

Transducers

Mikołaj Bojańczyk

March 17, 2026

Contents

Introduction	3
Part A: Mealy machines	7
A.1 Mealy machines	7
A.1.1 Equivalence, compositions and continuity	9
A.2 The Krohn-Rhodes Decomposition Theorem	12
A.2.1 Map lifting	14
A.2.2 State transformations	16
A.2.3 Counter-free Mealy machines	20
Part B: Rational functions	22
B.1 Rational relations	22
B.1.1 Composition and continuity	25
B.1.2 Undecidable equivalence	26
B.2 Rational functions	28
B.2.1 Bimachines	28
B.2.2 Decomposition into primes	33
B.2.3 Mealy machines as a subset of the rational functions	34
B.3 Rational relations and weighted automata	36
B.3.1 Weighted automata	37
B.3.2 Decidable equivalence	39
B.3.3 Decidability of equivalence for weighted automata over a field	41
B.4 Machine independent characterisations	43
B.4.1 Mealy machines	43
B.4.2 Sequential functions	46
B.4.3 Subsequential functions	47
B.4.4 Rational functions	53
Part C: Regular functions	55
Part D: Polyregular functions	56

Introduction

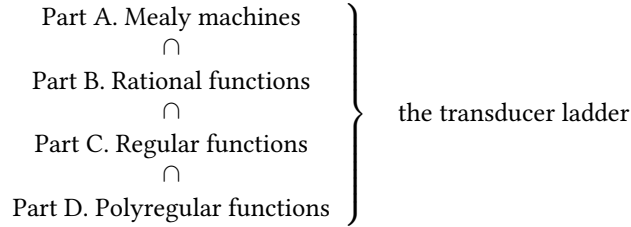
This book is a collection of lecture notes for a course on transducers, taught at the University of Warsaw in 2024. Unlike the study of regular languages, which can be viewed as functions with Boolean outputs

$$L : A^* \rightarrow \{\text{yes}, \text{no}\},$$

this course is about string-to-string functions

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

where A and B are some finite alphabets. In this course, 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:



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*

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

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 metric 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. 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

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

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, whose proof is left to the reader, is the sequence of words with factorial lengths

$$b^1, b^{2!}, b^{3!}, \dots$$

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

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

where all outputs are in the sequence, and furthermore each output is used for finitely many inputs. This function will necessarily be continuous, because 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, and thus regular. 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. $\square$

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:

$$A^* \xrightarrow{f} B^* \xrightarrow{L} \{\text{yes, no}\}.$$

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 $a_1 \cdots a_n \mapsto a_n \cdots a_1$.

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

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

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

Exercise .0.5. Show that the class of continuous functions is closed under composition.

Exercise .0.6. Show that there are uncountably many continuous functions, and therefore some of them are not computable. Hint: use the factorials.

Exercise .0.7. For words over an alphabet A , define a metric so that the distance between two strings $w \neq w'$ is

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

Show that this is a metric, and in fact it satisfies the stronger inequality

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

Exercise .0.8. Show that all string-to-string functions are continuous with respect to the above metric.

Exercise .0.9. Show that the continuous functions, in the sense of Definition .0.1, are exactly the uniformly continuous functions with respect to the above metric.

Part A: Mealy machines

In this part, we study with the simplest transducer model, namely Mealy machines. Mealy machines are barely different from regular languages, and in fact the early literature on automata theory did not even distinguish between the two. From this perspective, Mealy machines are a good starting point the study of transducer, since many of the results that we care about – such as continuity, composition and equivalence – are easy to prove, which we do in Section B.4.1. 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 not 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*

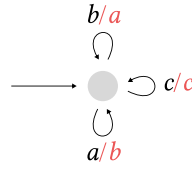
$$\underbrace{A}_{\substack{\text{input} \\ \text{alphabet}}} \quad \underbrace{B}_{\substack{\text{output} \\ \text{alphabet}}} \quad \underbrace{Q}_{\substack{\text{state} \\ \text{space}}} \quad \underbrace{q_0 \in Q}_{\substack{\text{initial} \\ \text{state}}} \quad \underbrace{\delta : Q \times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

