 1, 2, \ldots\} \}.$$

Solution. Encode a configuration of a Turing machine as a string which contains the contents of the tape, with the current state inserted at the position of the head. For a fixed Turing machine, the function which maps a configuration to the next one is rational: a bimachine copies the input string, except in the two positions next to the state, which it rewrites according to the transition function. (On strings that do not encode a configuration, the function can do anything, say leave the string unchanged.) The n-th iterate of this function maps a configuration to the configuration after n steps.

The set in the exercise is finite if and only if $f^n = f^{n+k}$ for some n and some $k \geq 1$, i.e. if and only if there is a bound n such that the behaviour of the machine on *every* configuration becomes periodic after at most n steps. For a machine which

stays in its halting configuration once it has halted, and which never visits the same configuration twice before halting, this says that every configuration halts after at most n steps.

It remains to reduce the halting problem to this property. Given a machine M and an input string x, consider the machine which, on a tape of length m, simulates M on the input x, and which halts as soon as the simulation would need more than m tape cells. We can assume that M never repeats a configuration, by adding a step counter to the tape; in particular, if M does not halt on x, then its space grows without bound. If M halts on x, then every configuration halts within a number of steps that does not depend on m, and the set of iterates is finite. If M does not halt on x, then the number of steps before the simulation runs out of space grows with m, and hence there is no such bound.

Exercise B.2.18. Define a *grammar compression* for a string to be a context-free grammar that generates the string and nothing else. We say that a string-to-string function is *compatible with compression* if for every input string with a grammar compression of size n, there is a compression of the output string that has size polynomial in n[6] Show that rational functions are compatible with compression.

Solution. This follows from a stronger result: for a context-free language $L \subseteq A^*$ and a rational relation $R \subseteq A^* \times B^*$, the image

$$\{\, v \in B^* \mid \ (w, v) \in R \text{ for some } w \in L \,\}$$

is context-free and can be computed in polynomial time (using grammars as representation of languages and nondeterministic transducers as representation of relations). The proof is a straightforward product construction. In the special case when the input language is a singleton and the relation is a function, the image is also a singleton.

[6]One could consider a stronger version, where the compressed output can be computed in polynomial time. Our positive reults will ensure the stronger version, and our negative results will exclude the weaker version.

$$
\begin{array}{c|c}
x + (y + z) = (x + y) + z & x \times (y \times z) = (x \times y) \times z \\
x + y = y + x & x \times 1 = x = 1 \times x \\
x + 0 = x & x \times 0 = 0 = 0 \times x
\end{array}
$$
$$
x \times (y + z) = x \times y + x \times z
$$
$$
(x + y) \times z = x \times z + y \times z
$$

Figure 1: The semiring axioms.

B.3 Rational relations and weighted automata

In this section, we prove that one can decide if two rational functions are equivalent. We describe the connections of rational relations (including their functional fragment) with another automaton model, called weighted automata. They could have an arbitrary semiring for outputs, but decidability of equivalence was shown when the semiring is a field. Using this connection, and a certain decidability result for weighted automata, we will show that equivalence of rational functions is decidable This connection will also be used in the future, to decide equivalence for a more powerful transducer model, namely the regular functions.

B.3.1 Weighted automata

We begin by defining weighted automata. For the equivalence algorithm, we will be mainly interested in weighted automata that have outputs in a field, but for other results, we will use the more general variant with outputs in an arbitrary semiring. Intuitively speaking, a semiring is like a field, except that we do not require any kind of inverses (both additive and multiplicative), and multiplication can be non-commutative. Here is the formal definition.

Definition B.3.1 (Semiring)**.** *A semiring is a set $\mathbb{S}$ equipped with two binary operations $+$ and $\times$, and two distinguished elements 0 and 1, subject to the axioms in Figure 1.*

Instead of carefully reading the semiring axioms, for our applications it will be more useful to see the examples.

Example 9. [Fields and rings] Every field, such as the field of rationals $(\mathbb{Q}, +, \times)$, is a semiring. This is true also for rings, such as the ring of integers $(\mathbb{Z}, +, \times)$. Finally, if we restrict the integers to the natural numbers, the result $(\mathbb{N}, +, \times)$ is still a semiring. □

Example 10. [Boolean semiring] In this semiring, the underlying set is $\{0, 1\}$, the semiring addition is $x \vee y$, and the semiring multiplication is $x \wedge y$. Both operations are commutative, but neither of them has an inverse. □

Example 11. [Semiring of regular languages] In this semiring, the underlying set is the regular languages, semiring addition is union $L \cup K$, and semiring multiplication is concatenation $L \cdot K$. □

Definition B.3.2 (Weighted automaton)**.** *A weighted automaton over a semiring $\mathbb{S}$ is defined in the same way as an* NFA *with output, except that:*

1. *instead of output strings, transitions use elements of the output semiring; and*
2. *we require that for every input string, there are finitely many accepting runs.*

The semantics of a weighted automaton is the function which maps an input string w to the sum

$$\sum_{\rho} \text{output of } \rho,$$

where the sum ranges over accepting runs over the input string, and the output of a run is the multiplication (in the semiring) of the weights of its transitions. If multiplication in the semiring is non-commutative, then it is important to multiply the transition weights in the order in which they are taken in the run. The assumption that there are finitely many accepting runs for every input string ensures that the above sum is well-defined.

Example 12. [Weighted automata over the Boolean semiring] When the semiring is the Boolean semiring, then a weighted automaton is the same as a nondeterministic automaton. Therefore, the functions computed by such weighted automata are exactly the regular languages. □

Just as regular languages arise as a special case of weighted automata in the Boolean semiring, the rational relations will arise for a different choice of semiring, namely the semiring of regular languages. This is shown in the following example.

Example 13. [Weighted automata over the semiring of regular languages] When the semiring is the semiring of regular languages, then weighted automata are the same as rational relations, i.e. a relation

$$R \subseteq A^* \times B^*$$

is rational if and only if the function

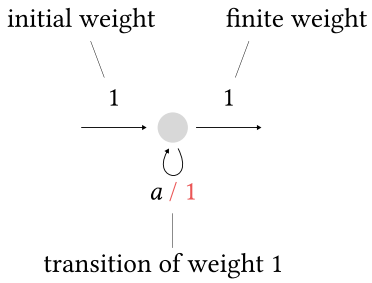
$$w \in A^* \quad \mapsto \quad \{ v \in B^* \mid wRv \}$$

is computed by a weighted automaton with outputs in the semiring of regular languages. This is not hard to see. To go from a rational relation to a weighted automaton, we simply use outputs on transitions that are singleton regular languages. In the opposite direction, a transition

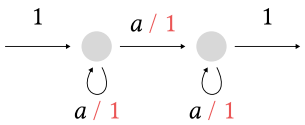
$$p \xrightarrow{a/L} q$$

in a weighted automaton can be replaced by a copy of an NFA that recognises the language L. □

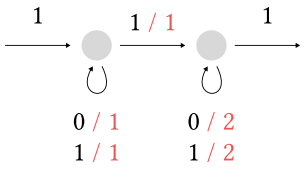
Example 14. [Weighted automata over $\mathbb{N}$] Let us give some examples of weighted automata over the semiring of natural numbers. Consider first the following automaton, which has a one-letter input alphabet:



This weighted automaton has exactly one run of weight 1 for each input word, and therefore it computes the constant function $w \mapsto 1$. If we want to compute the length of the input string, then we can use the following weighted automaton, which has one run of weight 1 for each input position:



Finally, consider the following weighted automaton with input alphabet $\{0, 1\}$:



For each position with label 1 in the input string, this automaton has a run whose weight is 2^i, where i is the distance until the end of the input string. Therefore, this automaton maps an input string to the number that is represented by it in binary notation. □

B.3.2 Decidable equivalence

In this section, we leverage the connection with weighted automata to decide equivalence of rational functions. To do this, we will use decidability of equivalence for weighted automata over the field of rationals[7].

[7] Schützenberger's orginal paper describes minimisation, see M. P. Schützenberger, "On the definition of a family of automata", 1961, Section B. From minimisation one can deduce an algorithm for zeroness, and from that an algorithm for equivalence. All of that, however, is not see easy to identify in the original paper.

Theorem B.3.3 (Schützenberger)**.** *Given two weighted automata over the field of rationals, it is decidable whether they compute the same function.*

The above result is well-known, but in the interest of completeness, we give a proof of it in the next section. We have stated it for the field of rationals, which is the field that we will use in our applications to string-to-string transducers. Nevertheless, the same proof will work for any other field, as long as field elements can be represented in a finite way and field operations can be computed. Let us now show how to use the above result to decide equivalence of rational functions. (The history of Theorem B.3.4 is discussed in Section B.5.)

Theorem B.3.4. *The equivalence problem $f = g$ is decidable for rational functions.*

Proof. We reduce the problem to equivalence for weighted automata over the field of rationals. For this reduction, we need three observations: (a) output strings can be injectively represented using rational numbers; (b) weighted automata are closed under pre-composition with rational functions; and (c) equivalence is decidable for weighted automata over the field of rationals. Here is a picture of the reduction:

$$A^* \underset{g}{\overset{f}{\rightrightarrows}} B^* \xhookrightarrow{\iota} \mathbb{Q}$$

In the picture, f and g are two rational functions, and ι is the injective weighted automaton from observation (a). By injectivity, the equivalence problem $f = g$ has the same answer as the equivalence problem $f \cdot \iota = g \cdot \iota$. By the closure from observation (b), the latter is an instance of equivalence for weighted automata, and hence decidable by observation (c). The decidability from observation (c) is a well-known result, but we give a self-contained proof in the next section for completeness. It remains to prove the two observations (a) and (b).

For observation (a) about an injective representation, we can use the weighted automaton from Example 14, which returns the natural number (a special case of a rational number), that is represented by the input string in binary (or any other base, for input alphabets with more than two letters). The only obstacle is leading zeros in the input string; to circumvent this obstacle the weighted automaton can begin by prepending the digit 1.

Observation (b) is proved in the following lemma. We prove the lemma for arbitrary semirings, and not just the field of rationals, since the more general result will be useful in the future.

Lemma B.3.5. *Weighted automata (over any semiring, not necessarily the field of rationals) are closed under pre-composition with rational functions, i.e. the composition*

$$A^* \underbrace{\xrightarrow{f}}_{\substack{\text{a rational} \\ \text{function}}} B^* \underbrace{\xrightarrow{h}}_{\substack{\text{a weighted} \\ \text{automaton}}} \mathbb{S}$$

is necessarily a weighted automaton.

Proof. We use a product construction. To make this construction correct, we want to ensure that there is a one-to-one correspondence between accepting runs of the product automaton and runs of the second weighted automaton. This will be achieved by making the first automaton (for the rational function f) unambiguous, and appropriately aligning the two runs in the product constructions.

By Lemmas B.2.4 and B.2.5, we assume that the automaton for the rational function f is: (a) unambiguous; and (b) for every nonempty input string, each transition used in the unique accepting run consumes exactly one input letter. Condition (a) will be useful to ensure that there is no double counting, and condition (b) will be used to align the runs of the two automata in the product construction.

The weighted automaton for the composition is now constructed as follows. The states of the new weighted automaton for the composition are pairs (state of f, state of h), plus two extra states $\vdash$ and $\dashv$. The initial states are $\vdash$ and every pair (p, q) that is initial on both coordinates, in the corresponding automata. The state $\dashv$ is the unique final state. Transitions between states that are pairs are of the form

$$(p, q) \xrightarrow{a/s} (p', q')$$

where a is an input letter and $s \in \mathbb{S}$ is the sum of weights of runs in the weighted automaton h that have the following properties: the source state is q, the target state is q', and the input string is equal to the output string of some transition in f that goes from state p to q. To reach the unique final state from pair states, we have transitions of the form

$$(p, q) \xrightarrow{\varepsilon,s} \dashv,$$

where s is the sum of weights for runs in the weighted automaton that have the following property: the source state is q, the target state is final, and the input string is empty. Finally, we need a transition that takes care of the empty input string, which is of the form

$$\vdash \xrightarrow{\varepsilon,s} \dashv,$$

where s is the output of the weighted automaton on the input string $f(\varepsilon)$. □

□

In the proof above, we established that if a function is rational, then weighted automata are closed under pre-composition with it. This can be seen as a variant of continuity, except that instead of regular languages (i.e. weighted automata over the Boolean semiring), we are using weighted automata over the field of rationals. As it turns out, this variant of continuity is exactly the same as rationality.

Theorem B.3.6. *A string-to-string function $f : A^* \to B^*$ is rational if and only if weighted automata over all (not necessarily commutative) semirings are closed under pre-composition with f.*

Proof. The left-to-right implication was shown in Lemma B.3.5. For the right-to-left implication, consider the semiring of regular languages (which is not commutative when the alphabet has more than one letter), and the function

$$w \in B^* \quad \mapsto \quad \{w\} \in \mathbb{S}.$$

By closure under pre-composition, it follows that

$$w \in A^* \quad \mapsto \quad \{f(w)\} \in \mathbb{S}$$

is computed by a weighted automaton over the semiring of regular languages. By Example 13, this means that the function f is rational. □

B.3.3 Decidability of equivalence for weighted automata over a field

In this section, we show that equivalence of weighted automata over the field of rationals is decidable. We first observe that the essential problem is zeroness, i.e. equivalence with the zero function. This is because

$$f = g \qquad \text{iff} \qquad f - g = 0,$$

and weighted automata can be subtracted from each other. It remains to show decidability of the zeroness problem.

Theorem B.3.7 (Schützenberger)**.** *The zeroness problem is decidable for weighted automata over the field of rationals*[8]*.*

Proof. Assume that Q is the set of states in the weighted automaton, and A is the input alphabet. We begin by recasting the semantics of the weighted automaton in terms of vectors in $\mathbb{F}^Q$ and linear maps that act on them. Define the *initial vector*

$$q_0 \in \mathbb{F}^Q$$

as follows: on the initial states it has 1, and on the non-initial states it has 0. For each input letter $a \in A$, define its *update map*

$$\delta_a : \mathbb{F}^Q \to \mathbb{F}^Q$$

to be the linear map for which in the corresponding $Q \times Q$ matrix, the coefficient in cell (p, q) stores the sum of weights of runs in the automaton that go from state p to state q on the input string a, and which consume the letter a in the last transition. Define the *final map*

$$F : \mathbb{F}^Q \to \mathbb{F}$$

as follows: it is the linear map where the coefficient next to state q is the sum of weights of runs in the automaton that start in state q and end in a final state, and which

[8] The proof works for any computable field, where field elements can be represented in a finite way and field operations can be computed.

consume no input letters. The data above was constructed so that if we take an input string, then the output of the weighted automaton on this input string is obtained as follows: start in the initial vector, then apply the update maps corresponding to the input letters, and then apply the final map.

Define a *reachable vector* to be a vector that can be obtained from the initial vector by applying a finite sequence of linear maps corresponding to input letters. The zeroness problem is equivalent to

(*) F outputs zero for all reachable vectors.

Since the map F is a linear map, the above is equivalent to

(**) F outputs zero for all linear combinations of reachable vectors.

Because the update maps are linear maps, the above is equivalent to:

(***) F outputs zero on the least subset of $\mathbb{F}^Q$ that contains the initial vector, and is closed under applying update maps and taking linear combinations.

The subspace (***) can be computed using a simple saturation procedure, where we start with a basis that contains only the initial vector, and we keep adding new basis vectors by applying update maps, until no new basis vectors can be added. This must terminate with at most $|Q|$ basis vectors. At the end, we check if the final map is zero on all basis vectors. □

B.4 Machine independent characterisations

In this section, we present machine independent characterisations of transducers, such as Mealy machines and the rational functions. The idea is to describe a class of functions uniquely in terms of the semantic properties of the functions – such as continuity – and not in terms of some computational model. For Mealy machines, the machine independent characterisation will be straightforward, but it will be substantially harder for the rational functions. In fact, our characterisation for rational functions will require developing characterisations for other intermediate classes that lie between Mealy machines and rational functions.

B.4.1 Mealy machines

As a warmup, we give a machine independent characterisation of Mealy machines. We say that a function is prefix preserving if

$$w \text{ is a prefix of } v \quad \Longrightarrow \quad f(w) \text{ is a prefix of } f(v).$$

Similarly, a length preserving function is one in which the length of the input string is the same as the length of the output string. Clearly Mealy machines have these two properties, and they are continuous. It turns out that these properties exactly characterise Mealy machines, as the following theorem shows.

Theorem B.4.1. *A function $f : A^* \to B^*$ is computed by a Mealy machine if and only if it is: (a) continuous; (b) prefix preserving; and (c) length preserving.*

Proof. Clearly a Mealy machine satisfies all conditions in the theorem. Let us now look at the converse. For each output letter $b \in B$, consider the language

$$\{ w \in A^* \mid f(w) \text{ ends with } b \}, \tag{3}$$

which is regular by continuity. Because the function preserves the length and prefixes, we know that each input position contributes exactly one output letter, and furthermore, we can determine which one using finite state control, thanks to the regularity of the languages (3) defined above. Formally speaking, the state space of the Mealy machine is the product of the state spaces of all languages from (3). □

Using the above theorem, we show that the Mealy machines, seen as a subset of the rational functions, are decidable.

Theorem B.4.2. *One can decide if a rational function is computed by a Mealy machine.*

Proof. It is enough to show that we can check if a rational function satisfies the three conditions in Theorem B.4.1. Continuity is for free, since rational functions are necessarily continuous. Let us discuss the other two conditions.

Lemma B.4.3. *One can decide if a rational function is length-preserving.*

Proof. We give two arguments. The first argument is based on semilinear sets. Since we only sketch this argument, we do not recall the definition of semilinear sets. Consider the set of input/output length pairs

$$\{ (|w|, |f(w)|) \mid \ w \in A^* \} \subseteq \mathbb{N} \times \mathbb{N}.$$

This set is semilinear and can be computed using the Parikh image. The function is length-preserving if and only if this set is contained in the diagonal $\{ (n, n) \mid \ n \in \mathbb{N} \}$. Since the diagonal is semilinear and containment is decidable for semilinear sets, we get a proof of the lemma. This concludes the first argument.

Let us now present in more detail a second argument, which is more direct and does not use semilinear sets. Fix an NFA with output that computes f, and which has states Q. Define a *typing* for this automaton to be a function

$$\tau : Q \to \mathbb{Z}$$

such that for every run ρ of the automaton from an initial state to q, we have

$$|\text{output string of } \rho| = |\text{input string of } \rho| + \tau(q).$$

The typing, if it exists, is clearly unique.

Claim B.4.4. *The function f is length-preserving if and only if the typing exists, and all accepting states are mapped to zero.*

Proof. The right-to-left direction is immediate. For the opposite direction, we assume implicitly that every state is productive, i.e. used in some accepting run. To prove the opposite direction, we observe that: (1) if the typing does not exist, then one can find two runs that reach the same state q and have different differences between the output and input lengths, which means that the function cannot be length-preserving; and (2) if the typing exists, but some accepting state has nonzero type, then also the function cannot be length-preserving, since we can find a run that reaches this accepting state and has a difference between output and input lengths equal to this nonzero type. □

Therefore, it remains to check if the conditions from the above claim are satisfied. The unique candidate for the typing can easily be computed, by looking at runs that reach every state. Once we have a candidate for the typing, we need to check if

$$|w| = |v| + \tau(q) - \tau(p) \quad \text{for every transition } p \xrightarrow{v/w} q$$

and all accepting states have type zero. This concludes the second argument for proving the lemma. □

It remains to check the prefix-preserving property. Before doing this, we prove a characterisation of the length-preserving functions.

Lemma B.4.5. *If a rational function is length-preserving, then it is computed by an NFA with output where for each transition, the input and output strings have the same length.*

Proof. In this proof, it will be convenient to use possibly negative strings, which can be seen as living in the free group. Formally speaking, the free group over generators B, which is denoted by $B^{(*)}$, consists of strings over the alphabet

$$\{\, b, b^{-1} \mid \ b \in B \,\}$$

which are reduced, i.e. they do not contain two consecutive letters that cancel each other out. The multiplication operation in the free group is defined by concatenation followed by reduction. The free group will also be used later in this chapter, in the proof of Theorem B.4.8.

Consider a rational function f that is length preserving and computed by some NFA with output. By Claim B.4.4, there must be some typing $\tau : Q \to \mathbb{Z}$ which maps all accepting states to zero. Define a new NFA with output as follows. The states are pairs (q, x) where x belongs to the free group $B^{(*)}$ and has length $\tau(q)$. This allows for negative lengths, since we assume that the inverse letters have length -1. The transitions of the automaton are defined by

$$\underbrace{(q,x) \xrightarrow{w/v} (p,y)}_{\substack{\text{is a transition in the} \\ \text{new automaton}}} \quad \text{if and only if} \quad \underbrace{q \xrightarrow{w/x^{-1}vy} p}_{\substack{\text{is a transition in the} \\ \text{original automaton}}}. \tag{4}$$

In the above, w and v are required to be normal strings, without negative letters, and also $x^{-1}vy$ must be a normal string, since it appears in a transition of the original automaton. Furthermore, the string x must be either purely positive, or purely negative. These transitions in the new automaton are length-preserving, as required by the conclusion of the lemma, because

$$|w| = |x^{-1}vy| + \tau(q) - \tau(p) = |v|.$$

The initial states are those which use an initial state of the original automaton, and likewise for the accepting states. Observe that both initial and final states are annotated by empty strings, since these states have type zero: initial states have type zero by definition and final states have type zero by assumption that the automaton is length-preserving. The condition (4) that defines transitions in the automaton also extends to runs, and thus the new automaton computes the same function as the original automaton. □

We now complete the proof of the theorem. Consider some NFA which computes a rational function. We first check if it is length-preserving, using Lemma B.4.3, and if it is, then we use Lemma B.4.5 to compute a new NFA in which all transitions are length-preserving. (This can be done because the proof of Lemma B.4.5 is constructive.) By possibly splitting transitions, we can assume without loss of generality that every transition consumes zero or one letters (both on input and output). As usual, we also assume that all states are productive, i.e. every state appears in some accepting run. The function computed by the automaton is *not* prefix-preserving if and only if one can find a pair of transitions

$$p_1 \xrightarrow{v_1/w_1} q_1 \qquad p_2 \xrightarrow{v_2/w_2} q_2$$

such that the following three conditions are satisfied:

1. the inputs v_1 and v_2 are the same; and
2. the outputs w_1 and w_2 are different; and
3. p_1 and p_2 can be reached from initial states using a common input string.

Since each transition consumes at most one letter, it follows from conditions 1 and 2 that the inputs and outputs of the two transitions use exactly one letter. To check if a pair of transitions with the above conditions exists, one can enumerate all pairs of transitions, and use a product construction to check if the two states p_1 and p_2 can be reached from initial states using a common input string. □

B.4.2 Sequential functions

Our ultimate goal in this chapter is to characterise the rational functions. Before we do that, however, we will characterise an intermediate model that lies between Mealy machines and rational functions, which is called the *sequential functions*. These are defined like Mealy machines, except that the transition function has type

$$Q \times A \to Q \times B^*,$$

which means that a transition can produce an output string of variable length, instead of exactly one letter.

Theorem B.4.6 (Ginsburg and Rose)**.** *A function $f : A^* \to B^*$ is sequential if and only if it: (a) is continuous; (b) is prefix preserving; (c) outputs ε when the input is ε; and (d) has the following bounded increase property:*

$$\overbrace{\sup_{\substack{w \in A^* \\ a \in A}} |f(wa)| - |f(w)|}^{\substack{\textit{maximal increase in output length due} \\ \textit{to extending input with one letter}}} \quad < \quad \infty.$$

Proof. Again, it is clear that sequential functions have the properties in the theorem. Let us prove the converse. Consider a function f that satisfies the four properties in the theorem. Define the *derivative* (this is not the same as in the proof of Theorem A.2.8) of this function to be

$$\begin{aligned} A^+ &\xrightarrow{f'} B^* \\ wa &\mapsto (f(w))^{-1} \cdot f(wa). \end{aligned}$$

This function tells us what is the contribution of the last letter to the output. The derivative is well-defined thanks to the prefix-preserving property, and it has finite image thanks to the bounded increase property. The output string of f can be computed by concatenating outputs of the derivatives

$$f(a_1 \cdots a_n) = f'(a_1) \cdot f'(a_1 a_2) \cdots f'(a_1 \cdots a_n).$$

Therefore, in order to show that f is a sequential function, it suffices to show that the derivative f' is computed by a finite automaton, i.e. for each of the finitely many output values, the corresponding set of inputs is a regular language. We do this in two steps.

- We first show that a finite automaton can determine the length of $f'(w)$. To do this, use the continuity assumption to compute the length

$$|f(w)| \mod k,$$

  where k is some number that is bigger than the maximal length of an output of a derivative. This gives the length of the derivative, since the length of $f'(wa)$ is the unique number $\ell < k$ such that

$$|f(w)| + \ell \equiv |f(wa)| \mod k.$$

- Once we know the length of the derivative $f'(wa)$, we determine its output in the same way as for Mealy machines, i.e. we can use continuity to ask what are the last $|f'(wa)|$ letters of the output $f(wa)$.

□

B.4.3 Subsequential functions

In this section, we extend the characterisation of sequential functions by considering a model that adds two new features: (a) the function is partial, i.e. it might have undefined outputs; and (b) the function knows when the input string ends. The model is called the *subsequential functions*, and the corresponding transducer is defined like a sequential transducer, except that its syntax is extended with an *end-of-input function*, which is a partial function of type $Q \to B^*$. The end-of-input function is applied to the last state in the computation. If the result is undefined, then the entire output string is undefined, and otherwise the result is appended to the previously produced output string.

Example 15. [Append b] One of the advantages of the end-of-input function is that it allows producing nonempty output on an empty input. For example, the function $a^n \mapsto a^n b$ is subsequential. We use a single state, in which the input is copied, and then the end-of-input function adds one more letter b. □

The subsequential functions seem to be a minor variation on the sequential ones, and yet their machine independent characterisation will be considerably harder than the one for sequential functions. Nevertheless, this effort will pay off in the next section, where subsequential functions will be used as a crucial ingredient in the characterisation of rational functions.

One of the difficulties in subsequential functions is that the condition on preserving prefixes is no longer relevant. This condition will no longer be necessary: because of the end-of-input function, the functions will no longer be prefix-preserving, such as the function in Example 15. It will also not be sufficient, since any function can be turned into a partial function that is prefix preserving.

Example 16. [Endmarkers] Take any string-to-string function f. We can extend the input alphabet with a distinguished endmarker $\dashv$, and define a new function by

$$f_{\dashv}(w) \quad \stackrel{\text{def}}{=} \quad \begin{cases} f(v) & \text{if } w = v \dashv \text{ for some } v \text{ that does not use } \dashv \\ \text{undefined} & \text{otherwise.} \end{cases}$$

In the new function, the domain contains only words that end with the endmarker $\dashv$ and do not use it anywhere else. Such words are incomparable with respect to the prefix relation, and therefore the new function is prefix-preserving. This way, we can turn any continuous function into one that is partial and prefix-preserving, and as we have seen in the introduction, there are plenty of ill-behaved continuous functions. Also, this example shows that the end-of-input function is somehow less important than the partiality feature, since f is subsequential if and only if $f_{\dashv}$ is subsequential without using the end-of-input function. Nevertheless, the end-of-input function will make the characterisation of subsequential functions more elegant, and so we keep it in the definition. $\square$

Because of the difficulty highlighted in the above example, we need a variant of the condition on preserving prefixes. This variant will be based on modifying strings by minor variations on the suffix (such as removing an endmarker), and is formalised in the following definition.

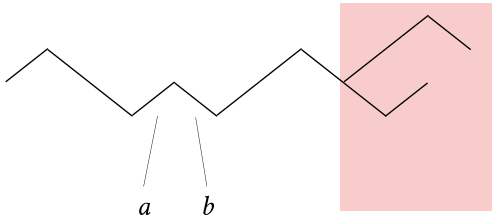
Definition B.4.7 (Left distance). *For two strings w_1 and w_2, define their* left distance

$$\|w_1, w_2\|$$

to be the smallest k such that the strings can be decomposed as

$$w_1 = vv_1 \quad w_2 = vv_2 \quad \textit{with } |v_1|, |v_2| \leq k$$

In the decomposition from the above definition, the string v will necessarily be the longest common prefix of w_1 and w_2, as explained in the following picture:



directions indicate letters left distance $k = 3$

The characterisation of subsequential functions will say that they are exactly the functions which are: (a) continuous; and (b) have a property called bounded variation, which is defined in terms of the left distance. The continuity condition is the same as in Definition .0.1, except that we use inverse images for partial functions in the natural way. The bounded variation property is actually the same as uniform continuity with respect to the left distance, but we prefer to avoid the word continuity to avoid a name clash with condition (a).

Theorem B.4.8 (Choffrut). *A partial function $f : A^* \to B^*$ is subsequential if and only if it is continuous and has the following property:*

- ***Bounded variation:*** *for every w_1 and w_2 in A^*, we have*

$$\sup_w \|f(ww_1), f(ww_2)\| < \infty,$$

where w ranges over strings such that both $f(ww_1)$ and $f(ww_2)$ are defined.

Proof. The left-to-right direction is easy to see, and we only explain the proof of the right-to-left direction. Assume that f is a continuous function with bounded variation. We will show that it can be computed by a subsequential transducer.

We begin with a simple pumping argument which shows that if a string can be extended to a string in the domain of f, then a short extension can be used.

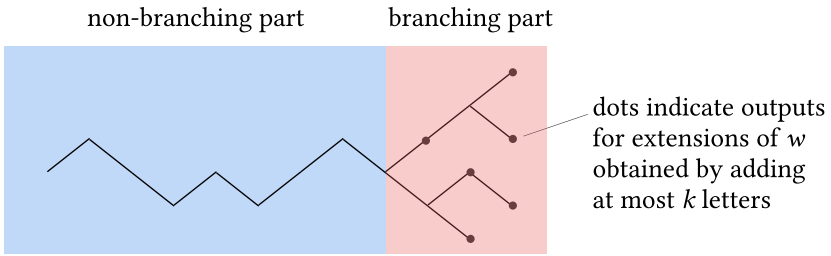
Claim B.4.9. *There is some $k \in \mathbb{N}$ such that for every $w \in A^*$, if there is some $v \in A^*$ such that wv is in the domain of f, then one can choose v so that it has length at most k.*

Proof. The domain is a regular language by continuity, and hence a pumping argument can be applied to make v short. □

Fix the constant k from the above claim for the rest of this proof. Consider a string $w \in A^*$. We will be interested in the outputs of f on input strings that are obtained from w by extending it with at most k letters. (By definition of k, such short extensions are enough to enter the domain of the function, if this is possible at all.) Since these extensions are at bounded left distance from each other, it follows from the bounded variation property that the outputs will also be at bounded left distance from each other. Summing up, there is some $K \in \mathbb{N}$ such that for every $w \in A^*$, all strings in the set

$$\{\ f(wv) \mid\ v \in A^* \text{ has length at most } k\ \} \tag{5}$$

are at left distance at most K from each other. This means that the set of strings in (5) can be visualised as a tree (where ancestors are prefixes), in which there is an initial non-branching part (of unbounded length), followed by some branching part of length at most K, as shown in the following picture:



Define $\alpha(w)$ to be the non-branching part for the input string w, which is formally speaking the longest common prefix of all strings in the set (5). The non-branching part might be undefined, but thanks to Claim B.4.9, it is defined if and only if w has some extension that is in the domain of f. In particular, if we think of the non-branching part as a string-to-string function

$$\alpha : A^* \to B^*,$$

then it is a partial function with a prefix-closed domain. Ultimately, we will establish that this function is subsequential.

Let us now discuss the branching part, which we denote by $\beta(w)$. Formally speaking, the branching part is the partial function

$$\begin{aligned} A^{\leq k} &\xrightarrow{\beta(w)} B^* \\ v &\mapsto f(wv) \text{ with the prefix } \alpha(w) \text{ removed.} \end{aligned}$$

From the branching and non-branching parts, we can recover the output of f:

$$f(w) = \alpha(w) \cdot \underbrace{\beta(w)(\varepsilon)}_{\substack{\text{branching part evaluated} \\ \text{at the empty string}}} \tag{6}$$

The above is undefined if either of the two concatenated strings on the right-hand side of the equality is undefined, and it is defined otherwise. We will show that the branching part $\beta(w)$ has finitely many possible values and the right value can be selected by a finite automaton. Therefore, the function f will be subsequential, because the contribution of the non-branching part $\alpha(w)$ is subsequential, and the contribution of the branching part $\beta(w)$ can be computed by using the end-of-input feature.

Computing the branching part. We begin by discussing the branching part, and only later we will discuss the non-branching part. The following claim shows that the contribution of the branching part can be computed using the end-of-input feature in a subsequential transducer.

