The single-use restriction for register automata and transducers over infinite alphabets

Rafał Stefański

June 2021

Contents

Acknowledgements

I would like to start by directing words of gratitude to the entire community of the Faculty of Mathematics, Informatics and Mechanics of the University of Warsaw, among whom I have had the opportunity to develop as a computer scientist over the past 10 years. Thanks to them, almost every day spent physically at work is a real pleasure. I especially want to thank Mikołaj Bojańczyk – I could not have imagined a better supervisor. Further thanks go to Edon Kelmendi for introducing me to the world of theoretical computer science and helping me write my first review; to Nathan Lhote for our invaluable cooperation during the lockdown; and to Janusz Schmude for our joint PhD adventure.

For the past year, I had the opportunity to work in the PPLV group at University College London. It was a very valuable time for me. I sincerely thank everyone with whom I had the opportunity to share it. I would like to give special thanks to Samson Abramsky for getting me excited to work with category theory, for his words of advice, and for his patience.

I would also like to thank the community associated with the 14th High School in Warsaw, especially to my counselling teachers – the late Stanislaw Lipiński and the late Jack Banasik; my math teachers, Jerzy Konarski and Filip Smętek; and my computer science teachers, Hanna Stachera and Joanna Śmigielska. I would also like to thank Maciej Matraszek for the work he put into running the computer science club.

Recalling my period of high school, it is impossible not to appreciate the people involved in the organization of the Polish Olympiad in Informatics and the editors of the monthly magazine Delta. They opened up new perspectives for me and steered me towards the academic world. At this point, I would also like to thank Damian Niwiński for his article "Impossible Shortcut"[1].

A very important stage in my education was the time spent at the 16th Primary and Middle School in Warsaw. My further thanks go to all my teachers from that period. Especially to Agnieszka Nowińska-Samsel for introducing

[1] Article in Polish: `https://deltami.edu.pl/temat/matematyka/logika/2012/07/30/Niemozliwy_skrot/`

me to the principles of mathematical rigour, and to Ewa Kietlińska-Zaleska for believing in my abilities.

I warmly thank my parents Ewa Stefańska and Grzegorz Stefański for the love, inspiration and effort they put into my education, and my grandmothers Teresa Rybak and Monika Stefańska for their care and devotion.

I save my final words of gratitude for my future wife Klaudia Grygoruk: Thank you for being who you are.

Podziękowania

Podziękowania chciałbym rozpocząć od skierowania słów wdzięczności do całej społeczności Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego, pośród której przez ostatnie 10 lat miałem okazję rozwijać się jako informatyk. Dzięki wszystkim tworzącym ją ludziom prawie każdy dzień spędzony fizycznie w pracy jest prawdziwą przyjemnością. Szczególnie pragnę podziękować Mikołajowi Bojańczykowi – trudno mi sobie wyobrazić lepszego promotora. Dalsze podziękowania kieruję do Edona Kelmendi za wprowadzenie mnie w tajniki świata teoretycznej informatyki i pomoc w napisaniu pierwszej recenzji; do Nathana Lhote'a za nieocenioną współpracę podczas lockdownu oraz do Janusza Schmude za wspólną doktorancką przygodę.

Przez ostatni rok miałem okazję pracować w grupie PPLV na University College London. Był to dla mnie bardzo cenny czas. Serdecznie dziękuję wszystkim, z którymi dane mi było go spędzić. Szczególne podziękowania kieruję pod adresem Samsona Abramsky'ego za inspirację do pracy z teorią kategorii, dobre rady oraz cierpliwość.

Słowa podziękowań należą się również środowisku związanemu z XIV L.O. im. Stanisława Staszica w Warszawie, a zwłaszcza moim wychowawcom – ś.p. Stanisławowi Lipińskiemu oraz ś.p. Jackowi Banasikowi; nauczycielom matematyki – Jerzemu Konarskiemu oraz Filipowi Smętkowi, a także nauczycielom informatyki – Hannie Stacherze i Joannie Śmigielskiej. Ponadto chciałbym podziękować Maciejowi Matraszkowi za pracę włożoną w prowadzenie kółka informatycznego.

Wspominając okres liceum, nie sposób nie docenić osób zaangażowanych w organizację Olimpiady Informatycznej oraz redakcję miesięcznika „Delta". To one otworzyły przede mną nowe perspektywy i pokierowały mnie w stronę świata akademickiego. W tym miejscu chciałbym również podziękować Damianowi Niwińskiemu za jego artykuł „Niemożliwy Skrót"[2]

Bardzo ważnym etapem mojego rozwoju był czas spędzony w Społecznej Szkole Podstawowej i Społecznym Gimnazjum nr 16 STO. Kolejne podziękowa-

[2] `https://deltami.edu.pl/temat/matematyka/logika/2012/07/30/Niemozliwy_skrot/`

nia kieruję do wszystkich nauczycieli obu tych placówek, w tym w sposób szczególny do Agnieszki Nowińskiej-Samsel za wprowadzenie mnie w zasady matematycznego rygoru oraz do Ewy Kietlińskiej-Zaleskiej, która zawsze wierzyła w moje możliwości.

Gorąco dziękuję moim rodzicom Ewie i Grzegorzowi Stefańskim za miłość, inspirację i wysiłek włożony w moje wykształcenie oraz moim babciom Teresie Rybak i Monice Stefańskiej za troskę i oddanie.

Wreszcie, wieńczące słowa kieruję do mojej przyszłej żony Klaudii Grygoruk: Dziękuję Ci za to kim jesteś.

Introduction

Regular languages

Regular languages are great. One reason for this is their remarkable definitional robustness: They have many substantially different definitions, all of which turn out to be equivalent. Examples include one-way deterministic finite automata, two-way deterministic finite automata, finite monoids and MSO logic on words. (Other well-known definitions include regular expressions and nondeterministic automata, but we do not mention them because they do not fit the narrative of this thesis.)

Apart from being aesthetically pleasing, having so many equivalent definitions can be used to simplify proofs and algorithms. For example, the simplest way to test a regular language, for nonemptiness, is to look at the language's automaton representation and check if some accepting state is reachable. Thanks to the equivalence of the definitions, this gives us an algorithm for every other representation of a regular language. On the other hand, the declarative syntax of MSO formulas is usually more convenient for defining properties than the operational, low-level syntax of automata. Also monoids and two-way automata have their advantages – e.g. monoids play an important role in the renowned Krohn-Rhodes theorem[3], and two-way automata are very useful when defining regular transductions[4].

These desirable properties of regular languages have inspired an effort to extend them to object such as trees, graphs or words over infinite alphabets. This line of research has already seen many important results: for example, Büchi showed that automata and MSO coincide for ω-words, and Rabin showed the same for infinite trees. More recent examples include work on regular languages of graphs[5] or regular languages over abstract monads[6].

The direction relevant to this thesis is the study of infinite alphabets. This

[3] See [KR65, Equation 2.2] or Theorem 7.

[4] See [EH01] or Item 3 in the introduction to Chapter 3.

[5] See [CE12].

[6] See [Boj20].

line of research, started Kaminski and Francez in the paper [KF94], has proved itself to be challenging. The study of regularity over infinite alphabets can be divided into three phases:

1. **Register automata.** This phase was initiated by [KF94]. The idea is to equip finite automata with registers so that they can store the input letters and compare them with each other[7]. (In this model, letters from the infinite alphabet can only be compared for equality; this restriction will be true for all models discussed in this thesis.) However, it was quickly[8] discovered that most of the models of language recognizers over infinite alphabets are pairwise inequivalent, with only trivial inclusions being valid. In particular one-way register automata, two-way register automata, orbit-finite monoids[9], and $\text{MSO}^{\sim}$ logic[10] on words are pairwise nonequivalent. Moreover, already two of those models – two-way register automata and $\text{MSO}^{\sim}$ – have undecidable emptiness[11]. Those results oriented the first phase towards finding possibly strongest models that still have decidable emptiness. Examples of such models include data automata[12] and alternating automata with one register[13]. For more details about the first phase, see [Boj19, Part 1].

2. **Orbit-finite automata**. This second phase was initiated by [Boj13] and [LKB14], inspired by work on nominal sets [Pit13]. The general goal of this phase is to build a definitional framework, in which the various existing automata models for infinite alphabets become simply the same as the classical models for finite alphabets, except with new notions of finite sets and functions between them. The appropriate framework turned out to be the already existing[14] category of nominal sets with finitely supported functions (denoted as $X \to_{\text{fs}} Y$), but enhanced with a novel notion of *oribit-finiteness*. Notable successes of this phase include Myhill-Nerode theorem and Angluin-style learning for deterministic orbit-finite automata[15]. More recent results include development of a theory of orbit-finite-dimensional vector spaces and decidability for equivalence of weighted register automata[16]. For more details about the second phase, see [Boj19, Sections 3 and 5].

[7] See Section 1.1 for details.

[8] Some of the inequivalence results are already present in [KF94], see [NSV04] for a complete overview. A few of them are also presented in Section 1.4.1 of this thesis.

[9] A natural extension of finite monoids for infinite alphabets. See [Boj13, Defnition 3.1].

[10] A natural extension of the MSO logic for infinite alphabets. See [NSV04, Section 2.4].

[11] For proof for two-way register automata see Section 1.4.1.3 or [NSV04, Theorem 5.3]. For $\text{MSO}^{\sim}$, see [NSV04, Theorem 3.2 and Theorem 5.1] combined with the fact that $\text{MSO}^{\sim}$ closed under compositions.

[12] See [BDM+11] or [Boj19, Section 2.1].

[13] See [DL09], [JL11], and [Boj19, Section 1.3].

[14] See [Pit13], or Section 1.2 for details and more bibliographical notes.

[15] See [LKB14, Theorem 5.2] and [MSS+17].

[16] See [BKM21] for details on both of those results.

3. **Single-use automata**. Finally (what I hope can be described as) the third phase of studies of infinite alphabets was started by [Ste18] and [BS20] and is further developed in this thesis (with some of the ideas traceable back to [Boj13] and [CLP15]). The key observation is that we can recover the definitional robustness of regular languages by introducing the *single-use restriction.* In the setting of register automata, this restriction amounts to the requirement that every read access to a register should have the side effect of destroying the register's value[17]. In a more abstract setting, the single-use restriction means replacing finitely supported functions ($X \to_{\text{fs}} Y$) with *single-use functions*[18] (denoted as $X \multimap Y$). Under the single-use restriction, one recovers many of the equivalences that were true for finite alphabets but failed for infinite alphabets, including the equivalence of the following models: one-way and two-way automata, orbit-finite monoids, and a version of MSO$^{\sim}$[19], which is the first such four-way equivalence result for infinite alphabets. Presenting and extending the research about the third phase is the main contribution of this thesis.

Transductions

Another important extension of the theory of regular languages is the study of transductions, i.e. functions of type $\Sigma^* \to \Gamma^*$. The theory of transducers is mainly interested in transduction classes of low complexity – a good litmus test is the decidability of the equivalence problem (i.e. given two transducers f and g from a given class, is it possible to check if $f(w) = g(w)$ for every w). The classification of transductions is finer than the one of languages. Relevant classes include:

Mealy machines ⊆ Letter-to-letter rational functions ⊆ Regular functions

Below, we present a brief description of each of these classes. For more details, see the introduction to Chapter 3.

Mealy machines[20] are a version of DFAs where all states are accepting, and every edge is additionally labelled by an output letter. Each such machine implements a length-preserving function, where every input position is replaced

[17] See Section 2.1 for details.

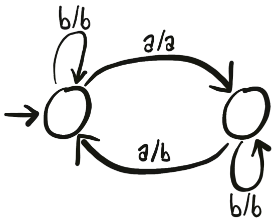
[18] The notion of single-use functions is a novel contribution of this thesis. See Defintion 7.

[19] Orbit-finite monoids were introduced in [Boj13]. Rigidly-guarded MSO$^{\sim}$ was introduced and shown to be equivalent to orbit-finite monoids in [CLP15]. One-way and two-way single-use register automata were introduced in [Ste18], and were shown to be equivalent to each other and to orbit-finite monoids in [BS20]. The definitions and the equivalence proof for orbit-finite monoids, single-use one-way automata and single-use two-way automata can also be found in this thesis. See Section 1.4.2, Definition 11, Definition 12, and Theorem 6. For a novel topological perspective see [UMB23].

[20] See [Mea55] or Item 1 in the introduction to Chapter 3.

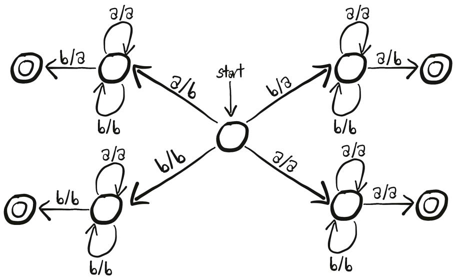
by the output label of the corresponding transition. Here is an example of a Mealy machine that implements the following function:

$$\text{“Change every second a to b”} \in \{\text{a, b}\}^* \to \{\text{a, b}\}^*$$



Rational functions[21] are all functions that can be expressed as a composition of a left-to-right Mealy machine and a right-to-left Mealy machine. Equivalent definitions include unambiguous nondeterministic Mealy machines and Mealy machines with a regular look-ahead. Here is an example of an unambiguous nondeterministic Mealy machine that implements the following function:

$$\text{“Swap the first and the last letter”} \in \{\text{a, b}\}^* \to \{\text{a, b}\}^*$$



Regular functions[22] are all functions that can be implemented by two-way transducers, i.e. a version of two-way automata, where every transition may be additionally labelled with an output letter. Regular functions exhibit a similar definitional robustness as regular languages. Equivalent definitions include MSO-transductions[23], string streaming transducers[24], regular expressions with output[25], and many others. Possibly for this reason, they have been the subject of significant research attention in recent years.[26] Here is an example of a two-way transducer that implements the following regular function:

$$\text{“Reverse the input word”} \in \{\text{a, b}\}^* \to \{\text{a, b}\}^*$$

[21] See [Eil74] or Item 2 in the introduction to Chapter 3.

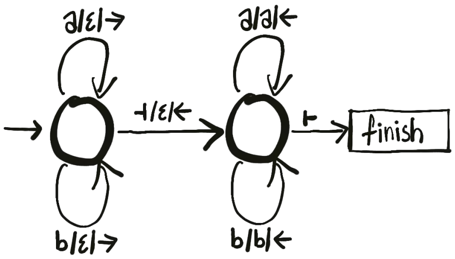
[22] See [EH01] Item 3 in the introduction to Chapter 3.

[23] See [Cou94, Section 2].