The semantics of the machine is a function

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

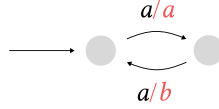
which is defined as follows. We consider the deterministic finite automaton which is obtained from the Mealy machine by ignoring the output letters. We run this automaton on the input string, and then we label each position by the output letter of the corresponding transition. This function is a *letter-to-letter*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

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

$\square$

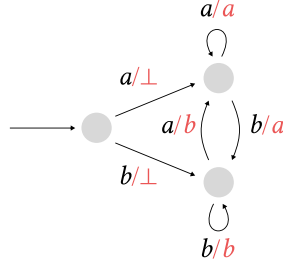
Example 4. [Delay] For an input alphabet A , consider the *delay function*, which shifts the input letters by one step to the right, as in the following picture:

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

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

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 alphabet. □

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.

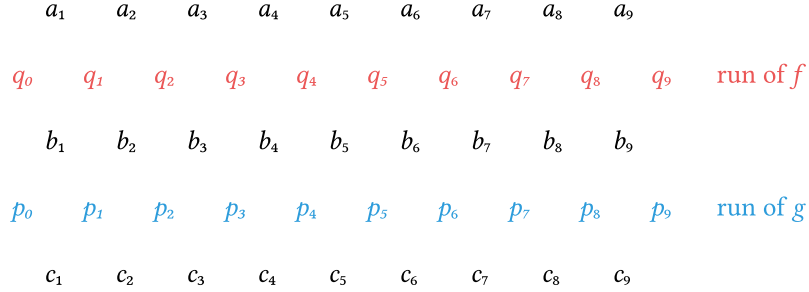
Compositions. We now establish closure under composition for Mealy machines. In this book, we use a dot to represent left-to-right composition of functions, i.e. $f \cdot g$ is the function where we first apply f , and then we apply g to the output of f . To denote the opposite order of composition,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

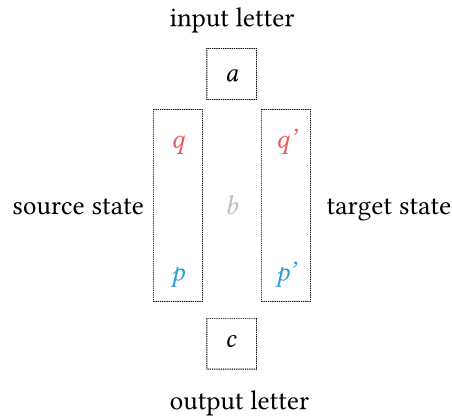
Proof. A straightforward product construction. Consider the composition $f \cdot g$ applied to some input string, with the runs of the two transducers shown in the following picture:



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 , and 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. $\square$

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

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

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. $\square$

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.

Exercise A.1.2. Show that a composition of n Mealy machines might require a number of states that is exponential in n .

Exercise A.1.3. We say that a Mealy machine

$$f : A^* \rightarrow A^*$$

is *invertible* if there is some Mealy machine

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

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

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

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

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

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 $f : B^* \rightarrow A^*$ such that

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

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

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.

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:

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

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 for Mealy machines is the most difficult one to prove (or at least not the easiest one).

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 $Q \rightarrow Q$ 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 machine:*

- *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 $ababa \dots$ 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. $\square$

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

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

where each Mealy machine $f_1, \dots, f_n$ is either reversible or a 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 $\#$, as follows:

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

If the underlying function f has type $A^* \rightarrow B^*$, then we denote its map lifting by

$$(A + 1)^* \xrightarrow{\text{map} f} (B + 1)^*,$$

where $+1$ stands for adjoining a fresh separator symbol to the alphabet. The following lemma shows that map lifting is compatible with decompositions into primes.

Lemma A.2.3. *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.

$$\text{map}(f; g) = (\text{map} f); (\text{map} g).$$

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.