Claim B.4.10. *The set of possible values of the branching part*

$$\{ \beta(w) \mid \ w \in A^* \}$$

is finite. Furthermore, for every β, the set of $w \in A^*$ such that $\beta(w) = \beta$ is regular.

Proof. The first part of the claim, about finitely many possible values, is a direct consequence of the fact that the branching part has length at most K, and thus there are finitely many possibilities for the corresponding tree. We now show the second part of the claim, about regularity. For every $v \in A^*$ of length at most k and every $u \in B^*$ of length at most K, the language

$$\{ w \in A^* \mid \ u \text{ is a suffix of } f(wv) \}$$

is regular by continuity. Therefore, a finite automaton which reads w can identify the last K letters of $f(wv)$ for all v of length at most k. This almost gives us the branching part $\beta(w)$, except that we do not know how the strings $f(wv)$ align with each other. To do this, we do a similar modulo-counting trick as in the proof of Theorem B.4.6: we determine the remainders

$$|f(wv)| \mod K+1$$

for all v of length at most k. This gives us the relative alignment of the outputs. Indeed, if v_1 and v_2 have length at most k then the difference

$$|f(wv_1)| - |f(wv_2)|$$

lies between $-K$ and K, and therefore this difference is uniquely determined by its value modulo $2K+1$, which is known to the automaton. With these differences, we know how the strings $f(wv)$ align with each other, and hence we can determine the branching part $\beta(w)$. □

Computing the non-branching part. We now show that the non-branching part $\alpha(w)$ can be computed by a subsequential transducer. For a nonempty input string $wa \in A^*$, let us define the offset as the difference

$$\delta(wa) \quad \stackrel{\text{def}}{=} \quad (\alpha(w))^{-1} \cdot \alpha(wa).$$

There is an important caveat in the definition of the offset: we do not necessarily know if $\alpha(w)$ is a prefix of $\alpha(wa)$. Therefore, the computation of the prefix might result in some negative letters, as in the following example:

$$\underbrace{(aa)^{-1}}_{\delta(wa)} = \underbrace{(abbaa)}_{\alpha(w)}^{-1} \cdot \underbrace{abb}_{\alpha(wa)}.$$

Formally speaking, the offset belongs to the free group that was defined in the proof of Lemma B.4.5. Using the offsets in the free group, we can compute the non-branching part as

$$\alpha(a_1 \cdots a_n) = \alpha(\varepsilon) \cdot \delta(a_1) \cdot \delta(a_1 a_2) \cdots \delta(a_1 \cdots a_n). \tag{7}$$

Formally speaking, the above equality is in the free group, i.e. we must first reduce the right-hand side before we can compare it to the left-hand side. We will worry about the reductions later, in Claim B.4.12. For now, let us show that the offsets can be computed by a finite automaton.

Claim B.4.11. *The set of possible values of the offset*

$$\{\, \delta(w) \mid \ w \in A^+ \,\}$$

is finite. Furthermore, for every δ, the set of $w \in A^+$ such that $\delta(w) = \delta$ is regular.

Proof. Consider a nonempty string of the form wa. To get its offset, we need to compare the branching parts $\beta(w)$ and $\beta(wa)$, and how they are aligned. All this information can be computed by a finite automaton, using the same proof as in Claim B.4.10. □

As explained in (7), the non-branching part $\alpha(w)$ is obtained by concatenating the offsets and then reducing them in the free group. We know that the ultimate result, after reduction, does not use any negative letters, since it is equal to the non-branching part $\alpha(w)$. The following claim shows that the negative letters can be eliminated from the transducer.

Claim B.4.12. *Let g be a partial sequential function whose output alphabet is*

$$\{\ b, b^{-1} \mid\ \ b \in B\ \},$$

and whose domain is prefix closed. Consider the function

$$g'(w) = \textit{reduced form of } g(w).$$

If all outputs of g' are in B^*, then it is a partial subsequential function.

Proof. The assumption that the domain is prefix-closed is true in our intended application, but it is not essential, since the argument can be easily adapted to work without this assumption. Because of the negative letters, it might be the case that extending the input results in a shorter output, i.e.

$$\{\ |g'(wv)| - |g'(w)| \mid\ \ w, v \in A^*\ \} \tag{8}$$

might contain negative numbers. The crucial observation is that the above set can only contain finitely many negative numbers. Indeed, otherwise the sequential transducer would have a loop with output of negative length, and iterating this loop would violate the assumption of the claim that all reduced outputs are positive. Therefore, there is some constant M such that for every input string w, only the last M letters of $g'(w)$ can ever be affected by cancellations in extension of the input. This means that g' can be computed by a subsequential transducer, which outputs $g'(w)$ without the last M letters, and keeps the last M letters in a buffer in the state. When the input ends, the end-of-input function flushes the buffer. □

Putting everything together. We now put the above ingredients together to obtain the desired conclusion that f is subsequential. Thanks to Claim B.4.11 and (7), we know that there is a subsequential function with outputs in the free group that computes the non-branching part $\alpha(w)$. Thanks to Claim B.4.12, we can improve this function so that it does not use negative letters at all, i.e. the function $\alpha(w)$ itself is subsequential. Thanks to Claim B.4.10, there is a finite automaton whose state can be used to indicate the value of the branching part. By taking the product of this automaton with the subsequential function for $\alpha(w)$, we can get a subsequential transducer for f, thanks to the equality (6). □

B.4.4 Rational functions

We finish this chapter with a machine independent characterisation of the rational functions. This characterisation is also based on bounded differences in the left distance.

Theorem B.4.13 (Reutenauer and Schützenberger)**.** *A function $f : A^* \to B^*$ is rational if and only if it is continuous, and the following equivalence relation on input strings has finite index:*

$$w_1 \sim w_2 \quad \stackrel{\text{def}}{=} \quad \sup_{w \in A^*} \|f(ww_1), f(ww_2)\| < \infty.$$

Proof. Since we have chosen to define rational functions as total functions, we assume that the function f is total. Nevertheless, the same proof would work for partial rational functions, since all the work related to undefined outputs is already done in the characterisation of subsequential functions.

The easy implication is from left-to-right. We have already established continuity for rational functions in Theorem B.1.5. To see that $\sim$ has finite index for a rational function, one can use bimachines: if w_1 and w_2 give the same state of the suffix automaton, then they will be equivalent.

The rest of the proof is devoted to the right-to-left direction. Assume then that f is continuous and $\sim$ has finitely many equivalence classes. We first observe that the relation $\sim$ in the statement of the theorem is a *left congruence*, which means that

$$w \sim w' \quad \implies \quad aw \sim aw' \qquad \text{for every } w, w' \in A^* \text{ and } a \in A.$$

This congruence property holds because when we prepend a, the range of the supremum in the definition of $\sim$ becomes smaller. Let Q be the finite set of equivalence classes of $\sim$. Thanks to the congruence property, there is an automaton structure on equivalence classes, which goes from right-to-left: if $q \in Q$ is an equivalence class and a is an input letter, then there is a well-defined equivalence class $aq \in Q$, which is obtained by taking any word in q, prepending a, and taking the equivalence class.

To finish the proof, we will show that the function f can be computed in two steps, both of which are rational functions, and is hence rational as a composition of rational functions. In principle, the second step is a partial function, but the outputs of the first step will always fall in the domain of the second step.

1. In the first step, we annotate each input letter with the equivalence class under $\sim$ of the remaining suffix. If the input is $a_1 \cdots a_n$, then the output of the first step is the string

   $$\xleftarrow{a_1} q_1 \xleftarrow{a_2} q_2 \cdots \xleftarrow{a_n} q_n \in (A \times Q)^*,$$

   where q_i is the equivalence class of the suffix $a_{i+1} \cdots a_n$. This function is rational, since it can be computed by a finite automaton which runs on the input string from right to left, and outputs the annotation as it goes.

2. In the second step, we use the string from the first step to compute the output of the original function f. Formally speaking, the second step is a partial function

$$g : (A \times Q)^* \to B^*,$$

which is defined if and only if the input is consistent (i.e. it is a possible output of the first step), and in this case it returns the output of f on the original input string. We now use Theorem B.4.8 to show that g is subsequential. Let us check that the conditions in the theorem are satisfied. Continuity of g is an easy consequence of continuity of f. For the bounded variation property, we need to show that for all strings w_1 and w_2 over the input alphabet of g,

$$\sup_w \|g(ww_1), g(ww_2)\| < \infty,$$

where w ranges over strings that make both outputs defined. Let v_1 and v_2 be the strings obtained from w_1 and w_2 by erasing the annotation with states Q. If w_1 and w_2 are inconsistent, or if they are consistent but the strings v_1 and v_2 are not equivalent under $\sim$, then the inequality above holds vacuously, since no string w can be found which makes both outputs defined. Otherwise, by definition of the function g, the supremum above is the same as

$$\sup_{v \in A^*} \|f(vv_1), f(vv_2)\|,$$

and is therefore bounded by the definition of $\sim$.

□

Example 17. [String reversal] Using the characterisation from the above theorem, we can show that the string reversal function is not rational. (We assume that the alphabet has at least two letters, since otherwise string reversal is the identity, and hence rational.) For string reversal, not only does the relation $\sim$ from the theorem fail to have finite index, but in fact all strings are pairwise non-equivalent. Indeed, consider two different strings w_1 and w_2. We want to show that

$$\sup_{w \in A^*} \|\mathrm{reverse}(ww_1), \mathrm{reverse}(ww_2)\| = \infty.$$

We can easily choose the string w so that the reversals differ in one of the first $|w_1|+1$ letters, which is a constant when w_1 and w_2 are fixed. Since the string w can be made arbitrarily long, this will give unbounded distance. □

Exercises

Exercise B.4.1. We order sequential transducers by the number of states. Show that minimal sequential transducers are unique up to isomorphism.

Solution. This is the Myhill-Nerode theorem for sequential transducers. The key point, which is also what distinguishes this exercise from the following one, is that the output of each transition is already determined by the function. Indeed, a sequential function is prefix preserving, and therefore the transition which reads the i-th letter must produce

$$f(a_1 \cdots a_{i-1})^{-1} \cdot f(a_1 \cdots a_i),$$

i.e. the contribution of the i-th letter, which is the derivative from the proof of Theorem B.4.6.

For an input string u, define its *residual* to be the function

$$f_u : A^* \to B^* \qquad f_u(v) \quad \stackrel{\text{def}}{=} \quad f(u)^{-1} f(uv),$$

which is well defined thanks to the prefix preserving property. If a sequential transducer computes f and it is in state q after reading u, then the residual f_u is determined by q, since the outputs produced after reading u depend only on q and on the letters that follow. In particular, the transducer has at least as many states as there are residuals. Also, in a minimal transducer all states are reachable, since otherwise we could remove the unreachable ones.

Consider now a minimal transducer. By the two observations above, the map which sends a state q to the residual f_u, for some u that reaches q, is well defined and surjective onto the residuals. It is also injective: if two states had the same residual, then we could merge them, since the outputs of the corresponding transitions would be equal, and the corresponding target states would again have equal residuals; this would give a transducer with fewer states. Therefore the map is a bijection, and by construction it preserves the initial state, the transitions and their outputs. In other words, every minimal transducer is isomorphic to the transducer whose states are the residuals, with the initial state $f_\varepsilon = f$ and transitions

$$f_u \xrightarrow{\ a \ / \ f_u(a)\ } f_{ua}.$$

Since this transducer depends on the function alone, all minimal transducers are isomorphic.

Exercise B.4.2. We order subsequential transducers by the number of states. Show that minimal subsequential transducers are not unique up to isomorphism.

Solution. The end-of-input function makes it possible to produce the same output either while reading the input, or at the very end, and the state structure does not resolve this ambiguity. Consider the function over a one-letter input alphabet defined by

$$\varepsilon \mapsto \varepsilon \qquad \text{and} \qquad a^n \mapsto c \quad \text{for } n \geq 1.$$

One state is not enough. Indeed, if the unique state had a transition with output x and end-of-input output y, then the output on the input string a^n would be $x^n y$, and this cannot be empty for $n = 0$ and equal to c for all $n \geq 1$.

Two states are enough, and there are two ways of using them. In both transducers, the states are the initial state q_0 and a second state q_1, the transitions are $q_0 \to q_1$ and $q_1 \to q_1$, the transition $q_1 \to q_1$ has empty output, and the end-of-input function maps q_0 to the empty string. The difference is in the remaining two values: in the first transducer, the transition $q_0 \to q_1$ outputs c and the end-of-input function maps q_1 to the empty string, while in the second transducer, this transition has empty output and the end-of-input function maps q_1 to c. Both transducers are minimal, and they are not isomorphic, since an isomorphism would have to match the initial states, and therefore also the two remaining states, but the outputs of the transitions $q_0 \to q_1$ are different.

As the example suggests, uniqueness is restored if one additionally requires the output to be produced as early as possible.

Exercise B.4.3. Consider bimachines ordered by the number of states in the suffix automaton. Show that minimal bimachines are not unique up to isomorphism.

Solution. This is already true for languages, i.e. for functions of type

$$A^* \to \{0, 1\},$$

which produce one output letter that says if the input string belongs to the language. Take the language of strings of even length, over a one-letter input alphabet.

In the first bimachine, the prefix automaton computes the parity of the prefix, and the suffix automaton tests if the suffix is empty. The output is empty in every gap except the last one, which is the one recognised by the suffix automaton, and in the last gap the output is the letter given by the parity in the prefix automaton. This machine has four states in total, two in each automaton. The second bimachine is symmetric: the prefix automaton tests if the prefix is empty, the suffix automaton computes the parity of the suffix, and the output is produced in the first gap. The two machines are not isomorphic, since already their prefix automata are not: in one of them the initial state can be reached again after reading a nonempty string, and in the other one it cannot.

It remains to explain why four states cannot be improved. Since the output is one letter for every input string, exactly one gap produces a nonempty output. Suppose that the suffix automaton has one state. Then the output in a gap depends only on the prefix, i.e. on the number of letters to the left of the gap, and therefore the gaps that produce a nonempty output form a fixed set of numbers, which does not depend on the input string. If this set has at least two elements, then a sufficiently long input string produces at least two letters of output; and if it has at most one element, then all sufficiently long input strings produce the same output. Both cases are impossible, and hence the suffix automaton has at least two states. The same argument, this time counting the letters to the right of the gap, shows that the prefix automaton has at least two states.

Exercise B.4.4. Give an example of a rational function which has two non-isomorphic unambiguous transducers of minimal size (the size is the number of states).

Solution. The parity function from Exercise B.4.3 can be used, i.e. the function which maps a^n to the one-letter string that says if n is even. We first observe that in every

transducer for this function, all cycles that appear in accepting runs have empty output: repeating such a cycle would give the unique accepting run of a longer input string, and its output would have more than one letter. Therefore the output letter is produced in a bounded part of the run, which can be at its beginning or at its end, and this is where the two transducers differ.

In the first transducer, the output is produced at the end. There are two states that track the parity of the number of letters read so far, plus a final state, which is entered by two transitions with empty input, one from each parity state, producing the corresponding output letter. In the second transducer, the output is produced at the beginning. From the initial state there are two transitions with empty input, which guess the parity of the input string and produce the corresponding output letter, and which lead to two states that check whether the guess was correct. Both transducers have three states, and both are unambiguous, since in both of them the accepting run is determined by the parity of the input string. They are not isomorphic, since in the first one the transitions with empty input enter the same state, and in the second one they leave the same state.

Three states are necessary. The input string a^0 needs an accepting run which uses transitions with empty input only, and which produces the letter for even parity; this run cannot use a cycle, since a cycle with empty input would give infinitely many accepting runs. On the other hand, the input string a^1 needs an accepting run which produces the letter for odd parity, and hence it cannot use any of the transitions in the run for a^0. A short case analysis on the two remaining states, which have to track the parity, shows that these two runs cannot be accommodated.

B.5 References

Rational functions and relations. To the best of my knowledge, the rational relations first appeared in [RS59, Definition 15] under the name of "two-tape one-way automata". The deterministic case of these automata does not correspond to the rational functions, since it describes a model where a deterministic automaton has access to both the input and output strings. To the best of my knowledge, the functional fragment first appears in [Sch61a, Section II] under the name of "transductions", where the corresponding model is the same as what we call a bimachine, see Definition B.2.2. Another early reference is [EM65], which uses introduces the approach that is followed by this book: we first introduce rational relations, and then isolate the functions as a special case. In [EM65], the relations and functions are called "recognised by generalised finite automata".

Rational and recognisable subsets of a monoid. None of the references discuss above use the name "rational", so we should explain why we use this name. It stems from a more general terminology, about rational and recognisable subsets of a monoid, which is popular in the French algebraic school. Let us explain this terminology.

In the automata literature, the name rational seems to appear for the first time in Schützenberger's paper about (what is now known as) weighted automata [Sch61b, Section III.A], where it is applied to describe (what was previously known as) regular expressions on (non-commutative) power series. The name rational is particularly well motivated in the setting of power series, which is easiest to see for the monoid $\mathbb{F}(x)$, which consists of fractions of polynomials over a field $\mathbb{F}$ that have one variable x, modulo common factors. This monoid is the least subset of all series (i.e. functions from monomials x^n to field coefficients) which contains all polynomials, is closed under addition and multiplication, and the following version of Kleene star:

$$1 + x + x^2 + \cdots + x^n + \cdots = \frac{1}{1-x}.$$

In a tradition dating back to at least Euler, fractions of polynomials as in $\mathbb{F}(x)$ are called rational functions, which explains why this submonoid of the monoid of all series is called the rational series, and more generally why regular expressions are called rational. This is the terminology typically used in the French algebraic tradition. One early example which uses this name for a monoid which is not the free monoid is [ES69] which talks about rational subsets of a commutative monoid – for example in the commutative monoid $\mathbb{N}^d$ with component-wise additions, the rational subsets are exactly the semilinear sets.

Let us now move to the recognisable subsets of a monoid, which are those that are recognised by a homomorphism into a finite monoid (this is the same as the congruence description from Definition B.1.3). The idea to use monoid to recognise languages goes back to Schützenberger, who defined the syntactic monoid of a language in [Sch56, p. 10]. An older reference which predates this is [Dub41, Theorem 3], which introduces the syntactic congruence of a subset of a monoid (not necessarily a free monoid, and the congruence is a right congruence, as in a DFA). Under the above terminology, the Kleene Theorem states that for the free monoid A^*, the rational

subsets are exactly the recognisable subsets; for example this is the statement used [Ber79, Chapter III, Theorem 2.1]. A more detailed discussion of this topic, with ample references, can be found in [Sak21].

Let us now return to the use of the name rational for string-to-string functions and relations. As we have observed in the beginning of this section, none of the early papers use the name rational for string-to-string functions and relations; this includes Schützenberger himself. As far as I can tell, this terminology starts to appear around the time of Eilenberg's book [Eil74, Chapter VII], maybe it is due to Eilenberg; Schützenberger seems to say so [RS91, Introduction]. Also Eilenberg introduced the name bimachine, and proved that it is equivalent to the rational functions, see [Eil74, Chapter XI, Theorem 7.1].

The equivalence problem. Undecidability of the equivalence problem for rational relations was first proved in [Gri68, p.410], although a related result on the undecidability of emptiness for intersection of rational relations can already be found in [RS59, Theorem 18]. The latter result is our Exercise B.1.4.

The equivalence problem for rational functions was shown decidable in [BH77, Theorem 1]. This is deduced from a more general result, namely functionality for rational relations (two rational functions are equal if and only if their union is a functional rational relation). The functionality problem for relations (and hence also the equivalence problem for functions) is solved in [BH77] using a a pumping arugment. According to the editors comments in Schützenberger's collected works [Sch09, p.4], the functionality problem for relations is also implicitly solved in [Sch76].

None of these decidability arguments above use a reduction to weighted automata, as we do here. This kind of reduction traces back to [AL85, Theorem 1] which proves a conjecture of Ehrenfeucht about string-to-string homomorphisms by embedding strings into the metabelian group and then applying the Hilbert Basis Theorem. The embedding itself is credited to Mal'cev [Mal53]. Of relevance to this book is also [AL85, Theorem 1], which uses the same approach to decide equivalence for HD0L equivalences; the latter equivalence is the same as copyful SST's from Exercise C.3.2 and it is the most powerful decidability result about equivalence that appears in this booi. A later result that is closer to our book because it uses weighted automata is [HK91]. It is still not exactly the same as our book, because (a) the source of the reduction is different (instead of equivalence for rational string-to-string functions, they studied equivalence for (tuple-of-string)-to-Boolean functions; and (b) the target of the reduction is different (instead of weighted automata over a field, they used weighted automata over a division ring). Also, since [HK91] uses weighted automata over division ring, it needs to extend Schützenberger's equivalence argument from a field to a division ring. The simple reduction from this book, which uses rational numbers to encode strings, appears in [SMK18, p. 21:4] and is also discussed in [Boj19, p. 6].

Machine independent characterisations. As remarked in the beginning of this book, the characterisation of Mealy machines in Theorem B.4.1 is in fact essentially the same as the original formulation of the Myhill-Nerode Theorem [Ner58, Lemma].

The characterisation of sequential functions in Theorem B.4.6 is from [GR66, p. 381]. The characterisation of subsequential functions in Theorem B.4.8 is from [Cho77, Proposition 3.4]. Finally, the characterisation of rational functions from Section B.4 is from [RS91, Theorem 1 on p. 674].

Part C: Regular functions

In this part, we move to the third step in the transducer ladder, namely the regular functions. Probably the quickest way to define the regular functions is in terms of prime functions, so we begin with that.

Example 18. [Map reverse and map duplicate] The map reverse function is obtained by applying the map lifting operation from Definition A.2.3 to string reversal. Here is an example of its application:

$$123\#45\#6789 \quad \mapsto 321\#54\#9876.$$

Similarly, we define the map duplicate function, by lifting string duplication $w \mapsto ww$. Here is an example of its application:

$$123\#45\#6789 \quad \mapsto 123123\#4545\#67896789.$$

Formally speaking these are not two functions, but a family of functions, one for each alphabet (the input and output alphabets are the same in this case). Using Theorem B.4.13, one can show that none of these functions is rational. □

In Example 17 we have shown that string reversal is not rational. It follows that map reverse is not rational either, since it can be used to implement string reversal by choosing an input string without separators. A similar argument can be made to show that neither duplicate nor map duplicate are rational. Therefore, we get a new step in the transducer ladder by adding these two functions, as we do in the following definition.

Definition C.0.1 (Regular functions)**.** *A string-to-string function is called* regular *if it can be obtained as a finite composition of functions, each of which is either:*

1. *a rational function;*
2. *one of the map reverse or map duplicate functions from Example 18.*

One could, of course, add just one of the two functions, or maybe even just duplicate and reverse without map. As we will see in this part, there is a good reason to consider the extension from the above definition. This is because the resulting

class has an abundance of equivalent models, including two-way automata with output, streaming string transducers, a transducer variant of monadic second-order logic, and a functional programming language based on combinators. These models will be introduced in the next few chapters.

Exercises

Exercise C.0.1. Show that regular functions are compatible with compression, as defined in Exercise B.2.18.

Solution. Functions compatible with compression are clearly closed under composition, and contain all rational functions by Exercise B.2.18. It remains to show that both map reverse and map duplicate are compatible with compression. It is easy to see that both reverse and duplicate (without map) are compatible with compression, and thus it remains to show the following claim.

Claim C.0.2. *Functions compatible with compression are closed under map lifting.*

Proof. The idea is that we can always improve a compression in polynomial time so that it is consistent with the separators in the following sense. Apart from the starting nonterminal, there are two kinds of nonterminals, called external and internal. Internal nonterminals generate strings without separators. For an external nonterminal, the rules must be of the form

$$X \to \underbrace{YZ}_{\text{both are external}} \qquad \text{or} \qquad X \to \#\underbrace{Y}_{\text{internal}}$$

The initial nonterminal has a rule of the form $S \to XY$ where X is internal and Y is external. The idea is that the internal nonterminals generate the blocks, and the external nonterminals generate sequences of blocks.

One can show without much difficulty that any grammar compression can be transformed into this form in polynomial time. The transformation is done by splitting the rules of the grammar at the separators, and introducing new nonterminals for the parts that are split off. After transforming into the form, we can apply the assumption on the original function to the blocks, which are the nonterminals Y in the rules $X \to \#Y$. □

C.1 The prime regular functions

So far, we have defined the regular functions in terms of compositions of certain prime functions, namely the rational functions, map reverse and map duplicate. Before introducing new models for this class, in this chapter we establish the basic properties of the regular functions as defined.

Continuity and composition. Almost immediately from the definition, we recover two of our favourite properties for transducer classes.

Theorem C.1.1. *Regular functions are continuous and closed under composition.*

Proof. Composition is built into the definition, so we only argue about continuity. Since continuous functions are closed under composition, and rational functions are continuous by Theorem B.1.5, it remains to show that map reverse and map duplicate are continuous. We first show that reverse and duplicate (without map) are continuous, and then we extend this with the map combinator.

Lemma C.1.2. *String reversal and string duplication are continuous.*

Proof. This lemma is a classical exercise in automata theory [9]. We need to show that for every regular language L, the inverse images

$$\{ w \in A^* \mid \text{reverse}(w) \in L \} \qquad \{ w \in A^* \mid ww \in L \}$$

are regular. We assume that L is given by a nondeterministic automaton. For string reversal, we modify the original automaton by reversing all transitions and swapping initial and accepting states. For string duplication, the inverse image is

$$\bigcup_q \{ w \in A^* \mid \begin{array}{l} \text{there is a run over } w \text{ from an initial state to } q \text{ and} \\ \text{there is a run over } w \text{ from } q \text{ to an accepting state} \end{array} \},$$

which is easily seen to be a regular language. □

It remains to lift the above lemma via map. We will show that the map lifting of every continuous function, not just reverse and duplicate, is continuous.

Lemma C.1.3. *If a string-to-string function is continuous, then the same is true for its map lifting.*

Proof. Consider the map lifting of some continuous function f. To prove the lemma, we need to show how an automaton $\mathcal{B}$ over the output alphabet of the map lifting is pulled back to an automaton $\mathcal{A}$ over the input alphabet. Given an input string,

$$w_1 \# \cdots \# w_n$$

the automaton $\mathcal{A}$ nondeterministically guesses states $p_1, q_1, \ldots, p_n, q_n$ of the automaton $\mathcal{B}$ and checks that the following conditions are all satisfied:

[9] Filip Murlak, Damian Niwiński, and Wojciech Rytter, *200 problems on languages, automata, and computation*, 2023, Problems 30 and 35

1. the state p_1 is initial and the state q_n is accepting;
2. for every $i \leq n$, the automaton $\mathcal{B}$ can go from p_i to q_i when reading $f(w_i)$;
3. for every $i < n$, the automaton $\mathcal{B}$ can go from q_i to p_{i+1} when reading $\#$.

The first and third conditions can easily be checked by an automaton. The second condition can also be checked by an automaton thanks to continuity of f, since it corresponds to an inverse image of a regular language. □

□

Equivalence. We can also show that equivalence is decidable for regular functions.

Theorem C.1.4. *Equivalence is decidable for regular functions.*

Proof. We use the same approach as in the equivalence algorithm for rational functions in Theorem B.3.4, which is a reduction to equivalence for weighted automata. As in the case of rational functions, it is enough to show that if we post-compose a regular function f with a weighted automaton g over the field of rationals

$$A^* \xrightarrow{f} B^* \xrightarrow{g} \mathbb{Q},$$

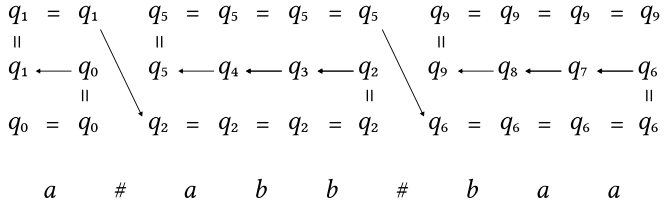
then the resulting function is also a weighted automaton, and can be computed. By choosing the weighted automaton g to be injective, we can faithfully represent the regular function f as a weighted automaton, thus reducing the equivalence problem for regular functions to the equivalence problem for weighted automata over a field.

Consider the class of functions which can be post-composed with weighted automata over a field, i.e.

$$\{\ A^* \xrightarrow{f} B^* \mid \text{ if } B^* \xrightarrow{g} \mathbb{Q} \text{ is a weighted automaton, then so is } f \cdot g.\ \}$$

We need to show that this class contains all regular functions. Since this class is easily seen to be closed under composition, and it contains all rational functions by Theorem B.3.6, it is enough to show that it contains map reverse and map duplicate.

- **Map reverse.** On each block between two separator symbols, the automaton for the post-composition $f \cdot g$ runs in reverse, and remembers the states at the beginning and end of the block. This is implemented using triples of states, as in the following picture:



(As we will see in the next chapter, this construction corresponds to a two-way automaton that computes the map reverse function.) Assuming that the original weighted automaton g had states Q, the new weighted automaton (for the post-composition $f \cdot g$) has states Q^3. We assume that the original automaton consumes at most one input letter in each transition, but ε-transitions are still allowed. The transitions are defined by

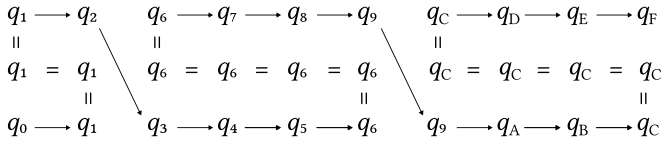
$$\underbrace{(r,q,s) \xrightarrow{a/x} (r,p,s)}_{\text{in the new automaton}} \qquad \Longleftrightarrow \qquad \underbrace{p \xrightarrow{a/x} q}_{\text{in the original automaton}} .$$

for a which is not the separator symbol $\#$, and by

$$\underbrace{(r,q,q) \xrightarrow{\#/x} (p,p,s)}_{\text{in the new automaton}} \qquad \Longleftrightarrow \qquad \underbrace{r \xrightarrow{\#/x} s}_{\text{in the original automaton}} .$$

The initial states are those which have an initial state on the first coordinate, and the accepting states are those which have an accepting state on the third coordinate. Note that this weighted automaton will multiply the weights of the original run in a different order, but this is not an issue since multiplication in a field is commutative. More generally, this construction would work for any commutative semiring. For non-commutative semirings, the construction cannot work, since we have established in Theorem B.3.6 that only rational functions can be closed under post-composition with weighted automata over arbitrary semirings, and map reverse is not rational.

- **Map duplicate.** The construction is similar to the one for map reverse, and illustrated in the following picture:



The details of the construction are left to the reader. Note that for a transition in the new automaton, its weight will be obtained by multiplying the weights of two transitions in the original automaton.

□

In the above proof, we showed that regular functions can be post-composed with weighted automata over the field of rationals, and the proof also extends to any commutative semiring. We conjecture that the converse also holds, thus giving a characterisation in the same spirit as Theorem B.3.6.

Conjecture C.1.5. *A string-to-string function $f : A^* \to B^*$ is regular if and only if it is closed under post-composition with weighted automata over all commutative semirings $\mathbb{S}$, i.e.*

$$B^* \xrightarrow{g} \mathbb{S} \text{ is a weighted automaton} \qquad \Longrightarrow \qquad A^* \xrightarrow{f \cdot g} \mathbb{S} \text{ is a weighted automaton.}$$

Note that continuity, in the sense of Definition .0.1, is a special case of the post-composition closure in the above conjecture, when one considers weighted automata over the Boolean semiring. It is also possible that the conjecture holds already for a fixed semiring, such as the field of rationals.

Exercises

Exercise C.1.1. In Definition C.0.1, we use map reverse and map duplicate over arbitrary alphabets of the form $A + 1$. Show that the definition remains the same if we only consider A with two letters.

Solution. One inclusion is immediate, since a two-letter alphabet is a special case. For the converse inclusion, we show how map reverse and map duplicate for an arbitrary alphabet A can be simulated using their two-letter counterparts, together with rational functions.

Fix an injective encoding of the letters of A as bit strings of a common length

$$\text{code} : A \to \{0,1\}^k,$$

and extend it to a homomorphism on $(A + 1)^*$, by mapping the separator to itself. This homomorphism is rational. Decoding is rational as well: a transducer reads the input in blocks of k letters, restarting at each separator, and outputs one letter of A per complete block. (A rational function must be total, so we also need to say what the decoder does on badly formatted inputs; this is irrelevant, so we can say that an incomplete last block is ignored.)

For map duplicate, we have the decomposition

$$\text{map duplicate over } A \quad = \quad \text{code} \cdot \text{map duplicate over } \{0,1\} \cdot \text{decode},$$

because duplicating the encoding of a string is the same as encoding its duplication.