[24] See [AČe10, Section 3] or Section 4.1.2.

[25] See [AFR14, Theorems 13 and 15].

[26] For examples, see [DFJL17], [DH19], [CHL+19], [NDRP20], or [BNê23].



Without a robust theory of regularity, one cannot hope for a good theory of transducers. In particular, deciding the equivalence of two transducers is at least as hard as the emptiness problem for their underlying automaton model. For example, the equivalence problem for (multiple-use) two-way register transducers is undecidable. In Chapters 3 and 4 of this thesis, we show that under the single-use restriction, one recovers a robust theory of transducers for infinite alphabets. The main results are as follows.

In Chapter 3, we define and study *single-use register Mealy machines.* In particular, we show that they:

1. admit a Krohn-Rhodes-like decomposition[27]; and
2. have an equivalent algebraic definition[28].

The Krohn-Rhodes decomposition theorem for single-use Mealy machine is the main technical contribution of this thesis (it constitutes around one fourth of its total volume). In Chapter 3, we also define an infinite-alphabet version of the rational functions, show that it also admits Krohn-Rhodes-like decompositions, and we develop a similar algebraic theory[29] for them.

Finally, in Chapter 4, we present the theory of single-use two-way transducers. We prove that the deterministic two-way single-use register[30]:

1. also admit a Krohn-Rhodes-like decomposition;
2. are closed under compositions;
3. have decidable equivalence;
4. are equivalent to the single-use variant of copyless SSTs over infinite alphabets (modification of SSTs from [AČ11, Section 2.2]);
5. are equivalent to the infinite alphabet variant of regular list functions (modification of [BDK18, Section 6]).

[27] This was initially shown in [BS20]. In this thesis this is Theorem 8.

[28] This is a novel contribution of this thesis. See Section 3.3.

[29] See Section 3.7.

[30] All of the following results were initially shown in [BS20].

We believe that the results presented in this thesis justify the name *regular languages over infinite alphabets* for the class of languages recognized by single-use deterministic register automata, and the name *regular functions over infinite alphabets* for the class of functions definable by single-use two-way register transducers.

Contributions

This thesis is based on a single publication [BS20], with the objective of delivering its results in a cohesive narrative. Additionally, this thesis contains two novel (unpublished before) contributions: The first one is defining *single-use functions* and using them to simplify the definitions of different models of single-use automata and transducers. The second one is developing the algebraic theory for single-use Mealy machines and single-use rational functions, and using this theory to simplify the proofs of the Krohn-Rhodes decomposition theorems for single-use Mealy machines and single-use two-way transducers. The results presented in this thesis are an outcome of a close collaboration with my advisor Mikołaj Bojańczyk, and it is mostly impossible to partition the contributions individually. However, for the sake of strengthening my Ph.D. application, it might be worth noting that the original idea for the single-use restriction, which initiated this line of research, was mine.

Chapter 1

Infinite alphabets

This chapter serves two purposes: The first purpose is to put this thesis in context. The chapter presents some previously studied models of computation over infinite alphabets. The second purpose is to lay out foundations for the upcoming chapters. We present the theory of *sets with atoms* and *orbit finiteness*, which is the modern abstract vocabulary for discussing infinite alphabets. In the chapter we follow the narrative of [Boj19] – we start with a rather concrete model of *deterministic register automata* and we abstract away towards *deterministic orbit-finite automata*. Then, we prove that the two models are equivalent. Although this proof is quite technical, we include it the thesis, because it illustrates some useful techniques for working with sets with atoms. Finally, in the last section we define some of the well-studied variants of register automata (including orbit-finite monoids), and we compare their expressive powers.

1.1 Deterministic register automata

We start describing the model of *deterministic register automata* with some intuition and examples. Let us fix a countably infinite alphabet and call it $\mathbb{A}$. Introducing infinite sets to automata theory is dangerous: infinite alphabets can only be processed using infinite state spaces, and automata with unrestricted infinite state spaces can recognize all possible languages. To avoid this, we say that elements of $\mathbb{A}$ can only be compared for equality. This means that we are only going to consider languages that can be defined in terms of equality, for example:

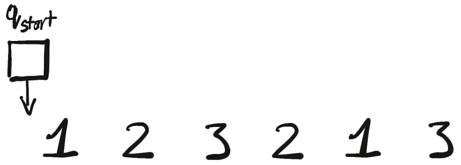
1. $\{w \in \mathbb{A}^* \mid \text{the first letter of } w \text{ appears again in } w\}$
2. $\{w \in \mathbb{A}^* \mid \text{the first and the last letter of } w \text{ are equal}\}$
3. $\{w \in \mathbb{A}^* \mid \text{there are at most three different letters in } w\}$.

(For now, we only talk about the intuition, and we do not define formally what it means for a language to be definable only in terms of equality.) A deterministic register automaton is a model that can recognize some languages of this type, including the three example languages from the list. It has a finite set of control states and a finite set of registers, in which it stores some of the letters it has seen in the input word. It can compare the values of the registers with each other and with the input letter. Here are two examples:

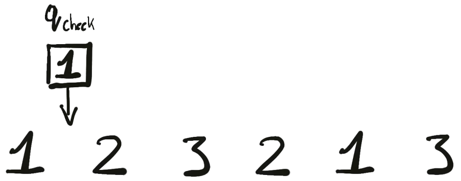
Example 1. Let us describe a deterministic register automaton, that recognizes the language.

$$\text{“The first letter appears again”} \subseteq \mathbb{A}^*$$

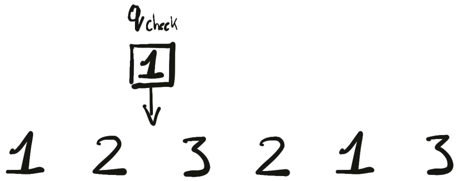
The automaton has one register and 3 control states: $\{q_{\text{start}}, q_{\text{check}}, q_{\text{found}}\}$. Let us go through the automaton's run on the word 1 2 3 2 1 3 $\in \mathbb{A}^*$ (we use natural numbers to denote elements of $\mathbb{A}$). Here is the initial configuration:



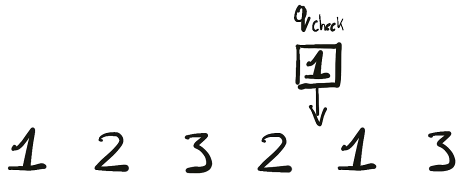
The automaton starts in q_{start} with an empty register (represented in the picture as the empty box). In its first step, the automaton stores the first letter in the register, sets its control state to q_{check}, and moves forward:



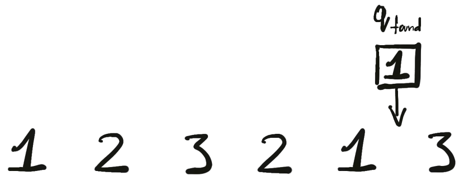
In the next step it compares the register value with the current letter. Since they are different, it simply moves forward:



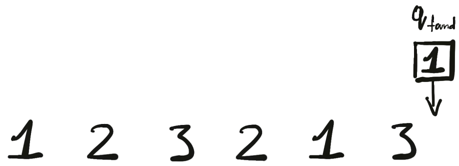
The same transition happens two more times:



Now, when the automaton compares the current letter with the register value, it finds out that they are equal. It has found a reappearance of the initial letter, so it sets its state to q_{found} and proceeds to the next letter.



In the state q_{found} the automaton ignores the input and keeps moving forward until the end of the word.

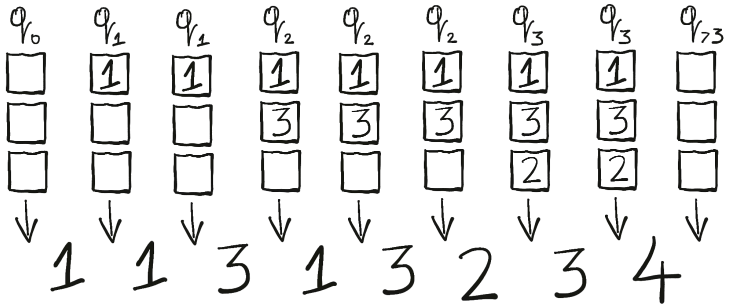


At the end of the word, the automaton accepts the input word if it is in the state q_{found}. $\triangleleft$

Example 2. The automaton that recognizes the language

$$\text{“There at most 3 letters in the word”} \subseteq \mathbb{A}^*$$

has three registers and four control states (q_0, q_1, q_2, q_3, and $q_{>3}$). After reading some part of the input, the automaton remembers the letters it has already seen (wihtout repretitions) unless it has already seen more than 3 of them. Here is an example run of the automaton:



◁

Now, let us discuss the transition function of a deterministic register automaton. The contents of the automaton's memory can be described as an element of the following set:

$$\underbrace{Q}_{\text{automatons's control state}} \times \underbrace{(\mathbb{A} + \perp)^R}_{\substack{\text{contents of each register, where} \\ \perp \text{ represents empty registers}}}$$

This means that the transition function should have the following type:

$$\underbrace{Q \times (\mathbb{A} + \perp)^R}_{\text{current memory state}} \times \underbrace{\mathbb{A}}_{\text{current letter}} \to \underbrace{Q \times (\mathbb{A} + \perp)^R}_{\text{updated memory state}}$$

As we have mentioned before, we cannot allow all transition function of this type, or else a deterministic register automaton would become an unrestricted infinite state machine (and such machines can recognize all languages, including "the length of the word is prime", and "the length of the word encodes a halting Turing machine"). In order to avoid that, we restrict the power of the transition function. The intuition behind this restriction is that the only allowed operations on registers should be:

1. comparing for equality two register values;

2. comparing for equality a register value with the input letter; and

3. saving the input letter in one of the registers.

We describe three ways of formalizing this restriction, and prove that they are equivalent:

1. **Syntactic equivariance** For this definition we provide syntax for specifying the transition function – every function that can be specified in this syntax is syntactically equivariant. A transition function is specified as a list of conditional commands, each of the following form:

$$\text{list of conditions} \to \text{list of actions}$$

To apply the transition, the automaton goes through the list of conditional comments (top-down), finds the first command in which every condition is satisfied, and performs all the actions from that command. (To make this a total function, we say that if there is no command whose all conditions are satisfied, then the automaton stays in the same configuration.) Conditions and actions are of the following types (each condition can appear both with $=$ and with $\neq$):

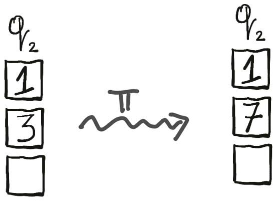
Example	Description
$\mathtt{r}_1 = \mathtt{r}_2$	Compare two register values
$\mathtt{r}_2 \neq \mathtt{input}$	Compare a register value with the current input letter
$\mathtt{r}_5 = \bot$	Check if a register is empty
$\mathtt{state} = \mathtt{q}_5$	Check that the automaton is in a particular state

Example	Description
$\mathtt{r}_2 := \mathtt{input}$	Save the current letter to a register
$\mathtt{r}_1 := \mathtt{r}_3$	Copy a register value into another register.
$\mathrm{swap}(\mathtt{r}_2, \mathtt{r}_4)$	Swaps the contents of two registers.
$\mathtt{r}_5 := \bot$	Clear the contents of a register
$\mathtt{state} := \mathtt{q}_3$	Update the state of the automaton

2. **Semantic equivariance** The intuition behind this definition is that the valid transition functions are the ones that do not discriminate between atoms i.e. the ones that commute with permutations of atoms. To express this formally, we have to extend atom permutations ($\mathbb{A} \to \mathbb{A}$) to act on the set of all memory states of a register automaton, i.e. on the set:

$$Q \times (\mathbb{A} + \bot)^R$$

We do it in a natural way, by applying the permutation to every register value, leaving the control state and the empty registers unchanged. Here is an example of an action of π that swaps atoms 3 and 7 (and does not touch other atoms):



We can naturally extend this action to the domain of the transition function:

$$(Q \times (\mathbb{A} + \bot)^R) \times \mathbb{A}$$

Finally, we say that a transition function δ is semantically equivariant if for every atom permutation π and for every x:

$$\delta(\pi(x)) = \pi(\delta(x))$$

3. Haskell-style equivariance In this definition, we use Haskell's type system. The key idea is to encode $\mathbb{A}$ as the polymorphic type:

$$\mathtt{Eq\,a} \;\Rightarrow\; \mathtt{a}$$

We continue by encoding the finite sets $R = \{r_1, \ldots, r_k\}$ and $Q = \{q_1, \ldots, q_n\}$ as variant types, explicitly enumerating all their elements:

$$\mathtt{type\,Q} = \mathtt{q_1} \,|\, \mathtt{q_2} \,|\, \ldots \,|\, \mathtt{q_n}$$

$$\mathtt{type\,R} = \mathtt{r_1} \,|\, \mathtt{r_2} \,|\, \ldots \,|\, \mathtt{r_k}$$

Then, we encode a configuration of a register automaton as:

$$\mathtt{Eq\,a} \Rightarrow (\mathtt{Q}\ ,\ \mathtt{R} \to \mathtt{Maybe\,a})$$

Finally, we define the valid transition functions to be the Haskell-definable total functions of the following type:

$$\mathtt{Eq\,a} \;\Rightarrow\; (\mathtt{Q},\, \mathtt{R} \to \mathtt{Maybe\,a}) \to \mathtt{a} \to (\mathtt{Q},\, \mathtt{R} \to \mathtt{Maybe\,a})$$

We prove that those definitions are all equivalent, starting with equivalence between semantic and syntactic equivariances. They are versions of definitions from [Boj19, Section 1.1] adapted for transition functions (original definitions work with transition relations), so this proof is very similar to the proof of Lemma 1.3 from [Boj19]:

Syntactic equivariance ⇒ Semantic equivariance: Notice that applying atom permutations to an element of the domain (i.e. $(Q \times (\mathbb{A} + \bot)^R) \times \mathbb{A}$) preserves all conditions from the syntactic definition. This means that x and $\pi(x)$ will be dispatched to the same conditional command. Also, all actions only move atoms around, so they commute with atom permutations.

Semantic equivariance ⇒ Syntactic equivariance: In this proof we look into the structure of the domains of transition functions. We say that two elements of $(Q \times (\mathbb{A} + \bot)^R) \times \mathbb{A}$ are in the same *orbit* if they differ only by an atom permutation. Formally, we define an orbit of x, to be the following set:

$$\{\pi(x) \,|\, \pi \text{ is an atom permutation}\}$$

Notice, that being in the same orbit is an equivalence relation – two orbits are either equal or disjoint. It is not hard to see that two elements of the domain belong to the same orbit, if and only if every condition from the definition of syntactic equivariance is satisfied either in both of them or in none of them. There are only finitely many of those conditions, so the following two claims are consequences of this observation:

Claim 1. *There are finitely many orbits in* $(Q \times (\mathbb{A} + \bot)^R) \times \mathbb{A}$.