For 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. This usefulness of the second state transformation is demonstrated by the following cancellation law:

$$\delta_w = \underbrace{(\gamma_u)^{-1}}_{\substack{\text{a permutation, and} \\ \text{hence can be inverted}}} \cdot \underbrace{\gamma_w}_{\substack{\text{where } u \text{ is the prefix of } w \text{ that ends in the last } \# \\ \text{call this the } \# \text{-prefix of } w}} \quad (1)$$

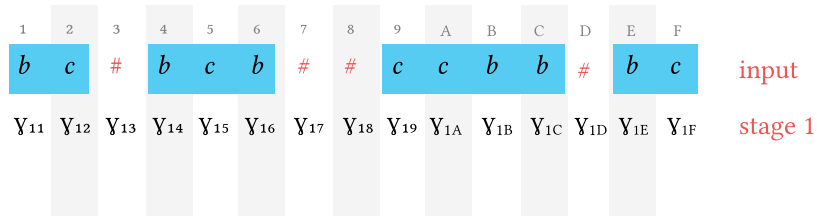
(In the edge case when w does not contain the separator symbol $\#$, prefix u is empty, and γ_u its state transformation is the identity permutation.) Using the cancellation law, we will decompose the map lifting of f into prime Mealy machines. The idea is that for each prefix of the input string, we 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

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

where Q 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} stands 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 the $\#$ -prefixes of the input string, as in the following picture:

1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
b	c	#	b	c	b	#	#	c	c	b	b	#	b	c	input
Y ₁₁	Y ₁₂	Y ₁₃	Y ₁₄	Y ₁₅	Y ₁₆	Y ₁₇	Y ₁₈	Y ₁₉	Y _{1A}	Y _{1B}	Y _{1C}	Y _{1D}	Y _{1E}	Y _{1F}	stage 1
			Y ₁₃	Y ₁₃	Y ₁₃	Y ₁₃	Y ₁₇	Y ₁₈	Y ₁₈	Y ₁₈	Y ₁₈	Y ₁₈	Y ₁₈	Y ₁₈	stage 2

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 $\delta_{1(i-1)}$. 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. $\square$

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:

$$\underbrace{A}_{\text{input alphabet}} \quad \underbrace{Q}_{\text{state space}} \quad \underbrace{\{\delta_a : Q \rightarrow Q\}_{a \in A}}_{\text{state transformations}}. \quad (2)$$

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

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

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

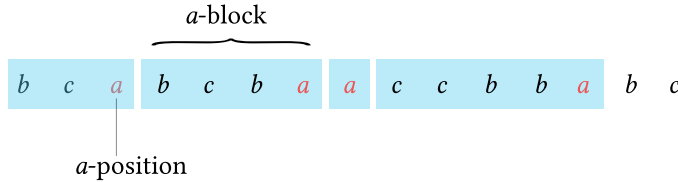
Before proving the lemma, we use it to conclude the proof of the Krohn-Rhodes Theorem. Take a Mealy machine, and consider its underlying pre-automaton. By applying the lemma, we get a composition of prime Mealy machines, which computes for each input position the state of the underlying pre-automaton 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.2.4. 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.

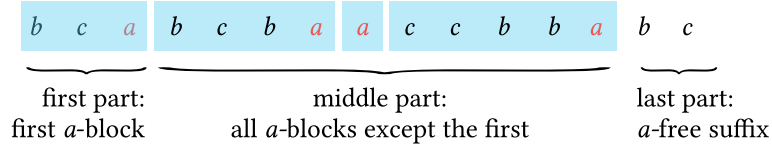
In the proof, we will be interested in a -positions, i.e. positions in the input string that are labelled with the letter a , and a -blocks, i.e. pieces of the input string that go from one a -position to the next one, including the latter a -position but not the former one. Here is a picture:



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

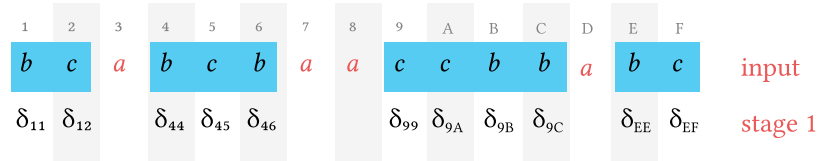
the a -blocks can be seen as individual letters for a pre-automaton with a smaller set of states. This will be made precise below.

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:



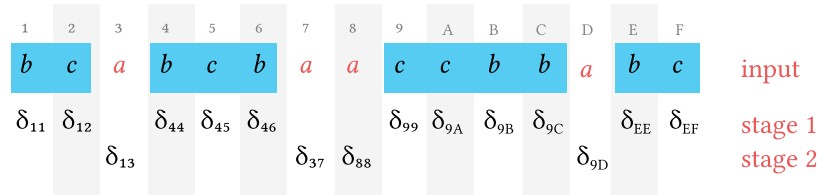
We refer to this decomposition as the *tripartite decomposition*. We now describe a sequential composition of prime Mealy machines, in which all state transformations in the tripartite decomposition are computed. This decomposition proceeds in five stages. We assume that in each stage, we have access 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 applying 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.