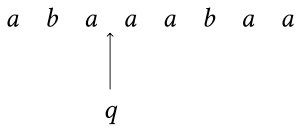
For map reverse, the same decomposition almost works, the problem being that reversing an encoded string reverses not only the order of the blocks, but also the bits inside each block. The second effect is undone before it happens: instead of the homomorphism above, we use the homomorphism which maps each letter to the reverse of its code. With this modification, reversing the encoded string yields the encoding of the reversed string, and the decomposition works as for map duplicate.

Exercise C.1.2. Show that weighted automata over arbitrary semirings are not closed under pre-composition with regular functions.

Solution. By Theorem B.3.6, a function is rational if and only if weighted automata are closed under pre-composition with it. Therefore, it is enough to show that there exists a regular function which is not rational. The string reverse function is regular, but not rational as we have shown in Example 17.

C.2 Two-way transducers

In this chapter, we define a transducer model for the regular functions, which is based on two-way automata, in which the head of the automaton can move in both directions. A configuration of the transducer consists of an input string, a state, and a head position, as in the following picture:



Note that the head points to a gap between positions, instead of a position. Usually, one defines two-way transducers (or Turing machines, which are a more powerful cousin) so that the head points to a position instead of a gap, but we choose the gap view so that it is more consistent with one-way automata. A configuration can be formally seen as an element of

$$\underbrace{A^*}_{\substack{\text{input}\\ \text{before}\\ \text{the head}}} \times \underbrace{Q}_{\text{state}} \times \underbrace{A^*}_{\substack{\text{input}\\ \text{after}\\ \text{the head}}}.$$

The transducer starts in an initial configuration, where the state is the chosen initial state, and the head is to the left of the entire input string. Next, in each step, the transducer looks at: (a) the letter to the left of the head, (b) the current state, and (c) the letter to the right of the head. Based on this information, it can either produce an output string and halt, or it can produce an output string, change its state, and move the head left or right. Note that the letters in (a) and (c) might be undefined, since the head can be at the leftmost or rightmost position, and hence they can be seen as elements of $A + 1$, where 1 is a special symbol that stands for the undefined letter. The formal definition is given below.

Definition C.2.1. *A two-way transducer consists of the following data:*

- *a finite input alphabet A;*
- *a finite output alphabet B;*
- *a finite set of states Q, with an initial state $q_0 \in Q$;*
- *a transition function*

$$\underbrace{(A+1)}_{\substack{\textit{letter}\\ \textit{left of}\\ \textit{head}}} \times \underbrace{Q}_{\textit{state}} \times \underbrace{(A+1)}_{\substack{\textit{letter}\\ \textit{right of}\\ \textit{head}}} \xrightarrow{\delta} \underbrace{B^*}_{\substack{\textit{produce an}\\ \textit{output string}\\ \textit{and halt}}} \overbrace{+}^{\textit{or}} \underbrace{(Q \times B^* \times \{-1,1\}).}_{\substack{\textit{if the transducer does not halt,}\\ \textit{then it chooses a new state,}\\ \textit{produces some output, and}\\ \textit{moves the head left or right.}}}$$

We now define the semantics of the transducer in more detail. The *configuration graph* is a directed edge labelled graph where vertices are

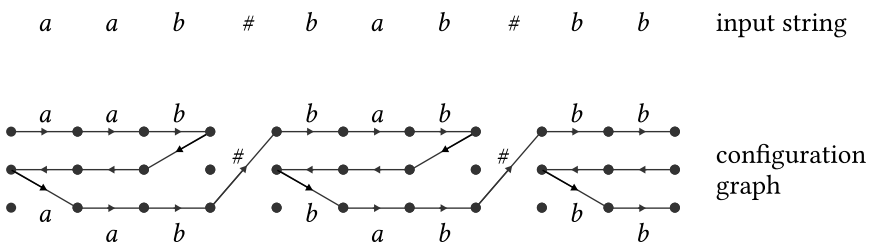
$$V = \underbrace{(A^* \times Q \times A^*)}_{\text{configurations}} + \underbrace{1}_{\text{halt}}$$

and the edges represent one step of computation, with

$$c \xrightarrow{w} d$$

saying that the transition applied in configuration c leads to d (which is defined in the natural way) and produces output w. By determinism, every vertex has at most one outgoing edge. (There might be no outgoing edge in case the transducer chooses to fall off the input, by moving left in the leftmost position or right in the rightmost position.) The output for an input string $w \in A^*$ is obtained by taking the corresponding initial configuration (ε, q_0, w), and reading the labels on the path from this configuration to the halting vertex. In particular, we require that this path exists, i.e. the transducer does not loop or fall off the tape.

Example 19. Consider the map duplicate function from Example 18. This function is computed by a two-way transducer that has three states, and whose configuration graph (restricted to the input string) is shown in the following picture:



(The reader might recognise this picture from the proof of Theorem C.1.4.) A very similar transducer works for the map reverse function, except that the output is produced during the movement in the left direction. □

C.2.1 Continuity

This section shows that two-way transducers are continuous, i.e. inverse images of regular languages are regular. Ultimately, we will show that two-way transducers are the same as the regular functions, and hence they must be continuous by Theorem C.1.1. However, the direct proof of continuity is worth the effort, since it will introduce some useful techniques.

Theorem C.2.2 (Rabin-Scott, Shepherdson)**.** *Every function computed by a two-way transducer is continuous.*

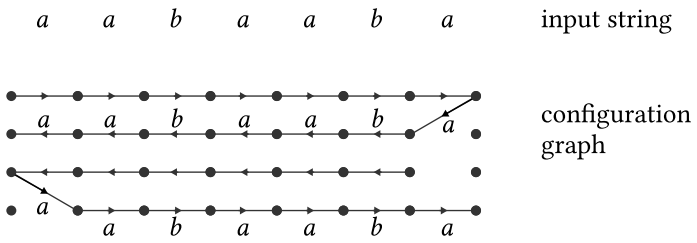
Proof. We want to show that if f is a two-way transducer and L is a regular language, then their composition (which corresponds to an inverse image)

$$A^* \xrightarrow{f} B^* \xrightarrow{L} \{\text{yes, no}\} \tag{9}$$

is a regular language. If we take a DFA that recognises the language L, then we can apply a straightforward product construction, yielding a two-way automaton (not transducer) which recognises the inverse image. Therefore, this theorem boils down to showing that two-way automata can only recognise regular languages[10].

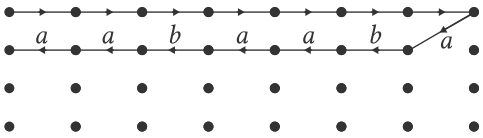
Later on in this section, we will want to prove that two-way transducers are closed under composition, which is the extension of (9) in which the language L is replaced by another two-way transducer. We try to phrase the present proof in a way which is amenable to that extension. The proof has two steps: (a) we first use a rational function to compute the configuration graph of f on the input string $w \in A^*$; and then (b) we use the configuration graph to check membership of $f(w) \in L$.

Computing the configuration graph. For an input string, we will be interested in its configuration graph, which is the configuration graph of the transducer restricted to configurations that come from the input string. We have seen a picture of such a configuration graph in Example 19. However, that picture was a bit misleading, since it only contained reachable configurations (except for isolated vertices). Here is a more illuminating picture, which shows a transducer that reverses the input string if the last letter is a, and otherwise keeps it unchanged:



A local inspection of the above configuration graph will not tell us which configurations are reachable. For this reason, we will be interested in the reachable part of the configuration graph, as in the following picture:

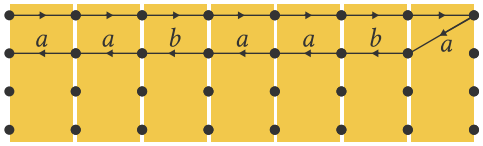
[10] Turning a two-way automaton into a one-way automaton was one of the original motivations for introducing nondeterministic automata in Michael O. Rabin and Dana Scott, "Finite automata and their decision problems", 1959. The same problem was solved independently in J. C. Shepherdson, "The Reduction of Two-Way Automata to One-Way Automata", 1959, which applied a direct construction without nondeterminism. In Note 4 of Shepherdson's paper one can also see a mention of the transducer variant, and thus this is probably the first appearance of two-way transducers in the literature.



reachable part of the configuration graph

The reachable part is a single path, which goes from the initial configuration to a halting configuration.

We will show that the reachable part of the configuration graph can be computed by a rational function. In order to do this, we need to represent it as a string over a finite alphabet. The idea is to slice the graph into smaller graphs, one for each input letter, as in the following picture:



string representation

Formally speaking, the alphabet for the string representation, call it C, consists of bipartite graphs, where the vertices are two copies of Q, edges are directed and labelled with output strings, each vertex has at most one outgoing edge, and this edge must go to the other copy of Q. This alphabet is finite, since we use only labels of edges that arise from transitions of the transducer. There is an edge case for the empty input string, in which the configuration graph looks like this:

Therefore, the alphabet C contains a special letter for this configuration graph.

Lemma C.2.3. *The function which maps an input string to the string representation of its reachable configuration graph is rational.*

Proof. The main observation is that configuration graphs form a regular language, i.e.

$$\{\, w \in C^* \mid \ w \text{ is a string representation of a reachable configuration graph } \} \quad (10)$$

is a regular language. To see this, we need to check that the letters are consistent with the transition function of the two-way transducer (which is a simple local consistency check) and that all arrows are reachable. For the latter property, about reachability of arrows, it is easier to consider the complement (since regular languages are closed under complementation). The complement can be checked by guessing a subset of arrows that is disconnected from the others.

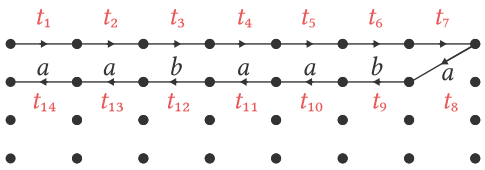
Once we know that the language (10) is regular, the lemma follows. The function in the lemma is computed by an NFA with output, which nondeterministically guesses the output, and then checks if it is correct using the language (10). There is a unique correct guess, and hence the function is rational. □

The following lemma shows that once we have computed the reachable configuration graph, we can check if the output string in this graph belongs to L.

Lemma C.2.4. *The following language is regular:*

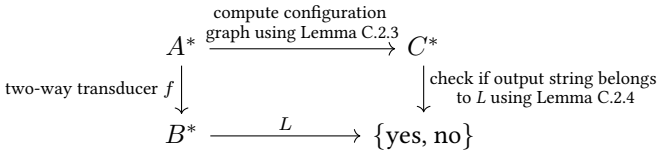
$$\{\, w \in C^* \mid \begin{array}{l} w \text{ is a string representation of a reachable configuration graph} \\ \text{and the corresponding output string belongs to } L \end{array} \,\}$$

Proof. The automaton checks that the input is indeed a reachable configuration graph (which can be done as shown in the previous lemma), and then it guesses a labelling of the edges in the graph by transitions of an automaton for L, as in the following picture (with transitions in red):



Once the transitions have been guessed, the automaton checks if they represent an accepting run, which is a purely local property. □

By putting the above two lemmas together, we complete the proof, as explained in the following diagram:



The right-down part describes a regular language by continuity of rational functions, and hence the down-right path also describes a regular language, thus proving continuity of the two-way transducer f. □

C.2.2 Closure under composition

We now prove that two-way transducers are closed under composition. This is a nontrivial result, because we cannot simply use a product construction, since the heads of the two transducers that are composed can move in different directions.

Theorem C.2.5 (Chytil and Jákl)**.** *Functions computed by two-way transducers are closed under composition, i.e.*

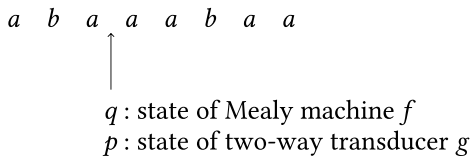
$$\mathrm{2way} \cdot \mathrm{2way} \subseteq \mathrm{2way}.$$

Proof. We will prove the theorem by gradually advancing up the transducer ladder. We will first show that two-way transducers are closed under pre-composition with Mealy machines, then we will extend this to pre-composition with rational functions, and finally we will show pre-composition with two-way transducers, as required by the theorem.

Lemma C.2.6. *Functions computed by two-way transducers are closed under precomposition with Mealy machines, i.e.*

$$\text{Mealy} \cdot \text{2way} \subseteq \text{2way}.$$

Proof. A natural idea would be to use a product construction, in which the two-way transducer for the composition $f \cdot g$ keeps a pair of states as in the following picture:



If the two-way transducer wants to move to the right, then we can easily update both states, by using the transition functions in the two automata. However, if the two-way transducer wants to move to the left, then we have a problem: we do not know how to update the state of the Mealy machine, since there might be several candidates which go to state q by reading the letter a. One way to solve this problem was proposed by Hopcroft and Ullman[11], using a clever construction in which a transducer retraces its steps. However, we will not need this construction, since we can leverage the Krohn-Rhodes Theorem, which says that every Mealy machine can be decomposed into prime Mealy machines. In light of this theorem, it is enough to show pre-composition with primes, i.e.

$$\text{(Reversible Mealy)} \cdot \text{2way} \subseteq \text{2way}$$
$$\text{(Flip-flop Mealy)} \cdot \text{2way} \subseteq \text{2way}.$$

For the reversible Mealy machines, we can simply use the product construction, since the Mealy machine is deterministic in both directions. It remains to deal with the case of flip-flops. For flip-flops, we adapt the product construction as follows. If the two-way transducer wants to move right, the pair of states can be easily updated. If it wants to move to the left, there are two cases: (i) the letter to the left of the head does not change the state of the Mealy machine; or (ii) the letter to the left of the head resets the state of the Mealy machine to some fixed state. In case (i), there is no problem. Finally, in case (ii), we can execute a subroutine which keeps on moving to the left until it reaches another resetting letter, stores the corresponding state, and then returns to the right to the original resetting letter. □

[11] This construction was introduced in J. E. Hopcroft and J. D. Ullman, "An approach to a unified theory of automata", 1967, Lemma 3. A description of the same construction can also be found in the lecture notes Mikołaj Bojańczyk, *An Automata Toolbox*, 2025, Lemma 17.4.

Corollary C.2.7. *Functions computed by two-way transducers are closed under pre-composition with rational functions, i.e.*

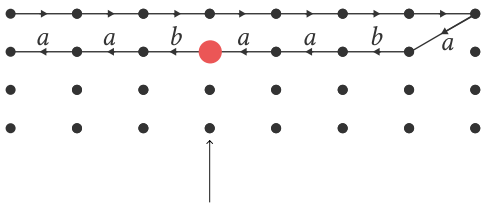
$$\text{Rational} \cdot \text{2way} \subseteq \text{2way}.$$

Proof. As explained in the proof of Theorem B.2.6, every rational function is a composition of: (a) Mealy machines, (b) a right-to-left variant of Mealy machines, (c) string homomorphisms, and (d) appending a separator. Pre-composition under (a) was shown in the previous lemma, and pre-composition under (b) can be shown by a symmetric argument. Pre-composition under (c) and (d) are easy constructions that are left to the reader. □

We are now ready to finish the proof of the theorem, by showing that the composition

$$A^* \xrightarrow{f} B^* \xrightarrow{g} C^*$$

can also be computed by a two-way transducer. By Lemma C.2.3, we can compute the configuration graph of f using a rational function. Since two-way transducers are closed under pre-composition with rational functions by Corollary C.2.7, we can compute the configuration graph of f using a two-way transducer. Finally, by Lemma C.2.4, it is enough to show that the output $g(f(w))$ can be computed by a two-way transducer which looks at the configuration graph of f on w. Once the configuration graph is present, we can do this easily using a product construction, which is explained in the following picture:



product automaton remembers:
- state of f (red circle); and
- state of g

□

Corollary C.2.8. *Every regular function is computed by a two-way transducer*

Proof. Two-way transducers can compute all rational functions by Corollary C.2.7, and they can compute map reverse and map duplicate by Example 19. Finally, they are closed under composition thanks to Theorem C.2.5. □

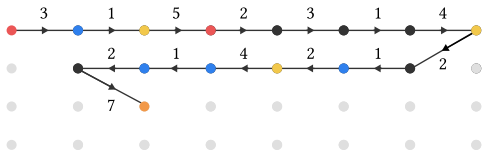
Later in this chapter we will show the converse inclusion, which is that two-way transducers can be decomposed into the prime rational functions (map reverse, map duplicate, and rational functions).

C.2.3 Decidability of equivalence

In the previous chapter, we have shown that equivalence is decidable for regular functions, as defined by compositions of certain prime functions. We will show that two-way transducers are the same as regular functions, and hence the equivalence algorithm from the previous chapter will apply. Nevertheless, it is worth pointing out that the proof from the previous chapter can be adapted to give an equivalence algorithm that works directly on two-way transducers, without going through the prime decomposition. To see this, consider a composition

$$A^* \xrightarrow{f} B^* \xrightarrow{g} \mathbb{Q}$$

of a two-way transducer followed by a weighted automaton. The composed function can be computed directly by a weighted automaton, in which each run describes a labelling of the configuration graph of f by transitions of g, as in the following picture, in which states of the weighted automaton are represented using colours:



The weight of a transition in the new automaton, which corresponds to a column in the above picture, is the product of transition weights of the original weighted automaton g along the column. For example, the second column in the above picture has weight $14 = 1 \times 2 \times 7$. It is helpful in the construction to assume that the weighted automaton g does not use ε-transitions. Such transitions can be eliminated using the same technique as in Lemma B.2.4. The details are left to the reader.

C.2.4 Decomposition into prime functions

A consequence of the closure under composition for two-way transducers was that all regular functions are computed by two-way transducers. In this section, we show the converse, thus establishing:

Theorem C.2.9. *Two-way transducers compute exactly the regular functions.*

Proof. The only part that we need to show is the left-to-right inclusion, i.e. that every two-way transducer is regular, i.e. it can be decomposed into prime functions. We begin by collecting some closure properties of the regular functions.

Lemma C.2.10. *If $f, g : A^* \to B^*$ are regular functions, then so are:*

- *their map liftings; and*
- *their concatenation $w \mapsto f(w) \cdot g(w)$; and*

- *the following conditional function*

$$w \in A^* \quad \mapsto \quad \begin{cases} f(w) & \text{if } w \in L \\ g(w) & \text{otherwise.} \end{cases} \qquad \text{where } L \subseteq A^* \text{ is regular.}$$

Proof. Since map commutes with composition, as we have remarked in the proof of Lemma A.2.4, it is enough to show that the map liftings of the prime functions are regular. Rational functions are closed under map lifting, so we only need to check the map lifting of map reverse and map duplicate, i.e. we have a double map lifting. In the double map lifting, there are two kinds of separators, which – up to a rational rewriting – can be represented using nested brackets (with two levels of nesting):

$$[[123, 45, 6], [78, 9]] \quad \xrightarrow{\text{map map rev}} \quad [[321, 54, 6], [87, 9]]$$

Once we see the double map lifting this way, it is clear that it is a regular function, since it can be implemented using the usual map lifting and a rational rearrangement of brackets.

Before dealing with the remaining two items in the lemma, about concatenation and conditionals, we introduce an auxiliary construction.

Claim C.2.11. *Consider two regular functions*

$$f_1 : A_1^* \to B_1^* \quad f_2 : A_2^* \to B_2^*$$

with disjoint input and output alphabets. The following function $f_1 + f_2$ is also regular:

$$w \in (A_1 + A_2)^* \;\mapsto\; \begin{cases} f_1(w) & \text{if } w \text{ uses only letters from } A_1 \\ f_2(w) & \text{if } w \text{ uses only letters from } A_2 \\ \bot & \text{otherwise, where } \bot \text{ is a string using both } B_1 \text{ and } B_2. \end{cases}$$

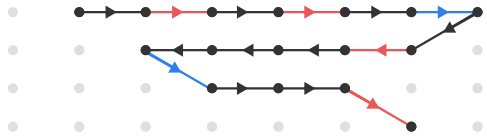
Proof. The construction in the claim is easily seen to be compatible with composition of functions. Therefore, it is enough to show it in the special case in which one of the functions is the identity function, and the other one is a prime function. When the prime function is rational, this is easy to see. When the prime function is map reverse or map duplicate, the argument is also straightforward and left to the reader. (If the input is in the part that should be unchanged, then we can insert a separator before every letter so that it can be recovered after map duplicate or map reverse.) □

Using the above claim, we implement concatenation of regular functions as follows:

$$w \underbrace{\mapsto}_{\text{append } \#} w\# \underbrace{\mapsto}_{\substack{\text{map duplicate} \\ \text{with } \# \text{ not being} \\ \text{a separator}}} w\#w\# \underbrace{\mapsto}_{\substack{\text{use disjoint alphabet} \\ \text{for second copy} \\ \text{and remove extra } \#}} w\#w \underbrace{\mapsto}_{\substack{\mathsf{map}(f+g) \\ \text{with } \# \text{ being} \\ \text{a separator}}} f(w)\#g(w).$$

Finally, a rational function can erase the separator and remove the colour information. A similar idea applies to the conditionals: we can first use a rational function to use an appropriate copy of the input alphabet (depending on membership in the condition language L), and then we can apply the function $f + g$. □

In the proof, we will be using snake graphs, which are pictorial descriptions of a run of a two-way transducer. Here is a picture of a snake graph, in which the colours of the arrows represent output letters (black represents the empty output):



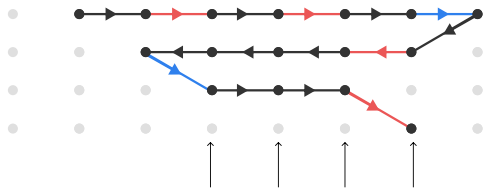
Formally speaking, a *snake graph* with states Q, length n and alphabet B is a directed graph with edges labelled by $B + 1$ that satisfies the following properties:

- each vertex consists of a row $q \in Q$ and a column $i \in \{0, 1, \ldots, n\}$;
- all edges in the graph are on a single directed path;
- edges can only go between adjacent columns.

Similarly to configuration graphs in the proof of Theorem C.2.2, snake graphs can be represented as strings over a finite alphabet C, once the set of states is fixed. Define the *output* of a snake graph to be the concatenation of the labels on the edges. Using the output of snake graphs, every two-way transducer f can be decomposed into two stages, as in the following diagram:

$$A^* \xrightarrow{\text{compute snake graph}} C^* \xrightarrow{\text{output of snake graph}} B^*$$

Since the first function is rational (and therefore also regular) by the same argument as in Lemma C.2.3, it remains to prove that the second function is regular. The second function is partial, since the input might not represent a snake graph. To make it total, we assume that if a string in C^* does not represent a snake graph, then its output is empty. To prove that the output of snake graph can be computed by a regular function, we use induction on a parameter of snake graphs that we call the width: this is the maximal number of times that the snake graph visits a single column. Here is a picture:

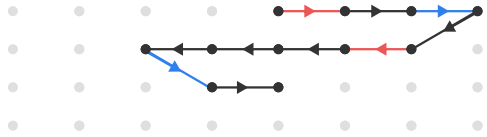


The maximal width of a snake graph is $|Q|$, and hence the following lemma will be sufficient to prove the theorem.

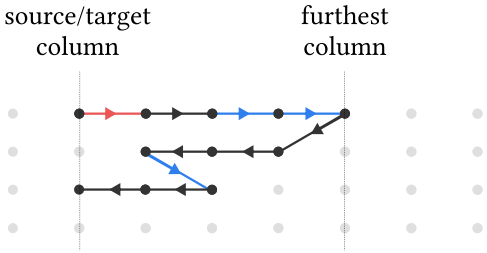
Lemma C.2.12. *Let C be the alphabet used to represent snakes with state Q and output alphabet B. For every $k \in \{1, \ldots, |Q|\}$, the following function is regular:*

$$w \in C^* \quad \mapsto \quad \begin{cases} \varepsilon \text{ if } w \text{ does not represent a snake graph of width} \leq k \\ \text{otherwise, the output of the snake graph.} \end{cases}$$

Proof. We begin by proving the induction step for a special case of a snake, in which the source and target are in the same column, as in the following picture:



A snake with this property is called a *looping snake*. If the column with the source and target is visited at least three times then we can decompose the snake into a concatenation of a constant number of snakes in which the source and target are the only vertices in their column. This is because, as we will see in a moment, each of the factors in this concatenation can be computed by a rational function, and the concatenation itself can be implemented by a regular function thanks to Lemma C.2.10. We are left with the case where the source and target vertices of the looping snake are the only vertices in their column. Among the columns visited by the snake, consider the one that is the furthest away from the source, as in this picture:



In order to reach this furthest column for the first time, the snake cannot use its full width, since it needs to return. Therefore, if we partition the snake into two parts: before and after the first visit to the furthest column, then both parts have smaller width than the original snake. This partition can be computed by a rational function, the outputs for the parts can be computed by induction assumption, and the results can be combined using the concatenation construction in Lemma C.2.10. This completes the proof of the induction step in the case of a looping snake.

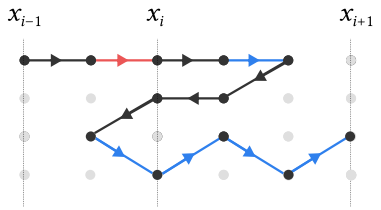
General case. Having dealt with the special case of looping snakes, we now deal with the general case. We can assume without loss of generality that the source column is before the target column. Indeed, using a regular language we can check if the source column is after the target column, and if it is, then we can reverse the snake, which does not change the output. (Formally speaking, when reversing a snake, we

need to change the direction of the arrows in the letters that represent the snake.) Also, we need to use here the cases construction from Lemma C.2.10 to check if the snake needs to be reversed.

From now on, we assume that the source column is before the target column. This means that the snake generally progresses in the left-to-right direction, but it might make loops along the way. In the rest of this proof, we will decompose a snake into parts which are looping and parts which are progressing. The looping parts are defined as follows: for a column x that is visited by the snake, define its *loop part* to be the part of the snake that starts in the first visit to x and ends in the last visit to x. The output of the loop part can be computed by a regular function, as we have explained above. To discuss the progressing part, we will be interested in special columns,

$$x_1 < \cdots < x_n,$$

which are called *record-breakers*. The first record-breaker x_1 is the source column. Assuming that the record-breaker x_i has been defined, we define a new record-breaker x_{i+1} as follows. Consider the rightmost column visited by the loop part of x_i, and then take the next column. (If the loop part of x_i is empty, then we simply take the next column after x_i.) If the next column is visited by the snake, then it becomes the new record-breaker x_{i+1}, otherwise the sequence of record-breakers ends at x_i. Here is a picture:



For each record-breaker x_i, define its *progress part* to be the part of the snake that starts in the last visit to x_i and ends in the first visit to the next record-breaker x_{i+1} (or the end of the snake if $i = n$). To compute the output of the progress part, we can use the induction assumption on smaller width, since every column visited by the progress part is also visited by the loop part, except for the last column which is visited only once. We now complete the proof of the induction step, which is done in several stages that are regular functions:

1. In the first stage, we first mark the record-breakers, i.e. we compute the string

$$w_0 \# w_1 \# \cdots \# w_n,$$

where w_i is the part of the input string between the record-breakers x_i and x_{i+1}. This stage can be implemented by a rational function, since a nondeterministic automaton with output can guess the record-breakers, and then check that they satisfy the conditions in the definition.

2. In the output of the first stage, each separator $\#$ represents a record-breaker. In the second stage, we compute for each separator the part of the input string that is around it, up to the nearest separator. This is represented by the following string

$$(w_0\#w_1)(w_1\#w_2)\cdots(w_{n-1}\#w_n).$$

In the above string, the parentheses are part of the output. As we will see later in the proof, the part $w_{i-1}\#w_i$ will contain both the loop and progress parts of the i-th record-breaker. This stage can be implemented by a regular function, which uses map duplicate to procure the two copies of each w_i, and then uses rational functions to arrange parentheses and remove the extra copies of w_0 and w_n.

3. For each block $(w_{i-1}\#w_i)$ from the previous stage, we will need two copies, one for the loop and one for the progress. For this reason, we duplicate each block from the previous stage, as represented by the string

$$\big((w_0\#w_1)(w_0\#w_1)\big)\big((w_1\#w_2)(w_1\#w_2)\big)\cdots\big((w_{n-1}\#w_n)(w_{n-1}\#w_n)\big).$$

4. We now show that the loop and progress parts of the i-th separator are contained in the block $w_{i-1}\#w_i$. This will finish the proof, since the outputs of these parts can be computed (by induction assumption for the progress part, and by the previous argument for the loop part), and regular functions are closed under applying the map combinator thanks to Lemma C.2.10. The loop part of the i-th record-breaker is contained in $w_{i-1}\#w_i$ because after visiting this record-breaker, the previous one is never visited. For the progress part, the argument is similar: the progress part never returns to its source (since otherwise such a return would be part of the corresponding loop) and it stops immediately upon visiting its target, which is the next record-breaker.

This completes the proof of the induction step in the general case, and hence the proof of the lemma. □

□

Exercises

Exercise C.2.1. Show that if the output alphabet is unary (one letter), then regular functions are exactly the same as rational functions.

Solution. One inclusion is immediate, since rational functions are regular. For the other one, consider a two-way transducer, and recall from Lemma C.2.3 that the function which maps an input string to the string representation of its reachable configuration graph is rational. In that representation, each input position carries one

slice of the graph, and the slice determines the output strings of the transitions that are performed while the head is in that position; there are boundedly many of them, since the reachable configuration graph is a path that crosses each position at most once per state.

Therefore, we can post-compose the rational function above with the letter-to-letter homomorphism which maps a slice to the concatenation of the output strings of its edges. This is again a rational function, and it produces the same letters as the two-way transducer, but ordered by input position rather than by the order in which the transducer produces them. Since the output alphabet has one letter, the two orders give the same string.

Exercise C.2.2. Consider languages (not functions) recognised by two-way deterministic automata. Show that this class is closed under Boolean combinations: union, intersection and complement. Furthermore, all constructions are polynomial in the automaton size.

Solution. As discussed in this book, a two-way automaton must eventually output "yes" or "no". This makes complementation trivial: just swap the outputs, without changing the number of states. For intersection: first run the first automaton, and if it outputs "yes", then return the head to the beginning of the input and run the second automaton, returning its output; otherwise output "no". Returning to the beginning of the input costs one extra state, which walks left until it falls off the input, and hence the number of states is the sum of the two, plus a constant. For union, the construction is the same, except that the second automaton is run when the first one outputs "no". (Alternatively, union is obtained from intersection and complementation, again without leaving the polynomial regime.)

Exercise C.2.3. Consider languages (not functions) recognised by two-way deterministic automata. Show that the shortest accepted string might be exponential in the automaton size.

Solution. In Exercise C.2.2, we showed that intersection can be achieved with a number of states that is the sum of the two automata. Therefore, one can write a deterministic two-way automaton which checks if the input length is divisible by each of the prime numbers $p_1, \ldots, p_n$, so that the number of states is proportional to the sum of the prime numbers. Checking divisibility by p_i needs p_i states in a single left-to-right pass. The shortest accepted string has length

$$p_1 \cdots p_n,$$

which is exponential in $p_1 + \cdots + p_n$, and hence exponential in the number of states.

Exercise C.2.4. Consider a deterministic two-way transducer which is not required to terminate, i.e. on some inputs it can loop forever. Show that the set of input strings where it terminates is a regular language.

Solution. We use the representation of the configuration graph from the proof of Theorem C.2.2: an input string is described by the string over the alphabet C in which each position carries the corresponding column of the configuration graph. As

observed in the proof of Lemma C.2.3, the strings over C that arise this way form a regular language, and the function which maps an input string to the representation of its configuration graph is rational.

The transducer terminates on an input string if and only if the graph of that string admits a path from the initial configuration to the halting vertex. This is a regular property of the (string representation of the) graph: a nondeterministic automaton reads the columns from left to right, and guesses the path through them, remembering only the states in which the path crosses the current position, and whether the halting vertex has already been reached. Since rational functions are continuous, the inverse image of this regular language is a regular language of input strings.

Exercise C.2.5. Improve the solution to the previous exercise, so that the regular language is polynomial, in terms of a deterministic two-way automaton that recognises it. (Hint: use a depth-first search on the run graph, but with a special trick.)

Solution. Without loss of generality we assume that the transducer halts in a fixed configuration, where the head is at the left end of the input string and the state is a fixed final state. Consider the configuration graph of the two-way transducer on some input string. This is a directed graph, in which every vertex has out-degree one or zero by determinism. We want to check if the unique initial configuration can reach the unique final configuration. If we look at the configurations that can reach the unique final configuration, then the corresponding part of the graph is a tree, since it does not contain cycles. This part of the graph can be explored by depth-first search, starting in the unique final configuration. If the search finds the initial configuration, one should accept, otherwise one should reject.