Claim 2. *For every orbit* $O \subseteq (Q \times (\mathbb{A} + \bot)^R)$*, there is a list of conditions, such that an element belongs to* O*, if and only if it satisfies all the conditions from the list.*

We finish the proof by showing that we can use actions from the syntactic definition to implement the transition function for each orbit:

Claim 3. *For every orbit, there is a universal sequence of actions, that transforms every* x *in that orbit into* $\delta(x)$.

Proof. We start by showing that there is a sequence of actions that transforms some x from the orbit into $\pi(x)$. Take some x. The actions can modify the state, clear the registers, and move the atoms around, so the only difficulty is to show that every atom that appears in $\delta(x)$ also appears in x. Assume towards a contradiction that there exists $a \in \mathbb{A}$ that appears in $\delta(x)$, but not in x. Let π be an atom permutation that swaps a with a fresh atom (i.e. an atom that appears neither in x nor in $\delta(x)$) and does not touch other atoms. This means that:

$$\pi(x) = x, \quad \text{but } \pi(\delta(x)) \neq \delta(x), \quad \text{so } \delta(\pi(x)) \neq \pi(\delta(x))$$

This contradicts the assumption that δ is semantically equivariant. It follows that there exists a sequence of actions that transforms x into $\delta(x)$. Call it a_δ, and let us prove that it is universal, i.e. that for every x' from the orbit of x:

$$\delta(x') = a_\delta(x')$$

If x and x' are in the same orbit, then $x' = \pi(x)$, for some π. The function δ is semantically equivariant, so:

$$\delta(x') = \pi(\delta(x))$$

By definition of a_δ:

$$\pi(\delta(x)) = \pi(a_\delta(x))$$

Notice that every individual action commutes with atom permutations. This means that a_δ, which a composition of individual actions, also commutes with atom permutations:

$$\pi(a_\delta(x)) = a_\delta(\pi(x))$$

Finally, since $x = \pi(x')$:

$$a_\delta(\pi(x)) = a_\delta(x')$$

□

Syntactic equivariance ⇒ Haskell-style equivariance: This implication is immediate – the syntax of syntactically equivariant functions can be directly translated into Haskell.

Haskell-style equivariance ⇒ Semantic equivariance: This implication follows (for free) from [Wad89, Section 3.4].

1.2 Sets with atoms

In this section, we introduce the abstract notion of *sets with atoms*. Intuitively, they are sets whose element can store a finite number of atoms. One example of a set with atoms, that we have already seen is:

$$Q \times (\mathbb{A} + \perp)^R$$

Before we start, we include a short bibliographical note: Sets with atoms were first studied by Fraenkel in 1920, and then by Mostowski in the 1930s. Both of those authors studied them as potential alternative models of set theory, which do not admit the axiom of choice. In computer science, they were rediscovered (under the name of nominal sets) by Gabbay and Pits in [GP02] who consider their applications to semantics. In the context of formal language theory, they were first studied by Bojańczyk in [Boj13] and by Bojańczyk, Klin and Lasota in [LKB14]. This section is mainly based on [Boj19, Chapter 2] and on [Pit13, Section 1, 2, and 5].

1.2.1 Action of atoms permutations and its supports

Semantic equivariance was defined only in terms of the action of atom permutations on the set $Q \times (\mathbb{A} + \perp)^R$. This means that we could define semantic equivariance for a function $X \to Y$, as long as we know that both X and Y are equipped with an action of the group of atom permutations. Another important property of $Q \times (\mathbb{A} + \perp)^R$ is that each of its elements contains only finitely many atoms. This can be abstractly defined in terms of *supports*:

Definition 1. Let X be a set equipped with an action of the group of atom permutations. We say that a subset of atoms $\alpha \subseteq \mathbb{A}$ supports an element $x \in X$, if for every permutation π:

$$\underset{\pi(a)=a}{\text{for every } a\in\alpha,} \quad \Rightarrow \quad \pi(x) = x$$

We say that x is *equivariant*, if it is supported by the empty set. ◁

For example, the element $(1, 2, 1) \in \mathbb{A}^3$ (equipped with the natural action) is supported by sets $\{1, 2\}$ and $\{1, 2, 3\}$, but not by $\{1\}$. More generally, every element $(x, y, z) \in \mathbb{A}^3$ is supported by the finite set $\{x, y, z\}$.

Definition 2. A *set with atoms* is a set equipped with an action of atom permutations such that all its elements have finite supports. ◁

For example, the set

$$Q \times (\mathbb{A} + \perp)^R$$

is a set with atoms, as long as R is finite. This is because every element of the set is supported by the set of at most $|R|$ atoms. Another example of set with atoms is the set of all finite words over atoms i.e. $\mathbb{A}^*$. This is because every word has a support no bigger than the word's length. On the other hand, the set of all subsets of atoms $P(\mathbb{A})$ is not finitely supported. This is because elements of $P(\mathbb{A})$ that are neither finite nor cofinite have infinite supports:

Lemma 1. *A subset $X \subseteq \mathbb{A}$ is finitely supported, if and only if either X or $(\mathbb{A} - X)$ is finite.*

Proof. Take a subset of atoms X, and its potential finite support α. An α-permutation π modifies X (meaning that $\pi(X) \neq X$), if and only if π transforms some atom from X into an atom from outside of X. This is only prevented if α contains all elements of X or all elements from $\mathbb{A} - X$. This is possible if and only if one of those sets is finite. □

It follows that sets with atoms are not closed under the powersets (i.e. $P(X)$ might not be a set with atoms even if X is). Instead, we define $P_{\text{fs}}(X)$, which is the set of all finitely supported subsets of X. For example, $P_{\text{fs}}(\mathbb{A})$ is the set of all finite and cofinite subsets of $\mathbb{A}$. It is worth pointing out that $P_{\text{fs}}(X)$ is usually different from the set of all finite and cofinite subsets of X. For example, the set $\{(x, 4) \mid x \in \mathbb{A}\}$ belongs to $P_{\text{fs}}(\mathbb{A}^2)$, and it is neither finite nor cofinite. It is not hard to see that if X is a set with atoms, then so is $P_{\text{fs}}(X)$.

Notice that equivariant subsets of sets with atoms are sets with atoms themselves (with permutation action inherited from the superset). Note that this is only true for equivariant subsets: consider $\{7, 8\}$ – a finitely supported subset of $\mathbb{A}$. It is very easy to find a π, for which $\pi(7) \notin \{7, 8\}$. This means that the permutation action inherited from the superset (i.e. $\mathbb{A}$) is not valid action for the set $\{7, 8\}$.

Another class of sets with atoms are the *atomless sets*, which are sets equipped with the trivial action:

$$\pi(x) = x \quad \text{for every } \pi$$

Every element of an atomless set is equivariant.

1.2.1.1 The finitely supported relations and functions

Sets with atoms are closed under many classical operations, including the product:

Lemma 2. *If X and Y are sets with atoms, then so is $X \times Y$, with the following permutation action:*

$$\pi((x, y)) = (\pi(x), \pi(y))$$

Proof. It is easy to see that, if x is supported by α and y is supported by β, then (x, y) is supported by $\alpha \cup \beta$. (Note that the lemma does not extend to infinite products.) □

A *finitely supported relation* between two sets with atoms X and Y is a finitely supported subset of $X \times Y$, i.e. an element of $P_{\text{fs}}(X \times Y)$. This means that the permutation action on relations is defined as follows:

$$x\ (\pi\!\sim)\ y \quad \overset{\text{def}}{\iff} \quad \pi(x) \sim \pi(y)$$

Sets with atoms are closed under equivariant quotients: Let $(\sim) \subseteq X \times X$ be an equivariant equivalence relation, then $X_{/\sim}$ is a set with atoms. Its permutation action defined as follows:

$$\pi\left([x]_{\sim}\right) = [\pi\, x]_{\sim}$$

It is easy to see that $[x]_{\sim}$ is supported by whatever supports x

A *finitely supported function* is a finitely supported relation that happens to be a function. (i.e. for all x there is exactly one y such that $x \sim y$). This leads to the following definition of permutation action on functions:

$$\pi(f) = \pi \circ f \circ \pi^{-1}$$

It follows that a function is supported by α if, and only if it for every π that is an α-permutation:

$$f(\pi(x)) = \pi(f(x))$$

We denote the set of all finitely supported functions between two sets with atoms as $X \to_{\mathrm{fs}} Y$. Similarly, we write $X \to_{\mathrm{eq}} Y$ for the set of all equivariant functions. The intuition behind the next lemma is that finitely supported functions cannot create new atoms:

Lemma 3. *If x is supported by α and f is supported by β then $f(x)$ supported by $\alpha \cup \beta$. In particular, if f is equivariant, then $f(x)$ is supported by α.*

Proof. Choose any $(\alpha \cup \beta)$-permutation π. Then, because π is both an α-permutation and a β-permutation, we obtain that:

$$\pi(f(x)) \overset{\beta \text{ supports } f}{=} f(\pi(x)) \overset{\alpha \text{ supports } x}{=} f(x)$$

□

Equivariant and finitely supported functions are closed under compositions – if f is supported by α and g is supported by β, then $f \circ g$ is supported by $\alpha \cup \beta$.[1]

We say that two sets with atoms (X and Y) are isomorphic, if there exists an equivariant bijection $f : X \to_{\mathrm{eq}} Y$, i.e. an equivariant function that is a surjection and an injection. The following lemma states that equivariant bijections have equivariant inverses.

Lemma 4. *Every finitely supported bijection $f : X \to Y$, has a finitely supported inverse function $f^{-1} : Y \to X$. Moreover, f and f^{-1} have the same supports, i.e. for every finite $\alpha \subset \mathbb{A}$:*

$$\alpha \textit{ supports } f \quad \iff \quad \alpha \textit{ supports } f^{-1}$$

[1] This means that there are two types of categories over sets with atoms. The first one is the category of all functions supported by some α. When $\alpha = \emptyset$, this is the category of all equivariant functions – the **Nom** studied in [Pit13]. The other type of category is the broader category of all finitely-supported functions (where the support depends on the functions).

Proof. Functions are defined as relations, so $f \subseteq X \times Y$. This means that we can define:

$$f^{-1} = \{(y, x) \mid (x, y) \in f\}$$

Since f is a bijection, then f^{-1} is a function. It is easy to see that f^{-1} is the inverse of f and that it has the same supports as f. □

A similar argument shows that all finitely supported injections admit a partial one-sided inverse. However, as we will see in Section 1.3.1, there are equivariant surjections that do not admit equivariant (or even finitely supported) one-sided inverses.

1.2.2 Orbit-finite sets

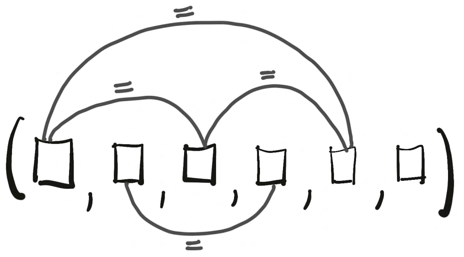
Finite sets with atoms are not very interesting – it is not hard to see that all finite sets with atoms have to be atomless. It turns out, however, that there exists a suitable analogue of finiteness for sets with atoms called *orbit finiteness*[2]. Intuitively, a set is orbit-finite if it has only finitely many elements *up to atom permutations.* Formally, we define an orbit[3] of an element $x \in X$ to be

$$\{\pi(x) \mid \pi \text{ is an atom permutation }\}.$$

Notice that every two orbits are either equal or disjoint, which means that they divide X into equivalence classes. We say that X is orbit-finite if it has finitely many orbits. For example, the set of all atoms $\mathbb{A}$ has only one orbit, so it is orbit-finite. Tuples of atoms are orbit-finite as well:

Lemma 5. *For every k, the set $\mathbb{A}^k$ is orbit-finite.*

Proof. An orbit of $\mathbb{A}^k$ can be defined by an equality pattern such as the one below (for $k = 6$):



It represents a tuple where atoms on positions 1, 3 and 5 are equal to each other (and different from all the other atoms); atoms on positions 2 and 4 are equal to each other (and different from all the other atoms); and the atom on position 6 is only equal to itself. For every k, there are only finitely many such patterns, so $\mathbb{A}^k$ is orbit-finite. □

[2] It was first introduced by Bojańczyk in [Boj13]

[3] This is the same orbit as in the proof of Semantic equivariance ⇒ Syntactic equivariance from Section 1.1.

The number of orbits in $\mathbb{A}^k$ is finite but very large, so it is sometimes useful to consider the set $\mathbb{A}^{(k)} \subseteq \mathbb{A}^k$ of tuples whose atoms are pairwise distinct – it has only one orbit (regardless of k).

An example of an orbit-infinite set is $\mathbb{A}^*$ – the length of the word is preserved by the permutation action, so there is at least one orbit for each word length. Perhaps surprisingly, the set $P_{\text{fs}}(\mathbb{A})$ is also orbit-infinite. The argument is the same as for $\mathbb{A}^*$ – the size of a set is preserved by the permutation action, so there is at least one orbit for each finite set size. For the same reason, the following set of finitely supported functions is orbit-infinite as well:

$$\mathbb{A} \to_{\text{fs}} \{\text{yes}, \text{no}\}.$$

This means that unlike classical finiteness, orbit finiteness is not preserved under powersets and function spaces. (In the next section, we will see that this causes some of the results from finite automata theory to fail for infinite alphabets.) Still, orbit finiteness is preserved by many classical combinators (and many classical results hold for infinite alphabets). Here are some of the operations that preserve orbit finiteness:

Lemma 6. *If X and Y are orbit-finite, then the following sets are orbit-finite as well: $X \times Y$, $X + Y$, and $X_{/\sim}$ (where $\sim$ is an equivariant equivalence relation).*

Proof. Cases $X + Y$ and $X_{/\sim}$ are easy: To show that $X + Y$ is orbit-finite, we notice that:

$$\#\text{orbits of } (X + Y) = \#\text{orbits of } X + \#\text{orbits of } Y$$

To show that $X_\sim$ is orbit-finite, we notice that:

$$x \text{ is in the same orbit as } y \quad \Rightarrow \quad [x]_\sim \text{ is in the same orbit as } [y]_\sim.$$

It follows that $X_\sim$ has as most as many orbits as X.