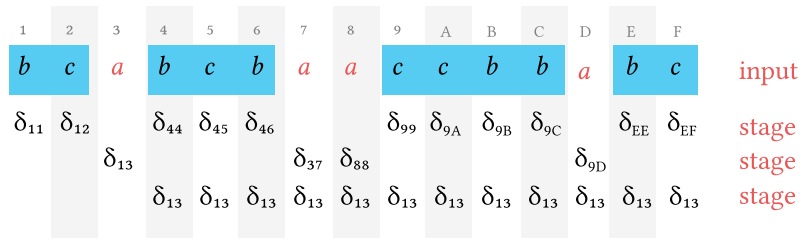
2. In the second stage, we compute the state transformations of individual a -free blocks, and we store them in the a -positions that end them, as in the following picture:



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 Counter-free 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.2.5. *Reversible Mealy machines are closed under composition.*

A much more interesting class is the second one, namely compositions of flip-flop machines.

Exercises

Exercise A.2.1. Show that every flip-flop machine can be obtained as a sequential composition of two-state flip-flop machines.

Exercise A.2.2. Show that the delay function from Example 4 is not a composition of reversible Mealy machines.

Exercise A.2.3. Show that the function from Example 3 is not a composition of flip-flop Mealy machines.

Exercise A.2.4. Is every invertible Mealy, as defined in Exercise A.1.3 machine necessarily reversible? The other way round?

Part B: Rational functions

In this part, we move one step up the transducer ladder, which is the rational relations. 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 name of non-proliferation of 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 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 an initial 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.

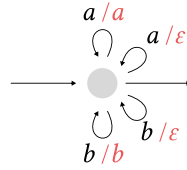
Definition B.1.2 (Rational relation). *A rational relation is any relation that can be defined by a nondeterministic automaton with output.*

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.

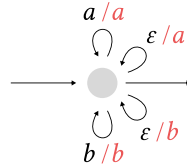
Example 6. [Subword relation] For every alphabet A , the subword relation

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

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

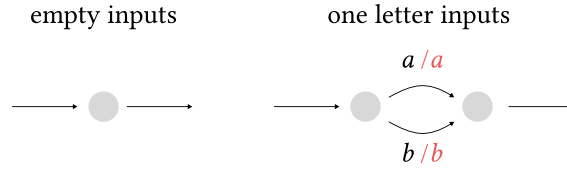


An important property of rational relations is that they are input/output symmetric. Therefore, the inverse of the subword relations, i.e. when letters can be added, is also rational, with the corresponding automaton looking like this:

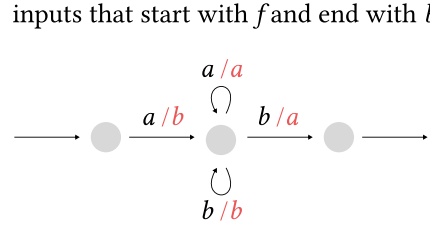


In particular, thanks to transitions that have ε on the input part, a rational relation can have infinitely many outputs for a single input. $\square$

Example 7. [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:



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. $\square$

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

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

In the free monoid A^* , the two notions coincide, but in other monoids they do not coincide. In the monoid $A^* \times B^*$, the rational relations, as defined in Definition B.1.2, 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.1.1.

We have still not explained why the name “rational” is used in Definition B.1.3. The motivation comes from power series.

B.1.1 Composition and continuity

We now show that 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*

$$R \subseteq A^* \times B^* \quad \text{and} \quad S \subseteq B^* \times C^*$$

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

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

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

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

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. 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. $\square$

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. We can prefer 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

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

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. $\square$

B.1.2 Undecidable equivalence

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

Theorem B.1.6. *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 $g, h : A^* \rightarrow B^*$, decide if there is some input string where both homomorphisms produce the same output. (Recall that a string homomorphism is a function that applies some function

of type $A \rightarrow B^*$ 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 $h : A^* \rightarrow B^*$ is a homomorphism, then its comcomplement 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 letter a after the correct input prefix, it does one of the following: (i) outputs a string that is a proper prefix of $g(a)$ and then produces no more output; or (ii) outputs a string that is incomparable with $g(a)$ under the prefix relation and then produces any output. $\square$

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:

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

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 universality problem for rational relations, and hence the latter problem must be undecidable. $\square$

The above problem is one of the reason 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.