The trick is that this depth-first search is performed by a two-way automaton whose number of states is polynomial. The current vertex of the search is a configuration, and hence it is stored for free: its position is the position of the head, and its state is remembered in the state of the searching automaton. To move down in the tree, the automaton needs to enumerate the children of the current vertex, i.e. the configurations which lead to it in one step; these are determined by a state and a direction, and hence there are boundedly many of them, and each one is at distance one from the current position. To move up in the tree, the automaton applies the transition function of the transducer, which is deterministic, and therefore it does not need a stack: the parent is computed from the child. The only extra information that has to be remembered is which child of the current vertex has been explored last, which is again bounded. Summing up, the state space of the searching automaton is the state space of the transducer times a bounded amount of bookkeeping.

Exercise C.2.6. Show that if a regular function has unbounded output size, then it has exactly linear output size in the following sense: the limit

$$\lim_{n\to\infty} \frac{\text{maximal output length on inputs of length at most } n}{n}$$

is defined, nonzero, and a rational number.

Solution. The size of the output and regularity are not affected once all letters in the output are replaced by a single letter a. Thus, the problem reduces to the case

of unary output, in which case regular functions coincide with rational functions by Exercise C.2.1, and the exercise was solved for rational functions in Exercise B.2.2.

Exercise C.2.7. One can imagine two nondeterministic variants of a two-way transducer, in which more than one output string can be generated:

1. for a given configuration, there are several possible transitions, and the transition is chosen nondeterministically; or
2. the input string from A^* is nondeterministically labelled by some auxilliary alphabet C, giving a new input string in $(A \times C)^*$, and then a deterministic two-way transducer is applied to this new input string.

Show that these models are incomparable in expressive power.

Solution. The first model is not contained in the second one, since the first model can have infinitely many output strings for a given input string, while the second one cannot. There is another rather silly reason why the second model is weaker, which is that the second model has to produce at least one output. However, this can be seen as mistake in the formalisation of the second model, which is rectified as follows: once the new input string is guessed, there is a regular language that describes which guesses are valid, and the transducer is only applied to valid guesses. This way, the second model can produce no output for a given input string, if all guesses are invalid.

Let us now show that the second model is not contained in the first one. The feature that is present in the second model but not in the first model is that the guess made for an input position persists after repeated visits. To see this consider the relation

$$\{ (w, vv) \mid \ |v| = |w| \},$$

where the input alphabet is $\{a\}$ and the output alphabet is $\{a, b\}$. In other words, given an input a^n, the transducer guesses some string in $v \in \{a, b\}^n$ and writes it twice. This can be easily achieved by the second model.

To see why it cannot be computed by the first model, consider a transducer of the first kind, and an input string a^n where n is large enough that 2^n is bigger than the number of configurations, which is the number of states times $n + 1$. For each $v \in \{a, b\}^n$ there is an accepting run which outputs vv; consider the configuration in which this run has produced exactly the first half of its output, i.e. exactly n output letters. There are 2^n strings v and only linearly many configurations, and therefore two different strings $v_1 \neq v_2$ give the same configuration c. But then we can cut and paste: follow the run for v_1 up to c, and then continue with the second half of the run for v_2. This is again an accepting run, and its output is $v_1 v_2$, which does not belong to the relation.

Exercise C.2.8. Consider the two models from the previous exercise. Show that both can be uniformised by deterministic two-way transducers: i.e. if a binary relations is recognised by one of the two nondeterministic models and it is total (each input has at least one output), then it contains a function recognised by a deterministic two-way transducer. In particular, if a relation that is computed by one of the two models, and it is a function, then it is computed by a deterministic two-way transducer.

Solution. We begin with the first model. As in the proof of Theorem C.2.2, a run of the transducer on an input string is presented by slicing its configuration graph, i.e. by a string over the alphabet C in which each input position carries the part of the run that crosses it. The pairs

$$(\text{input string},\ \text{slicing of an accepting run on it})$$

form a rational relation, since being a slicing of an accepting run is a regular property, checked slice by slice as in Lemma C.2.3. If the relation computed by the transducer is total, then this rational relation is total as well, and hence, by Lemma B.2.5, it contains a rational function, which chooses one accepting run for each input string. Once the run has been chosen, its output is produced by a letter-to-letter homomorphism, as in Exercise C.2.1, except that this time the letters have to be produced in the order of the run and not in the order of the input positions; this is done by a two-way transducer which follows the chosen run. Since regular functions are closed under composition, and a rational function followed by a two-way transducer is regular, we get a deterministic two-way transducer, as required.

For the second model, the argument is similar, and simpler. The pairs

$$(\text{input string},\ \text{a valid labelling of it by the auxiliary alphabet})$$

form a rational relation, indeed a letter-to-letter one, because the valid labellings form a regular language. If the relation is total, then Lemma B.2.5 gives a rational function which chooses one valid labelling for each input string, and the deterministic two-way transducer of the model is then applied to the result.

C.3 Streaming string transducers

So far we have proved that the regular functions are exactly those which can be computed by two-way transducers. In this section, we describe a one-way model for the regular functions, which is called *streaming string transducers* (SST's). The general idea is that the transducer processes the input string in a single left-to-right pass, and uses registers to store intermediate results, which are strings over the output alphabet. Before giving a formal definition of the model, we give some examples to illustrate the main ideas behind it.

Example 20. [Reverse] Consider the reverse function

$$\{a, b\}^* \xrightarrow{\text{reverse}} \{a, b\}^*.$$

To compute this function, we use a streaming string transducer that has one register X and one state q. This register starts with the empty string, and input letters are prepended to its content, with transitions of the form

$$\underbrace{q \xrightarrow{a|X \mapsto aX} q}_{\substack{\text{when reading input } a\text{, the state } q \text{ is not changed} \\ \text{and the register } X \text{ is updated by prepending } a}} \qquad q \xrightarrow{b|X \mapsto bX} q.$$

After reading the whole input, the content of the register X is the reverse of the input, and this is also the output of the transducer. □

Example 21. [Map reverse] Consider the map reverse function

$$\{a, b, \#\}^* \xrightarrow{\text{map reverse}} \{a, b, \#\}^*.$$

For this function, we also use one state q, but this time we have two registers. One register X stores the reverse of the current block, and the other register Y stores the output produced for the previous blocks. The transitions are as follows:

$$\underbrace{q \xrightarrow{a|\substack{X \mapsto aX \\ Y}} q}_{\substack{\text{upon reading input } a\text{,} \\ \text{letter } a \text{ is prepended to } X \\ \text{and } Y \text{ is left unchanged}}} \qquad \underbrace{q \xrightarrow{b|\substack{X \mapsto bX \\ Y}} q}_{\substack{\text{upon reading input } b\text{,} \\ \text{letter } b \text{ is prepended to } X \\ \text{and } Y \text{ is left unchanged}}} \qquad \underbrace{q \xrightarrow{\#|\substack{X \mapsto \varepsilon \\ Y \mapsto Y X \#}} q}_{\substack{\text{upon reading input } \#\text{,} \\ \text{register } X \text{ is cleared} \\ \text{and its content is appended to } Y}}.$$

After reading the whole input, the output is obtained as the concatenation YX. □

Example 22. [Iterated duplication] Using a register update of the form $X \mapsto XX$, we could implement a function with exponential output size. This will be forbidden in our model, by requiring that the register updates are copyless, which means intuitively that each register from the old valuation can be used at most once in the new valuation. □

Example 23. [Map duplicate] As we have explained in the previous example, we cannot copy registers. Nevertheless, there is some bounded duplication that is possible. For example, if we want to compute the map duplicate function, we can simply

have two registers X_1 and X_2 for the current block, which are updated independently. Then, the two registers are appended onto Y upon reading the update

$$\begin{matrix} X_1 \\ X_2 \\ Y \end{matrix} \mapsto \begin{matrix} \varepsilon \\ \varepsilon \\ Y X_1 X_2 \# \end{matrix}.$$

□

We now give a formal definition of the model of streaming string transducers. As explained in the above examples, the transducer model is based on updating registers that store strings over the output alphabet. Therefore, we begin by explaining in more detail how such register updates are formalised.

Registers and register updates. Suppose that we have a set $\mathcal{X}$ of register names and an output alphabet B. During its computation, the transducer will maintain a register valuation of type $\mathcal{X} \to B^*$, i.e. the registers will store strings over the output alphabet. To update the register valuation, we will use register updates of the form

$$u : \mathcal{X} \to (\mathcal{X} + B)^*,$$

which are *copyless* in the following sense: each register name from $\mathcal{X}$ can appear in at most one string of the form $u(X)$, and if it appears, then it appears exactly once in that string. In other words, if we concatenate all the strings $u(Y)$ ranging over $Y \in \mathcal{X}$, then each register name from $\mathcal{X}$ appears at most once in the resulting string. A copyless register update can be applied to a register valuation, yielding a new register valuation. The copyless restriction ensures that for each register update u there is some $k \in \mathbb{N}$ such that if the old register valuation had combined length n, then the new one after applying u has combined length at most $n + k$.

Transducer definition. Having defined register updates, we can now give a formal definition of streaming string transducers. The general idea is that this is a deterministic finite automaton, which processes the input string in a single left-to-right pass, and which uses register updates to compute the output string.

Definition C.3.1 (SST)**.** *A streaming string transducer (SST) consists of:*

- *a finite input alphabet A and a finite output alphabet B;*
- *a finite set Q of states, with an initial state $q_0 \in Q$;*
- *a finite set $\mathcal{X}$ of register names;*
- *a transition function*

$$\delta : Q \times A \to Q \times \textit{(copyless register updates)},$$

- *a final output function*

$$F : Q \to (B + \mathcal{X})^*.$$

In the final function, there is no copyless restriction. Nevertheless, we could impose such a restriction without changing the expressive power of the model, since the final function is applied once, and therefore any duplication could be treated by computing the appropriate registers in several independent copies. An SST as in the above definition computes a function of type $A^* \to B^*$ which is defined as follows. After reading an input string $w \in A^*$, the transducer is in a configuration, which consists of a state $q \in Q$ and a register valuation $\eta : \mathcal{X} \to B^*$. In the initial configuration, the initial state is used, and all registers are empty. Upon reading an input letter, the transition function is applied to determine a new state and a register update which is applied to the current register valuation. After reading the whole input string, the output is obtained by applying the final output function to the state at the end of the computation, and then substituting each register name for its contents in the final configuration.

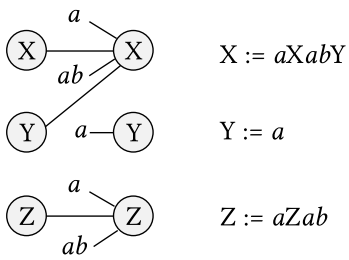
C.3.1 Equivalence with regular functions

In this section we show that the SST model is equivalent to the ones previously described in this part of the book.

Theorem C.3.2 (Alur and Černý)**.** *Streaming string transducers compute exactly the regular functions.*

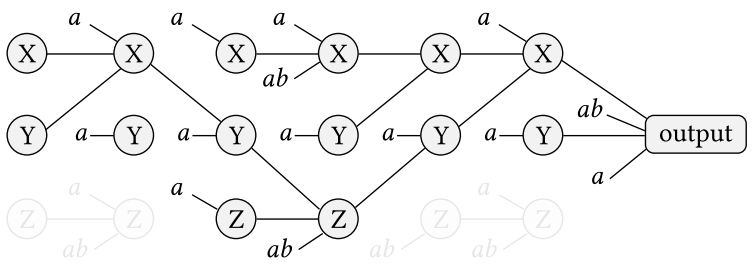
Proof. We begin with the easier implication of the theorem, which is that functions computed by SST's are regular. The converse implication, that regular functions are computed by SST's, is more involved, and relies on the decomposition results for the various classes of transducers that we have proved before.

SST to regular. Let us first show that every SST computes a regular function. Since the regular functions are the same as two-way transducers by Theorem C.2.9, we will prove this by converting an SST into an equivalent two-way transducer. A single register update of the SST can be visualised as a forest (of depth 1), as in the following picture:



The fact that the picture above is a forest, i.e. that the registers in the left column of the above picture have outdegree at most one, is a consequence of the copyless restriction on the register updates. If we compose all the register updates that are applied when

reading an input string of length n, and then we add the final output update, then we get a tree of depth $n+1$, as explained in the following picture (the unreachable parts are erased from the tree):



We use the name *register flow tree* for this tree. This tree can be computed by a rational function, under a suitable string representation, similar to the one for the configuration graph in Section C.2.1. Since two-way transducers are closed under precomposition with rational functions by Corollary C.2.7, it remains to show that there is a two-way transducer which inputs a (string representation of) a register flow tree, and outputs the corresponding output string. This is easy to see, since the output string is produced by traversing the register flow tree in a depth-first search, which can easily be implemented by a two-way transducer.

Regular to SST. In this part of the proof, we leverage the decomposition results on the various transducer classes that we have proved before. By definition, a regular function is a composition of prime functions, namely rational functions, map reverse, and map duplicate. We will show that SST's are closed under post-composition with these prime functions, i.e.

$$\text{SST} \cdot \text{primes} \subseteq \text{SST}.$$

This will imply that SST's are closed under post-composition with all regular functions, and hence SST's contain all regular functions. Among the prime functions we find the rational functions, and we will further decompose them using Theorem B.2.6 into: Mealy machines, right-to-left Mealy machines, homomorphisms, and the end-of-string function $w \mapsto w\#$. We now show how to post-compose SST's with these prime functions.

- **Homomorphisms and end-of-string.** Consider a composition

$$A^* \xrightarrow[\text{SST}]{f} B^* \xrightarrow{g} C^*,$$

  where g is either a homomorphism or the end-of-string function. We want to show that the composition $f \cdot g$ is computed by an SST. This is proved by a straightforward modification of the SST for f. In the case of a homomorphism, we apply the homomorphism when adding the letters to the registers, while in

the case of the end-of-string function, we simply add the letter # to the output in the final output function.

- **Map reverse.** Let us now give an SST for a composition

$$A^* \xrightarrow[\text{SST}]{f} (B+1)^* \xrightarrow{\text{map rev}} (B+1)^*.$$

To do this, we modify the SST f as follows. For a string stored in a register X of this SST, we will be interested in three parts, as illustrated in the following example:

$$\overbrace{abcd}^{\text{before first separator}}\overbrace{\#abbc\#abcdd\#abbcd\#abcca\#}^{\text{from first separator to last separator}}\overbrace{aaabbddd}^{\text{after last separator}}.$$

There is an edge case where the middle part is empty, i.e. there is no separator; in this case we assume that the first part is the whole string. If there is one separator only, then the middle part is #.

In the new SST computing the composition of f with map reverse, each register X of f is replaced by three registers X_1, X_2, X_3 that correspond to these three parts. The invariant, which compares the values of registers X_1, X_2, X_3 in the new SST with the value of register X in the original SST, is the following:

$$\overbrace{dcba}^{X_1 \text{ stores first part reversed}}\overbrace{\#cbba\#ddcba\#dcbba\#accba\#}^{X_2 \text{ stores middle part with map reverse applied}}\overbrace{dddbbaaa}^{X_3 \text{ stores third part reversed}}.$$

We now describe how the register updates are modified in the new SST. We only show the construction for atomic updates, which are:

$$\overbrace{X \mapsto aX}^{\text{prepend a letter}} \quad \overbrace{X \mapsto Xa}^{\text{append a letter}} \quad \overbrace{X \mapsto YZ}^{\text{concatenation}}.$$

  - **Prepending a letter.** If the original SST does a prepend operation $X \mapsto aX$ for a letter a that is not the separator symbol, then the new SST simulates this by adding the same letter to the left part in the opposite order, i.e. $X_1 \mapsto X_1 a$. If the prepended letter is the separator symbol #, then we need to move the left part into the middle part, which is implemented by the following update (updates are, as usual, simultaneous):

$$\begin{aligned} X_1 &\mapsto \varepsilon \\ X_2 &\mapsto \#X_1X_2. \end{aligned}$$

  - **Appending a letter.** The construction for appending letters is symmetric to the one for prepending, except that the third part is used instead of the first one.

– **Concatenation.** Suppose now that the original SST does a concatenation update $X \mapsto YZ$. There are several cases to consider depending on whether the middle part of Y and/or Z is empty or not. To manage this case distinction, the new SST keeps in its state the information about which middle parts are nonempty. When both middle parts are nonempty, then the new SST does the following updates:

$$\begin{aligned} X_1 &\mapsto Y_1 \\ X_2 &\mapsto Y_2 Z_1 Y_3 Z_2 \\ X_3 &\mapsto Z_3. \end{aligned}$$

The remaining cases, when some middle parts are empty, are handled by similar updates, which are left to the reader.

- **Map duplicate.** The construction for map duplicate is similar to the previous one. This time each register of the original SST is split into five registers instead of three, since we need an extra copy for each of the first and last parts.
- **Mealy machines.** We are left with Mealy machines, and right-to-left Mealy machines. Since the construction is symmetric, we only do it for the normal, left-to-right Mealy machines. Consider a composition

$$A^* \xrightarrow[\text{SST}]{f} B^* \xrightarrow[\text{Mealy}]{g} C^*$$

of an SST followed by a Mealy machine. Similarly to the previous cases, we will modify the SST by integrating the actions of the Mealy machine into its register updates. We begin with an erroneous construction, which will help explain the underlying issue. For each state q in the state space Q of the Mealy machine, let g_q be the Mealy machine obtained from g by making q the initial state. A natural idea is to create a new SST f', in which every register X of f is replaced by a family of registers $\{X_q\}_{q \in Q}$, subject to the following invariant: register X_q in the new SST stores g_q applied to the string stored in X by the original SST. However, this construction is not compatible with the copyless restriction. Indeed, if we want to simulate a concatenation update $X \mapsto YZ$ of the original SST, then the corresponding family of updates would be

$$X_q \mapsto Y_q Z_{q'},$$

where q' denotes the state of the Mealy machine after starting in q and reading the string in Y. However, it could be the case that the map $q \mapsto q'$ is not injective, and thus we would need to use several copies of a register $Z_{q'}$. To solve this issue, we will use the Krohn-Rhodes decomposition theorem, to decompose the Mealy machine into reversible machines and flip-flops, and do the construction for these simpler machines.

– **Reversible Mealy machines.** For reversible machines, the map $q \mapsto q'$ is injective, and thus the above construction works without any modification.

– **Flip-flop Mealy machines.** Consider now a flip-flop machine. In this case, we use a similar construction as for map reverse. For each register X of the original SST, we will care about two parts, depending on where the reset letters (those with a constant state transformation) are located, as explained in the following picture:

$$\overbrace{abbbababaaac}^{\text{up to first reset letter (in red)}}\overbrace{ababbbadababababaaaabcbbaabaaacbabbb}^{\text{remaining part}}.$$

In the edge case, where there is no reset letter in X, the first part is the whole string. For each register X in the original SST, the new SST will have $|Q| + 1$ registers: one register X_{right} for the right part, and for each state q of the Mealy machine, one register X_q which stores the left part of X transformed along g_q. It will also remember in its state: (a) the first constant letter in X; and (b) the last constant letter in X. A concatenation operation $X \mapsto YZ$ in the original SST is simulated by the following updates (with q ranging over states of the Mealy machine) in the new SST:

$$\begin{aligned} X_q &\mapsto Y_q \\ X_{\text{right}} &\mapsto Y_{\text{right}} Z_p Z_{\text{right}}, \end{aligned}$$

where p is the state of the Mealy machine after reading Y; the relevant state p can be inferred from the information stored in the state of the new SST. This is a copyless update, as required. The updates for prepending/appending letters are similar, and are left to the reader.

This completes the proof that SST's are closed under post-composition with the Mealy machines, and more generally, all prime functions used in the regular functions.

Having proved that SST's are closed under post-composition with all regular functions, we conclude that SST's contain all regular functions, and this completes the proof of Theorem C.3.2. □

Exercises

Exercise C.3.1. Write an SST over alphabet $\{a, b\}$ that sorts the input string, i.e. it outputs all the a's first, followed by all the b's.

Solution. We use one state and two registers X and Y, which store the letters a and the letters b that have been seen so far. The transitions are

$$q \xrightarrow{a \mid \substack{X \mapsto Xa \\ Y}} q \qquad\qquad q \xrightarrow{b \mid \substack{X \mapsto X \\ Y \mapsto Yb}} q,$$

and the final output function is XY. Both updates are copyless, since each register name is used once. After reading an input string, the register X stores as many letters a as the input has, and likewise for Y and b, and therefore the output is the sorted input string.

Exercise C.3.2. Consider copyful SST's, i.e. ones where the copyless restriction is lifted. Show that they are continuous.

Solution. Consider a composition

$$A^* \xrightarrow{\text{copyful SST}} B^* \xrightarrow{\text{regular language}} \mathsf{Bool}$$

For the regular language, we use a monoid homomorphism. We can now simulate the composition by modifying the copyful SST so that instead of storing a string in a register, one stores only the corresponding element in the monoid. After this simulation, we get a deterministic finite automaton, thus witnessing regularity of the composition.

Exercise C.3.3. Show that copyful SST's can have exponential size outputs, but not doubly exponential.

Solution. For the upper bound, observe that for each input letter, the combined length of the strings in the registers can be at most multiplied by some fixed constant $c \in \{1, 2, \ldots\}$, namely the maximal number of times that a register name is used in an update. Hence the output size is $\mathcal{O}(c^n)$, which is exponential and not doubly exponential.

The upper bound is attained. Consider the copyful SST with one register X, which starts empty, and the transitions

$$X \mapsto XXa,$$

with the final output function X. After reading n letters, the register stores a string of length $2^n - 1$, and hence the function $a^n \mapsto a^{2^n-1}$ is computed by a copyful SST.

Exercise C.3.4. Show that copyful SST's are not closed under composition.

Solution. By the previous exercise, the function $a^n \mapsto a^{2^n-1}$ is computed by a copyful SST. The composition of this function with itself is

$$a^n \mapsto a^{2^{2^n-1}-1},$$

which has doubly exponential output size, and hence it is not computed by a copyful SST, again by the previous exercise.

Exercise C.3.5. A polynomial automaton is an automaton with registers, like an SST, but the registers hold rational numbers instead of strings. The updates need not be copyless, and they can use both addition and multiplication. A polynomial function computes a string-to-rational-number function. Show that $a^n \mapsto 2^n$ can be computed by a polynomial automaton.

Solution. There is one register, which starts with 1 and is doubled in each step.

Exercise C.3.6. Show that $a^n \mapsto 2^{2^n}$ can be computed by a polynomial automaton.

Solution. There is one register, which starts with 2 and is squared in each step.

Exercise C.3.7. Assume that equivalence is decidable for polynomial automata[12]. Show that equivalence is decidable for copyful SST's.

Solution.

Essentially speaking, a polynomial automaton can simulate a copyful SST by replacing its output with a number that represents it injectively. Here are the details. Assume without loss of generality that the output alphabet is $\{0, 1\}$. For a string over the output alphabet, we will be interested in two numbers:

$$\alpha(w) = \text{number represented by } w \text{ in binary} \qquad \beta(w) = 2^{|w|}.$$

For a copyful SST, we replace each string register with two number registers that store the above two numbers. We can simulate concatenation polynomially as follows:

$$\alpha(w_1 \cdot w_2) = \alpha(w_1) \cdot \beta(w_2) + \alpha(w_2) \qquad \beta(w_1 \cdot w_2) = \beta(w_1) \cdot \beta(w_2).$$

Hence, the values of α and β on the output string can be computed by a polynomial automaton. Finally, the output string can be replaced by

$$\alpha(w) + 2 \cdot \beta(w),$$

which is an injective function. Thus the problem of equivalence of copyful SST's is reduced to the problem of equivalence of polynomial automata.

[12] This assumption is true by Michael Benedikt, Timothy Duff, Aditya Sharad, and James Worrell, "Polynomial automata: Zeroness and applications", 2017, Corollary 1.

C.4 Logic

In this chapter, we describe a logical perspective on regular languages and transducers. The underlying theme is the famous automata-logic correspondence, where regular languages (and also transductions) are defined using monadic second-order logic.

C.4.1 Monadic second-order logic

In this section, we begin by describing how logic can be used to define languages. In the next sections, we extend this to defining transductions.

The general idea behind the logical perspective is to describe a property of a word using a formula which quantifies over its positions, and has access to the order on positions as well as their labels. For example, the formula

$$\underbrace{\forall x}_{\substack{\text{for every}\\ \text{position } x}} \quad \underbrace{\exists y}_{\substack{\text{there exists}\\ \text{a position } y}} \quad \underbrace{a(y)}_{\substack{\text{such that}\\ y \text{ has}\\ \text{label } a}} \quad \wedge \quad \underbrace{y \leq x}_{\text{and is before } x}$$

describes the language aA^* of strings that begin with the letter a. The first logic that comes to mind is first-order logic, where formulas are constructed using

$$\underbrace{\exists x \quad \forall x}_{\substack{\text{quantification}\\ \text{over positions}}} \qquad \underbrace{\neg\varphi \quad \varphi_1 \wedge \varphi_2 \quad \varphi_1 \vee \varphi_2}_{\text{Boolean combinations}} \qquad \underbrace{x \leq y \qquad a(x)}_{\substack{\text{atomic formulas for}\\ \text{order and label tests}}},$$

where the letter a is taken from the input alphabet. We use the name *first-order logic over* A^* for this logic, where the finite alphabet A is a parameter of the logic. As we will see, formulas of this logic can only define regular languages.

Example 24. The language $(ab)^*$ is defined by the following formula

$$\bigwedge \begin{cases} \underbrace{\forall x\ \exists y\ a(y) \wedge y \leq x}_{\text{string begins with } a} \\ \underbrace{\forall x\ \exists y\ b(y) \wedge y \geq x}_{\text{string ends with } b} \\ \underbrace{\forall x\ \forall y\ x+1 = y \Rightarrow a(x) \wedge b(y) \vee b(x) \wedge a(y)}_{\text{string alternates between } a \text{ and } b} \end{cases}$$

In the above formula, the successor relation $x + 1 = y$ can be defined in terms of the order relation as follows:

$$x + 1 = y \quad \stackrel{\text{def}}{=} \quad x < y \wedge \neg\exists z\ x < z < y,$$

where $<$ is defined in terms of $\leq$ as usual. $\square$

As it turns out, first-order logic is not strong enough to define all regular languages. The classical example is parity: first-order logic over a one-letter alphabet

$\{a\}$ cannot define the regular language $(aa)^*$ of even-length strings. (A formal argument will be given at the end of this chapter.) For this reason, we extend first-order logic with an extra feature: the ability to quantify over sets of positions; the resulting logic is called *monadic second-order logic* (MSO). In MSO, we have two kinds of variables: first-order variables, which range over positions, and second-order variables, which range over sets of positions. We use the convention that first-order variables are denoted by lowercase letters x, y, z, and second-order variables by uppercase letters X, Y, Z. The logic MSO is obtained by extending first-order logic with the following features:

$$\underbrace{\exists X \quad \forall X}_{\substack{\text{quantification} \\ \text{over sets of positions}}} \qquad \underbrace{x \in X}_{\text{set membership}}$$

Example 25. The language $(aa)^*$ can be defined in MSO by saying that the positions can be labelled with two colours (odd and even), so that the colours alternate along the string, the first position is odd, and the last position is even. The corresponding formula is a variant of the formula for $(ab)^*$ in Example 24:

$$\exists X \bigwedge \begin{cases} \underbrace{\forall x\, \exists y\; y \in X \wedge y \leq x}_{\text{first position is in } X} \\ \underbrace{\forall x\, \exists y\; y \notin X \wedge y \geq x}_{\text{last position is not in } X} \\ \underbrace{\forall x\, \forall y\; x+1 = y \Rightarrow (x \in X \Leftrightarrow y \notin X)}_{\text{string alternates between } X \text{ and not } X}. \end{cases}$$

□

Theorem C.4.1 (Büchi, Elgot, Trakhtenbrot)**.** *A language $L \subseteq A^*$ is regular if and only if it is definable in* MSO*.*

Proof. We begin with the inclusion

$$\text{regular} \quad \subseteq \quad \text{MSO}.$$

To prove this inclusion, we show that the logic MSO is rich enough to formalise the definition of an accepting run. Indeed, consider a nondeterministic automaton with k transitions

$$q_1 \xrightarrow{a_1} p_1, \ldots, q_k \xrightarrow{a_k} p_k$$

The corresponding MSO formula (which is a variant of the parity formula from Example 25) says that the input string can be labelled by transitions in a way that represents

an accepting run (for readability, we use finite disjunctions $\bigvee_i$, whose ranges are described below the formula):

$$\exists X_1 \dots \exists X_k \quad \bigwedge \begin{cases} \text{(1) every position is labelled by exactly one transition:} \\ \forall x \ \bigvee_i x \in X_i \wedge (\bigwedge_{j \neq i} x \notin X_j) \\ \text{(2) consecutive transitions agree on the intermediate state:} \\ \forall x \ \forall y \ x+1 = y \Rightarrow \bigvee_{(i,j)} x \in X_i \wedge y \in X_j \\ \text{(3) the first transition has a source state that is initial:} \\ \forall x \ \exists y \ y \leq x \wedge \bigvee_i y \in X_i \\ \text{(4) the last transition has a target state that is accepting:} \\ \forall x \ \exists y \ y \geq x \wedge \bigvee_i y \in X_i. \end{cases}$$

The disjunctions in the four items range as follows: (1) i ranges over all transitions; (2) (i, j) ranges over all pairs of transitions such that the target state of i is equal to the source state of j; (3) i ranges over all transitions that have an initial source state; and (4) i ranges over all transitions that have an accepting target state. These are finite disjunctions, and hence they correspond to repeated use of $\vee$, and not some kind of quantification. The formula also uses the successor relation $x + 1 = y$, which can easily be defined in terms of the order. It should be easy to see that the formula is true exactly in the words that admit at least one accepting run.

Let us now prove the other inclusion

$$\text{regular} \quad \supseteq \quad \text{MSO}.$$

By induction on the size of a formula of MSO, we will show that the corresponding language is regular. Although the outermost formula does not have free variables, its subformulas will have free variables, and therefore we need to formally associate a language also to formulas that have free variables. This is done by encoding the values of the variables as bits that are stored in the labels of positions. For a string $w \in A^*$ and subsets $X_1, \dots, X_k$ of its positions, let us define

$$w \otimes X_1 \otimes \cdots \otimes X_k$$

to be the string over alphabet $A \times 2^k$ in which every position carries its original label from w, and also k bits of information which indicate the sets from $X_1, \dots, X_k$ that it belongs to. Using this notation, we can use languages to describe formulas with free variables. The following lemma, in the case of formulas that have no free variables, establishes the desired inclusion in the theorem.

Lemma C.4.2. *Let φ be an MSO formula over alphabet A, with*

$$\textit{free variables of } \varphi \quad \subseteq \quad \{\underbrace{x_1, \dots, x_k}_{\substack{\textit{first-order} \\ \textit{variables}}}, \underbrace{X_1, \dots, X_\ell}_{\textit{set variables}}\}.$$

Then the following language over alphabet $A \times 2^{k+\ell}$ is regular:

$$\{ \, w \otimes \{x_1\} \otimes \cdots \otimes \{x_k\} \otimes X_1 \otimes \cdots \otimes X_\ell \mid \ w \models \varphi(x_1, \dots, x_k, X_1, \dots, X_\ell) \, \}.$$

Proof. The regular language will always need to check that the input string is well-formatted, which means that the bits for the first-order variables are such that each variable selects exactly one position. This property is easily seen to be regular. The lemma is now proved by induction on the structure of the formula.

- Consider the atomic formula $x_i \leq x_j$. The corresponding language says that the input string is well-formatted, and the unique position selected by x_i is to the left of the unique position selected by x_j. This is easily seen to be regular. The other atomic formulas $x_i \in X_j$ and $b(x_i)$ are treated in the same way.
- For the Boolean connectives, use the fact that regular languages are closed under Boolean combinations. In the case of negation, we also need to check again if the input string is well-formatted.
- Consider an existential set quantifier $\exists X$. The corresponding automaton nondeterministically guesses for each position a bit 0 or 1 that is added to its label, and then runs the automaton from the induction assumption. For an existential first-order quantifier $\exists x$, we do the same, except that we need to ensure that only one position is selected. For the universal quantifiers, we use De Morgan's laws to reduce them to existential quantifiers and negation.

□

It is worth pointing out that the construction in the above lemma has a large computational cost. When we apply an existential quantifier, we create a nondeterministic automaton. When we apply complementation, we need to complement the automaton, which is done using determinisation. Therefore, whenever we alternate an existential quantifier with negation, the construction incurs an exponential blowup, thus leading to a tower of nested exponentials. This is indeed unavoidable, since one can show that the construction cannot be done so that it has a tower of exponentials of fixed height, independent of the formula. □

C.4.2 Rational functions in terms of logic

In the previous section, we showed how monadic second-order logic can be used to define languages. In this section, we extend the correspondence to cover rational functions. The relevant model is described in the following definition.