The most interesting case is $X \times Y$. We start the proof by introducing a way of representing orbit-finite sets:

Claim 4. *Every orbit-finite set is isomorphic*[4] *to a set of the form*

$$\mathbb{A}^{(k_1)}{}_{/\sim_1} + \mathbb{A}^{(k_2)}{}_{/\sim_2} + \ldots + \mathbb{A}^{(k_n)}{}_{/\sim_n}$$

where $\sim_i$ are equivariant equivalence relations.

Proof. Every orbit-finite set can be decomposed as a disjoint sum of its orbit, so it suffices to show that every single-orbit set is isomorphic to

$$\mathbb{A}^{(k)}{}_{/\sim}, \quad \text{for some } k \text{ and } \sim.$$

[4] Remember that we require the isomorphism to be an equivariant function.

Let X be a single-orbit set. Take some $x \in X$, and let α be a finite support of x. Arrange the elements of α in any order to form a tuple $\bar{\alpha} \in \mathbb{A}^{(k)}$, where $k = |\alpha|$. Define a function $f : \mathbb{A}^{(k)} \to X$ as follows (remember that a function is a special kind of a relation):

$$f = \{(\pi(\bar{\alpha}), \pi(x)) \mid \text{for each atom permutation } \pi\}$$

First, we show that f is well-defined (i.e. that it produces exactly one output for every argument): The set $\mathbb{A}^{(k)}$ only has one orbit, so for every $\bar{\beta} \in \mathbb{A}^{(k)}$ there is a π such that $\pi(\bar{\alpha}) = \bar{\beta}$. It follows that f produces at least one result for every argument. To show that it produces at most one result for each argument, we take some π_1 and π_2, such that $\pi_1(\bar{\alpha}) = \pi_2(\bar{\alpha})$, and we show that $\pi_1(x) = \pi_2(x)$: Both π_1 and π_2 are α-permutations, so $(\pi_2^{-1} \circ \pi_1)$ is an α-permutation as well. Since x is supported by α, it follows that:

$$\pi_2^{-1}(\pi_1(x)) = x$$

It follows that $\pi_1(x) = \pi_2(x)$. Now, let us show that f is equivariant: take a $\bar{\beta} \in \mathbb{A}^{(k)}$ and an atom permutation π. We know that there is some ρ, for which $\bar{\beta} = \rho(\bar{\alpha})$. It follows that:

$$\pi(f(\bar{\beta})) = \pi(f(\rho(\bar{\alpha}))) = \pi(\rho(x)) = f(\pi(\rho(\bar{\alpha}))) = f(\pi(\bar{\beta}))$$

It is also not hard to see that f is a surjection (because X is a single orbit). It follows that if we divide $\mathbb{A}^{(k)}$ by f's kernel, we get an isomorphism:

$$f' : \mathbb{A}^{(k)}{}_{/\ker f} \to X$$

To finish the proof of the claim, notice that since f is equivariant then so are $\ker f$ and f'. □

Now, take some orbit-finite X and Y and show that $X \times Y$ is also orbit finite. We start by applying Claim 4, to both X and Y. Then apply the distributivity of products over disjoint sums, to obtain that $X \times Y$ is isomorphic to a disjoint sum of products of the following form:

$$\mathbb{A}^{(k)}{}_{/\sim_i} \times \mathbb{A}^{(l)}{}_{/\sim'_j}$$

Disjoint sums preserve orbit finiteness, so we are left with showing that each such product is orbit-finite: First, notice that each such product is isomorphic to:

$$\left(\mathbb{A}^{(k)} \times \mathbb{A}^{(l)}\right)_{/(\sim_i \sim'_j)}$$

where $(\sim_i \sim'_j)$ is a relation that independently checks that $\sim_i$ holds on the first k coordinates and $\sim'_j$ holds on the last l coordinates. This is an equivariant equivalence relation, so dividing by it preserves orbit finiteness. This leaves us with showing that $\mathbb{A}^{(k)} \times \mathbb{A}^{(l)}$ is orbit-finite. Since $\mathbb{A}^{(k)} \times \mathbb{A}^{(l)} \subseteq \mathbb{A}^{k+l}$, this follows from the following claim (and Lemma 5):

Claim 5. *Equivariant subsets of orbit-finite sets are orbit-finite.*

Proof. Let Y be an equivariant subset of an orbit-finite X. Notice, that if two elements are in different orbits in Y, then they are also in different orbits in X. It follows that X contains at least as many orbits as Y. □

□

We conclude this section, with a table summarizing closure properties of sets with atoms and orbit-finite sets:

Operation	Preserves sets with atoms?	Preserves orbit-finite sets?
$X \times Y$	Yes	Yes
$X + Y$	Yes	Yes
$X_{/\sim}$ (for equivariant $\sim$)	Yes	Yes
Equivariant subsets	Yes	Yes
X^*	Yes	No
$P(X)$	No	No
$P_{\text{fs}}(X)$	Yes	No
$X \to Y$	No	No
$X \to_{\text{fs}} Y$	Yes	No

1.3 Deterministic orbit-finite automaton

In this section we introduce a model of computation that generalizes the deterministic finite automaton. This is a very natural model that deals with infinite (but orbit-finite) alphabets. Later, we argue that it is equivalent to register automata. The section is mostly based on [Boj19, Section 5.2 and 6.2], but it also discuses techniques from [BS20, Section B.1] and [Pit13, Section 4].

Definition 3. A *deterministic orbit-finite automaton* consists of:

1. an orbit-finite alphabet Σ;
2. an orbit-finite set of states Q;
3. an equivariant initial state $q_0 \in Q$;
4. an equivariant subset of accepting states $Q_{\text{acc}} \subseteq Q$;
5. and an equivariant transition function

$$f : Q \times \Sigma \to_{\text{eq}} Q.$$

◁

Such an automaton defines a language over Σ. To see this in action, let us define a deterministic orbit-finite automaton, recognizing the language

$$\{w \in \mathbb{A}^* \mid w \text{ has at most 3 different letters}\}$$

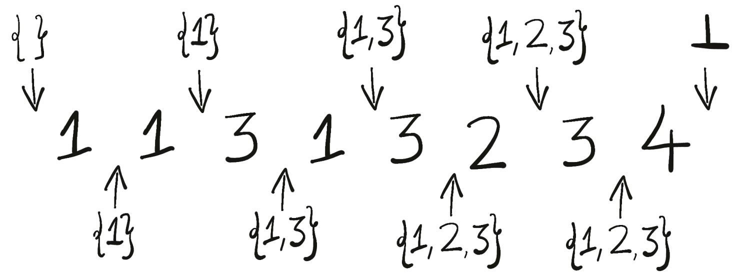
First, we notice that $\Sigma = \mathbb{A}$, which is an orbit-finite set. Then, let us define the automaton's set of states as:

$$Q = \underbrace{\binom{\mathbb{A}}{\leq 3}}_{\text{subsets of at most 3 atoms}} + \underbrace{\perp}_{\substack{\text{represents sets with} \\ \text{more than 3 atoms}}}$$

This is an orbit-finite set – it has 5 orbits: one orbit for each size of the set (i.e. 0, 1, 2, and 3) and one for the element $\perp$. The initial state is the empty set, and the accepting states are all the states except of $\perp$. The transition function is defined as follows:

$$f(q, a) = \begin{cases} q \cup \{a\} & \text{if } q \neq \perp \text{ and } |q \cup \{a\}| \leq 3 \\ \perp & \text{otherwise} \end{cases}$$

The transition function is easily seen to be equivariant. Here is an example (rejecting) run of this automaton:



1.3.1 Register automata, orbit-finite automata, and straight sets

We would like to prove that deterministic orbit-finite automata and deterministic register automata have equal expressive powers. However, the two models are slightly incompatible: register automata can only recognize languages over the alphabet[5] $\mathbb{A}$, whereas orbit-finite automata can recognize languages over every orbit-finite alphabet. The statement of the equivalence theorem has to account for this incompatibility:

Theorem 1. *Deterministic register automata and deterministic orbit-finite automata recognize the same languages over the alphabet* $\mathbb{A}$.

[5] Register automata are usually defined to work over the alphabet $\Sigma \times \mathbb{A}$ (where Σ is some finite set). Theorem 1 can be adapted to this type of automata as well.

We start with discussing the more difficult part of the proof, which is translating orbit-finite automata to register automata. Let us illustrate the problem with this translation, by discussing a failed attempt: Take an orbit-finite automaton $\mathcal{A}$ and let Q be its set of states. Thanks to Claim 4, we know that Q is isomorphic to:

$$\mathbb{A}^{(k_1)}{}_{/\sim_1} + \mathbb{A}^{(k_2)}{}_{/\sim_2} + \ldots + \mathbb{A}^{(k_n)}_{/\sim_n}$$

This set can be represented as:

$$\mathbb{A}^{(k_1)} + \mathbb{A}^{(k_2)} + \ldots + \mathbb{A}^{(k_n)}$$

Each element of this set can be represented as a memory state (where $k = \max(k_i)$):

$$Q' := \underbrace{\{1, 2, \ldots, n\}}_{\text{the index } i} \times \underbrace{(\mathbb{A} + \perp)^k}_{\substack{k_i \text{ atoms} \\ \text{followed by } \perp\text{'s}}}$$

It is important to point out that this representation is usually partial (i.e. some elements from Q' might not represent any element) and not injective (i.e. some elements from Q might have more than one representation in Q'), but it is always surjective (every element from Q has to have at least one representation in Q'). Now, we would like to lift $\mathcal{A}$'s transition function $\delta : Q \times \mathbb{A} \to Q$ to work on this representation, obtaining $\delta' : Q' \times \mathbb{A} \to Q'$. Here is an attempt:

$$\delta'(x, a) = \text{a representant of } (\delta(\text{element represented by } x, a))$$

Surprisingly, this is not always possible – we require δ' to *choose* a representant, and some sets with atoms do not admit choice:

Claim 6. *There is no finitely supported function $f : \binom{\mathbb{A}}{2} \to_{fs} \mathbb{A}^{(2)}$, that chooses a tuple representation of a set. In other words, there is no such f, that for all $a, b \in \mathbb{A}$ the value $f(\{a, b\})$ is either equal to (a, b) or to (b, a).*

Proof. Suppose that there is such f, and let α be a finite support of f. Take some two different atoms a and b from outside of α. Suppose without the loss of generality that $f(\{a, b\}) = (a, b)$. Let π be the atom automorphism that swaps a with b and does not touch other atoms. Notice that this π is an α-permutation, and that $\{a, b\} = \pi(\{a, b\})$. This leads to a contradiction:

$$(a, b) = f(\{a, b\}) = f(\pi(\{a, b\})) = \pi((a, b)) = (b, a).$$

□

The source of those problems with choice are symmetries such as $\{a, b\} = \{b, a\}$. In the representation from Claim 4 they manifest themselves as the equivalence relations $(\sim_i)$. This observation motivates the definition of *straight sets*, which are orbit-finite sets that do not exhibit those symmetries:

Definition 4. An orbit-finite set is *straight* if it is isomorphic to a set of the following form:

$$\mathbb{A}^{(k_1)} + \mathbb{A}^{(k_2)} + \ldots + \mathbb{A}^{(k_n)}$$

◁

For example, $\mathbb{A}^2$ is a straight set:

$$\mathbb{A}^2 \simeq \underbrace{\mathbb{A}}_{\text{pairs of the form } (x,x)} + \underbrace{\mathbb{A}^{(2)}}_{\text{pairs of the form } (x,y)} .$$

In general, every $\mathbb{A}^k$ is a straight set – it is isomorphic to the disjoint union of one $\mathbb{A}^{(k_i)}$ per equality pattern, where k_i is the number of distinct atoms in that equality pattern (see the proof of Lemma 5). Using a similar proof as the one of Lemma 6, we can show that straight sets are closed under Cartesian products and disjoint sums.

Let us now briefly discuss the structure of straight sets: One can think of an element $x \in \mathbb{A}^{(k_1)} + \mathbb{A}^{(k_2)} + \ldots + \mathbb{A}^{(k_n)}$ as a coloured tuple:

$$x = i(\bar{x}) \quad \text{for} \quad \underbrace{i \in \{1, 2, \ldots, n\}}_{\text{the colour}} \quad \underbrace{\bar{x} \in \mathbb{A}^{(k_i)}}_{\text{the tuple}}$$

We define

$$\dim x = k_i \quad \text{and} \quad \sigma(x) = \{\text{all the atoms from } \bar{x}\}$$

It is easy to see that both dim and σ are equivariant functions, and that $\sigma(x)$ supports x. Thanks to the following lemma we can extend the functions σ and dim to all straight sets:

Lemma 7. *For every straight X, for every two isomorphisms:*

$$f : X \to_{eq} \mathbb{A}^{(k_1)} + \ldots + \mathbb{A}^{(k_n)} \quad g : X \to_{eq} \mathbb{A}^{(l_1)} + \ldots + \mathbb{A}^{(l_m)},$$

and for every x, it holds that $\sigma(f(x)) = \sigma(g(x))$. (In particular, since $\dim(x) = |\sigma(x)|$*, it follows that* $\dim(f(x)) = \dim(g(x))$*.)*

Proof. Let $h = g \circ f^{-1}$. It is an isomorphism of the following type:

$$h : \mathbb{A}^{(k_1)} + \mathbb{A}^{(k_2)} + \ldots + \mathbb{A}^{(k_n)} \ \to_{\text{eq}} \ \mathbb{A}^{(l_1)} + \mathbb{A}^{(l_2)} + \ldots + \mathbb{A}^{(l_m)}$$

It is enough to show that for all $x \in \mathbb{A}^{(k_1)} + \mathbb{A}^{(k_2)} + \ldots + \mathbb{A}^{(l_n)}$, it holds that $\sigma(x) = \sigma(h(x))$. The set $\sigma(x)$ supports x, so by Lemma 3 it supports $h(x)$ as well. It follows that $\sigma(h(x)) \subseteq \sigma(x)$. The other inclusion can be proved in the same way, because by Lemma 4, the function h^{-1} is equivariant. □

Finally, let us show that straight sets admit choice:

Lemma 8 (Straight Uniformisation)**.** *Let X and Y be straight sets, and let R be an equivariant relation $R \subseteq X \times Y$. If for every $x \in X$, there exists at least one $y \in Y$ for which $x\,R\,y$, then there exists a* finitely supported *function $r : X \to Y$ such that $x\,R\,r(x)$, for every $x \in X$.*

Before we prove the lemma, let us notice that r does not have to be equivariant:

Example 3. Consider the following $R \subseteq \mathbb{A} \times \mathbb{A}^{(2)}$:

$$R = \{(x,\ (x,y)) \mid x, y \in \mathbb{A}, \text{ such that } x \neq y\}$$

It satisfies the conditions of Lemma 8, so it should have a finitely supported uniformization. Here is an example uniformization supported by $\{4, 5\}$:

$$r(x) = \begin{cases} (x,5) & \text{if } x = 4 \\ (x,4) & \text{otherwise} \end{cases}$$

On the other hand, it is not hard to see that R has no equivariant uniformization (this follows from Lemma 3). ◁

As we can see, the reason why there might not be an equivariant r, is that x might not have enough atoms to construct a matching element of Y. The following lemma formalizes this intuition. We prove it, before we prove Lemma 8:

Lemma 9. *If $R \subseteq X \times Y$ is as in Lemma 8, but additionally for every x there is a y such that $x\,R\,y$, and*

$$\sigma(x) \text{ supports } y,$$

then there is an equivariant *r that uniformizes R.*

Proof. Let us fix straight equivariant isomorphisms for X, Y:

$$f_X : X \to_{\text{eq}} \mathbb{A}^{(k_1)} + \ldots + \mathbb{A}^{(k_n)} \qquad f_Y : Y \to_{\text{eq}} \mathbb{A}^{(l_1)} + \ldots + \mathbb{A}^{(l_m)}$$

This means that elements of X and Y can be seen as coloured tuples. We take some $x \in X$, and we show how to construct $r(x) \in Y$ in an equivariant way:

1. First, we consider only those y's, such that xRy, and y is supported by $\sigma(x)$. (This means that $\sigma(y) \subseteq \sigma(x)$.)
2. Out of those y's, we prefer the ones labelled with a smaller colour.
3. To choose one of the remaining tuples, we annotate each atom in every remaining tuple y with its position in x (remember that x is a tuple of atoms). Then, we chose $r(x)$ to be the y whose annotation is lexicographically smallest.