Exercises

Exercise B.1.1. 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

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

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

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.

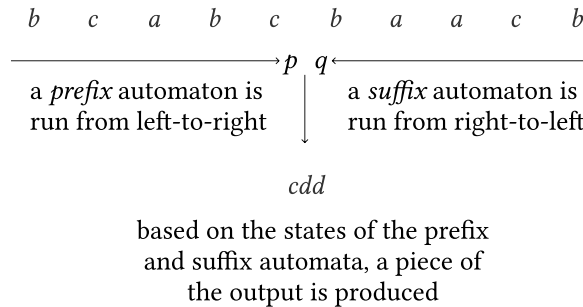
Definition B.2.1 (Rational function). *A rational function is the special case of a rational relation, in which each input string is related to exactly one output string.*

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). This advantage 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*

$$(\text{states of prefix automaton}) \times (\text{states of suffix automaton}) \rightarrow B^*$$

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

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

We run the prefix automaton on the prefix, and the suffix automaton on the reverse¹ 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. *The following models describe the same string-to-string functions:*

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

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

¹Using 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.

From bimachines to functional. We begin with the easiest inclusion

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

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

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

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:

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

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 one 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

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

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 input 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. $\square$

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 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 smaller initial states of the original automaton.
- **Final states.** The special state $\dashv$ is final. A state of the form $P \ni p$ is final if and only if p is final in 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 reach 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. $\square$

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

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

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 uses exactly one ε -transition (i.e. with empty input), and this is the last transition of the run. This property was ensured by the construction in the previous step, by the proof of Lemma B.2.5. 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), 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 last letter.

□

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 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 begin by appending the and we decompose the past and future automata into their prime components. This gives us two sequences of prime machines, which label each position in the input string with the states before and after reading it, for both the future and past automata. Next, we apply the last prime function, which allows us to deal with an input string that is empty. Finally, we apply a homomorphism, which produces the □

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 $f : A^* \rightarrow B^*$. 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. Show that rational relations are not closed under intersection.

Exercise B.2.2. Show that it is undecidable if two rational relations have nonempty intersection

Exercise B.2.3. Show that there is a function

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

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

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

the composition $f \cdot g$ is rational.

Exercise B.2.4. Consider two rational functions f and g , possibly with different input and output alphabets. We say that f factors through g if one can decompose f as

$$f = g_1 \cdot g \cdot g_2$$

for some rational functions g_1 and g_2 . Show an algorithm that decides this property, given rational functions f and g .

Exercise B.2.5. A set I of a rational functions is called an ideal if

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

i.e. functions that factor through the ideal must stay in the ideal. Show that there are finitely many ideals.

Exercise B.2.6. Consider a rational relation $R \subseteq A^* \times B^*$, such that for every input string there is at least one output. Show that there is a rational function $f : A^* \rightarrow B^*$ such that $wRf(w)$ holds for every input string.

Exercise B.2.7. Show that if a rational function $f : A^* \rightarrow B^*$ is surjective, then it has a one-sided inverse, i.e. a rational function $g : B^* \rightarrow A^*$ such that $g \cdot f$ is the identity on B^* .

Exercise B.2.8. Show that the following problem is undecidable: given a rational function f , we want to know if it is injective, i.e. different input strings are mapped to different output strings.

Exercise B.2.9. Show that the following problem is undecidable: given a rational function $f : A^* \rightarrow A^*$, we want to know if it generates finitely many functions under composition, i.e. if the following set is finite:

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

Exercise B.2.10. Show that the following two conditions are equivalent for a rational relation:

1. for every input string, all output strings have the same length as the input;
2. it is computed by an NFA with output, in which for every transition, the length of the input and output strings are the same.