Definition C.4.3 (MSO relabelling)**.** *An MSO relabelling consists of:*

1. *an input alphabet A;*
2. *an output alphabet B;*
3. *a finite set Φ of MSO formulas over the input alphabet, such that each formula $\varphi(x)$ in Φ has exactly one free variable, which is of first-order kind;*
4. *an output map of type $\Phi \to B^*$;*
5. *a string in B^* for the empty input.*

We require that for every input string and every position in it, there is a unique formula from Φ which is true in that position.

An MSO relabelling defines a function of type $A^* \to B^*$ as follows. If the input string is empty, then we use the string from the last item. Otherwise, if the input string is nonempty, then to each input position we associate the unique formula from Φ that is true in that position, and we output the string that the output map assigns to this formula. The resulting string – with the outputs concatenated from left to right – is the output of the function.

Theorem C.4.4 (Bloem-Engelfriet)**.** *A string-to-string function is rational if and only if it is definable by an MSO relabelling.*

Proof. We begin with the inclusion

$$\text{rational} \quad \subseteq \quad \text{MSO relabellings.}$$

Consider a rational function $f : A^* \to B^*$. By Theorem B.2.3 and Lemma B.2.4, we know that f is computed by a nondeterministic one-way transducer, which has the following properties: (1) for every input string there is exactly one accepting run; and (2) if the input string is nonempty, then every transition used in the accepting run consumes exactly one input letter. Let

$$\Delta \subseteq Q \times A \times B^* \times Q$$

be the finite set of transitions that are used in accepting runs over input strings; these transitions consume exactly one input letter. The following claim shows that we can define runs of the transducer in MSO.

Claim C.4.5. *For each transition $t \in \Delta$ there is an MSO formula $\varphi_t(x)$ over alphabet A such that for every $w \in A^*$ and every position x in w,*

$$w \models \varphi_t(x)$$

if and only if x is a position where transition t is used in the unique accepting run on w.

Proof. The same construction as in the proof of regular $\subseteq$ MSO in Theorem C.4.1, except that we need to additionally assert that transition t is used in position x. □

The set of formulas in the MSO relabelling is in one-to-one correspondence with the transitions:

$$\Phi = \{\ \varphi_t(x) \text{ from Claim C.4.5} \ \mid \ t \in \Delta\ \}.$$

Because the transducer is unambiguous, each position is selected by a unique formula, as required in the definition of an MSO relabelling. The output function maps $\varphi_t(x)$ to the output string in the transition t. It is easy to check that the MSO relabelling defined in this way computes the same function f as the original transducer.

Let us now prove the other inclusion

$$\text{rational} \quad \supseteq \quad \text{MSO relabellings.}$$

Consider an MSO relabelling defined by a set Φ of formulas.

Claim C.4.6. *The following language over alphabet $A \times \Phi$ is regular:*

$$\{ (a_1, \varphi_1) \cdots (a_n, \varphi_n) \mid \begin{array}{l} \text{for every } i \in \{1, \ldots, n\}, \text{ formula } \varphi_i \in \Phi \\ \text{is true in position } i \text{ of } a_1 \cdots a_n \in A^*. \end{array} \}.$$

Proof. Because each formula in Φ is a formula of MSO, it follows that the language in the statement of the claim is definable in MSO, the corresponding formula being

$$\bigwedge_{\varphi \in \Phi} \forall x \, (\varphi \text{ is in the label of } x) \Rightarrow x \text{ is selected by } \varphi \text{ after projecting letters to } A.$$

Thanks to Theorem C.4.1, the language is regular. □

The language above can be seen as a length-preserving rational relation

$$R \subseteq A^* \times \Phi^*,$$

which for input $a_1 \cdots a_n$ outputs all possible choices of $\phi_1 \cdots \phi_n$ such that the corresponding zipped word belongs to the language from the above claim. Because of the unambiguity requirement in the definition of an MSO relabelling, we know that this relation is in fact a function. Therefore, in order to produce the output of the MSO relabelling, we can apply this rational function, followed by a homomorphism which maps each formula from Φ to the corresponding output string in B^*. Since rational functions are closed under composition, it follows that the function defined by the MSO relabelling is rational. □

Exercises

Exercise C.4.1. Show that a function $f : A^* \to B^*$ is computed by a Mealy machine if and only if it is definable by an MSO relabelling with the following additional restrictions imposed on Definition C.4.3:

1. the string for empty input in item 5 is empty;
2. the output map in item 4 maps each formula to a one-letter string;
3. the formulas in item 3 depend only on the past, which means that $w \models \varphi(x)$ depends only on the prefix of w up to and including position x.

Solution. Consider first a function computed by a Mealy machine. As the set of formulas we take one formula $\varphi_t(x)$ for each transition t, which says that transition t is used in position x; this can be written in MSO as in Claim C.4.5. Since the machine is deterministic, exactly one of these formulas is true in each position, as required in the definition of an MSO relabelling. The three additional restrictions are satisfied: a Mealy machine maps the empty string to the empty string; the output map, which sends φ_t to the output letter of the transition t, produces one-letter strings; and the formulas

depend only on the past, because the transition used in a position is determined by the prefix up to and including this position.

Consider now an MSO relabelling which satisfies the three restrictions. Because of the third restriction, each formula $\varphi \in \Phi$ is determined by the language

$$L_\varphi \quad \stackrel{\text{def}}{=} \quad \{\, w \in A^* \mid \; w \text{ is nonempty and } w \models \varphi(x) \text{ for the last position } x \text{ of } w \,\},$$

in the sense that φ is true in a position of an input string if and only if the prefix up to and including this position belongs to L_φ. This language is definable in MSO, and hence it is regular by Theorem C.4.1. The Mealy machine for the relabelling is the product of deterministic automata for the languages L_φ, with φ ranging over Φ. The state of this product after reading a prefix of the input string tells us which formulas are true in the last position of this prefix; by the uniqueness requirement in the definition of an MSO relabelling, exactly one formula is true, and by the second restriction, it is mapped to a single output letter. Therefore, each transition of the product can be labelled by the output letter that corresponds to its target state, and the resulting Mealy machine computes the same function as the relabelling; for the empty input string, both produce the empty string thanks to the first restriction.

C.4.3 Regular functions in terms of logic

In this section, we define the logical model for the regular functions. This model is called *MSO transductions*. The idea is that we use formulas of MSO logic to define a new order on the input positions, and to change the labels. For example, the reverse function is defined by saying that the labels are the same as in the input string, but the order is reversed. A more sophisticated formula is needed to define the order after applying map reverse, but this can still be done. For functions that create new positions (such as map duplicate), we will need an extra duplication feature in the model. To make this feature easier to use, as well as to prepare the ground for more advanced models of logical transduction in the next part, we begin by introducing an extended syntax for MSO logic.

Extended syntax for MSO logic. We will want to quantify over objects such as gaps in strings, configurations, or sets of configurations. For example, the set of gaps in a string of length n has $n+1$ elements, and therefore quantifying over a gap corresponds to either quantifying over a position, or otherwise considering the special case of the extra gap that is not associated with any position. Such quantification can be handled directly in MSO, by considering different cases inside formulas. Nevertheless, in order to handle such constructions in a succinct and yet precise way, we consider an extended notation for MSO logic, in which first-order variables can have types such as:

$$\underbrace{x : n+1}_{\substack{\text{a variable}\\ \text{that ranges}\\ \text{over gaps}}} \qquad \underbrace{x : Q \times (n+1)}_{\substack{\text{a variable that}\\ \text{ranges over}\\ \text{configurations}}} \tag{11}$$

To implement this, we will use an extended syntax, in which first-order variables are assigned types that are polynomials of the form

$$\tau = n^{d_1} + \cdots + n^{d_k}.$$

Example 26. Using a variable $x : n^2$, we can quantify over pairs of positions. To access the components of this variable, we will have a feature in the logic that will unpack it as a pair (x_1, x_2) of variables of type n. □

Example 27. The summation in a polynomial is used to represent alternative variants. For example, the polynomial $n^0 + n^0$, which we denote by $1 + 1$, allows us to quantify over Boolean values. To access a variant type, we will have a feature in the logic that will allow us to identify which variant is used. □

The polynomials are used as types for first-order variables. We extend this notation to second-order variables, so that we use variables such as

$$\underbrace{X : 2^{Q \times (n+1)}}_{\substack{\text{a variable that} \\ \text{ranges over} \\ \text{sets of configurations}}}.$$

For the second-order variables, the types will be of the form 2^τ, such that the polynomial τ is linear, i.e. it only uses monomials of degree zero or one. (The linear restriction corresponds to the fact that we are working with the monadic fragment of second-order logic.) We have the same formula constructors as in MSO logic, except that the quantifiers are now typed:

$$\underbrace{\exists x : \tau \quad \forall x : \tau}_{\text{first-order quantifiers}} \qquad \underbrace{\exists X : 2^\tau \quad \forall X : 2^\tau}_{\text{second-order quantifiers}} \qquad \underbrace{\vee \ \wedge \ \neg}_{\text{Boolean connectives}} \qquad \underbrace{x \leq y \quad a(x)}_{\substack{\text{atomic formulas} \\ \text{for order and labels}}} \qquad \underbrace{x \in X}_{\text{set membership}}.$$

For the atomic formulas, the variables need to represent positions, i.e. their types must be n, while for set membership, the types need to match: if x has type τ then X must have type 2^τ. Finally, we add a feature for unpacking the polynomial types. For every variable x that has a type

$$\tau = n^{d_1} + \cdots + n^{d_k},$$

and for every $i \in \{1, \ldots, k\}$ we have an atomic formula

$$x = i(\underbrace{x_1, \ldots, x_{d_i}}_{\text{these variables have type } n}),$$

which says that x uses the i-th monomial, and identifies the coordinates of it.

Example 28. If we have a Boolean variable $x : 1 + 1$ then we can check if it is true or false using the atomic formulas $x = 1()$ and $x = 2()$. □

Here is a more sophisticated example of the logic, which computes the transitive closure of the one-step reachability relation of a two-way transducer.

Example 29. Consider a two-way transducer which has input alphabet A and states Q. If an input string has length n, then it has $n + 1$ gaps, with one gap before the input string and one gap after each position. Therefore, the set of configurations of this automaton is produced by the polynomial

$$\tau = Q \cdot (n+1) = \underbrace{n + \cdots + n}_{|Q| \text{ times}} + \underbrace{1 + \cdots + 1}_{|Q| \text{ times}}.$$

We first observe that one-step reachability can be defined in MSO. This means that we can write a formula

$$\delta(x : \tau, y : \tau)$$

which says that the transducer can go in one step from configuration x to configuration y. This is done by unpacking the state and position in x, similarly for y, and then comparing them with respect to the definition of transitions in the transducer. The more interesting observation is that we can also define multi-step reachability. The corresponding formula

$$\delta^*(x : \tau, y : \tau)$$

says that for every set $X : 2^\tau$ of configurations that is closed under the one-step reachability relation we have

$$x \in X \implies y \in X.$$

Observe that we can quantify over sets of configurations, since the type τ has degree at most one, which corresponds to the fact that the number of configurations is linear in the length of the input string. □

String-to-string MSO transductions. We now explain how MSO logic can be used to define regular string-to-string functions. The idea is that the positions of the output string are defined by some formula $\varphi(x : \tau)$, where τ is a linear polynomial. This means that we can duplicate each input position a constant number of times, and then possibly add a constant number of extra positions.

Definition C.4.7. *A string-to-string MSO transduction is given by:*

- *an input alphabet A;*
- *an output alphabet B;*
- *a type $\tau \in \mathbb{N}[n]$, which is linear;*

- *a family of formulas*

$$\underbrace{\varphi_{univ}(x:\tau)}_{\text{universe formula}} \qquad \underbrace{\varphi_b(x:\tau)}_{\substack{\text{one letter formula} \\ \text{for each } b \in B}} \qquad \underbrace{\varphi_{\leq}(x:\tau, y:\tau)}_{\substack{\text{order} \\ \text{formula}}},$$

which are MSO *formulas (in extended syntax) over the alphabet A.*

We require that for every input string $w \in A^*$, every element $x \in \tau(w)$ selected by the universe formula satisfies exactly one of the letter formulas, and that the order formula defines a linear order on the elements selected by the universe formula

The semantics of an MSO transduction is a function of type $A^* \to B^*$, which is defined as follows. For an input string $w \in A^*$, the output string is obtained by taking the elements of $\tau(w)$ that satisfy the universe formula, ordering them according to the order formula, and then labelling them according to the letter formulas. The assumption that the type τ is linear will ensure that the output has linear size, as required in a regular function. Later on in this book, we will consider an extension where the linearity requirement is dropped. Also, MSO transductions can be used to define functions on other structures, such as graphs or trees, but this is beyond the scope of this book.

Example 30. [Reverse as an MSO transduction] Let us show that string reversal over alphabet A can be defined as an MSO transduction. Since the positions in the output string are the same as the positions in the input string, we use the type $\tau = n$. The labels are inherited from the input string, so the letter formula $\varphi_a(x)$ is $a(x)$. Finally, the order is reversed, so the order formula is

$$\varphi_{\leq}(x:n, y:n) \quad \stackrel{\text{def}}{=} \quad x \geq y.$$

□

Example 31. [Outputs of constant length] Consider a string-to-string function

$$f : A^* \to B^*$$

in which all outputs have at most one letter. In this case, we can use the type $\tau = 1$. For each output letter $b \in B$, the label formula

$$\varphi_b(x:1)$$

is an MSO formula that describes the regular language of input strings that are mapped to b. (The formula ignores the variable x, since it carries no information.) The order formula is "true", since there is at most one position to order. The same construction would work for functions with outputs of length at most k, by using the type $\tau = 1 + \cdots + 1$ with k copies of 1, so that all output positions can be covered. □

Example 32. [Duplication as an MSO transduction] We describe an MSO transduction for the duplicating function $w \mapsto ww$. In this case, the type is $\tau = n + n$. For each

letter $a \in A$ in the alphabet, the corresponding letter formula says that its argument is one of the copies of an input position with label a, i.e.

$$\varphi_a(x : \tau) \quad \stackrel{\text{def}}{=} \quad \exists y : n \quad a(y) \wedge (x = 1(y) \vee x = 2(y)).$$

The order formula uses the lexicographic order, with the first copy of the input string coming before the second copy:

$$\varphi_{\leq}(x : \tau, y : \tau) \quad \stackrel{\text{def}}{=} \quad \bigvee \begin{cases} \text{both from first copy:} \\ \exists x' : n \; \exists y' : n \quad x = 1(x') \wedge y = 1(y') \wedge x' \leq y' \\ \text{from different copies:} \\ \exists x' : n \; \exists y' : n \quad x = 1(x') \wedge y = 2(y') \\ \text{both from second copy:} \\ \exists x' : n \; \exists y' : n \quad x = 2(x') \wedge y = 2(y') \wedge x' \leq y' \end{cases}$$

□

Theorem C.4.8 (Engelfriet-Hoogeboom)**.** *String-to-string* MSO *transductions define exactly the regular functions.*

Proof. As our model for the regular functions, we use two-way transducers. We begin with the easier direction

$$2\text{way} \subseteq \text{MSO transductions}.$$

One approach would be to show that MSO transductions are closed under composition, see the exercises, which would reduce the proof to the case of prime functions. However, the proof for general two-way transducers is easy enough so that we do not need to decompose into prime functions. Indeed, we can simply use MSO to formalise the semantics of a two-way transducer. We assume without loss of generality that the two-way transducer outputs at most one letter in each transition; this can be ensured by using extra transitions. As the affine type, we use the type

$$\tau = Q \cdot (n+1) = \underbrace{n + \cdots + n}_{|Q| \text{ times}} + \underbrace{1 + \cdots + 1}_{|Q| \text{ times}}$$

that was used in Example 29. The universe formula says that $x \in \tau(w)$ represents a reachable configuration such that the transducer outputs one letter in this configuration. This can be defined in MSO as explained in Example 29. The letter formulas identify the letter produced by the configuration, while the order is defined as in Example 29.

We now turn to the more difficult direction

$$2\text{way} \supseteq \text{MSO transductions}.$$

from MSO transductions to two-way transducers. We begin by reducing to the special case where the type τ is n, which will make the proof easier.

Lemma C.4.9. *We can assume without loss of generality that the type τ is n.*

Proof. The crucial observation is that types more complicated than n can be reduced away by using rational functions:

(*) every MSO transduction can be obtained by composing a rational function with an MSO transduction that uses type n.

Once we have proved (*), the lemma will follow, since two-way transducers are closed under precomposition with rational functions. It remains to prove (*). Consider an MSO transduction that has type $\tau = \alpha n + \beta$. The rational function copies each input letter α times, and then appends β extra letters, as in the following example:

$$123 \quad \mapsto \quad 1^\alpha 2^\alpha 3^\alpha \#^\beta.$$

If f is this rational function, then there is a one-to-one correspondence between $\tau(w)$ and positions in $f(w)$. This correspondence and the structure of order and labels can be defined in MSO, and thus the output of the original MSO transduction can be obtained by a new MSO transduction that uses type n. □

Thanks to the above lemma, we assume that the type τ is n, which means that the output positions are in one-to-one correspondence to some subset of the input positions. To design the two-way automaton, we use the following lemma.

Lemma C.4.10. *Let Φ be a finite set of MSO formulas over some alphabet A, which have either one or two free variables of type n. There is a letter-to-letter rational function*

$$f : A^* \to C^*$$

with the following property for every formula $\varphi \in \Phi$:

- *If φ has one free variable, then there is a subset $F_\varphi \subseteq C$ such that*

$$w \models \varphi(x : n) \quad \iff \quad F_\varphi \text{ contains the label of } x \text{ in } f(w).$$

- *If φ has two free variables, then there is a regular language $L_\varphi \subseteq C^+$ such that*

$$w \models \varphi(x : n, y : n) \quad \iff \quad L_\varphi \text{ contains the infix } [x..y] \text{ in } f(w).$$

Proof. For the formulas that have one free variable, there is no difficulty: the label in the output of f can simply store which positions are selected by which formulas, which can be done thanks to the results in the previous section. The more interesting case is that of a formula $\varphi(x : n, y : n)$ with two free variables. As we have shown in the proof of the previous section, the language

$$\{\ w \otimes \{x\} \otimes \{y\} \mid \ w \in A^+ \text{ and } x \leq y \text{ are distinguished positions with } \varphi(x, y)\ \}$$

is recognised by some deterministic automaton, call it $\mathcal{A}$. In the proof, we will be discussing state transformations of this automaton. Consider some input string $w \in C^+$ with two distinguished positions $x \leq y$. In order to check whether

$$w \models \varphi(x, y)$$

holds, it is enough to know the state transformation of the string

$$w \otimes \{x\} \otimes \{y\}. \tag{12}$$

This state transformation can be obtained by composing three state transformations

$$\underbrace{w[..x-1] \otimes \emptyset \otimes \emptyset}_{\substack{\text{we write } \delta_{<x} \text{ for the} \\ \text{state transformation} \\ \text{of this part}}} \quad \underbrace{w[x..y] \otimes \{\text{first}\} \otimes \{\text{last}\}}_{\substack{\text{we write } \delta_{xy} \text{ for the} \\ \text{state transformation} \\ \text{of this part}}} \quad \underbrace{w[y+1..] \otimes \emptyset \otimes \emptyset}_{\substack{\text{we write } \delta_{>y} \text{ for the} \\ \text{state transformation} \\ \text{of this part}}}.$$

The letter-to-letter rational function f in the statement of the lemma labels each position x with the state transformations $\delta_{<x}$ and $\delta_{>x}$. Once we have the range $[x..y]$ of the relabelled string $f(w)$, we can compute the state transformation in (12) as follows: the state transformation $\delta_{<x}$ is stored in the first position, the state transformation $\delta_{>y}$ is stored in the last position, and the state transformation δ_{xy} can be computed by looking at the letters in the range $[x..y]$. All of this can be computed by a regular language L_φ, as required in the conclusion of the lemma. □

Let Φ be the following set of formulas:

1. the letter formulas from the transduction;
2. unary formulas that select the first and last positions of the output order;
3. a formula which selects positions that belong to the output string, have a defined successor in the output order, and this successor is to the left in the input order;
4. a formula that represents the successor relation in the output order, and also one for the predecessor relation.

The formulas in the first three items have one argument, while the two formulas in the last item have two arguments. Apply Lemma C.4.10 to this set of formulas, yielding a letter-to-letter rational transduction

$$g : A^* \to C^*.$$

Since two-way transducers are closed under precomposition with rational transductions, it is enough to show that the output of the original MSO transduction f can be computed by a two-way transducer that inputs the output of g. In other words, the following diagram commutes

$$A^* \xrightarrow{g} C^* \xrightarrow{h} B^* \qquad A^* \xrightarrow{f} B^*$$

for some two-way transducer h. The two-way transducer h is described in the following program. In the program, the transducer can ask questions about formulas with one argument from Φ, since the answer to such a question is encoded in the label of the current position.

1. If the input string is empty, output $f(\varepsilon)$. Otherwise, go to step 2.
2. Scan the input string from left to right, and in each position ask if it is the first position. The answer to this question can be determined by looking at the label in $g(w)$, thanks to Lemma C.4.10. Find the position x that is the first in the output order, and go to step 3.
3. Let x be the current position. Ask which of the label formulas is true in it, and produce this letter on the output. Ask if the position x is the last in the output order, and if so terminate the computation. Otherwise, go to step 4.
4. We know that x is not the last position in the output order. Ask if the successor of x, call it y, is to the left or right in the input order. If y is to the right, then start moving to the right, while running the automaton for the language $L \subseteq C^*$ that is associated by Lemma C.4.10 to the formula with two arguments that represents the successor relation on output positions. The position y is the first one such that the infix $[x..y]$ belongs to L. If y is to the left, do the same, but using the language for the predecessor relation. Go to step 3.

□

C.4.4 The first-order fragment

Recall the notion of an aperiodic string-to-string function from Definition A.2.7, and the corresponding condition on state transformations in Lemma A.2.11. The latter condition only talks about state transformations, and hence it can be meaningfully applied to deterministic automata without output: a DFA is called *aperiodic* if for every state transformation δ, the sequence $\delta^1, \delta^2, \ldots$ eventually reaches a stable value. The following theorem shows that the characterisation in Theorem A.2.8, which talked about aperiodic Mealy machines and flip-flops, happens to identify the same class as first-order logic.

Theorem C.4.11 (McNaughton-Papert, Schützenberger)**.** *A language is definable in first-order logic if and only if it is recognised by an aperiodic* DFA*.*

Proof. This theorem is stated in terms of languages, while Theorem A.2.8 was stated in terms of functions, but the two results are closely related, since languages and Mealy machines are essentially the same. In fact, we will use the correspondence between languages and functions in this proof. It just so happens that compositions of flip-flops are more naturally connected to functions, while first-order logic is more naturally connected to languages.

From aperiodicity to first-order logic. We begin with the implication

$$\text{aperiodic DFA} \quad \subseteq \quad \text{first-order definable.}$$

This is usually considered the more difficult implication in the theorem, but it so happens that we have already done the work for this implication, by proving the Krohn-Rhodes decomposition theorem and its aperiodic variant in Theorem A.2.8. Consider

a regular language $L \subseteq A^*$, and let

$$f : A^* \to \{\text{yes, no}\}^*$$

be the corresponding Mealy machine, in which the n-th letter of the output string says if the first n letters of the input string belong to L. We prove the difficult implication in the following three steps:

$$\begin{array}{c} L \text{ is computed by an aperiodic DFA} \\ \Downarrow \text{by definition} \\ f \text{ is computed by an aperiodic Mealy machine} \\ \Downarrow \text{Theorem A.2.8} \\ f \text{ is a composition of flip-flops} \\ \Downarrow \text{see below} \\ L \text{ is first-order definable} \end{array}$$

It remains to show the last step, i.e. if f is a composition of flip-flops, then L is first-order definable. For this, we introduce the notion of first-order definable Mealy machine, which means that for every output letter b, the language

$$\{ w \in A^* \mid \ f(w) \text{ ends with letter } b \}$$

is first-order definable. Flip-flop machines are easily seen to be first-order definable, and it is also easy to see – using substitution of formulas – that first-order definable Mealy machines are closed under composition. Hence, if f is a composition of flip-flops, then it is first-order definable, and therefore also L is first-order definable.

From first-order logic to aperiodicity. Let us now turn to the other implication of the theorem, which is

$$\text{aperiodic DFA} \quad \supseteq \quad \text{first-order definable.}$$

To prove this implication, we will associate to each first-order formula a corresponding aperiodic DFA. This will be based on a technique that is called *compositionality of logic*. To explain this technique, it will be easier to work with a more combinatorial representation of first-order formulas, which uses nested sets.

Definition C.4.12 (k-types)**.** *For $k \in \{0, 1, \ldots\}$, the k-type of a string $w \in A^*$, denoted by $tp_k(w)$, is defined inductively as follows:*

$$\begin{aligned} tp_0(w) &= \emptyset \\ tp_{k+1}(w) &= \{ \ (tp_k(w_1), a, tp_k(w_2)) \mid \ a \in A \text{ and } w = w_1 a w_2 \ \}. \end{aligned}$$

By definition, once the alphabet A is fixed there are finitely many k-types for each k, although the number grows very rapidly, with an exponential increase for

each step $k \mapsto k+1$. We now establish that k-types are closely related to first-order formulas. The number k will correspond to the *quantifier-rank* of a first-order formula, which is the maximal number of quantifiers that can be found along a branch in its syntax tree. Equivalently, the quantifier-rank of atomic formulas is 0, Boolean combinations do not increase the quantifier rank (i.e. one uses the maximal quantifier rank of the formulas in the Boolean combination), and quantifiers increment it by one. The following lemma establishes the connection between k-types and first-order formulas of quantifier rank at most k.

Lemma C.4.13. *Two strings in A^* have the same k-type if and only if they satisfy the same first-order formulas of quantifier rank at most k.*

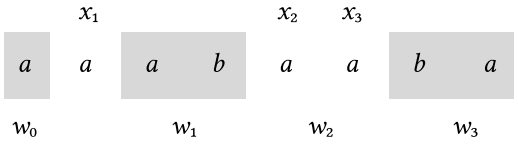
Proof. For our applications, we will not need the right-to-left implication, but we prove it anyway for the sake of completeness. For the right-to-left implication, we observe that for each possible k-type, the corresponding set of strings can be defined in first-order logic with quantifier rank k. The formula quantifies over a position x that corresponds to the letter a in a split $w = w_1 a w_2$, and then uses formulas from the induction assumption to describe the types for the parts w_1 and w_2. The formulas from the induction assumption need to have their quantification restricted to positions that are $< x$ for w_1 and $> x$ for w_2, but this does not affect the quantifier rank.

Let us now prove the left-to-right implication. To prove this implication, we need a more refined statement that deals with free variables.

Claim C.4.14. *Let $\varphi(x_1, \ldots, x_n)$ be a first-order formula of quantifier rank k. Then*

$$w \models \varphi(x_1, \ldots, x_n) \quad \text{with } x_1 < \cdots < x_n$$

depends only on the labels of the distinguished positions $x_1, \ldots, x_n$ and the k-types of the strings $w_0, \ldots, w_n$ between them, defined as in the following picture:



Proof. Induction on the structure of φ. The only interesting case is when the formula uses a quantifier, say an existential quantifier

$$\exists x_{n+1}\ \varphi(x_1, \ldots, x_{n+1}).$$

The quantified variable will necessarily fall either on one of the distinguished positions or in some interval between them. If it falls on one of the distinguished positions x_i, then we can apply the induction assumption for the formula obtained from φ by replacing x_{n+1} with x_i. Otherwise, the quantified position falls in one of the intervals $w_0, \ldots, w_n$, and we can use the available information and the induction assumption to check if it is true. □

In the case of a formula without free variables, the information in the claim is the same as the k-type of the string, and therefore the claim implies the left-to-right implication of the lemma. □

We now show that the k-types are compatible with operations on strings, which will allow us to define an automaton structure on them, and that the resulting automaton is aperiodic. (In the lemma below, a sequence is called aperiodic if it eventually reaches a stable value, i.e. from some point on all elements in the sequence are the same.)

Lemma C.4.15. *For every $k \in \{0, 1, \ldots\}$ and $w, v \in A^*$, we have:*

1. ***Refinement:*** *if $k > 0$ then $\mathrm{tp}_k(w)$ uniquely determines $\mathrm{tp}_{k-1}(w)$;*
2. ***Congruence:*** *$\mathrm{tp}_k(w)$ and $\mathrm{tp}_k(v)$ uniquely determine $\mathrm{tp}_k(wv)$.*
3. ***Aperiodicity:*** *the sequence $\mathrm{tp}_k(w^1), \mathrm{tp}_k(w^2), \mathrm{tp}_k(w^3), \ldots$ is aperiodic.*

Proof. For the induction basis of $k = 0$ there is nothing to do, since there is only one possible type. Let us now prove the induction step. Suppose that we have proved all three properties for k and we want to prove them for $k + 1$.

- **Refinement.** Choose a triple in $\mathrm{tp}_{k+1}(w)$ and use the congruence property for k to obtain $\mathrm{tp}_k(w)$. The answer does not depend on the choice of triple. If there is no triple, then w is the empty string, and the type $\mathrm{tp}_k(w)$ is also empty.
- **Congruence.** By definition of types,

$$\begin{aligned}\mathrm{tp}_{k+1}(wv) =& \{\ (\mathrm{tp}_k(w_1), a, \mathrm{tp}_k(w_2 v))\ \mid\ a \in A \text{ and } w = w_1 a w_2\ \} \cup \\ & \{\ (\mathrm{tp}_k(w v_1), a, \mathrm{tp}_k(v_2))\ \mid\ a \in A \text{ and } v = v_1 a v_2\ \}.\end{aligned}$$

  By the congruence property for k, the first set in the union depends only on the set of triples in $\mathrm{tp}_{k+1}(w)$ and the individual value $\mathrm{tp}_k(v)$. Since the latter value depends only on $\mathrm{tp}_{k+1}(v)$ by the refinement property, it follows that the first set depends only on the types $\mathrm{tp}_{k+1}(w)$ and $\mathrm{tp}_{k+1}(v)$. Since the same is true for second set, we obtain the congruence property for $k + 1$.
- **Aperiodicity.** By definition, the type $\mathrm{tp}_{k+1}(w^n)$ is equal to the following set

$$\{\ (\mathrm{tp}_k(w^{n_1} w_1), a, \mathrm{tp}_k(w_2 w^{n_2}))\ \mid\ a \in A, w = w_1 a w_2, n = n_1 + 1 + n_2\ \}.$$

  By the congruence and aperiodicity properties for k, the two sequences

$$\begin{aligned}&\mathrm{tp}_k(w^1 w_1), \mathrm{tp}_k(w^2 w_1), \mathrm{tp}_k(w^3 w_1), \ldots \\ &\mathrm{tp}_k(w_2 w^1), \mathrm{tp}_k(w_2 w^2), \mathrm{tp}_k(w_2 w^3), \ldots\end{aligned}$$

  are both aperiodic. From this it follows that the set in the definition of $\mathrm{tp}_{k+1}(w^n)$ must also reach a stable value, since the powers n_1 and n_2 can be capped to some fixed threshold.

□

Using the above lemma, we show that if a language is first-order definable, then it is recognised by an aperiodic DFA, thus completing the proof of the theorem. Let k be the quantifier rank of a first-order formula that defines the language. Define the set of states Q to be the set of possible k-types of strings in A^*. As we have remarked before, this is a finite set. By the congruence property in the above lemma, we can define an automaton structure on this set, since the k-type of wa depends only on the k-type of w and the letter a. By the aperiodicity property in the lemma, the automaton is aperiodic. Finally, since the language is defined by a first-order formula of quantifier rank at most k, it follows that strings that reach the same state cannot be distinguished by the language, and hence we can meaningfully define an accepting set of states in the automaton. □

First-order definable rational functions. In the above theorem, we described the languages $L \subseteq A^*$ that can be defined in first-order logic. One can also lift this characterisation to string-to-string functions. For the rational functions, the corresponding fragment is the *first-order relabellings*, which are defined in the same way as the MSO relabellings from Definition C.4.3, except that instead of MSO formulas, we use first-order formulas. The corresponding automaton model is bimachines, in which both the prefix and suffix automata are aperiodic (call these aperiodic bimachines).

Theorem C.4.16. *A string-to-string function is a first-order relabelling if and only if it is computed by an aperiodic bimachine.*

Proof. We begin with the direction from aperiodic bimachines to first-order relabellings. Consider the prefix automaton, which is aperiodic. Thanks to Theorem C.4.11, we can describe runs of this automaton in first-order logic. In particular, for each transition t of the prefix automaton, we can write a first-order formula $\varphi_t(x)$ which selects input positions in which t is the transition used by the prefix automaton. Similarly, we can describe in first-order logic the transitions of the suffix automaton. By combining these formulas, we can describe in first-order logic the pairs (transition of the prefix automaton, transition of the suffix automaton) that are used on each input position, and using these pairs we can compute the corresponding parts of the output string.