□

We are now ready to proof the Straight Uniformization Lemma:

Proof. Let $\dim X$ be the maximal dimension of an element from X (and analogously for $\dim Y$). Let $k = \dim(X) + \dim(Y)$, and define $R' \subseteq (X \times \mathbb{A}^{(k)}) \times Y$, to be a relation that ignores its $\mathbb{A}^{(k)}$ part, and otherwise is equal to R:

$$R' = \{((x, \bar{a}), y) \mid (x, y) \in R,\ \bar{a} \in \mathbb{A}^{(k)}\}$$

Now, we would like to apply Lemma 9 to R'. The following claim proves that R satisfies the lemma's assumptions:

Claim 7. *For every $\bar{a} \in \mathbb{A}^{(\dim(X)+\dim(Y))}$ and for every $x \in X$, there is a $y \in Y$, such that $(x, \bar{a})\, R'\, y$, and y is supported by $\sigma(x, \bar{a})$.*

Proof. Take some $\bar{a} \in \mathbb{A}^{(\dim(X)+\dim(Y))}$ and some $x \in X$. By assumption, we know that there exists $y \in Y$, such that $x\, R\, y$. Define π to be a permutation that moves all atoms from $\sigma(y) - \sigma(x)$ into $\sigma(\bar{a}) - \sigma(x)$, and does not touch any atoms from $\sigma(x)$. We know that such a π always exists, because

$$|\sigma(y) - \sigma(x)| \leq \dim Y \quad \text{and} \quad |\sigma(\bar{a}) - \sigma(x)| \geq \dim Y$$

It follows that:

$$\sigma(\pi(y)) \subseteq \sigma(\bar{a}) \cup \pi(\sigma(x))$$

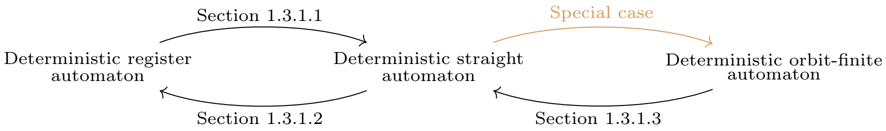
Moreover π is a $\sigma(x)$-permutation, x is supported by $\sigma(x)$, and R is equivariant, so $x\, R\, \pi(y)$. It follows that $(x, \bar{a})\, R'\, y$. □

Let r' be the equivariant uniformization of R' produced by Lemma 9. Pick any tuple of atoms $\bar{a} \in \mathbb{A}^{(\dim(X)+\dim(Y))}$, and define r as:

$$r(x) = r'(x, \bar{a})$$

It is easy to see that r is an $\bar{a}$-supported uniformization of R. □

We are now ready to prove Theorem 1. We define a *deterministic straight automaton* to be a variant of the orbit-finite automaton, where the alphabet and the set of states have to be straight. We use this model to break the proof of Theorem 5 into smaller steps:


Section 1.3.1.1
Special case
Deterministic register automaton
Deterministic straight automaton
Deterministic orbit-finite automaton
Section 1.3.1.2
Section 1.3.1.3

1.3.1.1 Register automaton ⇒ Straight automaton

The set of all possible memory configurations of a register automaton

$$Q \times (\mathbb{A} + \perp)^R$$

is straight and as such it can be directly used as the set of states of a straight automaton. The transition function, initial configuration, and set of accepting functions are equivariant by definition.

1.3.1.2 Straight automaton ⇒ Register automaton

Let the state space of the straight automaton be equal to :

$$Q \simeq \mathbb{A}^{(k_1)} + \mathbb{A}^{(k_2)} + \ldots + \mathbb{A}^{(k_n)}$$

As mentioned before, one can think of this set as non-repeating tuples of atoms coloured in one of the n colours and therefore elements of Q can be represented as:

$$Q' = \underbrace{\{1, 2, \ldots, n\}}_{\text{colour of the tuple}} \times \underbrace{(\mathbb{A} + \bot)^{(\max k_i)}}_{\substack{\text{atoms of the tuple} \\ \text{(followed by } \bot\text{'s)}}}$$

This representation is partial, but injective and surjective. It follows, by a similar argument as in Lemma 4, t has a (total) one-sided inverse. This means that every element of Q has a canonical representation in Q'. We use it to lift the transition function δ, to work on representations:

$$\delta'(x, a) = \textit{the} \text{ representation of } (\delta(\text{the element represented by } x, a))$$

We are now left with some technical details: The initial state of the straight automaton is equivariant, so its representation has to be of the form $c_0(\bot, \bot, \ldots, \bot)$, for some colour c_0. We choose this colour to be the initial state of the register automaton. Moreover, since the register automaton accepts by control state and not by memory configuration, every time it moves forward it has to check if its configuration is accepting, and remember this information in its control state.

1.3.1.3 Orbit-finite automaton ⇒ Straight automaton

This construction makes use of the Lemma 8 to fix the failed attempt from the beginning of this section. Take an orbit-finite automaton $\mathcal{A}$. Its set of states (Q) is orbit-finite, so thanks to Claim 4 it is isomorphic to:

$$\mathbb{A}^{(k_1)}{}_{/\sim_1} + \mathbb{A}^{(k_2)}{}_{/\sim_2} + \ldots \mathbb{A}^{(k_n)}_{/\sim_n}$$

It can be therefore represented as the straight set:

$$\bar{Q} = \mathbb{A}^{(k_1)} + \mathbb{A}^{(k_2)} + \ldots \mathbb{A}^{(k_n)}$$

This time the representation function (call it $h : \bar{Q} \to Q$) is surjective, but not injective. Let f be the transition function of $\mathcal{A}$. Consider the relation:

$$F \subseteq (\bar{Q} \times \mathbb{A}) \times \bar{Q}$$

$$F = \{((q,\, a),\; p) \mid q, p \in \bar{Q},\; a \in \mathbb{A}, \text{ such that } \delta(h(q),\, a) = h(p)\}$$

Since r is surjective, F satisfies the assumption from Lemma 8. We use it to obtain a *finitely supported* transition function $\bar{\delta} : \bar{Q} \times \mathbb{A} \to \bar{Q}$. The set of accepting states is simply the set of all representations of accepting states. For the initial state, we repeat the uniformization construction. Define $I = \{\bullet\}$ to be the atomless singleton and define:

$$B \subseteq 1 \times \bar{Q} \quad B = \{(\bullet\,,\, q) \mid r(q) \text{ is the initial state of } \mathcal{A}\}$$

Let $\bar{i}$ be B's uniformization, and pick $\bar{i}(\bullet)$ as the initial state of the straight automaton. This almost finishes the construction, but so far we have constructed a *finitely supported* transition function, and we require an equivariant one. There are two ways to fix this: First, we can show that orbit-finite sets can be represented by straight sets in a way that *reflects supports*, which means that for all α and x:

$$\alpha \text{ supports } r(x) \Rightarrow \alpha \text{ supports } x$$

With this stronger representation, we can use Lemma 9 and obtain an equivariant transition function and initial state, finishing the proof of Theorem 1. Another approach to prove the theorem would be to show how to remove the surplus atoms from the support of the straight automaton. Both of those approaches illustrate techniques that are going to be useful later in this thesis, so in the next two subsections we present both of them. (Although any one of them could finish the proof of Theorem 1.)

1.3.2 Support-reflecting straight representations

Most of this section is dedicated to proving the following straight representation lemma. The proof is based on the proof of [Boj19, Lemma 6.2].

Lemma 10. *For every orbit-finite set X there is a straight set $\bar{X}$ and a surjective representation function:*

$$f : \bar{X} \to X$$

that reflects supports*, i.e. for every x and α*

$$\alpha \textit{ supports } f(x) \implies \alpha \textit{ supports } x$$

Proof. Straight sets are closed under disjoint unions, so it is enough to prove the claim for an X that is a single orbit. We start the proof by applying Claim 4 to obtain an isomorphism:

$$\mathbb{A}^{(n)}/_{\sim} \to X$$

This defines a natural straight representation:

$$f : \mathbb{A}^{(n)} \to X$$

This straight representation may or may not reflect supports. If it does, then there is nothing more to do. If it does not, then there is a $\bar{x} \in \mathbb{A}^{(k)}$ and α, such that α supports $f(\bar{x})$, but it does not support $\bar{x}$. This means that, there is an atom in $\bar{x}$ that is not present in α. Assume without loss of generality that this is x_n – the last atom of $\bar{x}$. Interestingly, this means that f does not depend on its last argument:

Claim 8. *For all pairwise distinct atoms $y_1, y_2, \ldots, y_n, y'_n$:*

$$f(y_1, \ldots, y_{n-1}, y_n) = f(y_1, \ldots, y_{n-1}, y'_n)$$

Proof. Since f is equivariant, we can prove the claim, by proving it for some particular $y_1, y_2, \ldots, y_n, y'_n$ that we choose (this is because for every $z_1, \ldots, z_n, z'_n$, there is a π that maps it to $y_1, y_2, \ldots, y_n, y'_n$). We choose to prove it for $x_1, \ldots, x_n, x'_n$, where $x_1, \ldots, x_n$ are elements of $\bar{x}-$inherited from the proof of Lemma 10−, and x'_n is some atom that does not appear in $\bar{x}$ or in α. Define $\bar{x}' := x_1, \ldots, x_{n-1}, x'_n$, and let us show that $f(\bar{x}) = f(\bar{x}')$. Let π be the permutation that swaps x_n with x'_n and does not touch other atoms. Such π is an α-permutation, so:

$$f(\bar{x}) = \pi(f(\bar{x})) = f(\pi(\bar{x})) = f(\bar{x}')$$

□

Using the claim, we define an equivariant $f' : \mathbb{A}^{(n-1)} \to X$ in the following way:

$$f'(\bar{x}) = \text{the only element of } \{f(\bar{x}, x_n) \mid x_n \in \mathbb{A}\}$$

In this way we obtain a representation of a lower dimension. We repeat this process until we obtain a representation that reflects the supports. (Note that, this process will always end, because for $n = 0$, we obtain a representation $I \to X$, which always reflects supports, because I is atomless.) □

Having proved the lemma, we go back and finish the translation *Orbit-Finite Automata ⇒ Straight Automata*: we prove that if r is a straight and support-reflecting representation of $\mathcal{A}$'s set of states, then, by Lemma 3, the following transition relation satisfies the condition of Lemma 9:

$$F \subseteq (\bar{Q} \times \mathbb{A}) \times \bar{Q} \qquad F = \{((\bar{p},\, a),\ \bar{q}) \mid f(r(\bar{p}),\ a) = r(\bar{q})\}$$

It follows that we can use (Lemma 9) and obtain an equivariant transition function, for the straight automaton.

An interesting consequence of Lemma 10 is the following theorem about least supports. It was first proved as [GP02, Proposition 3.4], but the proof presented below follows the lines of [Boj19, Section 6].

Definition 5. We say that $\alpha \subseteq_{\text{fin}} \mathbb{A}$ is the *least support* of x if:

1. α supports x; and
2. for every other β, that supports x, it holds that $\alpha \subseteq \beta$.