Hint: for two states p and q in an NFA with output, consider the set

$$\Delta_{pq} = \{ (|w| - |v|) \mid \text{there is a run } p \xrightarrow{v/w} q \}.$$

We only prove the non-trivial implication from 1 to 2. Consider an NFA with output that computes the relation. Consider the set Δ_{pq} from the hint. This set cannot contain two different numbers, since otherwise we would get a violation of condition 1. Therefore, this set is just a single number (possibly undefined if there is no run from p to q), which we denote by δ_{pq} . We have the following transitivity law

$$\delta_{pq} = \delta_{pr} + \delta_{rq},$$

which holds whenever the two number on the right-hand side are defined. This implies that for every state q we can assign a number δ_q such that

$$\delta_{pq} = \delta_q - \delta_p.$$

We would like to improve the NFA with output so that the numbers δ_q are all zero. For this, we define a new automaton in which the states are pairs (q, u) , such that q is a state of the original automaton and u is a string of length δ_q . The general idea is that u is a piece of the output string that was produced before, awaiting some transition. Transitions are defined by

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

whenever the original automaton had a transition of the form

$$p \xrightarrow{v/w'} q$$

such that $uv = w'u'$. This implies that

$$|u| - |u'| = |w'| - |v|.$$

The left-hand side is equal to δ_{pq} because of the lengths of the strings u and u' , and the right-hand side is equal to $|w| - |v| + \delta_{pq}$, and thus $|w| = |v|$, as desired. Finally, one can check that the new automaton computes the same relation.

B.3 Rational relations and weighted automata

In this section, we give describe the connections of rational relations (including their functional fragment) with another automaton model, called weighted automata. 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.

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

Figure 1: The semiring axioms.

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 8. [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. $\square$

Example 9. [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. $\square$

Example 10. [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 $L \cdot K$. $\square$

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 11. [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. $\square$

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

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

is rational if and only if the function

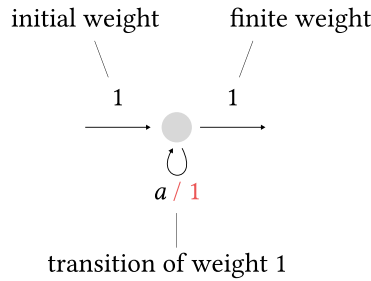
$$w \in A^* \mapsto \{ v \in B^* \mid wRv \}$$

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

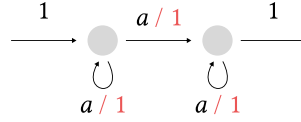
$$p \xrightarrow{a/L} q$$

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

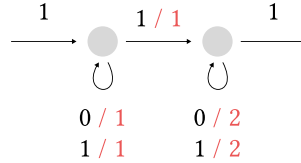
Example 13. [Weighted automata over $\mathbb{N}$] Let us give some examples of weighted automata over the semiring of natural numbers. Consider first the following automaton, which has a one-letter input alphabet:



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 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. $\square$

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.

Theorem B.3.3. *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.

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; and (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:

$$A^* \begin{array}{c} \xrightarrow{f} \\ \xrightarrow{g} \end{array} B^* \xrightarrow{\iota} \mathbb{Q}$$

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 13, 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*

$$A^* \xrightarrow{f} B^* \xrightarrow{h} \mathbb{S}$$

$\underbrace{\quad}_{\text{a rational function}} \quad \underbrace{\quad}_{\text{a weighted automaton}}$

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 f) unambiguous, and appropriately aligning the two runs in the product constructions.

By Lemmas B.2.4 and Lemma 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 in the conclusion of the theorem 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 that have the following properties: the source state is q , the target state is q' , the input string is a , and the letter is consumed in the last transition. 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 $f : A^* \rightarrow B^*$ is rational if and only if weighted automata 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, and the function

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

By closure under pre-composition, it follows that

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

is computed by a weighted automaton over the semiring of regular languages. By Example 12, 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

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

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

Theorem B.3.7. *The zeroness problem is decidable for weighted automata over the field of rationals².*

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*

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

to be the linear map for which in the corresponding $Q \times Q$ matrix, the coefficient in cell (p, q) stores the sum of weight 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*

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

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 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 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 is 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. $\square$

²The proof works for any computable field, where field elements can be represented in a finite way and field operations can be computed.

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

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

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

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

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). $\square$

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 claim.