Let us now prove the opposite direction, from first-order relabellings to aperiodic bimachines. Let k be the maximal quantifier rank of the first-order formulas used in the relabelling. Consider an input string $a_1 \cdots a_n \in A^*$. The output produced by the first-order relabelling on the i-th position depends on which formula from Φ is true in the i-th position. By compositionality, this depends only on the following three pieces of information:

$$\overbrace{a_1 \cdots a_{i-1}}^{\substack{\text{(1)}\\ k\text{-type of}\\ \text{this prefix}}} \overbrace{a_i}^{\substack{\text{(2)}\\ \text{this}\\ \text{letter}}} \overbrace{a_{i+1} \cdots a_n}^{\substack{\text{(3)}\\ k\text{-type of}\\ \text{this suffix}}}.$$

To retrieve this information, we will use the prefix and suffix automata in the bimachine. After reading a prefix of the form $a_1 \cdots a_i$, the prefix automaton will keep track

of information (1) and (2), i.e. the k-type of $a_1 \cdots a_{i-1}$ and the letter a_i. This can be done by an aperiodic automaton thanks to Theorem C.4.11. Similarly, after reading the suffix $a_{i+1} \cdots a_n$, the suffix automaton will keep track of information (3), i.e. the k-type of this suffix. Therefore, if we know the states of these two automata for a gap of the form

$$\overbrace{a_1 \cdots a_i}^{\text{prefix}} \overbrace{a_{i+1} \cdots a_n}^{\text{suffix}},$$

with $i > 0$, then we have sufficient information to compute the string produced by the relabelling on the i-th input position. If the input string is empty, then the output can be computed by a separate transition. This shows that the first-order relabelling can be computed by an aperiodic bimachine. □

First-order definable regular functions. Finally, it would be useful to give a machine characterisation of the first-order transductions, i.e. the functions that can be defined by MSO transductions in which all formulas are first-order. We only state the appropriate result, and we leave the proof for a future edition of these notes: a string-to-string function is a first-order transduction if and only if it can be obtained by composing: map reverse, map duplicate and first-order rational functions (as in Theorem C.4.16).

Exercises

Exercise C.4.2. Consider first-order formulas that define languages. Show that for every $n \in \{1, 2, \ldots\}$ there is a first-order formula of size polynomial in n such that the corresponding language contains exactly one string which has length at least

$$\exp(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2^{\exp(n-1)} & \text{if } n > 1. \end{cases}$$

Solution. By induction on $n \in \{0, 1, \ldots\}$ define the *strings of order* n as follows. If $n = 0$ then there is only one such string and it is empty. Consider now $n > 0$ and let

$$w_1, \ldots, w_\ell$$

be the strings of order $n-1$, listed in lexicographic order. The new alphabet is obtained by extending the previous alphabet with two fresh letters 0 and 1. A string string of order n is defined to be any string of the form

$$a_1 w_1 a_2 w_2 \cdots a_\ell w_\ell \qquad \text{where } a_1, \ldots, a_\ell \in \{0, 1\}.$$

At each increment n, the number of strings of order n increases exponentially. We will now show that the first string of order n can be defined by a first-order formula that is polynomial in n, thus completing the exercise. To prove this, we show a stronger claim which is amenable to induction.

Claim C.4.17. *There is a first-order formula*

$$\varphi_n(x_1, x_2, y_1, y_2)$$

of size polynomial in n which holds if and only if

- *$x_1 < x_2$ and the corresponding infix is a string of order n; and*
- *$y_1 < y_2$ and the corresponding infix is a string of order n; and*
- *the above two infixes are consecutive in the lexicographic order on strings of order n.*

Proof. The induction step uses φ_{n-1} to identify, inside two infixes of order n, the pairs of sub-infixes of order $n-1$ that carry the same index i: these are the pairs that can be reached from the first sub-infixes by applying the "consecutive" relation of order $n-1$ the same number of times. Being consecutive in the lexicographic order on strings of order n then says that the bit sequences $a_1 \cdots a_\ell$ of the two infixes are consecutive binary numbers, i.e. there is an index where the first has 0 and the second has 1, before it the bits agree, and after it the first has only 1s and the second only 0s. All of this is first-order, once the sub-infixes of order $n-1$ can be compared.

The only delicate point is the size. A naive translation uses φ_{n-1} a constant number $c > 1$ of times, which would give formulas of size exponential in n. This is avoided by the standard trick of quantifying over the arguments: instead of writing several copies of φ_{n-1} with different arguments, one writes a single copy applied to universally quantified arguments,

$$\forall u_1 u_2 v_1 v_2 \ \big(\text{the arguments are one of the finitely many tuples we care about} \Rightarrow \ \varphi_{n-1}(u_1, u_2, v_1, v_2) \big),$$

so that the size of φ_n is the size of φ_{n-1} plus a constant. The details are left to the reader. □

Exercise C.4.3. In MSO, set quantification is restricted to sets of positions. In full second-order logic, we can quantify over sets of pairs, or sets of triples, etc. Show that second-order logic can define non-regular languages.

Solution. Consider the language $\{ a^n b^n \mid \ n \geq 0 \}$, which is not regular. It is defined by a formula of second-order logic which says that: (a) every position labelled a is before every position labelled b; and (b) there is a binary relation R on positions which is the graph of a bijection between the a-positions and the b-positions. The first condition is first-order. The second one uses a quantifier over a set of pairs of positions, and says that every pair in R consists of an a-position and a b-position, that every a-position appears in exactly one pair, and that every b-position appears in exactly one pair; all of this is first-order, once the relation R is available. Such a bijection exists if and only if the two sets have the same size, and hence the formula defines the language above.

Exercise C.4.4. Consider the variant of first-order logic on strings in which the only available predicate is the successor relation, i.e. the order relation is not available. Show that this logic is strictly weaker than first-order logic with order.

Solution. The separating language is

$$\{ w \in \{a,b,c\}^* \mid \begin{array}{l} w \text{ has exactly one letter } b \text{ and exactly one letter } c, \\ \text{and the } b \text{ comes before the } c \end{array} \}.$$

With the order relation, this is defined by saying that there are positions $x < y$ labelled b and c respectively, and no other position is labelled b or c.

To prove that it cannot be defined with the successor relation only, we use an Ehrenfeucht-Fraïssé argument. Fix a number k of rounds, and consider the two strings

$$a^m b a^m c a^m \qquad \text{and} \qquad a^m c a^m b a^m,$$

where m is much bigger than 2^k. The first one belongs to the language and the second one does not. In the game with the successor relation only, the duplicator wins in k rounds: the invariant is that after i rounds, the chosen positions are matched in such a way that two positions are at the same distance in both strings whenever that distance is at most 2^{k-i}, and are far apart in both strings otherwise. This invariant can be maintained, because the only difference between the two strings is the order of the letters b and c, which are at distance more than 2^k from each other. Since the duplicator wins, no formula of quantifier rank k with successor only can distinguish the two strings, and hence no such formula defines the language.

C.5 Combinators

The elegant syntax of regular expressions is one of the main attractions of regular languages. Regular expressions have no variables or states, and they are compositional – bigger regular expressions are built from smaller ones using a few simple combinators, namely union, concatenation and Kleene star. In this section, we describe an attempt at a similar syntax for regular functions. One could argue that the prime decompositions earlier in this book are such a syntax, see Definition C.0.1, with only one combinator, namely composition. However, those decompositions are motivated mainly by a mathematical desire to find a minimal core of a given transducer class. The beauty of regular expressions is that they are both a minimal core, and are also usable in everyday programming practice. The formalism in this section is intended to go in that direction, and to be more grounded in programming practice, by using functions like

$$\underbrace{A^{**} \xrightarrow{\texttt{concat}} A^*}_{\substack{\text{flatten a list of lists} \\ \text{into a list}}} \qquad \underbrace{\mathbb{1} + A \times A^* \xrightarrow{\texttt{cons}} A^*}_{\substack{\text{a list is constructed from} \\ \text{either an empty list, or} \\ \text{a head and a tail}}},$$

which are familiar to any user of a functional programming language, such as Haskell.

Types. The functions in the above examples are not based purely on strings, but have more structured types. For example, the domain of `concat` uses lists of lists, while the domain of `cons` is constructed using products, co-products, lists and a unit type. Such types can be encoded into strings using separators, but we will prefer a more direct approach which does not use such representations. The types that we care about are defined in the following way.

Definition C.5.1 (Types)**.** *Types are expressions built using the following constructors:*

$$\underbrace{\mathbb{1}}_{\substack{\textit{unit type} \\ \textit{which contains} \\ \textit{a unique element } 1 \in \mathbb{1}}} \quad \underbrace{A \times B}_{\substack{\textit{product} \\ \textit{type}}} \quad \underbrace{A + B}_{\substack{\textit{co-product} \\ \textit{type}}} \quad \underbrace{A^*}_{\substack{\textit{list} \\ \textit{type}}}.$$

Each type also represents a set in the natural way, and we will use the same notation for the type and the set that it represents. For example $\mathbb{1}^*$ represents list over the unit type; this set is in bijection with the natural numbers. Using multiple disjoint copies of the unit type we can represent finite sets, e.g. $\mathbb{1}+\mathbb{1}$ represents a binary alphabet, and $\mathbb{1} + \mathbb{1} + \mathbb{1}$ represents a ternary alphabet. Formally speaking, for the ternary alphabet we should choose one of the two bracketings $\mathbb{1} + (\mathbb{1} + \mathbb{1})$ or $(\mathbb{1} + \mathbb{1}) + \mathbb{1}$, but these two types are isomorphic and we will not pay attention to the difference.

Type-to-type functions. A type-to-type function is defined to be any function

$$A \xrightarrow{f} B$$

where A and B are types. As we have explained above, types can be used to describe finite alphabets and lists over these alphabets, and hence string-to-string functions are a special case of type-to-type functions. However, the opposite inclusion is also essentially true, since types can be represented using strings, and thus type-to-type functions can be represented as string-to-string functions. As the alphabet for the string representations, we use an eight-letter alphabet that we denote by 8 because it has eight letters:

$$(\quad)\quad[\quad]\quad,\quad 1\quad \mathsf{Left}\quad \mathsf{Right}.$$

The string representation for elements of types is defined by induction: the unique element of $\mathbb{1}$ is represented by 1; elements of $A + B$ are represented by $\mathsf{Left}\ a$ or $\mathsf{Right}\ b$; pairs are represented by (a, b), and lists are represented by $[a_1, \ldots, a_n]$. Using this representation, we can lift transducer classes to describe type-to-type functions.

Definition C.5.2 (Regular under string representation)**.** *A type-to-type function*

$$A \xrightarrow{f} B$$

is called regular under string representation *if there is some regular string-to-string function f' such that the following diagram commutes:*

$$\begin{array}{ccc} A & \xrightarrow{\ f\ } & B \\ {\scriptstyle \text{string representation}}\downarrow & & \downarrow{\scriptstyle \text{string representation}} \\ 8^* & \xrightarrow[f']{} & 8^*. \end{array}$$

Similarly we define rational functions between types.

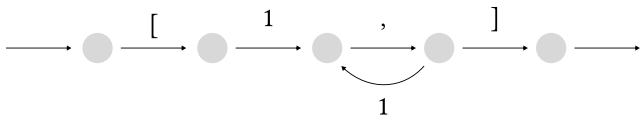
Example 33. [Parity under string representation] Consider the function

$$\mathbb{1}^* \xrightarrow{f} \mathbb{1} + \mathbb{1}$$

which returns the parity of the input list. We think of the codomain as representing the Booleans, with the first copy of $\mathbb{1}$ representing "yes" and the second copy representing "no". Here are two examples of input/output pairs for this function under string representation:

$$[1,1,1] \mapsto \underbrace{\mathsf{Right1}}_{\text{"no" means odd}} \qquad [1,1] \mapsto \underbrace{\mathsf{Left1}}_{\text{"yes" means even}}.$$

The function f is regular under string representation. The transducer f' counts the appearances of the letter 1 modulo two, and outputs Left1 or Right1 accordingly. Note that not all input strings over the input alphabet 8 are representations of the domain, but the transducer f' will still produce some output for such inputs. However, the input strings which are correctly formatted, i.e. represent some element of the domain, form a regular language, as witnessed by the following automaton:



Therefore, we could improve f' so that it produces the empty string for inputs that are not correctly formatted. □

C.5.1 Regular terms

As we mentioned earlier in this section, when defining type-to-type functions, we would like to work directly on the types instead of using string representations as is done in Definition C.5.2. This is the purpose of the following definition, in which type-to-type functions are built by applying four standard combinators to certain atomic functions. We begin by describing two atomic functions that are not completely standard.

Example 34. [Split] For types A and B, the split function

$$(A + B)^* \xrightarrow{\texttt{split}} A^* \times (B \times A^*)^*$$

splits the input list according to appearances of elements from B, as illustrated in the following example where elements of A are letters and elements of B are red digits:

$$[a, b, c, 1, d, e, 2, 3, f, g, h] \mapsto ([a, b, c], [(1, [d, e]), (2, []), (3, [f, g, h])]).$$

A defining property of the split function is that it is a bijection, and its inverse is the concatenation of all its entries in the left-to-right order. □

Example 35. [Group prefix multiplication] For a finite group G, the group prefix multiplication function

$$G^* \xrightarrow{\texttt{pref}} G^*$$

takes a list of elements from G and returns the list of all prefix products. This means that the output list has the same length as the input list, and the i-th element of the output list is the result of applying the group operation to the first i elements in the input list. For example, if G is $\{0, 1, 2\}$ with addition modulo 3, then we have

$$[0, 1, 2] \mapsto [0, 1, 0].$$

Note that this function is parameterised not just by the underlying set of the group, but also its group operation. □

Apart from the above two examples, all other atomic functions are standard built-in functions of Haskell, and we believe that the combinators are also self-explanatory. Here is the definition.

Definition C.5.3 (Regular term)**.** *A* regular term *is any expresion that is built by applying the combinators from Item 2 to the atomic terms in Item 1. In the following A, B, C range over types, and G ranges over groups whose underlying set is a finite type.*

1. *The atomic functions are:*

   Identity $A \xrightarrow{\texttt{id}} A$.

   Projections $A \times B \xrightarrow{\texttt{fst}} A$ *and* $A \times B \xrightarrow{\texttt{snd}} B$.

   Co-projections $A \xrightarrow{\mathsf{Left}} A + B$ *and* $B \xrightarrow{\mathsf{Right}} A + B$.

   Distributivity $A \times (B + C) \xrightarrow{\texttt{distR}} (A \times B) + (A \times C)$.

   List constructor $\mathbb{1} + A \times A^* \xrightarrow{\texttt{cons}} A^*$.

   List deconstructor $A^* \xrightarrow{\texttt{uncons}} \mathbb{1} + A \times A^*$.

   Reverse $A^* \xrightarrow{\texttt{reverse}} A^*$.

   Concatenation $A^{**} \xrightarrow{\texttt{concat}} A^*$.

   Split $(A + B)^* \xrightarrow{\texttt{split}} A^* \times (B \times A^*)^*$.

   Group prefix multiplication $G^* \xrightarrow{\texttt{pref}} G^*$.

2. *The combinators are:*

   Composition *composition of functions.*

   Pairing $A \xrightarrow{(f,g)} B \times C$ *for two functions* $A \xrightarrow{f} B$ *and* $A \xrightarrow{g} C$.

   Co-pairing $A + B \xrightarrow{\mathsf{either}\ f\ g} C$ *for two functions* $A \xrightarrow{f} C$ *and* $B \xrightarrow{g} C$.

   Map $A^* \xrightarrow{f^*} B^*$ *for a function* $A \xrightarrow{f} B$.

Each term has syntax (an expression built from the atomic functions and combinators) and semantics (a type-to-type function), but we will not distinguish between the two, and we will use the same notation for both.

Theorem C.5.4. *A type-to-type function is regular under string representation if and only if it is defined by some regular term.*

The rest of Section C.5.1 is devoted to the proof of Theorem C.5.4. We begin with the easy direction, which is

$$\text{defined by a regular term} \quad \Rightarrow \quad \text{regular under string representation.}$$

This is a straightforward induction on the structure of the term. The induction basis says that all atomic terms are regular under string representation, which is easy to check, with the only non-trivial part being parsing the string representation of the input. Let us illustrate this on one example, which is distributivity.

Lemma C.5.5. *For every types A, B, C, the distributivity function*

$$A \times (B + C) \xrightarrow{\texttt{distR}} (A \times B) + (A \times C)$$

is regular under string representation.

Proof. The input, if correctly formatted, has one of the forms $(a, \mathsf{Left}b)$ or $(a, \mathsf{Right}c)$, where a is a string representation of some element in A, and similarly for b and c. The transducer that implements the distributivity function first checks if the input is correctly formatted, and then checks if it uses the left or right variant, and finally outputs the corresponding output. In all of these steps, we use the fact that parsing can be implemented by a finite automaton, as stated in the following claim, which is proved by a straightforward induction on the structure of the type.

Claim C.5.6. *For every type A, the following is a regular language*

$$\{\, w \in 8^* \mid \ w \textit{ represents an element of } A \,\}.$$

Using the above claim, an automaton can check if the input is correctly formatted, and if so, it can check if the input uses the left or right variant. Also, the output can be produced, by identifying the separation between the two components of the input. □

The remaining atomic functions are handled in a similar way, and the combinators are also easily seen to preserve regularity, thus proving the induction step.

The rest of this proof is devoted to the more difficult implication

$$\text{defined by a regular term} \quad \Leftarrow \quad \text{regular under string representation},$$

which says that the terms are expressively complete. Ultimately, we need to prove expressive completeness for all type-to-type functions, but we begin with the string-to-string case, which contains the essence of the proof. In the string-to-string case, we can profit from the prime decomposition theorems (which are, technically speaking, our formal definition of the transducer class). Since terms are syntactically closed under composition, it suffices to show that the following prime string-to-string functions are definable by terms, which is done in Lemmas C.5.7 - C.5.14 as described in the following table.

Lemma C.5.7	string-to-string homomorphisms
Lemma C.5.10	appending an endmarker $w \mapsto w\#$
Lemma C.5.12	map reverse and map duplicate
Lemma C.5.13	flip-flop Mealy machines
Lemma C.5.14	reversible Mealy machines
symmetric	right-to-left variants of prime Mealy machines

In the last line of the above table, the symmetry is resolved by the string reversal function, since a right-to-left Mealy machine can be decomposed as: (1) reverse; (2) a left-to-right Mealy machine; (3) reverse again. From the decomposition result on rational functions in Theorem B.2.6 and the definition of regular functions, the above lemmas will imply that all regular string-to-string functions are definable by terms. We will then conclude the proof in Lemma C.5.15 by extending the completeness from string-to-string functions to all type-to-type functions.

Let us begin with the case of string-to-string homomorphisms, which is easy and also establishes some infrastructure that will be used in the other cases.

Lemma C.5.7. *Every string-to-string homomorphism is definable by a regular term.*

Proof. We say that a type is of the form $\mathbb{1} + \ldots + \mathbb{1}$ if it is a co-product of finitely many copies of the unit type. (As mentioned before, a formal description would need to specify the bracketing, but all possible bracketings are isomorphic using regular terms.) The following claim shows that such types cover all finite types.

Claim C.5.8. *Every finite type admits a bijection, which is definable by a regular term, to a type of the form* $\mathbb{1} + \ldots + \mathbb{1}$.

Proof. Induction on the structure of the finite type, with distributivity used in the case of products. It is worth noting that the proof needs two kinds of distributivity:

$$A \times (B + C) \xrightarrow{\texttt{distR}} (A \times B) + (A \times C)$$
$$(A + B) \times C \xrightarrow{\texttt{distL}} (A \times C) + (B \times C).$$

Our atomic functions contain only the first kind, but the second kind can be derived from the first one, since we have a function for swapping coordinates:

$$A \times B \xrightarrow{\texttt{swap}} B \times A,$$

which can be derived using pairing and projections. □

Claim C.5.9. *Every type-to-type function with a finite domain is definable by a regular term.*

Proof. By Claim C.5.8, we can assume that the domain is of the form $\mathbb{1} + \ldots + \mathbb{1}$. By using co-pairing, we can reduce to the case of a function with domain $\mathbb{1}$. We now use induction on the structure of the codomain. The interesting case is a function

$$\mathbb{1} \xrightarrow{f} A^*$$

with a list codomain. Suppose that the output is a list $[a_1, \ldots, a_n]$. A term defining the function is constructed as follows. If $n = 0$, i.e. the output is the empty list, then we can use the composition

$$\mathbb{1} \xrightarrow{\texttt{Left}} \mathbb{1} + A \times A^* \xrightarrow{\texttt{cons}} A^*.$$

If the output list is non-empty, then use pairing to construct the pair (head, tail) for the list, with both the head and tail constructed using the induction hypothesis. Finally, use the list constructor to construct the output list from the head and tail. (There is a nested induction here on the length of the output list.) □

By combining the above two claims, we see that if A and B are finite types, then every function of type $A \to B^*$ is definable by a regular term. Every string-to-string homomorphism then arises by taking such a function f, and using the following composition

$$A^* \xrightarrow{f^*} B^{**} \xrightarrow{\texttt{concat}} B^*.$$

□

Lemma C.5.10. *The function $w \mapsto w\#$ is definable by a term.*

Proof. When formalising this function, we think of $\#$ as being a fresh symbol. Therefore, we can think of its type as being

$$A^* \to (A + \mathbb{1})^*.$$

(This type of thinking is justified by Claim C.5.9, which says that any function with a finite domain is definable by a term, which covers bijections between finite types.) By the symmetry principle, it will be easier to define a prepending function $w \mapsto \#w$. We begin by defining functions into the unit type (which are unique).

Claim C.5.11. *For every type A, the function $A \xrightarrow{!} \mathbb{1}$ is definable by a term.*

Proof. Induction on the structure of the type, with the identity used for the induction bases, projections used for products, co-pairing used for co-products, and the list deconstructor used for lists. □

Using the ! function from the above claim, we can define prepending as follows.

$$A^* \xrightarrow{((!;\mathsf{Right}),\mathsf{Left}^*)} (A + \mathbb{1}) \times (A + \mathbb{1})^* \xrightarrow{\mathsf{Right};\mathtt{cons}} (A + \mathbb{1})^*.$$

□

Lemma C.5.12. *Map reverse and map duplicate are definable by terms.*

Proof. We begin by defining some helper functions:

$$\begin{aligned}
\mathbb{1} &\xrightarrow{\mathtt{empty}} A^* && \mathsf{Left};\mathtt{cons} \\
A &\xrightarrow{\mathtt{pure}} A^* && (\mathtt{id}, (!;\mathtt{empty}));\mathsf{Right};\mathtt{cons} \\
A^* \times A^* &\xrightarrow{\mathtt{binconc}} A^* && (\mathtt{id} \times \mathtt{pure});\mathsf{Right};\mathtt{cons};\mathtt{concat}.
\end{aligned}$$

In the last line above, we use a derived derived combinator for applying two functions in parallel to separate coordinates of a product, which is defined by

$$f \times g = ((\mathtt{fst}; f), (\mathtt{snd}; g)).$$

Finally, we define the inverse of `split`, which is a form of concatenation that was discussed earlier in Example 34. This inverse is the composition of

$$A^* \times (B \times A^*)^* \xrightarrow{\mathsf{Left} \times (\mathsf{Right} \times \mathsf{Left}^*)^*} (A + B)^* \times ((A + B)^* \times (A + B)^*)^*$$

followed by the below form of concatenation where $C = A + B$:

$$C^* \times (C \times C^*)^* \xrightarrow{\mathtt{id} \times (\mathsf{Right};\mathtt{cons})^*} C^* \times C^{**} \xrightarrow{\mathsf{Right};\mathtt{cons}} C^{**} \xrightarrow{\mathtt{concat}} C^*$$

Equipped with the above helper functions, we can define map reverse and map duplicate, as required by the lemma. We assume that these functions are typed as

$$(A + \mathbb{1})^* \to (A + \mathbb{1})^*.$$

The term for map reverse is the following composition:

$$(A + \mathbb{1})^* \xrightarrow{\texttt{split}} A^* \times (\mathbb{1} \times A^*)^* \xrightarrow{(\texttt{reverse} \times (\texttt{id} \times \texttt{reverse})^*)} A^* \times (\mathbb{1} \times A^*)^*,$$

followed by the inverse of `split`. In the presence of these helper functions, map duplicate is defined in the same way as map reverse, except that we use $(\texttt{id}, \texttt{id}); \texttt{binconc}$ instead of string reversal. □

Lemma C.5.13. *Every flip-flop Mealy machine is definable by a term.*

Proof. Consider a flip-flop Mealy machine

$$A^* \xrightarrow{f} B^*$$

with states Q. We assume that the states are a finite type. The input alphabet partitions into two parts: the identity letters which leave the state unchanged, and the reset letters which reset the state to a fixed state. To define this function using a term, we will split the input string according to appearances of the reset letters. The exact construction has several steps, which are described below.

1. Before splitting along the reset letters, we will need to annotate them with the states that they produce. This is implemented by a string-to-string homomorphism

   $$A^* \xrightarrow{h} (A + Q)^*$$

   in which the identity letters are left unchanged, and each reset letter a is replaced by the two-letter string aq where q is the state after reading a.

2. In the second step, we apply `split` to the output of the first step in order to split along the states, thus yielding an element of

   $$(w_0, [(q_1, w_1), \ldots, (q_n, w_n)]) \in A^* \times (Q \times A^*)^*.$$

   Thanks to the way that the first step was defined, we know that: (1) each of the strings $w_0, \ldots, w_n$ contains at most one reset letter, and if it contains a reset letter then it is the last one; and (2) q_i is the state of the Mealy machine after reading $w_0 \cdots w_{i-1}$. In particular, thanks to (1), we know that the state of the Mealy machine will not change while reading any of the strings $w_0, \ldots, w_n$ (except at the end, when the state no longer influences the output). We will leverage this in the following steps, by simulating the Mealy machine using a homomorphism.

3. Thanks to the way that the previous step was defined, the output produced by the Mealy machine on each pair (q_i, w_i) is obtained by applying the function

   $$\begin{aligned} Q \times A^* &\xrightarrow{g} B^* \\ (q, v) &\mapsto h_q(v) \end{aligned}$$

where h_q is the string-to-string homomorphism that replaces each letter by the corresponding output of the Mealy machine assuming that the current state is q. Using Lemma C.5.7, as well as distributivity and co-pairing, we can show that g is definable by a term.. (The variant of distributivity is the left-hand one that was given in the proof of Claim C.5.8.) For the initial string w_0, we can simply apply the homomorphism h_{q_0}, where q_0 is the initial state of the Mealy machine.

4. As a result of the previous step, we get an element of

$$B^* \times (B^*)^*$$

which should be concatenated to get the output string. This concatenation can be done by a term.

□

Lemma C.5.14. *Every reversible Mealy machine is definable by a term.*

Proof. Consider a reversible Mealy machine

$$A^* \xrightarrow{f} B^*$$

with states Q. Let us also choose some arbitrary group structure on the input alphabet A, say a cyclic group.

- Let G be the group of permutations of the finite set Q; this is a finite group and we can describe its underlying set using some type. In the first step we apply the letter-to-letter homomorphism

$$A^* \to (A \times G)^*$$

where each letter is paired with its state transformation.

- Let the output of the first step be

$$[(a_1, g_1), \ldots, (a_n, g_n)] \in (A \times G)^*.$$

Thanks to the group structure on A, the alphabet $A \times G$ can be seen as a group. In the second step, we apply group prefix multiplication.

- The result of the previous step is

$$[(a'_1, g'_1), \ldots, (a'_n, g'_n)] \in (A \times G)^*,$$

where a'_i is the product of the first i input letters (according to our arbitrary group structure) and g'_i is the state transformation induced by the first i input letters. Recall the delay machine from Example 4, which is a flip-flop Mealy machine (and hence definable by a term thanks to Lemma C.5.13). Using this machine, we can label each pair with the previous one, so that each pair in the list becomes a four-tuple $(a'_i, g'_i, a'_{i-1}, g'_{i-1})$, with the edge cases of a'_0 and g'_0 being defined to be the identity elements of the respective groups.

- Using group inverses we can recover the original input letters, because
$$a_i = (a'_{i-1})^{-1} \cdot a'_i.$$
Therefore, for each item in the previous list, we can compute the pair (a_i, g'_{i-1}) which describes the original input letter, and the state transformation before reading it. This is all the information needed to get the corresponding output letter in the Mealy machine, and hence a string-to-string homomorphism can be used to get the output string.

□

We have thus proved that all prime regular string-to-string functions are definable by terms, and hence all regular string-to-string functions are definable by terms. It remains to lift the result to type-to-type functions. Consider a type-to-type function
$$A \xrightarrow{f} B$$
that is regular under string representation. To define this function by a regular term, we compose the terms for: (a) the string representation of A; (b) the regular string-to-string function that defines f on string representations; and (c) the inverse of the string representation of B. Step (b) is already done, and we now need to show that steps (a) and (c) can be defined by terms. This will be proved in the following lemma. The inverses in the lemma are understood as one-sided inverses, i.e. a function such that the composition
$$A \xrightarrow{\text{string representation}} 8^* \xrightarrow{\text{inverse of string representation}} A$$
is the identity on A.

Lemma C.5.15. *For every type, its string representation and a one-sided inverse can be defined by terms.*

Proof. Induction on the structure of the type. We only do the induction step for a product type $A \times B$, with the other cases being similar. The string representation is easy, and we focus on its inverse. If correctly formatted, the input string is of the form (a, b), where a and b are string representations of elements of A and B, respectively. Consider the rational function
$$8^* \xrightarrow{f} (8 + \mathbb{1})^*$$
which identifies the separating comma between a and b (there might be other commas in the representations of a and b), and writes it using a fresh symbol. This function is defined by a term, since we have already proved expressive completeness for terms in the case of string-to-string functions. Apply the function f, followed by split, leading to an element of the type
$$8^* \times (\mathbb{1} \times 8^*)^*. \tag{13}$$
We know that if the original input was correctly formatted, then the list on the second coordinate will have exactly one element. We will now extract it using a version of the head operation.

Claim C.5.16. *For every type A there is a term $A^* \xrightarrow{\texttt{head}} A$ which returns the first element when the input list is nonempty (and an arbitrary element when it is empty).*

Proof. The difficulty is inventing a value when the input list is empty. By a simple induction on the structure of the type A, we show that there is a term

$$\mathbb{1} \xrightarrow{\texttt{invent}} A.$$

Using this term, the claim is proved as follows

$$\texttt{head} = \texttt{uncons}; \mathsf{either}\ \texttt{invent}\ \texttt{fst}$$

□

By applying `head` to the second coordinate in (13), we get an element of

$$8^* \times (\mathbb{1} \times 8^*).$$

By projecting away the $\mathbb{1}$, removing the outer brackets (which are the first letter of the first 8^* and the last letter of the second 8^*), and applying the induction assumption to both coordinates, we get the desired element of $A \times B$. This completes the induction step for the product type, and the other cases are similar. □

C.5.2 Rational terms

In Section C.5.1, we proved that the regular type-to-type functions are exactly those that can be defined by regular terms. In this section, we isolate a fragment which corresponds to the rational and not regular functions.

C.6 References

The name "regular" for the functions studied in this part comes from [EH01, p. 217]. Although we begin with a definition in terms of prime functions, this is a later development historically, and the original definition was in terms of two-way transducers. The decomposition into primes is implicit in [BDK18] and explicit in [BS20, Theorem 18], although the latter paper discuss a more general framework of orbit-finite sets.

Two-way transducers. Two-way automata as acceptors (and not transducers) appear in [She59, Definition 2] and [RS59, Definition 13]. Both of these papers prove that such acceptors define exactly the regular languages, which covers the continuity result from Theorem C.2.2. Shepherdson also sketches a transducer variant of two-way automata, providing the first known appearance of the two-way transducer model that we use here [She59, Note 4]. The transducer model is also somewhat recognisable in papers from the 1960s and 1970s such as [HU67] or [AHU69]. In the latter case, our two-way transducer model is the same as balloon automata with no auxilliary storage.

Closure of two-way transducers under pre-composition with Mealy machines was first proved in [HU67, Theorem 5] using an ingenious construction. In our proof the same result, see Lemma C.2.6, we avoid ingenuity by appealing to the decompositions into prime functions. Closure of two-way transducers under composition, which is Theorem C.2.5 in this book, was first proved in [CJ77, Theorem 2]. The authors of [CJ77] credit this result to [AU70]. I believe that they are referring to [AU70, Theorem 2], but that theorem talks about inverse images of languages under two-way automata, and not about compositions of two transducers, although the proof does seem to contain the right ingredients for composition of transducers.