It is not hard to see that if a least support exists, then it is unique. If it exists, we denote it as $\text{supp}(x)$. ◁

Theorem 2. *Every element of a set with atoms has a least support.*

Proof. Choose a set with atoms X and its element $x \in X$. Let $X_x \subseteq X$ be the orbit of x in X i.e.:

$$X_x = \{\pi(x) \mid \pi \text{ is an atom permutation}\}$$

Let $h : \mathbb{A}^{(n)} \to \bar{X}_x$ be a straight and support-reflecting representation function of X_x (as defined in Lemma 10). By combining the fact h reflects supports, with Lemma 3, it is not hard to see that the least support of $h(\bar{x})$ is $\sigma(\bar{x})$. This finishes the proof, because h is surjective. □

1.3.3 Eliminating redundant atoms and name abstraction

In this section, we show an alternative approach of finishing the translation *Orbit-Finite Automata ⇒ Straight automata.* It relies on the following lemma, which shows that we can eliminate redundant atoms from the automaton's support:

Lemma 11. *If a language L is equivariant and recognized by a* finitely supported *orbit-finite automaton*[6]*, then it is also recognized by an* equivariant *orbit-finite automaton. Moreover, if L is recognized by a* finitely supported *straight automaton, then it is also recognized by an* equivariant *straight automaton.*

To illustrate the intuition behind the proof of the lemma, we start by proving it for register automata:

1.3.3.1 Register automata and atom placeholders

A finitely supported register automaton is a register automaton, whose transition function is *syntactically finitely supported*, i.e. it can be defined using the syntax for an equivariant transition function extended with the following two types queries and one type of action:

$$\mathtt{r}_1 = a, \quad \mathtt{input} = a, \quad \mathtt{r}_1 := a$$

It is not hard to see that a transition function is syntactically finitely supported, if and only if it is a finitely supported function. The proof is almost the same as the one for syntactically equivariant functions from Section 1.1, but this time it uses α-orbits, defined as:

$$\{\pi(x) \mid \pi \text{ is an } \alpha\text{-permutation of atoms}\}$$

Now, let us take L recognized by a finitely supported register automaton $\mathcal{A}$, and let us construct an equivariant $\mathcal{A}'$ that recognizes the same language. Notice that the group of atom permutations has a natural action on the set of register automata[7], which means that we can talk about $\pi(\mathcal{A})$. It is not hard

[6] i.e. by an orbit-finite automaton whose transition function, initial state, and subset of accepting states does not have to be equivariant (but has to be finitely supported)

[7] The action applies π to its set of states, initial state, transition function, and set of accepting states. The only non-equivariant of those components is the transition function, so this boils down to applying π to the transition function.

to see that the function that maps an automaton to its language is equivariant, i.e. $\pi(\mathcal{A})$ recognizes $\pi(L)$. Since L is equivariant, it follows that for every π:

$$\pi(\mathcal{A}) \text{ recognises } L$$

Every word $w \in \mathbb{A}^*$ is finitely supported, so we can always find a permutation π_w, such $\pi_w(\mathcal{A})$ does not use any letter from w as a constant. The idea of the proof is to construct one automaton $\mathcal{A}'$, whose run on every w simulates the run of $\pi_w(\mathcal{A})$. Choose a finite, atomless set P of *atom placeholders*, such that $|P| = |\mathrm{supp}(\mathcal{A})|$. This means that there exists a bijection $p : \mathrm{supp}(\mathcal{A}) \to_{\mathrm{fs}} P$ (the bijection is supported by $\mathrm{supp}(\alpha)$). Let us now consider a variant of a register automaton, which can use its register to store both atoms or placeholders. Now, by replacing queries and actions of $\mathcal{A}$ in the following way

$$\begin{array}{ccc} \mathtt{r}_1 = a & \rightsquigarrow & \mathtt{r}_1 = p_a \\ \mathtt{input} = a & \rightsquigarrow & \mathtt{input} = p_a \\ \mathtt{r}_1 := a & \rightsquigarrow & \mathtt{r}_1 := p_a \end{array}$$

we obtain an equivariant register automaton with placeholders $\mathcal{A}'$. Note that the query that compares an atom with a placeholder always returns “No”. Since no atom from α appears in w, it follows that the run of $\mathcal{A}'$ on w simulates the run $\pi_w(\mathcal{A})$ on w. To finish the proof, notice that a placeholder automaton can easily be simulated by a register automaton, that uses its control state to keep track of the placeholder values.

1.3.3.2 Name abstraction

Before we prove Lemma 11 in its generality, let us show how to introduce placeholders into abstract sets with atoms. This operation is described in [Pit13, Section 4] under the name of *name abstraction*. The placeholders are introduced using operations of the following kind: “Replace all the occurrences of an atom $a \in \mathbb{A}$ in x by a placeholder” (where x is an element of a set with atoms X). We denote this operation as $\langle a \rangle x$. Note that it is not injective: For example, if we consider $X = \mathbb{A}^{(2)}$, then

$$\langle 5 \rangle (5, 2) = \langle 4 \rangle (4, 2)$$

because in both cases the first element is replaced with the placeholder. Before we define $\langle a \rangle x$ formally, let us discuss the inverse operation: “Replace the placeholder in $\langle a \rangle x$ with $b \in \mathbb{A}$.” This operation is denoted as $(\langle a \rangle x)@b$. If $\sigma_{(a\ b)}$ denotes the atom permutation that swaps a and b, then @ is defined in the following way:

$$(\langle a \rangle x)@b = \begin{cases} x & \text{if } a = b \\ \sigma_{(a\ b)}\, x & \text{if } b \notin \mathrm{supp}(x) \\ \text{undefined} & \text{otherwise} \end{cases}$$

The fact that $(\langle a\rangle x)@b$ is undefined when $b \neq a$ and $a \notin \mathrm{supp}(x)$ represents the intuition that the placeholder is different from all the atoms in $\langle a\rangle x$. For example, if we consider $X = \mathbb{A}^{(2)}$, then:

$$\langle 6\rangle(6,4)@3 = (3,4) \quad \text{but} \quad \langle 6\rangle(6,4)@4 \text{ is undefined}$$

Observe that if we had simply replaced 6 with a 4 in $(6,4)$, we would have obtained $(4,4)$ which does not belong to $\mathbb{A}^{(2)}$.

Intuitively, in order to check if $\langle a\rangle x$ is equal to $\langle b\rangle y$, we take a *fresh* $c \in \mathbb{A}$ – i.e. such c that $c \notin (\{a,b\} \cup \mathrm{supp}(x) \cup \mathrm{supp}(y))$ – and check if $\langle a\rangle x@c = \langle b\rangle y@c$. Notice that this does not depend on the choice of c – if this equality holds for some fresh c, then it holds for all fresh c. We can define this as the following relation:

$$\langle a\rangle x \sim \langle b\rangle y \iff \langle a\rangle x@c = \langle b\rangle y@c$$

As shown in [Pit13, Lemma 4.1], this is an equivalence relation. We use it to define the set of all possible $\langle a\rangle x$'s, denoted as $[\mathbb{A}]X$:

$$[\mathbb{A}]X = (\mathbb{A} \times X)_{/\sim},$$

We define $\langle a\rangle x$ to be the equivalence class of (a,x). For example:

$$[\mathbb{A}]\mathbb{A}^2 \simeq \underbrace{\mathbb{A}^2}_{\substack{\text{none of the atoms}\\ \text{is the placeholder}\\ \text{e.g.}\langle 5\rangle(1,2)}} + \underbrace{\mathbb{A}}_{\substack{\text{the first atom}\\ \text{is the placeholder}\\ \text{e.g.}\langle 5\rangle(5,2)}} + \underbrace{\mathbb{A}}_{\substack{\text{the second atom}\\ \text{is the placeholder}\\ \text{e.g.}\langle 5\rangle(1,5)}} + \underbrace{1}_{\substack{\text{both of the atoms}\\ \text{are placeholders}\\ \text{e.g.}\langle 5\rangle(5,5)}}$$

Notice that $[\mathbb{A}]X$ is a set with atoms, with the following action of the group permutation:

$$\pi(\langle a\rangle x) = \langle \pi a\rangle(\pi x)$$

Operation $[\mathbb{A}](\cdot)$ commutes with many classical combinators:

Lemma 12. *For all sets with atoms X and Y:*

$$[\mathbb{A}](X \times Y) \simeq [\mathbb{A}]X \times [\mathbb{A}]Y \quad [\mathbb{A}](X + Y) \simeq [\mathbb{A}]X + [\mathbb{A}]Y \quad [\mathbb{A}](X^*) \simeq ([\mathbb{A}]X)^*$$

Proof. The isomorphism for products and coproducts is proved in [Pit13, Equations 4.27, 4.28]. For the words, we use the following isomorphism:

$$W : [\mathbb{A}](X^*) \to ([\mathbb{A}]X)^*$$

$$W(w) = (\langle a\rangle(w@a)_1, \langle a\rangle(w@a)_2, \ldots, \langle a\rangle(w@a)_n) \quad \substack{\text{where } a \text{ is any atom}\\ \text{that does not appear in supp}(w)}$$

Note that $W(w)$ does not depend on the choice of a. □

Let us cite two important properties of $[\mathbb{A}]$:

Lemma 13 ([Pit13, Proposition 4.5])**.** *For every $x \in X$ and every atom a:*

$$supp(\langle a\rangle x) = supp(x) - \{a\}$$

Lemma 14 ([Pit13, Proposition 4.14])**.** *The following sets are isomorphic:*

$$[\mathbb{A}](X \to_{fs} Y) \simeq [\mathbb{A}]X \to_{fs} [\mathbb{A}]Y$$

The isomorphism is given by the following formula:

$$(\langle a\rangle f)x = \langle b\rangle \left((\langle a\rangle f)@b\right)(x@b),$$

where $b \in \mathbb{A}$ is some atom that does not appear in x or in $supp(\langle a\rangle f)$. (As usual, the result does not depend on the choice of c.)

Now let us show that $\langle c\rangle(f)$ preserves identities and function compositions[8].

Lemma 15. *For every $\langle a\rangle id_X = id_{[\mathbb{A}]X}$ and $\langle a\rangle(f \circ g) = (\langle a\rangle f) \circ (\langle a\rangle g)$.*

Proof. We start with two claims that follow immediately from the definitions:

Claim 9. *If b is fresh for $\langle a\rangle x$, then $\langle a\rangle x = \langle b\rangle(\sigma_{(a\ b)}x)$.*

Claim 10. *If a is fresh for x, then $(\langle a\rangle f)x = \langle a\rangle(f(x@a))$.*

Let us take some $x \in X$ and an atom b that is fresh for x, and notice that:

$$(\langle a\rangle id_X)\, x \overset{\text{Claim 9}}{=} (\langle b\rangle id_X)\, x \overset{\text{Claim 10}}{=} \langle b\rangle(x@b) = x$$

For the second part, we need to show that for all a, f, g, x:

$$(\langle a\rangle f)\ ((\langle a\rangle g)\, x) = (\langle a\rangle(f \circ g))\, x$$

Take some a, f, g, x, and let b be a fresh atom. Define $f' := \sigma_{(ab)}f$, $g' := \sigma_{(ab)}g$, and observe that:

$$(\langle a\rangle f)\ ((\langle a\rangle g)\ x) \overset{\text{Claim 9}}{=} (\langle b\rangle f')\ ((\langle b\rangle g')\ x) \overset{\text{Claim 10}}{=} (\langle b\rangle f')\ (\langle b\rangle(g'\, x@b)) \overset{\text{Claim 10}}{=}$$

$$= \langle b\rangle\ (f'(\langle b\rangle(g'\, x@b)@b)) = \langle b\rangle(f'\ (g'\, x@b)) =$$

$$= \langle b\rangle((f' \circ g')\ x@b) \overset{\text{Claim 10}}{=} (\langle b\rangle(f' \circ g'))\ x \overset{\text{Claim 9}}{=} \langle a\rangle(f \circ g)\ x$$

□

Finally, we show one more property of $\langle c\rangle(\cdot) : X \to_{\text{fs}} [\mathbb{A}]X$:

Lemma 16. *For all $a \in \mathbb{A}$, $f : X \to Y$ and $x \in X$ it holds that:*

$$\langle a\rangle(fx) = (\langle a\rangle f)(\langle a\rangle x)$$

This means that the following diagram commutes[9]

[8] This means that the following mapping is a functor: $X \mapsto [\mathbb{A}]X$, $f \mapsto \langle c\rangle f$. To avoid potential confusion, it is worth pointing out that this functor very similar to the $[\mathbb{A}]$ functor defined in [Pit13, Section 4.4], but not exactly the same. For this reason the proof of the lemma is very similar to the proof of [Pit13, Lemma 4.10]

[9] This means that $\langle c\rangle(\cdot) : X \to_{\text{fs}} [\mathbb{A}]X$ is a natural transformation between the identity functor and the functor $\langle c\rangle(\cdot)$ from the previous footnote. The same is true for the functor $[\mathbb{A}]$ (see [Pit13, Equation 4.19] and the previous footnote).

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "\langle c\rangle(\cdot)"'] & Y \arrow[d, "\langle c\rangle(\cdot)"'] \\
{[\mathbb{A}]X} \arrow[r, "\langle c\rangle f"] & {[\mathbb{A}]Y}
\end{tikzcd}$$

Proof. The lemma follows from Claim 10:

$$(\langle a\rangle f)(\langle a\rangle x) \overset{\text{Claim 10}}{=} \langle a\rangle(f\ \langle a\rangle x@a) = \langle a\rangle(fx)$$

□

1.3.3.3 Redundant atoms in orbit-finite automata

We are now ready to prove the general version of Lemma 11. Take an orbit-finite automaton $\mathcal{A}$ that recognizes an equivariant language. We pick some $a \in \text{supp}(\mathcal{A})$ and we construct an automaton $\langle a\rangle\mathcal{A}$ such that:

1. $\langle a\rangle\mathcal{A}$ recognizes the same language as $\mathcal{A}$;
2. $\text{supp}(\langle a\rangle\mathcal{A}) = \text{supp}(\mathcal{A}) - \{a\}$; and
3. if $\mathcal{A}$ is straight, then so is $\langle a\rangle\mathcal{A}$.