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

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

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 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. $\square$

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. $\square$

So far, we have used only binary rational relations, but the same notion applies to ternary rational relations, with transitions in the corresponding automaton being labelled by triples of words. We claim that if the function is length-preserving, then

$$\{ (x, y_1, y_2) \in A^* \times B^* \times B^* \mid \begin{array}{l} y_1 \text{ is a prefix of some output in } f(xA^*) \text{ and} \\ y_2 \text{ is a prefix of some output in } f(xA^*) \\ |x| = |y_1| = |y_2| \end{array} \}$$

It remains to check prefix-preserving property. Before doing this, 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 $\tau : Q \rightarrow \mathbb{Z}$ which maps all accepting states to zero. Define a new a 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

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

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. These transitions in the new automaton are length-preserving, as required by the conclusion of the lemma, because

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

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. $\square$

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

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

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. $\square$

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

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

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

Theorem B.4.6. *A function $f : A^* \rightarrow B^*$ is sequential if and only if it is: (a) continuous; (b) prefix preserving, and (c) has the following bounded increase property:*

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

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 three properties. Define the *derivative* of this function to be

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

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 output of the derivatives. This can give 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 $Q \rightarrow B^*$. 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 14. [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 14. It will also not be sufficient, since any function can be turned into a partial function that is prefix preserving.

Example 15. [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

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

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 if 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.

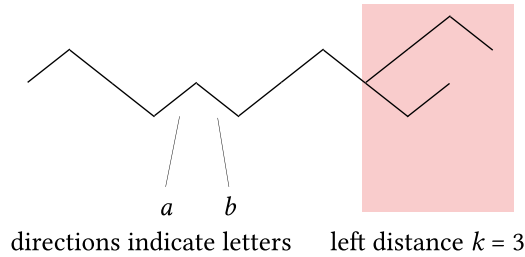
Definition B.4.7 (Left distance). *For two strings w_1 and w_2 , define their left distance*

$$\|w_1, w_2\|$$

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

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

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:



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 function 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. *A partial function $f : A^* \rightarrow B^*$ 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.

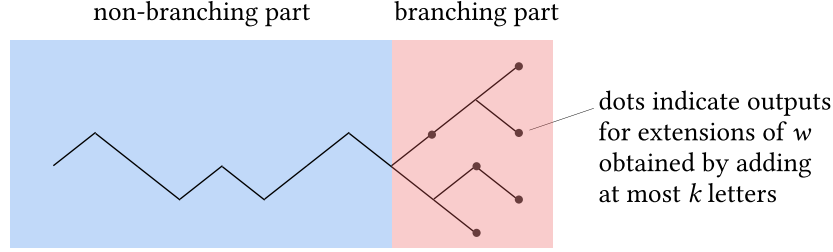
Claim B.4.9. *There is some $k \in \mathbb{N}$ such that for every $w \in A^*$, if there is some $v \in A^*$ such that wv is in the domain of f , then one can choose v so that it has length at most k .*

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

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 w . In particular, if we think of the branching part as a string-to-string function

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

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

$$A^{\leq k} \xrightarrow{\beta(w)} B^*$$

$$v \mapsto f(wv) \text{ with the prefix } \alpha(w) \text{ removed.}$$

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

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

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 sequential, 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

$$|f(wv)| \pmod{K+1}$$

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

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

is at most K , and therefore this difference is uniquely determined by its value modulo $K+1$, 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 $\beta(w)$. $\square$

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

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

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:

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

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

$$\alpha(a_1 \cdots a_n) = \alpha(\varepsilon) \cdot \delta(a_1) \cdot \delta(a_1 a_2) \cdots \delta(a_1 \cdots a_n). \quad (7)$$

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. $\square$

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

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

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

Proof. The assumption that the domain is prefix-closed is true in our intended application, but it is not essential, since 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.

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

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 sequential transducer, which outputs $g'(w)$ without the last M letters, and keeps the remaining M letters in its state. $\square$

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). $\square$

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. *A function $f : A^* \rightarrow B^*$ is rational if and only if it is continuous, and the following equivalence relation on input strings has finite index:*

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

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

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

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

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

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 every 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 $\sim$.

□

Part C: Regular functions

Part D: Polyregular functions