Decidability of equivalence. Decidability of equivalence for two-way transducers was first proved in [Gur82, Theorem 1]. Our statement from Theorem C.1.4 deviates from that result in that it uses a different model (compositions of primes) which is only later proved to be equivalent, and a proof that is based on weighted automata, see the comments in Section B.5.

Streaming string transducers. This model was introduced in [AČ10, Section 3], where it was shown to be equivalent to two-way transducers and MSO transductions in Theorems 1,2,3. Many of the main ideas behind this model can be traced back to [BE00, Theorem 17]. If we restrict the result from [BE00] from trees to strings, and translate it to our terminology, then it says that the MSO transductions are exactly those that can be computed by an SST with regular look-ahead (regular look-ahead can be formalised as a pre-composition with a right-to-left Mealy machine). Now it is known that the regular look-ahead can be eliminated, since already without look-ahead SST's are equivalent to the regular functions. Another early transducer take on the regular functions (again, using MSO transductions and the more general tree-to-tree case) can be found in [EM99, Theorem 7.1]; the paper uses macro tree transducers, which are nicely explained in [Ngu24] using a terminology similar to the one of this

book. Despite the earlier work described above, one should note that the attractive operational presentation of SSTs from [AČ10] has proved instrumental in attracting attention to the topic.

Logic. The original references for Theorem C.4.1 about the equivalence of MSO and regular languages are [Büc60, Theorems 1 and 2], [Elg61, Theorem on p. 35] and [Tra62, Теорема 10 и 3]. Interestingly enough, the logical corollary that was the initial goal of this work, which is that weak MSO is decidable on the natural numbers with order, is credited to unpublished – and possibly automata-free – work of Ehrenfeucht in [Elg61, Section 5.8]. The connections between MSO and automata are a big topic, a good introduction is the survey paper [Tho97].

A version of Theorem C.4.4 about MSO relabelings can be found in [BE00, Theorem 10], with some minor differences: it is letter-to-letter and uses trees instead of strings. The logical characterisation of the regular functions from Theorem C.4.8 is due to [EH01], where it was shown that MSO transductions have the same expressive power as two-way transducers. In my opinion, this is a foundational paper for the regular functions, since it shows that they can be descibed in at least two different ways, including logic (also, this paper establishes the name "regular" for this class.

The automata characterisation of the first-order fragment that is discussed in Section C.4.4 is a combination of two results: a breakthrough result of Schützenberger which showed that star-free regular expressions are the same as aperidodic monoids [Sch65, Main Property], and an important later addition of McNaughton and Papert which showed that star-free regular expressions are the same as first-order logic [MP71, Theorem 10.5]. Our proof leverages the Krohn-Rhodes Theorem, this option was already noticed in [Mey69, Footnote 1].

Combinators. Theorem C.5.4 was originally proved in [BDK18, Theorem 6.1]. There is an alternative approach to regular expressions for transducers, which does not use types [AFR14, Theorem 15].

Part D: Polyregular functions

In this part, we move to the fourth and final (for this book) step in the transducer ladder, namely the polyregular functions. All transducer classes that we have seen before had linear output size, and the purpose of the polyregular functions is to achieve polynomial output size.

As was the case for the regular functions, probably the quickest way to define the polyregular functions is in terms of prime functions, so we begin with that. We will simply add one more prime function, which has quadratic output size.

Example 36. [Marked squaring] The marked squaring function is illustrated in the following example:

$$1234 \quad \mapsto \quad \underline{1}234\ \underline{12}34\ \underline{123}4\ \underline{1234}.$$

More formally, if the input string has n letters, then it is copied n times, and in the i-th copy the first i letters are underlined[13]. In our example, we have spaces in the output for readability, but the function does not produce them, i.e. the output length is n^2. (Using a variant with spaces would not change the class of polyregular functions.) As was the case for map reverse and map duplicate, marked squaring is not one function, but a family of functions, with one function of type

$$A^* \to (\underbrace{\underline{A} + A}_{\substack{\text{two copies of the} \\ \text{input alphabet, one with} \\ \text{underlines and one without}}})^*$$

for each input alphabet A. $\square$

Definition D.0.1 (Polyregular functions)**.** *A string-to-string function is* polyregular *if it can be obtained as a finite composition of functions, each of which is either regular, or the marked squaring function from Example 36. In other words:*

$$\textit{polyregular} \quad \stackrel{\textit{def}}{=} \quad (\textit{regular} \cup \textit{marked squaring})^*.$$

[13]The underlines might seem mysterious at first, but we will show in Exercise D.0.2 that they are needed.

Of course, the regular functions can be further decomposed into primes, using prime decompositions of regular functions, rational functions, and Mealy machines. The above definition is pleasingly short, but it does not adequately convey the expressive power and algorithmic strength of polyregular functions. To do that, we will present several characterisations of the polyregular functions in this part, using models coming from different fields, such as automata or programming languages.

Before we do that, let us discuss our favourite three properties of transducer classes: continuity, closure under composition, and decidable equivalence. For closure under composition, there is nothing to do, since the class is defined in terms of composition. Decidable equivalence, regrettably, is not known and is a major open problem in this field. We are left with continuity.

Theorem D.0.2. *Polyregular functions are continuous.*

Proof. Since continuous functions are closed under composition, and regular functions are continuous by Theorem C.1.1, it remains to show that marked squaring is continuous. Consider then a composition

$$A^* \xrightarrow{\text{marked squaring}} (A + A)^* \xrightarrow{L} \mathsf{Bool} \tag{14}$$

of marked squaring with a regular language L. We want to show that this composition, which corresponds to the inverse image of L under marked squaring, is a regular language. Suppose that the language L is recognised by a DFA $\mathcal{A}$ with states Q. For a string $w \in A^*$ and a position i in this string, let us write

$$\delta^i_w : Q \to Q$$

for the state transformation of the automaton $\mathcal{A}$ when reading the string that is obtained from w by underlining the first i input positions. We will decompose the language in (14) into two steps, such that the first step is a rational function, and the second step is a regular language. Composing these two steps will give a regular language, by continuity of rational functions, and will hence prove that the language in (14) is regular.

1. Suppose that the input is a string $w \in A^*$. In the first step, we replace each input position i by the state transformation δ^i_w. The result is a string which has the same length as w, but is over the alphabet of state transformations $Q \to Q$. We will argue below that this first step is a rational function.

2. If we compose the state transformations in the output of the first step, then we get the state transformation of the automaton $\mathcal{A}$ on the result of applying marked squaring to w. Therefore, in the second step, we: (a) compose all the state transformations in the output of the first step; and then (b) check if the resulting state transformation takes the initial state of $\mathcal{A}$ to an accepting state. The second step can be implemented by a DFA, whose states are state transformations, and is hence a regular language.

As we have explained, the composition of the above two steps gives the same language as the composition in (14), and hence proves regularity of the latter language.

To conclude the proof, we will explain why the first step is a rational function. We will compute it using an unambiguous one-way transducer with output, in which the states are pairs (δ, γ) of state transformations. The invariant is going to be that: (*) δ represents the underlined state transformation of the past, and γ represents the non-underlined state transformation of the future. The transitions are of the form

$$(\delta_1, \gamma_1) \xrightarrow{a/\delta_2 \cdot \gamma_2} (\delta_2, \gamma_2)$$

such that the following conditions are satisfied:

$$\delta_2 = \delta_1 \cdot \underbrace{\delta_{\underline{a}}}_{\substack{\text{underlined state} \\ \text{transformation of } a \in A}} \qquad \text{and} \qquad \gamma_1 = \underbrace{\delta_a}_{\substack{\text{non-underlined state} \\ \text{transformation of } a \in A}} \cdot \gamma_2.$$

The initial states are those where δ is the identity transformation, and the accepting states are those where γ is the identity transformation. This automaton has a unique accepting run on every input string, which satisfies the invariant (*) described above. □

Exercises

Exercise D.0.1. Show that marked squaring is not compatible with compression, as defined in Exercise B.2.18.

Solution. Suppose that the input to marked squaring is the string a^N with $N = 2^n$, which has a grammar compression of size linear in n, obtained by repeated doubling. We will show that the output of marked squaring has no grammar compression of size subexponential in n. The crucial observation is that the distances between consecutive underlined letters in the output take exponentially many values, while a grammar can only produce a linear number of such distances.

Indeed, the output of marked squaring on a^N is the concatenation of the strings $\underline{a}^i a^{N-i}$, with i ranging from 1 to N. Between the last underlined letter of the i-th block and the first underlined letter of the next block there are exactly $N-i$ letters that are not underlined, and hence the distances between consecutive underlined letters take all the values $1, \ldots, N-1$. It remains to prove the following claim, in which b plays the role of the underlined letter.

Claim D.0.3. *Consider a grammar compression of a string $w \in \{a, b\}^*$. Then*

$$\{\, i \in \{0, 1, \ldots\} \mid \ w \text{ contains an infix of the form } ba^i b \,\}$$

has size at most linear in the size of the grammar compression of w.

Proof. Each rule in the grammar can add at most one new value of i. Indeed, consider a rule $X \to YZ$. Every infix of the form $ba^i b$ in the string generated by X is either

an infix of the string generated by Y, or of the string generated by Z, or it crosses the boundary between them. In the last case, its first letter is the last b of the string generated by Y, and its last letter is the first b of the string generated by Z, and therefore there is at most one such infix. □

Exercise D.0.2. Suppose that we replace marked squaring with the (unmarked) squaring operation from Exercise .0.3. Show that the resulting class of functions is a strict subset of the polyregular functions.

Solution. (Unmarked) squaring is compatible with compression. Indeed, given a grammar compression of w, the length $|w|$ is computed in polynomial time, as a number with polynomially many bits, and a grammar for $w^{|w|}$ is obtained by adding $\mathcal{O}(\log |w|)$ rules that double the string, together with a bounded number of extra rules for the remaining copies, according to the binary representation of $|w|$. Therefore the resulting class of functions would be compatible with compression, while polyregular functions are not, by Exercise D.0.1. The result would continue to hold if we added map lifting to the closure properties, while still keeping the squaring operation unmarked. This is because map lifting preserves compatibility with compression, as we have shown in the solution to Exercise C.0.1.

D.1 For-transducers

We begin our discussion of polyregular functions with *for-transducers*, a model that was introduced in [Boj18, Section 3]. Roughly speaking, a for-transducer is an imperative program, which uses for loops to range over positions in the input string. We will use Python syntax for the code examples.

Example 37. [Marked squaring] Here is the source code of a for-transducer that implements the marked squaring function from Example 36. The program has two nested loops: the variable x in the outer loop ranges over copies of the input string, and the variable y in the inner loop ranges over positions inside a copy.

```
def markedSquare(w):
    for x in positions(w):
        for y in positions(w):
            if y>x and w[y] == 'a':
                output('a')
            if y<=x and w[y] == 'a':
                output('a̲')
            if y>x and w[y] == 'b':
                output('b')
            if y<=x and w[y] == 'b':
                output('b̲')
```

This program contains the main capabilities of for-transducers: we can loop over input positions, we can test the labels and order on these positions, and we can output letters. □

Apart from the features illustrated in the above example, the remaining features of for-transducers are loops that range over positions in the reverse order, and Boolean variables. We illustrate these features in the next two examples.

Example 38. [Reverse] To implement the reverse function, we use a loop that ranges over positions in the reverse order, as in the following program:

```
def reverse(w):
    for x in positions_reverse(w):
        if w[x] == 'a':
            output('a')
        if w[x] == 'b':
            output('b')
```

□

Finally, we can use Boolean variables, as illustrated in the following example.

Example 39. [Parity] The following for-transducer outputs the parity of the length of the input string:

```
  def parity(w):
    q = True  # a Boolean variable
    for x in positions(w):
        q = not q
    #at the end, we output the parity
```

```
    if q:
        output('even')
    else:
        output('odd')
```

Using k Boolean variables, we could compute the length modulo any number $\leq 2^k$. □

The above examples illustrate all features of for-transducers. Nevertheless, we give below a more formal definition of the syntax of the programming language. In a for-transducer, there are two kinds of variables: position variables that range over positions in the input string, and Boolean variables that range over {true, false}. Apart from that, there is a special variable `w` that represents the input string, which is an argument of the for-transducer. There are two kinds of loops

$$\texttt{for x in positions(w)} \qquad \texttt{for x in positions\_reverse(w)}$$

Such a loop creates a position variable `x`, which ranges over positions of the input string `w` in either forward or reverse order. The atomic statements are

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

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

$$\underbrace{\texttt{X}}_{\substack{\text{value of a} \\ \text{Boolean variable}}} \qquad \underbrace{\texttt{x == y} \qquad \texttt{x <= y}}_{\substack{\text{equality and order} \\ \text{tests on positions}}} \qquad \underbrace{\texttt{w[x] ==' a'}}_{\substack{\text{label of} \\ \text{position}}}$$

We will not write variable declarations in the programs, with the types being implicit: the variables bound by the for loops are position variables, and the remaining ones are Booleans. The Booleans are initialised to false.

This completes the syntax of for-transducers[14]. The semantics of a for-transducer is a string-to-string function which is defined in the expected way (we have presented the model as a fragment of Python, and the semantics are consistent). The output size of a for-transducer is polynomial – if it has at most k nested loops, then the output size will be $\mathcal{O}(n^k)$. This bound is not tight: one can show that the map reverse function – which has linear output size – needs two nested loops in order to be implemented by a for-transducer.

Example 40. The order test $x \leq y$ is in fact redundant, and it can be implemented using equality tests only: run a for loop of first-to-last kind, and check which of the

[14] In the examples, we use some syntactic sugar, such as `q = not q` for negation of a Boolean variable, and `output(w[y])` for outputting the label of a position variable. Both of these can be implemented using conditionals from the basic syntax.

variables x or y is seen first in the loop, with the result remembered in a Boolean variable. Similarly, we can implement a successor test $x = y + 1$. As we will see later, we can implement any test $\varphi(x_1, \ldots, x_n)$ that can be defined using a formula of monadic second-order logic. □

D.1.1 Equivalence with polyregular functions

In this section, we show that for-transducers are equivalent to polyregular functions, as defined in Definition D.0.1.

Theorem D.1.1. *A string-to-string function is polyregular if and only if it is computed by a for-transducer.*

Proof. We begin with the left-to-right inclusion

$$\text{polyregular} \subseteq \text{for-transducers}.$$

Before proving this inclusion, we would like to remark that it is non-trivial. In fact, already the fact that for-transducers can compute all regular functions is not obvious, at least if we think of regular functions as being computed by two-way transducers. Indeed, it is not immediately clear how a for-transducer can simulate the movement of the head in a two-way transducer, since the loops in a for-transducer are required to move in a single direction. As we will see, the problem can be solved by the decomposition theorems for regular and rational functions that we have proved in the previous parts of this book.

By the definition of the polyregular functions and the decomposition theorem for rational functions in Theorem B.2.6, we know that each polyregular function can be obtained by composing: (1) marked squaring; (2) map reverse; (3) map duplicate; (4) Mealy machines in both directions; (5) homomorphisms; and (6) the end-of-input function $w \mapsto w\#$. We will show that each of these functions can be computed by a for-transducer, and that for-transducers are closed under composition. Marked squaring was treated in Example 37. A for-transducer for map reverse is given in Figure 2, and a similar one can be written for map duplicate. Mealy machines – in both directions – can be implemented using the same construction as in Example 39, with the Boolean variables used to hold a state. Homomorphisms and the end-of-input function are straightforward to implement. We are left with closure under composition.

Before proving closure under composition, we begin by describing a normal form.

Definition D.1.2 (Prenex form)**.** *A for-transducer is in* prenex form *if it can be written as*

```
for x_1 in τ_1:
  for x_2 in τ_2:
    ...
      for x_k in τ_k:
        body(x_1,...,x_k)
epilogue
```

such that:

```
def mapReverse(w):
  #in this loop, x ranges over separator symbols
  for x in positions(w):
      if w[x]=='#':
          #output the block ended by x, in reverse
          #this variable will have four possible values,
          #which are: 'before', 'separator', 'inside', and '
              after'
          q = 'before'
          for y in positions_reverse(w):
              if y == x:
                  q = 'separator'
              if y < x and q == 'separator':
                  q = 'inside'
              if q == 'inside' and w[y]=='#':
                  q = 'after'
                  output('#')
              if q == 'inside':
                  output(w[y])
          #special code for dealing with leftmost block
          if q == 'inside':
              output('#')
  #an extra loop to process the rightmost block
  q = 'inside'
  for y in positions_reverse(w):
      if w[y]=='#':
          q = 'after'
      if q == 'inside':
          output(w[y])
```

Figure 2: A for-transducer that implements the map reverse function. We use a variable q that can have four possible values; such a variable can be implemented using two Boolean variables.

- *each* τ_i *is either* `positions(w)` *or* `positions_reverse(w)`*;*
- *the subprograms* body *and* epilogue *do not use any loops;*
- *the subprogram* body *produces at most one output letter in each loop iteration.*

The subprogram body is allowed to refer to the position variables $x_1, \ldots, x_k$, but the subprogram epilogue is not. There is, however, some form of communication between the two programs, since both of these subprograms refer to the same Boolean variables, and the epilogue can be sensitive to the changes in the Boolean variables that were effected by the various iterations of the body. For example, the following program which checks for emptiness is in prenex form:

```
def isEmpty(w):
  for x in positions(w):
    q = True
  if q:
    output('not␣empty')
```

Lemma D.1.3. *Every for-transducer is equivalent to one in prenex form.*

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

Using the prenex form, we can show that for-transducers are closed under composition.

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

Proof. The construction is easiest to see if we assume that both composed for-transducers are in the prenex form of Lemma D.1.3, as follows:

First for-transducer

```
for x_1 in τ_1:
  for x_2 in τ_2:
    ...
      for x_k in τ_k:
        body1(x_1, ..., x_k)
epilogue1
```

Second for-transducer

```
for y_1 in σ_1:
  for y_2 in σ_2:
    ...
      for y_ℓ in σ_ℓ:
        body2(y_1, ..., y_ℓ)
epilogue2
```

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

Our goal is to write a for-transducer for the composed function, in which the second for-transducer is executed on the output of the first for-transducer. In this composition, the variables $y_1, \ldots, y_\ell$ of the second for-transducer range over outputs of the first one, which are in one-to-one correspondence with those tuples $(x_1, \ldots, x_k)$ where the body of the first transducer produces an output letter. Therefore, each loop in the second transducer can be replaced by k loops, giving a for-transducer with $\ell \times k$ variables which looks like this (when multiplying kinds of loops, we assume that forward loops are represented by the number 1 and reverse loops are represented by the number -1):

Composed for-transducer

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

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

This completes the proof of the left-to-right inclusion in the theorem, since we have shown that for-transducers are closed under composition, and that they can compute all the prime polyregular functions.

Let us now move to the other inclusion

$$\text{for-transducers} \subseteq \text{polyregular}.$$

Thanks to Lemma D.1.3, it suffices to show that for-transducers in prenex form compute polyregular functions. Consider such a for-transducer, with its code being

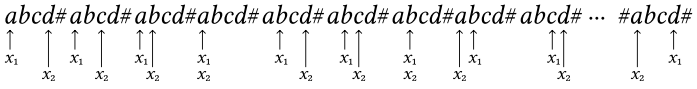
```
for x_1 in τ_1:
  for x_2 in τ_2:
    ...
      for x_k in τ_k:
        body(x_1, ..., x_k)
epilogue
```

Each iteration of the body is called for some valuation of the position variables. Using the representation described in Section C.4, an input string $w \in A^*$ together with a valuation of the variables $x_1, \ldots, x_k$ can be represented as a string over an extended alphabet:

$$w \otimes x_1 \otimes \cdots \otimes x_k \in (A \times 2^k)^*.$$

We will simulate the for-transducer by a composition of the following functions:

1. In the first phase, we loop through all valuations of the variables $x_1, \ldots, x_k$, in the order given by the nested for loops, and for each one we output the string representation of the valuation, as described above. The output of this phase is illustrated in the following picture, assuming that the input string is $abcd$ and there are two position variables x_1 and x_2, of types left-to-right and right-to-left respectively:



   The construction producing this output is by induction on the number of variables. If we have computed the output for i variables, corresponding to the outermost i loops, then the output for $i + 1$ variables is obtained as follows: apply marked squaring, and then use a rational function to filter out the ill-formatted blocks. The details are left to the reader. In particular, if the loop for the variable x_{i+1} is of the reverse kind, then we need a reverse version of marked squaring

$$abcd \quad \mapsto \quad \underline{a}bcd \; a\underline{b}cd \; ab\underline{c}d \; abc\underline{d},$$

   which can be implemented using marked squaring, two instances of reverse, and a rational function.

2. Once we have the output of the first phase, the output of the nested loops in the for-transducer can be computed by a rational function, which maintains the Boolean variables of the for-transducer in its state, scans each block (corresponding to one iteration of the body), and produces at most one output letter per block. The epilogue can then be implemented using one more pass, which only needs to scan one of the copies of the input string.

This completes the translation from for-transducers to polyregular functions, and therefore also the proof of Theorem D.1.1. □

Exercises

Exercise D.1.1. Consider a first-order formula that defines a language. Show that the same language, viewed as a string-to-string function with outputs "yes" and "no", can be computed by a for-transducer whose source code is polynomial in the size of the formula.

Solution. The translation replaces quantifiers by for loops, and the truth values of subformulas by Boolean variables. For a subformula $\exists x\, \psi$, we use a Boolean variable which is set to false, and then a loop over all positions x, which sets the variable to true whenever the translation of ψ evaluates to true; a universal quantifier is treated dually. The atomic formulas – the order and equality tests on positions, and the tests on labels – are available directly in the syntax of for-transducers, and the Boolean connectives are available in the conditions. Finally, after the outermost loop, the epilogue outputs "yes" or "no", depending on the Boolean variable for the whole formula.

The size is polynomial, and in fact linear: each quantifier of the formula contributes one loop and one Boolean variable, and each atomic subformula contributes one test. Note that this uses the fact that the loops of a for-transducer can be nested arbitrarily, and that Boolean variables can be reset inside a loop, so that the inner quantifiers are re-evaluated for every valuation of the outer ones.

Exercise D.1.2. Show that there is no algorithm that runs in elementary time and solves the following computational version of continuity of for-transducers:

- **Input.** Source code of a for-transducer f and an NFA over its output alphabet;
- **Output.** NFA for its preimage.

Solution. Already the size of the output NFA can be non-elementary, and hence no algorithm can produce it in elementary time. Indeed, consider a first-order sentence φ as in Exercise C.4.2, of size polynomial in n, whose language contains exactly one string, of length at least $\exp(n)$. By Exercise D.1.1, there is a for-transducer f_φ, of size polynomial in φ, which outputs "yes" or "no" according to φ. Apply the hypothetical algorithm to f_φ and to the one-state NFA which accepts the language $\{\text{yes}\}$. The answer is an NFA for the language of φ. An NFA whose language contains a string of length ℓ and no longer string must have more than ℓ states, since otherwise the accepting run would repeat a state and could be pumped. Therefore the answer has at least $\exp(n)$ states, which is not elementary in the size of the input.

Exercise D.1.3. A *forward for-transducer* is a for-transducer in which all loops are of the first-to-last kind. Show that the functions computed by forward for-transducers are exactly the composition closure of marked squaring and rational (not regular) functions.

Solution. Both inclusions follow the proof of Theorem D.1.1, with an eye on the direction of the loops.

From forward for-transducers to compositions. Consider the proof of Theorem D.1.1, which translates a for-transducer into a composition of marked squaring and rational functions. In the first phase, the valuations of the position variables are enumerated, using one application of marked squaring per variable, followed by a rational function which filters out the ill-formatted blocks. The only place where a function beyond the rational ones is used is the reverse variant of marked squaring, which is needed for a variable whose loop is of the last-to-first kind, and which is implemented using string reversal. If all loops are forward, this never happens, and the first phase is a composition of marked squarings and rational functions. The second phase is a rational function, as is the epilogue, and hence the whole for-transducer is a composition of marked squarings and rational functions.

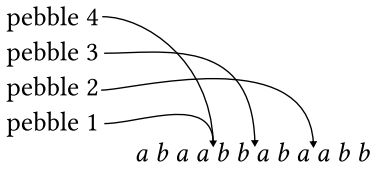
From compositions to forward for-transducers. For the converse inclusion, we need three things. First, marked squaring is computed by a forward for-transducer, as we have seen in Example 37.

Second, every rational function is computed by a forward for-transducer. By Theorem B.2.3, a rational function is computed by a bimachine, and, as observed after the definition of bimachines, we may assume that the suffix automaton is run on the suffix in the forward direction, since the only role of the suffix automaton is to split the suffixes into finitely many regular languages. The forward for-transducer then has an outer forward loop over the positions x, which maintains the state of the prefix automaton in Boolean variables. Inside this loop, and before producing any output, an inner forward loop over the positions y computes the state of the suffix automaton on the suffix that starts after x: the Boolean variables for this state are reset at the beginning of the inner loop, and are updated only for those y that satisfy $x < y$. Once both states are known, the body outputs the corresponding string. The gap after the last position is handled by the epilogue.

Third, forward for-transducers are closed under composition. This is the construction of Lemma D.1.4: each loop of the second transducer is replaced by a group of loops, one for each loop of the first transducer, and the direction of a new loop is the product of the directions of the two loops that it comes from. If both transducers are forward, then all these products are forward.

D.2 Pebble transducers

We now describe a second model for polyregular functions, which is called *pebble transducers*. A pebble transducer extends a two-way transducer by having not one head, but a stack of *pebbles* that point to the input string, as in the following picture:



Similarly to one-way and two-way transducers (but unlike for-transducers), our pebbles point to gaps between positions, and not positions. Importantly, the pebbles are organised in a stack, with pebble 1 at the bottom of the stack, and a pebble can only be modified by the transducer if it is at the top of the stack[15]. The stack can be potentially empty, and – more importantly – its height is bounded by a constant k that is given in the pebble automaton. (There are models with unbounded numbers of pebbles [EHS07].) We use the name *head* for the topmost pebble on the stack. Apart from the pebble stack, the automaton has a state from a finite set. To update its configuration, the pebble transducer looks at its state, and the answers to the following *questions*, where $i \in \{1, \ldots, k\}$ range over pebble numbers:

1. what are the two input letters adjacent to pebble i?
2. which other pebbles are in the same place as pebble i?

Formally speaking, if the input alphabet is A, then the set of possible answers to the above questions is described by the set

$$\overbrace{\sum_{\ell \in \{0,\ldots,k\}}}^{\substack{\text{how many}\\ \text{pebbles are}\\ \text{in the stack}}} \overbrace{\prod_{i \in \{1,\ldots,\ell\}}}^{\substack{\text{for each pebble}\\ \text{on the stack}\\ \text{we answer the}\\ \text{questions}}} \overbrace{(A+1) \times (A+1)}^{\substack{\text{the two letters}\\ \text{adjacent to the}\\ \text{pebble, with 1 meaning}\\ \text{that there is no letter}}} \times \overbrace{\mathsf{P}\{1, \ldots, \ell\}}^{\substack{\text{the set of pebbles}\\ \text{in the same place}\\ \text{as pebble } i}}.$$

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

1. output letter a;
2. move the head by one position to the left/right;
3. push a new pebble, placed in the first gap, onto the pebble stack;

[15] Without stack discipline the model would be the same as multi-head automata, which correspond to logarithmic space Oscar H. Ibarra, "Characterizations of Some Tape and Time Complexity Classes of Turing Machines in Terms of Multihead and Auxiliary Stack Automata", 1971, Corollary 3.5.

4. pop the topmost pebble from the pebble stack;

5. terminate.

Formally speaking, a *configuration* of the pebble transducer is defined to be an input string, together with a state and a stack $x_1, \ldots, x_\ell$ of gaps in the input string whose height ℓ is in $\{0, \ldots, k\}$.

The transducer begins in the configuration where the stack is empty and the state is a designated initial state. Next, it keeps on updating its configuration according to the above transition rules. The resulting sequence of configurations is called the *run* of the pebble transducer, and it stops when the "terminate" action is executed. The *output string* is defined to be the sequence of output letters that are produced by output actions during the run, in the order that they were executed. The output string is undefined if one of the following errors occurs: (a) the run never terminates; or (b) the head moves out of the input string; or (c) the stack bound is exceeded, by either popping in an empty stack or pushing pebble $k + 1$. We will only care about pebble transducers which define total functions, which means that for every input string none of the errors (a) – (c) occur. We use the name k-pebble transducer for a pebble transducer where the bound on the stack height is k. In the case of $k = 1$, we recover two-way transducers.

D.2.1 Continuity

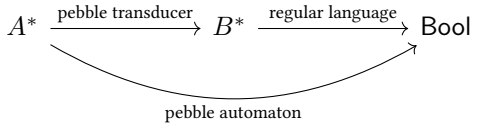
In Theorem D.2.4 later in this section, we will show that pebble transducers compute the same string-to-string functions as for-transducers, and therefore they are continuous and closed under composition. Nevertheless, it is instructive to give a direct proof of continuity, which does not rely on the equivalence with for-transducers, and which will also explain the relevance of the stack discipline. We begin by showing that without stack discipline, the model would not be continuous.

Example 41. [Importance of stack discipline] Consider the non-regular language $\{ a^n b^n \mid n \in \mathbb{N} \}$. We will show that this language can be computed by an extension of pebble transducers that does not have stack discipline. (The transducer will output "yes" or "no" depending on whether the input string is in the language or not.) The transducer begins by placing the pebble one at the beginning of the input string, and pebble two at the end. Then, the transducer enters a loop, such that in each iteration pebble one is moved one step to the right, and pebble two is moved one step to the left, until both pebbles meet. We require that, before meeting, the labels traversed by pebble one have label a and the labels traversed by pebble two have label b. Formally speaking, this transducer needs a third pebble, which shuttles between the first two pebbles to implement the loop. Stack discipline is violated by this transducer, since pebble one is moved without removing pebble two. As we have remarked in the footnote on stack discipline, without stack discipline the model can compute the same functions as deterministic Turing machines with logarithmic space. □

We now show that with stack discipline, the model becomes continuous.

Theorem D.2.1. *Pebble transducers compute continuous functions.*

Proof. In the proof, it will be simpler to deal with *pebble automata*, which are the variant of pebble transducers that compute languages, i.e. string-to-Boolean instead of string-to-string. A pebble automaton is defined like a pebble transducer, except that instead of outputting letters, at some point during its run it makes a decision which is either "accept" or "reject". We first observe that the composition of a pebble transducer with a regular language is a pebble automaton, as explained in the following diagram:



This is easy to show, by using a product of the pebble transducer with a DFA that recognises the regular language. Therefore, in order to prove the theorem, it is enough to show that pebble automata recognise only regular languages. (They also recognise all regular languages, since they generalise DFA.)

Consider a pebble automaton that has k pebbles and states Q. If the input string has length n, then the set of configurations can be described by the polynomial

$$\tau(n) = \sum_{\ell \in \{0,\ldots,k\}} Q \times (1+n)^{\ell}. \tag{15}$$

The main observation in the proof is that reachability of configurations can be defined in MSO, as stated in the following lemma.

Lemma D.2.2. *Let τ be as in* (15)*. There is an* MSO *formula*

$$\varphi(s : \tau, t : \tau)$$

which holds if there is a run that begins in configuration s and ends in configuration t.

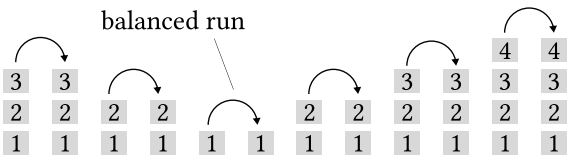
Before proving the lemma, we use it to conclude the proof of the theorem. By the lemma, the language recognised by a pebble automaton is defined by an MSO formula which says that there is a run from the initial configuration to an accepting configuration. Since MSO can only define regular languages thanks to Theorem C.4.1, it follows that the language recognised by the automaton is regular, as required in the theorem. It remains to prove the lemma.

Proof. The most natural idea would be to define one-step reachability using a formula ψ, and to then express reachability as a fixpoint, as follows

$$\underbrace{\forall X : 2^{\tau}}_{\substack{\text{for every}\\ \text{set } X \text{ of}\\ \text{configurations}}} \quad \underbrace{(s \in X}_{\substack{\text{if } X\\ \text{contains}\\ \text{the source}}} \wedge \underbrace{\forall c : \tau\, \forall d : \tau\, \psi(c,d) \wedge c \in X \Rightarrow d \in X)}_{\text{and is closed under taking transitions}} \quad \Rightarrow \quad \underbrace{t \in X}_{\substack{\text{then } X\\ \text{contains}\\ \text{the target}}}.$$

Unfortunately, this approach does not work directly, since we cannot quantify over sets of type 2^{τ}. This is because τ contains tuples of positions (if the automaton has $k > 1$ pebbles), and monadic second-order logic can only quantify over sets of individual positions. To work around this issue, we will have to appeal to the stack discipline.