This is enough to prove the lemma, because if we repeat this construction $|\text{supp}(\mathcal{A})|$ times, we obtain an equivariant automaton, recognizing the same language as $\mathcal{A}$. Thanks to the third assumption, this construction preserves straight automata.
If $\mathcal{A} = (Q, \Sigma, q_0, Q_{acc}, f)$, then we define $\langle a\rangle\mathcal{A}'$ as:

$$([\mathbb{A}]Q, \quad [\mathbb{A}]\Sigma, \quad \langle a\rangle(q_0), \quad \{\langle a\rangle q \mid a \in Q_{acc}\}, \quad \langle a\rangle(\delta))$$

This definition results in a slight mismatch of alphabets – the alphabet of $\langle a\rangle\mathcal{A}$ is $[\mathbb{A}]\Sigma$, but we want it to recognize languages over Σ. The following lemma shows a natural way to inject Σ into $[\mathbb{A}]\Sigma$:

Claim 11. *For every X, the following ι_X function is an injection $X \hookrightarrow \mathbb{A}[X]$:*

$$\iota_X(x) = \langle a\rangle x \quad \textit{for any } a \notin \textit{supp}(x)$$

Proof. It is not hard to see that the definition does not depend on the choice of a. To show that that ι_X is an injection, let take some $x, y \in X$, such that $\iota_X(x) = \iota_X(y)$, and show that $x = y$. If we pick a that is fresh for x and y, then:

$$x = (\langle a\rangle x)@a = \iota_X(x)@a = \iota_X(y)@a = (\langle a\rangle y)@a = y$$

□

Formally, this means that, when $\langle a\rangle\mathcal{A}$ is used to recognize languages over the alphabet Σ, it has the following transition function:

$$\langle a\rangle\delta(\iota_\Sigma(\cdot),\cdot) : (\Sigma \times [\mathbb{A}]Q) \ \to \ [\mathbb{A}]Q \quad \text{defined as} \quad (a,q) \mapsto (\langle a\rangle\delta)(\iota_\Sigma\,(a),q)$$

Thanks to Lemma 13, we know that $\mathrm{supp}(\langle a\rangle\mathcal{A}) = \mathrm{supp}(\mathcal{A}) - \{a\}$, and it is also not hard to show that if Q is straight then so is $[\mathbb{A}]Q$. This leaves us with showing that $\langle a\rangle\mathcal{A}$ recognizes the same language as $\mathcal{A}$. This proof is similar to the proof for register automata. Take some $w \in \Sigma^*$ and an atom b that is fresh for w and $\mathcal{A}$, and define $\mathcal{A}'_w := \sigma_{(a\ b)}(\mathcal{A})$. Since the language of $\mathcal{A}$ is equivariant, we know that $\mathcal{A}$ and $\mathcal{A}'_w$ recognize the same language. Moreover, since b is fresh for $\mathcal{A}$, it follows from Claim 9 that $\langle a\rangle\mathcal{A} = \langle b\rangle\mathcal{A}'_w$. This means that it is enough to show that $\langle b\rangle\mathcal{A}'_w$ accepts w if and only if $\mathcal{A}$ accepts w. The key observation is that since b is fresh for w, then for every i, it holds that $\iota_\Sigma(w_i) = \langle b\rangle(w_i)$. It follows that:

$$(\langle b\rangle\delta)(\iota_\Sigma(w_i),\cdot) = (\langle b\rangle\delta)(\langle b\rangle w_i,\cdot)$$

Notice that b is fresh for every state that appears in the run of $\langle b\rangle\mathcal{A}'_w$ on w – this follows from Lemma 3, because b does not appear in the initial state, in the transition function, or in any w_i. It follows that we can use Claim 10 to further simplify the transition function:

$$(\langle b\rangle\delta)(\langle b\rangle w_i,q) = \langle b\rangle(\delta(w_i,q@b))$$

By the formula from Lemma 14, it follows that the transition function for w_i is equal to:

$$\langle b\rangle(\delta(w_i,\cdot))$$

If we treat, the initial state as a function $1 \to_{\mathrm{fs}} Q$, and the accepting set as a function $Q \to_{\mathrm{fs}} \{\mathrm{Yes},\mathrm{No}\}$, then this leaves us with showing that the following diagram commutes:

$$\begin{tikzcd}
 & Q \arrow[r, "\delta(w_1{,}\cdot)"] & Q \arrow[r, "\delta(w_2{,}\cdot)"] & Q & \cdots & Q \arrow[r, "\delta(w_n{,}\cdot)"] & Q \arrow[dr, "Q_{acc}"] & \\
1 \arrow[ur, "q_0"] \arrow[dr, "\langle b\rangle q_0"'] & & & & & & & \{\mathrm{Yes},\mathrm{No}\} \\
 & {[\mathbb{A}]Q} \arrow[r, "\langle b\rangle\delta(w_1{,}\cdot)"] & {[\mathbb{A}]Q} \arrow[r, "\langle b\rangle\delta(w_2{,}\cdot)"] & {[\mathbb{A}]Q} & \cdots & {[\mathbb{A}]Q} \arrow[r, "\langle b\rangle\delta(w_n{,}\cdot)"] & {[\mathbb{A}]Q} \arrow[ur, "\langle b\rangle Q_{acc}"'] &
\end{tikzcd}$$

To prove that, we draw some auxiliary arrows so that each face of the diagram becomes an instance of Lemma 16:

$$\begin{tikzcd}
 & Q \arrow[r, "\delta(w_1,\cdot)"] \arrow[dd, "\langle b\rangle(\cdot)"] & Q \arrow[r, "\delta(w_2,\cdot)"] \arrow[dd, "\langle b\rangle(\cdot)"] & Q \arrow[dd, "\langle b\rangle(\cdot)"] & \cdots & Q \arrow[r, "\delta(w_n,\cdot)"] \arrow[dd, "\langle b\rangle(\cdot)"] & Q \arrow[dr, "Q_{acc}"] \arrow[dd, "\langle b\rangle(\cdot)"] & \\
1 \arrow[ur, "q_0"] \arrow[dr, "\langle b\rangle q_0"'] & & & & & & & \{Yes, No\} \\
 & {[A]Q} \arrow[r, "\langle b\rangle\delta(w_1,\cdot)"] & {[A]Q} \arrow[r, "\langle b\rangle\delta(w_2,\cdot)"] & {[A]Q} & \cdots & {[A]Q} \arrow[r, "\langle b\rangle\delta(w_n,\cdot)"] & {[A]Q} \arrow[ur, "\langle b\rangle Q_{acc}"'] &
\end{tikzcd}$$

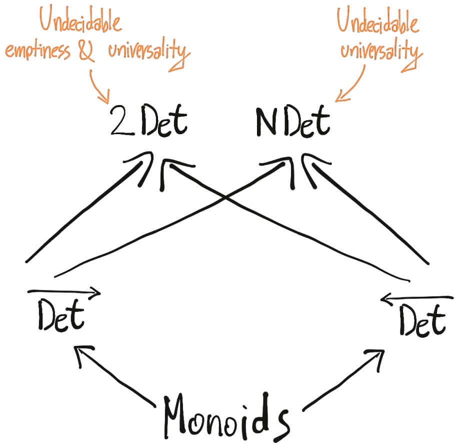
This finishes the proof of Lemma 11, which in turn finishes the (second) proof of Theorem 1.

1.4 Other models for infinite alphabets

The theory of infinite alphabets is notorious for the abundance of non-equivalent models that it features – the expressive powers of different models are well illustrated by [Boj19, Figure 1.1] or [NSV04, Figure 1]. In this section we present the following part of this landscape:

1. Right-to-left deterministic orbit-finite automata;
2. Two-way deterministic orbit-finite automata;
3. One-way nondeterministic orbit-finite automata;
4. Orbit-finite monoids

The following picture summarizes the results presented in this section:



1.4.1 Variants of orbit-finite automata

We begin the discussion with variants of orbit-finite automata. This section is based on [Boj19, Section 1.4]:

1.4.1.1 Right-to-left orbit finite automata

Notice that the class of languages recognized by orbit-finite automata is not closed under the reverse:

Lemma 17. *There is a language L, such that it is recognized by a deterministic orbit-finite automaton, but its reverse is not.*

Proof. Consider the following language over the alphabet $\mathbb{A}^*$:

$$L_{\text{First}} = \text{“The first letter appears again”}$$

As shown by Example 1, it can be recognized by an orbit-finite register automaton. Consider now its reverse:

$$L_{\text{Last}} = \text{“The last letter appears before”}$$

We show that this language cannot be recognized by a deterministic orbit-finite automaton. First, let us notice that every orbit-finite set has a limit on the size of its supports:

Claim 12. *For every orbit-finite set A there is a number k such that for every $a \in A$, it holds that $|supp(x)| \leq k$.*

Proof. The following function is equivariant:

$$a \mapsto |\text{supp}(a)|$$

It follows that the size of least supports is fixed in every orbit. There are only finitely many orbits in A, so we can set k to be the maximal size of the support among those orbits. □

Suppose that L_{Last} is recognized by a deterministic orbit-finite automaton $\mathcal{A}$ and let k be the maximal support size for the states of $\mathcal{A}$. Let q be the state of $\mathcal{A}$ after it has read a prefix consisting of $k + 1$ different atoms:

$$a_1 \quad a_2 \quad \ldots \quad a_{k+1}$$

Since $\mathcal{A}$'s states have supports of size at most k, one of the input letters must be fresh for q (i.e. $a_i \notin \text{supp}(q)$). Say that it is a_j. This leads to a contradiction because, if we choose a_{k+2} different from all a_i's, then $\mathcal{A}$ cannot distinguish between:

$$a_1\ a_2\ \ldots\ a_{k+1}\ a_j \in L_{\text{Last}} \quad \text{and} \quad a_1\ a_2\ \ldots\ a_{k+1}\ a_{k+2} \notin L_{\text{Last}}$$

□

A consequence of Lemma 17 is that deterministic (left-to-right) orbit-finite automata recognize a different class of languages than their right-to-left counterparts.

We finish the discussion on right-to-left and left-to-right deterministic orbit-finite automata, by mentioning that both of those models are computationally quite simple: They have decidable emptiness (special case of [Boj19, Theorem 1.7]) and, since they are closed under complements, decidable universally.

1.4.1.2 Nondeterministic orbit-finite automata

The definition of a *nondeterministic orbit-finite automaton* is not surprising – it looks just like a deterministic orbit-finite automaton, but (a) it may have more than one initial state and (b) instead of transition functions, it has a transition relation:

$$f \subseteq_{\text{fs}} Q \times \Sigma \times Q$$

Nondeterministic orbit-finite automata can recognize the language

“Last letter appears before”:

A nondeterministic automaton can nondeterministically guess a position, save its letter to a register, and at the end of the word verify that it is equal to the last letter. (In general, it is easy to see that the languages recognized by nondeterministic automata are closed under the reverse.)

It follows that nondeterministic orbit-finite automata are strictly more expressive than their deterministic counterparts. A reason for this mismatch in expressive powers is that orbit-finite sets are not closed under finitely supported powersets and therefore they do not admit the powerset construction.

Non-deterministic orbit-finite automata are computationally quite complex: They have decidable emptiness [Boj19, Theorem 1.7], but undecidable universality[10] [Boj19, Theorem 1.8].

1.4.1.3 Two-way deterministic orbit-finite automaton

Two-way deterministic orbit-finite automaton is variant of a deterministic orbit-finite automaton that is not forced to read the input left-to-right or right-to-left. Instead, in each transition, it decides whether it wants to go left or right. When it leaves the word, it gets notified and might choose to go back. To finish its run it has to explicitly accept or reject the input[11]. To accommodate those features, its transition function has the following type:

$$(\Sigma + \underbrace{\{\vdash, \dashv\}}_{\substack{\text{end of word}\\ \text{markers}}}) \times Q \quad \longrightarrow_{\text{eq}} \quad Q \times \{\leftarrow, \rightarrow\} + \{\text{accept}, \text{reject}\}$$

Two-way orbit-finite automata extend two-way finite automata (defined in [She59, Defniiton 2]). Over finite alphabets, two-way and one-way automata are equivalent ([She59, Theorem 2]). However, this equivalence does not hold for orbit-finite alphabets: It is not hard to see that two-way orbit-finite automata can recognize both L_{first} and L_{last}. It follows they are strictly stronger than

[10] It might be worth mentioning that universality is decidable for the *unambiguous* nondeterministic orbit-finite automata. One way to show this is to show that orbit-finite weighted automata have decidable equivalence. See [BKM21] for details.

[11] It may also loop and never finish its run. We assume that this means that the automaton rejects the input.

one-way deterministic automata.

Two-way register automata are computationally very strong. The following theorem says that they belong more to complexity theory than to automata theory.

Theorem 3 ([NSV04, Theorem 3.8 (b)])**.** *Take a language over a finite alphabet $L \subseteq \Sigma^*$ and define $L_{\mathbb{A}} \subset (\Sigma \times \mathbb{A})^*$ to be the language of those words whose Σ-part belongs to L and whose atoms are pairwise distinct. Then:*

$$L \in \textsc{LogSpace} \iff L_{\mathbb{A}} \textit{ is recognisable by a two-way orbit-finte automaton}$$

Proof. The key idea is that when the input alphabet is $\Sigma \times \mathbb{A}$, and all the atoms are distinct, then remembering the atom value is the same as remembering the atom's position.

We start with the easier right-to-left implication ($\Leftarrow$): The alphabet $\Sigma \times \mathbb{A}$ is straight, so using the techniques presented in the proof of Theorem 1, we can transform a two-way orbit-finite automaton that recognizes $L_{\mathbb{A}}$ into a two-way register automaton over the alphabet $\Sigma \times \mathbb{A}$ (this type of an automaton is a natural extension of the standard register automaton). In order to simulate this two-way register automaton with a LogSpace Turing machine, we notice that instead of storing an atom, it is enough to store its position (in binary).

The ($\Rightarrow$) implication is more involved. We start with the following claim:

Claim 13. *There is a two-way orbit-finite automaton, recognizing the language "Each letter appears only once" (over $\mathbb{A}$).*

Proof. We show that a two-way orbit-finite automaton can simulate the following program:

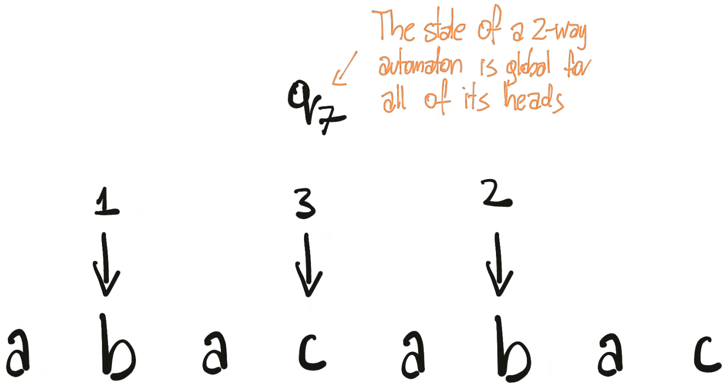
```
for i  in 1..n:
    r = atom at i-th position
    for j  in (i + 1)..n:
        if r is equal to the atom at j-th position:
            reject
accept
```

The execution is mostly straightforward. The only problem appears, when the automaton finishes the inner loop, and has to go back to position $i + 1$. At this point the automaton knows that the value r appears exactly once in the word, so it can go back to the ith position by locating the value r. □

We are left with showing that once a two-way automaton has checked that all the input atoms are pairwise distinct, it can simulate a logspace Turing machine. For this we use an intermediate model of *two-way multi-head deterministic finite automaton.* Such an automaton resembles a two-way deterministic finite automaton, but it has many heads that move independently from each other. A transition function of a two-way multi-head deterministic finite automaton has the following type:

$$Q \times \underbrace{(\Sigma + \{\vdash, \dashv\})^k}_{\text{what each head is seeing}} \longrightarrow Q \times \underbrace{\{\leftarrow, \rightarrow\}^k}_{\substack{\text{instructions}\\\text{for each of the heads}}} + \{\text{accept}, \text{reject}\}$$

Here is an example configuration of a multi-head automaton:

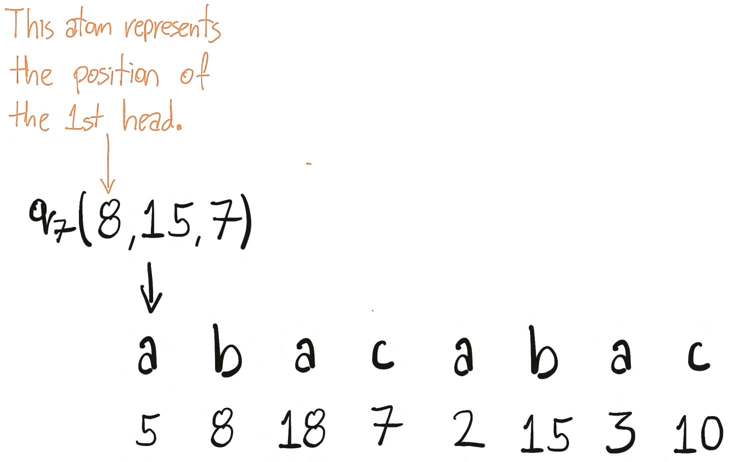

The state of a 2-way automaton is global for all of its heads
q7
1
3
2
a b a c a b a c

The multi-head automata are equivalent to logspace Turing machines in the following sense[12]:

Theorem 4. *A language $L \subseteq \Sigma^*$ belongs to* LogSpace *if and only if it is recognizable by a two-way deterministic multi-head automaton.*

So it suffices to show that, as long as its input is equipped with a unique atom on every position, a two-way orbit-finite automaton can simulate a deterministic two-way multi-head automaton. This is straight forward: Instead of remembering the position of each head, an orbit-finite automaton can simply remember the atom from that position. For example, the configuration from the example above can be represented as follows:

[12] To the best of my knowledge, the earliest reference to this result is [Iba71, Corollary 3.5]. However, in place of a proof, the author states that this is a well-known unpublished result by Alan Cobham and others. The earliest reference that contains the proof is [Har72, Page 338]. I would like to thank Nguyễn Lê Thành Dũng for pointing me to those references.