The proof is based on the analysis of special runs, which we call *balanced runs*: this is a run in which the source and target have the same height, and the top pebble is never popped (but it can be moved). Every run of the pebble automaton can be decomposed into a sequence of at most $4k$ runs, each of which is either a single transition, or a balanced run, as explained in the following picture, which shows the evolution of the stack:



Since the decomposition has constant length, it remains to show that we can define in MSO runs which are either a single transition, or a balanced run. For single transitions, this is achieved by simply formalising the semantics of a pebble automaton in the logic. The interesting case is that of balanced runs.

When formalising balanced runs, we will think of a configuration with stack of height $\ell > 0$ as being a pair of the form (x, y) where

$$\underbrace{x : (1+n)^{\ell-1}}_{\substack{\text{prefix of the stack} \\ \text{that has height } \ell-1}} \qquad \underbrace{y : \overbrace{Q \times (1+n)}^{\substack{\text{we write } \sigma \\ \text{for this type}}}}_{\text{the head}}.$$

The point of this decomposition is that a balanced run modifies y but not x. The following claim shows that balanced runs can be defined in MSO, and thus concludes the proof of the lemma.

Claim D.2.3. *For every $\ell \in \{1, \ldots, k\}$, there is an MSO formula*

$$\varphi_\ell(x : (1+n)^{\ell-1}, y_1 : \sigma, y_2 : \sigma)$$

which holds if and only if there is a balanced run from (x, y_1) to (x, y_2).

Proof. The proof is by induction on ℓ, but in the opposite order: the induction starts with $\ell = k$ and proceeds down until $\ell = 1$. The base case is proved in the same way as the induction step, so we will only describe the induction step. Define a *basic balanced run* to be a balanced run which is either: (a) a single transition that moves pebble ℓ; or (b) a run that starts by pushing pebble $\ell+1$, then does a balanced run, and finishes with popping pebble $\ell+1$. A balanced run is a composition of basic balanced runs. Using the induction assumption (if $\ell < k$), we can write an MSO formula

$$\psi(x : (1+n)^{\ell-1}, y_1 : \sigma, y_2 : \sigma)$$

which says that there is a basic balanced run from (x, y_1) to (x, y_2). To finish the proof of the claim, we will formalise in MSO the composition of basic balanced runs, using a fixpoint similar to the one at the beginning of the proof of the lemma.

$$\underbrace{\forall Y : 2^\sigma}_{\substack{\text{for every}\\ \text{set } Y \text{ of}\\ \text{head}\\ \text{positions}}} \quad \underbrace{(y_1 \in Y}_{\substack{\text{if } Y\\ \text{contains}\\ \text{the source}}} \wedge \underbrace{\forall c : \sigma\, \forall d : \sigma\, \psi(x, c, d) \wedge c \in Y \Rightarrow d \in Y)}_{\text{and is closed under taking basic balanced runs}} \quad \Rightarrow \quad \underbrace{y_2 \in Y}_{\substack{\text{then } Y\\ \text{contains}\\ \text{the target}}}.$$

The difference with respect to the erroneous fixpoint construction at the beginning of the lemma is that the type σ is linear, and therefore quantification over sets of type 2^σ is allowed in MSO, as explained below:

$$2^\sigma = 2^{Q \times (1+n)} = \overbrace{(2^1 \times 2^n) \times \cdots \times (2^1 \times 2^n)}^{\substack{\text{quantifying over a subset of } Q \times (1+n) \text{ is like}\\ \text{quantifying over } |Q| \text{ bits and } |Q| \text{ subsets of } n}}.$$

This completes the proof that balanced runs can be defined in MSO. □

As we have remarked at the beginning of the proof of the lemma, every run can be decomposed into a constant number of runs, each of which is either a single transition or a balanced run. Since both of these can be defined in MSO, it follows that reachability can be defined in MSO, as required in the lemma. □

□

D.2.2 Equivalence with for-transducers

We now prove that pebble transducers compute the same string-to-string functions as for-transducers, and therefore they compute exactly the polyregular functions.

Theorem D.2.4. *Pebble transducers and for-transducers compute the same string-to-string functions.*

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

The interesting direction is simulating a pebble transducer with a for-transducer. The difficulty is that a pebble transducer can move its head both ways, while a loop in a for-transducer has a fixed direction, which is either first-to-last or last-to-first. Therefore, converting a pebble transducer into a for-transducer can be seen as a sort of "untwisting", to use the terminology of [Bas+17]. A variant of untwisting was already presented when decomposing two-way transducers into primes in Theorem C.2.9, and our proof will draw heavily on the ideas behind that decomposition. It is worth pointing out that this untwisting will increase the depth of recursion. For example, map reverse is a function with linear output, which can be implemented by a pebble transducer with one pebble, but the corresponding for-transducer in Figure 2 has three loops, two of them nested.

To transform a pebble transducer into a for-transducer, we will show that a for-transducer can be used to produce the entire computation of a pebble transducer, represented as a list of configurations. A configuration of a pebble transducer can be represented as a string over an extended alphabet, as explained in the following example:

$$\underbrace{q}_{\text{state}}\, abaax_1x_3ababa\, \underbrace{x_2}_{\substack{\text{pebble } i\\ \text{is letter } x_i}}\, ababaax_4abababa.$$

More formally, the *string representation* of a configuration with ℓ pebbles is a string in

$$Q \cdot (A \cup \{x_1, \ldots, x_\ell\})^*$$

where each pebble name from $\{x_1, \ldots, x_\ell\}$ appears exactly once, and if there are consecutive letters that represent pebbles, then they are listed in increasing order. Define the *string representation* of a run to be the concatenation of the string representations of the configurations used in the run.

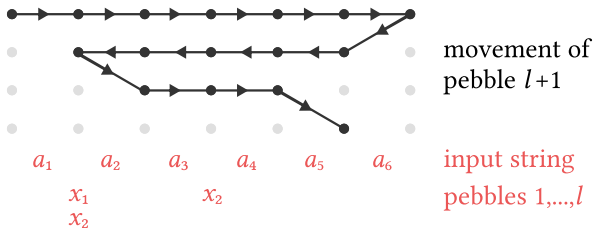
To prove that the computation of a pebble transducer can be produced by a for-transducer, we will appeal to the tree structure in a run of a pebble transducer, which we now explain. For a configuration with stack of height ℓ that appears in this run, its *parent* is defined to be the most recent earlier configuration with stack of height $\ell - 1$. If we assume that the initial configuration is the unique configuration of height 0 in every run, then this parent relation defines a tree structure on the configurations, with the initial configuration being the root. The assumption that the initial configuration is the unique configuration of height 0 in every run can be made without loss of generality, since any pebble transducer can be easily transformed into one that satisfies this assumption. The main technical ingredient in the proof of Theorem D.2.4 is the following lemma, which says that the children of a configuration in this tree structure can be computed by a for-transducer.

Lemma D.2.5. *For every k-pebble transducer there is a for-transducer which inputs a string representation of a configuration, and outputs all children of this configuration, represented as the concatenation of their string representations in order of execution.*

Before proving the lemma, we show how to use it to conclude the proof of the theorem. Functions computed by for-transducers are closed under composition, which was shown in Lemma D.1.4. They are also closed under the map combinator, which is left as an exercise for the reader. Thanks to these closure properties and the lemma, we can use a for-transducer to output all configurations used in a run: first we output the initial configuration, then we append to it the list of its children, then we append (using map) to each child a list of all of its children, and so on k times, until we have output all configurations in the run. Once we have a list of all configurations, the output string can be produced using a rational function which scans each configuration and outputs the letter (if any) that is produced when its transition is executed. It remains to prove the lemma.

Proof of Lemma D.2.5. Fix some k-pebble transducer for the rest of the proof. Suppose that the input is a configuration with stack of height ℓ. If $\ell = k$, then there is nothing

to do, since the stack is already at full height and there can be no children. Suppose now that $\ell < k$. The children of the input configuration can be described as a directed graph, in which an edge represents a run between two configurations of height $\ell + 1$ that are not separated by any configuration of height $\ell + 1$. Here is a picture of this graph, in which $\ell = 3$.



The picture also contains additional annotation (in red), which gives us the input string and the positions of pebbles $\{1, \ldots, \ell\}$ that stay fixed throughout the run. We use the name *child configuration graph* for this graph. The child configuration graph can be represented as a string over a fixed finite alphabet, whose length is the same as the input string. This string contains the red annotation of the input string together with the positions of pebble $\ell + 1$ in each child. Using this string representation, it is meaningful to talk about a for-transducer which outputs the child configuration graph.

Claim D.2.6. *There is a for-transducer which inputs the string representation of a configuration, and outputs the string representation of its child configuration graph.*

Proof. In fact, we will show that the function in the claim is already a rational function, and therefore a special case of a for-transducer. To prove this, as our model for rational functions we use unambiguous one-way transducers with output. The transducer will simply guess the output string, and then check if it is correct. To complete the proof, it remains to show that the set of string representations of child configuration graphs is a regular language. Since regular languages are exactly those that can be defined in MSO by Theorem C.4.1, it is enough to show that set of string representations of child configuration graphs is definable in MSO. This follows from Lemma D.2.2, in which we showed that reachability of configurations can be defined in MSO. □

Thanks to the above claim, in order to complete the proof of the lemma, it is enough to show that there is a for-transducer which inputs the string representation of the child configuration graph, and outputs the concatenation of the string representations of the configurations that are stored in this graph, as stated in the following claim.

Claim D.2.7. *There is a for-transducer which inputs the string representation of a child configuration graph, and outputs the concatenation of the string representations of the corresponding child configurations.*

Proof. This is essentially the same proof as in Lemma C.2.12, in which we transformed a configuration graph into its output string. The difference is that in this claim, for each vertex in the child configuration graph we need to output the entire child configuration, which requires producing a copy of the input string. In particular, the transformation has quadratic output size.

As in Lemma C.2.12, the lemma is proved by induction on the width of the input graph, i.e. the maximal number of times that a given input gap is visited. In particular, in the inductive argument we will use subgraphs of the child configuration graph, in which the input is restricted by removing edges. In the induction basis, when the width is 1, then the path in the child configuration graph goes in one direction only, and the output can be produced by a for-transducer that has an outer loop which traverses the snake, and an inner loop which is used to copy the input string. (Depending on the direction of the path, the outer loop is either first-to-last or last-to-first.) In the induction step, we first deal with the case of looping snakes, and then we decompose a general snake into a sequence of loops, exactly as we did in the proof of Lemma C.2.12. □

By composing the for-transducers in Claims D.2.6 and D.2.7, we get a for-transducer which inputs the string representation of a configuration, and outputs the concatenation of the string representations of its children, as required in the lemma. □

D.3 References

Historically the first model for this class was the pebble transducers.

Pebble transducers. Automata with pebbles first appeared as language acceptors in [GH96, Definition 4.1], where it was shown that they accept exactly the regular languages[GH96, Theorem 4.2]. This corresponds to saying that pebble transducers are continuous, i.e. Theorem D.2.1. The transducer variant was introduced in [MSV03, Section 3.1], where a tree-to-tree variant was used because of XML applications. The string-to-string variant was explicitly identified by Engelfriet and Hoogeboom, who proved that the corresponding class of functions is closed under composition [EM02, Theorem 2]. (The composition closure does not hold for the tree variant as defined in [MSV03], because of exponential size outputs.)

It is worth pointing out that there are two ways of counting pebbles. Like [MSV03] but unlike [GH96; EM02], we count the head as just another pebble. Therefore, a transducer with k pebbles in our notation becomes, in the notation of [GH96; EM02], a transducer with $k-1$ pebbles and a head. The motivation for our choice is that a transducer with k pebbles in our notation will have output size $\mathcal{O}(n^k)$. This is not tight by a wide margin: for every k there is a polyregular function of quadratic growth that cannot be computed by a pebble transducer with k pebbles[Boj22, Theorem 3.3].

For-transducers and primes. For-transducers and the prime polyregular functions were introduced in the unpublished manuscript [Boj18, Sections 1 and 3]. The manuscript also contains other characterisations of the class, and introduces the name "polyregular". A published overview of that manuscript is [Boj22]. In a future version of this book, we will also include a characterisation of the polyregular functions in terms of the λ-calculus [Boj18, Section 4]. Another planned addition is the logical characterisation from [BKL19, Theorem 7], which uses (not necessarily linear) MSO interpretations.

References

[AČ10] Rajeev Alur and Pavol Černý. "Expressiveness of Streaming String Transducers". In: *FSTTCS 2010*. Vol. 8. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2010, pp. 1–12. DOI: 10.4230/LIPIcs.FSTTCS.2010.1. URL: https://doi.org/10.4230/LIPIcs.FSTTCS.2010.1.

[AFR14] Rajeev Alur, Adam Freilich, and Mukund Raghothaman. "Regular Combinators for String Transformations". In: *CSL-LICS 2014*. A full version, with the proofs and constructions omitted for space, is available as arXiv:1402.3021. ACM, 2014, 9:1–9:10. DOI: 10.1145/2603088.2603151. URL: https://doi.org/10.1145/2603088.2603151.

[AHU69] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. "A General Theory of Translation". In: *Mathematical Systems Theory* 3.3 (Sept. 1969), pp. 193–221. DOI: 10.1007/BF01703920. URL: https://doi.org/10.1007/BF01703920.

[AL85] Michael H. Albert and John Lawrence. "A Proof of Ehrenfeucht's Conjecture". In: *Theoretical Computer Science* 41 (1985), pp. 121–123. DOI: 10.1016/0304-3975(85)90066-0. URL: https://doi.org/10.1016/0304-3975(85)90066-0.

[AU70] Alfred V. Aho and Jeffrey D. Ullman. "A Characterization of Two-Way Deterministic Classes of Languages". In: *Journal of Computer and System Sciences* 4.6 (Dec. 1970), pp. 523–538. DOI: 10.1016/S0022-0000(70)80027-7. URL: https://doi.org/10.1016/S0022-0000(70)80027-7.

[Bas+17] Félix Baschenis, Olivier Gauwin, Anca Muscholl, and Gabriele Puppis. "Untwisting two-way transducers in elementary time". In: *LICS 2017*. IEEE Computer Society, 2017, pp. 1–12. DOI: 10.1109/LICS.2017.8005138. URL: https://doi.org/10.1109/LICS.2017.8005138.

[BDK18] Mikołaj Bojańczyk, Laure Daviaud, and Shankara Narayanan Krishna. "Regular and First-Order List Functions". In: *LICS 2018*. An extended version, 32 pages against the 10 of the proceedings version, is available as arXiv:1803.06168; numbering differs between the two. ACM, 2018, pp. 125–134. DOI: 10.1145/3209108.3209163. URL: https://doi.org/10.1145/3209108.3209163.

[BE00] Roderick Bloem and Joost Engelfriet. "A Comparison of Tree Transductions Defined by Monadic Second Order Logic and by Attribute Grammars". In: *Journal of Computer and System Sciences* 61.1 (Aug. 2000), pp. 1–50. DOI: 10.1006/jcss.1999.1684. URL: https://doi.org/10.1006/jcss.1999.1684.

[Ben+17] Michael Benedikt, Timothy Duff, Aditya Sharad, and James Worrell. "Polynomial automata: Zeroness and applications". In: *LICS 2017*. IEEE Computer Society, 2017, pp. 1–12. DOI: 10.1109/LICS.2017.8005101. URL: https://doi.org/10.1109/LICS.2017.8005101.

[Ber79] Jean Berstel. *Transductions and Context-Free Languages*. Cited according to the electronic edition of 14 December 2009, which contains the first four chapters and is available at https://www-igm.univ-mlv.fr/~berstel/LivreTransductions/LivreTransductions14dec2009.pdf. Stuttgart: Teubner, 1979.

[BH77] Meera Blattner and Tom Head. "Single-valued a-transducers". In: *Journal of Computer and System Sciences* 15.3 (Dec. 1977), pp. 310–327. DOI: 10.1016/S0022-0000(77)80033-0. URL: https://doi.org/10.1016/S0022-0000(77)80033-0.

[BKL19] Mikołaj Bojańczyk, Sandra Kiefer, and Nathan Lhote. "String-to-String Interpretations With Polynomial-Size Output". In: *ICALP 2019*. 2019. DOI: 10.4230/LIPIcs.ICALP.2019.106. URL: https://doi.org/10.4230/LIPIcs.ICALP.2019.106.

[Boj18] Mikołaj Bojańczyk. "Polyregular Functions". In: *CoRR* abs/1810.08760 (2018). DOI: 10.48550/arXiv.1810.08760. URL: https://doi.org/10.48550/arXiv.1810.08760.

[Boj19] Mikołaj Bojańczyk. "The Hilbert Method for Transducer Equivalence". In: *ACM SIGLOG News* 6.1 (Feb. 2019), pp. 5–17. DOI: 10.1145/3313909.3313911. URL: https://doi.org/10.1145/3313909.3313911.

[Boj22] Mikołaj Bojańczyk. "Transducers of polynomial growth". In: *LICS 2022*. ACM, 2022, 1:1–1:27. DOI: 10.1145/3531130.3533326. URL: https://doi.org/10.1145/3531130.3533326.

[Boj25] Mikołaj Bojańczyk. *An Automata Toolbox*. Lecture notes. 2025. URL: https://www.mimuw.edu.pl/~bojan/paper/automata-toolbox-book.

[BS20] Mikołaj Bojańczyk and Rafał Stefański. "Single-Use Automata and Transducers for Infinite Alphabets". In: *ICALP 2020*. Vol. 168. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020, 113:1–113:14. DOI: 10.4230/LIPIcs.ICALP.2020.113. URL: https://doi.org/10.4230/LIPIcs.ICALP.2020.113.

[Büc60] J. Richard Büchi. "Weak Second-Order Arithmetic and Finite Automata". In: *Zeitschrift für mathematische Logik und Grundlagen der Mathematik* 6.1-6 (1960), pp. 66–92. DOI: 10.1002/malq.19600060105. URL: https://doi.org/10.1002/malq.19600060105.

[Cho77] Christian Choffrut. "Une caracterisation des fonctions sequentielles et des fonctions sous-sequentielles en tant que relations rationnelles". In: *Theoretical Computer Science* 5.3 (Dec. 1977), pp. 325–337. DOI: 10.1016/0304-3975(77)90049-4. URL: https://doi.org/10.1016/0304-3975(77)90049-4.

[CJ77] Michal P. Chytil and Vojtěch Jákl. "Serial Composition of 2-Way Finite-State Transducers and Simple Programs on Strings". In: *ICALP 1977*. Ed. by Arto Salomaa and Magnus Steinby. Vol. 52. Lecture Notes in Computer Science. Springer, 1977, pp. 135–147. DOI: 10.1007/3-540-08342-1_11. URL: https://doi.org/10.1007/3-540-08342-1_11.

[Dub41] Paul Dubreil. "Contribution à la théorie des demi-groupes. I". In: *Mémoires de l'Académie des Sciences de l'Institut de France* 63.3 (1941). Tome 63 covers 1936–39; the memoir is no. 3 of the volume, with its own pagination, pp. 1–52. URL: https://gallica.bnf.fr/ark:/12148/bpt6k3278g/f355.

[EH01] Joost Engelfriet and Hendrik Jan Hoogeboom. "MSO Definable String Transductions and Two-Way Finite-State Transducers". In: *ACM Transactions on Computational Logic* (2001). DOI: 10.1145/371316.371512. URL: https://doi.org/10.1145/371316.371512.

[EHS07] Joost Engelfriet, Hendrik Jan Hoogeboom, and Bart Samwel. "XML transformation by tree-walking transducers with invisible pebbles". In: *PODS 2007*. 2007. DOI: 10.1145/1265530.1265540. URL: https://doi.org/10.1145/1265530.1265540.

[Eil74] Samuel Eilenberg. *Automata, Languages, and Machines. Volume A*. Vol. 59-A. Pure and Applied Mathematics. New York: Academic Press, 1974. ISBN: 978-0-12-234001-7.

[Elg61] Calvin C. Elgot. "Decision problems of finite automata design and related arithmetics". In: *Transactions of the American Mathematical Society* 98.1 (1961), pp. 21–51. DOI: 10.1090/S0002-9947-1961-0139530-9. URL: https://doi.org/10.1090/S0002-9947-1961-0139530-9.

[EM02] Joost Engelfriet and Sebastian Maneth. "Two-Way Finite State Transducers with Nested Pebbles". In: *Mathematical Foundations of Computer Science 2002*. 2002. DOI: 10.1007/3-540-45687-2_19. URL: https://doi.org/10.1007/3-540-45687-2_19.

[EM65] C. C. Elgot and J. E. Mezei. "On relations defined by generalized finite automata". In: *IBM Journal of Research and Development* 9.1 (Jan. 1965), pp. 47–68. DOI: 10.1147/rd.91.0047. URL: https://doi.org/10.1147/rd.91.0047.

[EM99] Joost Engelfriet and Sebastian Maneth. "Macro Tree Transducers, Attribute Grammars, and MSO Definable Tree Translations". In: *Information and Computation* 154.1 (Oct. 1999), pp. 34–91. DOI: 10.1006/inco.1999.2807. URL: https://doi.org/10.1006/inco.1999.2807.

[ES69] Samuel Eilenberg and M. P. Schützenberger. "Rational Sets in Commutative Monoids". In: *Journal of Algebra* 13.2 (Oct. 1969), pp. 173–191. DOI: 10.1016/0021-8693(69)90070-2. URL: https://doi.org/10.1016/0021-8693(69)90070-2.

[GH96] Noa Globerman and David Harel. "Complexity Results for Two-Way and Multi-Pebble Automata and their Logics". In: *Theoretical Computer Science* 169.2 (Dec. 1996), pp. 161–184. DOI: 10.1016/S0304-3975(96)00119-3. URL: https://doi.org/10.1016/S0304-3975(96)00119-3.

[GR66] Seymour Ginsburg and Gene F. Rose. "A Characterization of Machine Mappings". In: *Canadian Journal of Mathematics* 18 (1966), pp. 381–388. DOI: 10.4153/CJM-1966-040-3. URL: https://doi.org/10.4153/CJM-1966-040-3.

[Gri68] Timothy V. Griffiths. "The Unsolvability of the Equivalence Problem for Λ-Free Nondeterministic Generalized Machines". In: *Journal of the ACM* 15.3 (July 1968), pp. 409–413. DOI: 10.1145/321466.321473. URL: https://doi.org/10.1145/321466.321473.

[Gur82] Eitan M. Gurari. "The Equivalence Problem for Deterministic Two-Way Sequential Transducers is Decidable". In: *SIAM Journal on Computing* 11.3 (Aug. 1982). A preliminary version appeared at FOCS 1980, pp. 83–85, https://doi.org/10.1109/SFCS.1980.46, pp. 448–452. DOI: 10.1137/0211035. URL: https://doi.org/10.1137/0211035.

[HK91] Tero Harju and Juhani Karhumäki. "The Equivalence Problem of Multitape Finite Automata". In: *Theoretical Computer Science* 78.2 (Jan. 1991), pp. 347–355. DOI: 10.1016/0304-3975(91)90356-7. URL: https://doi.org/10.1016/0304-3975(91)90356-7.

[HU67] J. E. Hopcroft and J. D. Ullman. "An approach to a unified theory of automata". In: *8th Annual Symposium on Switching and Automata Theory (SWAT 1967)*. 1967, pp. 140–147. DOI: 10.1109/FOCS.1967.4. URL: https://doi.org/10.1109/FOCS.1967.4.

[Iba71] Oscar H. Ibarra. "Characterizations of Some Tape and Time Complexity Classes of Turing Machines in Terms of Multihead and Auxiliary Stack Automata". In: *Journal of Computer and System Sciences* 5.2 (Apr. 1971), pp. 88–117. DOI: 10.1016/S0022-0000(71)80029-6. URL: https://doi.org/10.1016/S0022-0000(71)80029-6.

[KR65] Kenneth Krohn and John Rhodes. "Algebraic theory of machines. I. Prime decomposition theorem for finite semigroups and machines". In: *Transactions of the American Mathematical Society* 116 (1965), pp. 450–464. DOI: 10.1090/S0002-9947-1965-0188316-1. URL: https://doi.org/10.1090/S0002-9947-1965-0188316-1.

[Mal53] A. I. Mal'cev. "Nilpotent semigroups". In: *Uchenye Zapiski Ivanovskogo Gosudarstvennogo Pedagogicheskogo Instituta. Fiziko-Matematicheskie Nauki* 4 (1953). In Russian. The source of the embedding of free monoids into the metabelian group that Albert and Lawrence use in their proof of Ehrenfeucht's conjecture, pp. 107–111.

[Mea55] George H. Mealy. "A method for synthesizing sequential circuits". In: *The Bell System Technical Journal* 34.5 (Sept. 1955), pp. 1045–1079. DOI: 10.1002/j.1538-7305.1955.tb03788.x. URL: https://doi.org/10.1002/j.1538-7305.1955.tb03788.x.

[Mey69] Albert R. Meyer. "A Note on Star-Free Events". In: *Journal of the ACM* 16.2 (1969), pp. 220–225. DOI: 10.1145/321510.321513. URL: https://doi.org/10.1145/321510.321513.

[MNR23] Filip Murlak, Damian Niwiński, and Wojciech Rytter. *200 problems on languages, automata, and computation*. Cambridge University Press, 2023. DOI: 10.1017/9781009072632. URL: https://doi.org/10.1017/9781009072632.

[Moo56] Edward F. Moore. “Gedanken-Experiments on Sequential Machines”. In: *Automata Studies*. Ed. by C. E. Shannon and J. McCarthy. Annals of Mathematics Studies 34. Princeton, NJ: Princeton University Press, 1956, pp. 129–153. DOI: 10.1515/9781400882618-006. URL: https://doi.org/10.1515/9781400882618-006.

[MP71] Robert McNaughton and Seymour Papert. *Counter-Free Automata*. M.I.T. Research Monograph 65. Cambridge, MA: MIT Press, 1971. ISBN: 0-262-13076-9.

[MSV03] Tova Milo, Dan Suciu, and Victor Vianu. “Typechecking for XML transformers”. In: *Journal of Computer and System Sciences* 66.1 (Feb. 2003), pp. 66–97. DOI: 10.1016/S0022-0000(02)00030-2. URL: https://doi.org/10.1016/S0022-0000(02)00030-2.

[Ner58] Anil Nerode. “Linear automaton transformations”. In: *Proceedings of the American Mathematical Society* 9.4 (1958), pp. 541–544. DOI: 10.1090/S0002-9939-1958-0135681-9. URL: https://doi.org/10.1090/S0002-9939-1958-0135681-9.

[Ngu24] Lê Thành Dũng Nguyên. “Two or three things I know about tree transducers”. In: *CoRR* abs/2409.03169 (2024). Unpublished note collecting known results. DOI: 10.48550/arXiv.2409.03169. URL: https://doi.org/10.48550/arXiv.2409.03169.

[Pin25] Jean-Éric Pin. *Mathematical Foundations of Automata Theory*. MPRI lecture notes. Version of 24 March 2025; profinite methods are Chapter X, *Profinite words*, pp. 175–188. Living document, so the version date matters when citing. 2025. URL: https://www.irif.fr/~jep/PDF/MPRI/MPRI.pdf.

[Pos46] Emil L. Post. “A Variant of a Recursively Unsolvable Problem”. In: *Bulletin of the American Mathematical Society* 52.4 (1946), pp. 264–268. DOI: 10.1090/S0002-9904-1946-08555-9. URL: https://doi.org/10.1090/S0002-9904-1946-08555-9.

[RS59] Michael O. Rabin and Dana Scott. “Finite automata and their decision problems”. In: *IBM Journal of Research and Development* 3.2 (1959), pp. 114–125. DOI: 10.1147/rd.32.0114. URL: https://doi.org/10.1147/rd.32.0114.

[RS91] Christophe Reutenauer and Marcel-Paul Schützenberger. “Minimization of Rational Word Functions”. In: *SIAM Journal on Computing* 20.4 (Aug. 1991), pp. 669–685. DOI: 10.1137/0220042. URL: https://doi.org/10.1137/0220042.

[Sak21] Jacques Sakarovitch. “Automata and Rational Expressions”. In: *Handbook of Automata Theory. Volume I: Theoretical Foundations*. Ed. by Jean-Éric Pin. An extended version, with the proofs and examples cut for space, is available as arXiv:1502.03573; theorem numbering differs between the two. Berlin: EMS Press, 2021, pp. 39–78. DOI: 10.4171/AUTOMATA-1/2. URL: https://doi.org/10.4171/AUTOMATA-1/2.

[Sch09] Marcel-Paul Schützenberger. *Œuvres complètes*. French. Ed. by Jean Berstel, Alain Lascoux, and Dominique Perrin. Vol. 8. Tome 8 collects the papers of 1971–1975, with editors' comments. Marne-la-Vallée: Institut Gaspard-Monge, Université Paris-Est, 2009. URL: https://www-igm.univ-mlv.fr/~berstel/Mps/Oeuvres-Completes/tome08.pdf.

[Sch56] M. P. Schützenberger. "Une théorie algébrique du codage". In: *Séminaire Dubreil. Algèbre et théorie des nombres* 9 (1956). Séminaire 1955–1956, exposé no. 15, pp. 1–24. URL: https://www.numdam.org/item/SD_1955-1956__9__A10_0/.

[Sch61a] M. P. Schützenberger. "A remark on finite transducers". In: *Information and Control* 4.2-3 (1961), pp. 185–196. DOI: 10.1016/S0019-9958(61)80006-5. URL: https://doi.org/10.1016/S0019-9958(61)80006-5.

[Sch61b] M. P. Schützenberger. "On the definition of a family of automata". In: *Information and Control* 4.2-3 (Sept. 1961), pp. 245–270. DOI: 10.1016/S0019-9958(61)80020-X. URL: https://doi.org/10.1016/S0019-9958(61)80020-X.

[Sch65] M. P. Schützenberger. "On finite monoids having only trivial subgroups". In: *Information and Control* 8.2 (1965), pp. 190–194. DOI: 10.1016/S0019-9958(65)90108-7. URL: https://doi.org/10.1016/S0019-9958(65)90108-7.

[Sch76] M. P. Schützenberger. "Sur les relations rationnelles entre monoides libres". In: *Theoretical Computer Science* 3.2 (Nov. 1976), pp. 243–259. DOI: 10.1016/0304-3975(76)90026-8. URL: https://doi.org/10.1016/0304-3975(76)90026-8.

[Sco67] Dana Scott. "Some Definitional Suggestions for Automata Theory". In: *Journal of Computer and System Sciences* 1.2 (Aug. 1967). A less polished version was presented at the Conference on the Algebraic Theory of Machines, Languages and Semigroups on 5 September 1966, under the title *A Modest Proposal Concerning Nondeterministic Automata*, pp. 187–212. DOI: 10.1016/S0022-0000(67)80014-X. URL: https://doi.org/10.1016/S0022-0000(67)80014-X.

[She59] J. C. Shepherdson. "The Reduction of Two-Way Automata to One-Way Automata". In: *IBM Journal of Research and Development* 3.2 (Apr. 1959), pp. 198–200. DOI: 10.1147/rd.32.0198. URL: https://doi.org/10.1147/rd.32.0198.

[Sip12] Michael Sipser. *Introduction to the Theory of Computation*. 3rd ed. Boston, MA: Cengage Learning, 2012. ISBN: 978-1-133-18779-0.

[SMK18] Helmut Seidl, Sebastian Maneth, and Gregor Kemper. "Equivalence of Deterministic Top-Down Tree-to-String Transducers Is Decidable". In: *Journal of the ACM* 65.4 (Apr. 2018), 21:1–21:30. DOI: 10.1145/3182653. URL: https://doi.org/10.1145/3182653.

[Tho97] Wolfgang Thomas. “Languages, Automata, and Logic”. In: *Handbook of Formal Languages. Volume 3: Beyond Words*. Ed. by Grzegorz Rozenberg and Arto Salomaa. Berlin: Springer, 1997, pp. 389–455. DOI: 10.1007/978-3-642-59126-6_7. URL: https://doi.org/10.1007/978-3-642-59126-6_7.

[Tra62] B. A. Trakhtenbrot. “Finite automata and the logic of one-place predicates”. In: *Sibirskii Matematicheskii Zhurnal* 3.1 (1962). In Russian. English translation in *American Mathematical Society Translations, Series 2*, volume 59, *Twelve Papers on Logic and Algebra*, 1966, pp. 23–55, https://doi.org/10.1090/trans2/059/02, pp. 103–131. URL: https://www.mathnet.ru/eng/smj4799.

[Wil99] Thomas Wilke. “Classifying Discrete Temporal Properties”. In: *STACS 99*. Vol. 1563. Lecture Notes in Computer Science. Springer, 1999, pp. 32–46. DOI: 10.1007/3-540-49116-3_3. URL: https://doi.org/10.1007/3-540-49116-3_3.