With this representation, a two-way orbit-finite automaton can simulate a transition of a multi-head automaton, by doing two sweeps: a left-to-right one to see the letter under each of the heads; and a right-to-left one to perform necessary updates in the heads' positions. □

The emptiness of LogSpace-Turing machines is easily seen to be undecidable, so a consequence of Theorem 3 is that two-way deterministic orbit-finite automata, have undecidable emptiness. As a deterministic model, they are closed under complement, which means that they also have undecidable universality.

We finish this section, by showing that one-way nondeterministic orbit-finite automata and two-way deterministic orbit-finite automata have incomparable expressive powers:

Lemma 18 ([KF94, Example 11]). *There is a language L, that can be recognized by a two-way deterministic orbit-finite automaton, but cannot be recognized by a one-way nondeterministic orbit-finite automaton.*

Proof. One example of such L is the language "Each letter appears only once". According to Claim 13 it can be recognized by a two-way deterministic orbit-finite automaton. It suffices to show that it cannot be recognized by a one-way nondeterministic orbit-finite automaton: Suppose that it is recognized by $\mathcal{A}$ whose set of states is Q. Let k be maximal support size in Q (see Claim 12). Consider the word consisting of $k+2$ pairwise distinct atoms:

$$a_1 \quad a_2 \quad \ldots \quad a_{k+1} \quad a_{k+2}$$

This word belongs to L, so $\mathcal{A}$ has to have an accepting run on it. Let q_k be $\mathcal{A}$'s state in this run just after it has processed a_k. One of the a_i (for $i \leq k+1$) is not present in the support of q_k. It follows that $\mathcal{A}$ also has to have an accepting run the following word, which does not belong to L:

$$a_1 \quad a_2 \quad \dots \quad a_{k+1} \quad a_i$$

This contradicts the assumption that $\mathcal{A}$ recognizes L. □

Lemma 19 ([Boj19, Exercise 26])**.** *There is a language L, that can be recognized by a one-way nondeterministic orbit-finite automaton, but cannot be recognized by a two-way deterministic orbit-finite automaton. (This lemma is conditional on* LogSpace $\neq$ NLogSpace*.)*

Proof. We start by defining the domino language over the alphabet $\mathbb{A}^2$:

$$L_{\text{Domino}} = \{(a_1, a_2)(a_2, a_3)(a_3, a_4) \dots (a_n, a_{n+1}) \mid a_1, \dots, a_{n+1} \in \mathbb{A} \ \}$$

Now, we define the language $L_{\text{SubDomino}}$ of words that have a valid domino subsequence that contains the first and the last letter. For example, the following word belongs to $L_{\text{SubDomino}}$ (its valid domino subsequence has been underlined):

$$\underline{(7,5)} \quad (1,2) \quad (5,9) \quad \underline{(5,4)} \quad (8,2) \quad \underline{(4,3)} \quad \underline{(3,2)}$$

It is easy to see that the language $L_{\text{SubDomino}}$ is recognized by a nondeterministic orbit-finite automaton. On the other hand, it can be shown that if $L_{\text{SubDomino}}$ is recognized by a deterministic two-way automaton, then the NLogSpace-complete problem of reachability in directed acyclic graphs belongs to LogSpace. (See [Boj19, Exercise 26] for details.) □

1.4.2 Orbit-finite monoids

We finish this introduction to languages over infinite alphabets with one more model – *orbit-finite monoids*. As noted in the introduction, it plays a central role in this thesis. This section is based on [Boj19], where the model was first introduced.

First we discuss the well-established model of *finite monoids* (for recognizing languages). A *monoid* is a set M, equipped with an associative binary operation $(\cdot)$ and a neutral element 1. This means that, for all $a, b, c \in M$ it holds that

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c \quad \text{and} \quad 1 \cdot a = a \cdot 1 = a.$$

A monoid is finite, if the set M is finite. A finite monoid M together with a function $h : \Sigma \to M$, and an accepting subset $F \subseteq M$ can be used to recognize a language over the finite alphabet Σ: to see if a word $w \in \Sigma^*$ belongs to the language, we check whether:

$$h(w_1) \cdot h(w_2) \cdot \ldots \cdot h(w_n) \in F$$

Example 4. For example, consider the following language (over a finite alphabet Σ):

"No letter appears twice in a row"

It is recognized by the following finite monoid:

$$M = \underbrace{\Sigma^2}_{\substack{(a,b) \text{ represents all non-empty words that} \\ \text{do not contain a repetition;} \\ \text{start with } a\text{; and end with } b}} + \underbrace{\perp}_{\substack{\text{represents} \\ \text{all the words that} \\ \text{contain a repetition}}} + \underbrace{1}_{\substack{\text{represents} \\ \text{the empty word}}}$$

M's operation is defined as follows:

$$\perp \cdot x = x \cdot \perp = \perp, \quad 1 \cdot x = x \cdot 1 = x, \quad (x_1, y_1) \cdot (x_2, y_2) = \begin{cases} (x_1, y_2) & \text{if } y_1 \neq x_2 \\ \perp & \text{otherwise} \end{cases}$$

The function h maps a letter x into (x, x), and the accepting subset F is $M - \{\perp\}$. ◁

It is a well-known fact, that the class of languages recognized by finite monoids is exactly equal to the class of regular languages:

Lemma 20. *A language is recognized by an orbit-finite monoid, if and only if it is recognized by a deterministic orbit-finite automaton.*

Proof. ($\Rightarrow$): Let $L \subseteq \Sigma^*$ be a language recognized by a finite monoid M (together with h and F). It is not hard to see that L is also recognized by an automaton where the set of states is M, the initial state is 1, the accepting subset of states is F, and the transition function is given as:

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

($\Leftarrow$): Let L be a language recognized by a finite automaton $\mathcal{A}$ whose set of states is equal to Q. It follows that L is recognized by the monoid $Q \to Q$, whose operation is given as $f \cdot g = g \circ f$, together with the following h and F:

$$h(a) = (q \mapsto \delta(q, a)) \qquad F = \{f \mid f \in Q \to Q, \substack{\text{such that } f(q_0) \\ \text{is an accepting state}}\}$$

The intuition behind this construction is that a word w can be characterized by its behaviour function $b_w \in Q \to Q$:

$$b_w(q) = \substack{\text{In which state will } \mathcal{A} \text{ exit } w \text{ on the right,} \\ \text{if it enters } w \text{ on the left in the state } q?}$$

(For a more detailed explanation see [Boj20, Theorem 22], or Section 2.3.4). □

Now, let us extend the theory of finite monoids to sets with atoms: An *orbit-finite monoid*, is monoid whose underlying set (i.e. M) is orbit-finite and whose operation $(\cdot)$ is an equivariant function. The language recognized by an orbit-finite monoid M is defined in the same way as for a finite M, but we require h and F to be equivariant, i.e.:

$$h : \Sigma \to_{\text{eq}} M \quad \text{and} \quad F \subseteq_{\text{eq}} M$$

For example, consider again the language from Example 4, but this time over an orbit-finite Σ:

$$\text{“No letter appears twice in a row”} \subseteq \Sigma^*$$

It is easy to see that this language is recognized by an orbit-finite monoid: In fact, the definitions of M, h and F remain the same as in the finite case. If Σ is orbit-finite then so is $1 + \Sigma^2 + \perp$, and both h and F are easily seen to be equivariant.

Let us present one more example:

Example 5. Consider the following language over the alphabet $\mathbb{A}$:

$$\text{“There are at most 3 different letters in the word ”}$$

It is recognized by the following orbit-finite monoid:

$$M = \underbrace{\binom{\mathbb{A}}{\leq 3}}_{\text{sets with at most 3 letters}} + \underbrace{\perp}_{\substack{\text{a representation of} \\ \text{sets with more than 3 atoms}}}$$

The monoid operation is defined as:

$$x \cdot y = \begin{cases} x \cup y & \text{if } x \neq \perp,\, y \neq \perp, \text{ and } |x \cup y| < 3 \\ \perp & \text{otherwise} \end{cases}$$

Function h is defined as $x \mapsto \{x\}$, and the accepting subset is defined as $F = M - \{\perp\}$. ◁

The expressive power of orbit-finite monoids is strictly weaker than the one of deterministic one-way orbit-finite automata. First, let us notice that an orbit-finite monoid can be translated into an orbit-finite automaton, using the same construction as in the proof of Lemma 20. To show that orbit-finite monoids recognize a different class of languages than deterministic orbit-finite automata, it suffices to show that orbit-finite-monoids are closed under reverse (we already know that the deterministic one-way orbit-finite automata are not):

Lemma 21. *If a language L is recognized by and orbit-finite monoid, then so is its reverse.*

Proof. Let L be recognized by M, h, F. Define $\overleftarrow{M}$, to be an orbit-finite monoid that has the same underlying set as M, but its operation $(\star)$ is defined as:

$$a \star b = b \cdot a$$

where $(\cdot)$ is the operation of the original monoid M. This operation is easily seen to be associative. This finishes the proof, because $\overleftarrow{M}$ (together with the original h and F) recognizes the reverse of L. □

It follows orbit-finite monoids are indeed strictly weaker than deterministic orbit-finite automata. The following claim gives an explicit witness of this non-containment:

Example 6. The language

$$\text{“First letter appears again”}$$

is not recognized by any orbit-finite monoid. If it were, then (by Lemma 21) so would be its reverse:

$$L_{\text{last}} = \text{“The last letter appears before”}$$

This would mean that the language L_{last} is recognized by some deterministic orbit-finite automaton, which was shown in the proof of Lemma 17 to be false. $\triangleleft$

It is worth pointing out that:

$$\text{orbit-finte monoids} \neq \binom{\text{left-to-right deterministic}}{\text{orbit-finite automata}} \cap \binom{\text{right-to-left deterministic}}{\text{orbit-finite automata}}$$

The class on the right is stronger, as witnessed by the following language:

“The first letter is equal to the last one
and it appears somewhere else in the word”

Chapter 2

Single-use restriction

In this section, we introduce the *single-use restriction*, which weakens register automata in a way that makes them equivalent to orbit-finite monoids. The restriction was first introduced in my master's thesis [Ste18], which proves that single-use register automata are not stronger than orbit-finite monoids, and later studied in the conference paper [BS20], which proves that the two models are actually equivalent. The contribution of this thesis is introducing the abstract class of *single-use functions* (Section 2.2), and defining the *single-use automaton* in terms of single-use functions (Section 2.3).

2.1 Single-use register automaton

We start the chapter with an informal discussion on the model of the *single-use register automaton*, which is a variant of the deterministic[1] register automaton, where every register value can be used at most once. This means that:

1. the automaton is not allowed to make copies of register values; and
2. whenever the automaton asks a query about a register value, this has the side effect of destroying that register's contents.

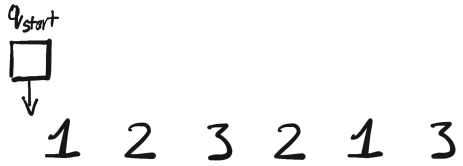
To see this model in practice, consider the following example:

Example 7. We want single-use automata to be equivalent to orbit-finite monoids. This means that the following language should not be recognized by any single-use register automaton (because, by Example 6, it is not recognized by any orbit-finite monoid):

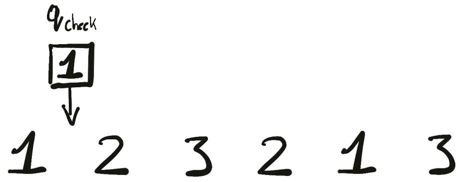
$$\text{“The first letter appears again”} \subseteq \mathbb{A}$$

[1] In this thesis we assume that all single-use models are deterministic unless we explicitly state otherwise. This is because combining the single-use restriction with nondeterminism, which we do not know how to resolve (see Section 2.4).

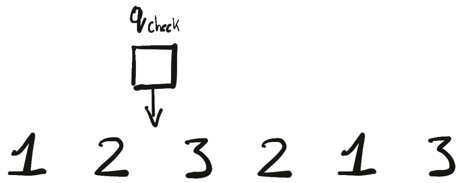
The formal proof of this fact follows from Lemma 39 presented later in this chapter. For now, let us simply show that the standard register automaton presented in Example 1 (in Chapter 1) fails to recognize the language under the single-use restriction. We follow the automaton's run on the word 1 2 3 2 1 3. It starts in the initial configuration:



The first transition proceeds without difficulties: the automaton stores the first letter in its register and moves to the second position:



Then, the automaton compares its register value with its current input value. It learns that they are different, but, as a consequence of the single-use restriction, it loses the register value:



At this point, the automaton does not remember the first atom any more, so it has no chance of checking if it appears later in the word. ◁

Let us now consider a positive example:

Example 8. The following language is recognized by a single-use automaton:

$$\text{“No letter appears twice in a row”} \subseteq \mathbb{A}^*$$

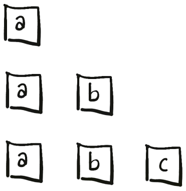
The automaton has one register in which it stores a copy of the previous letter. Whenever the automaton moves forward to a new position, it compares the new letter with the previous one. This has the side effect of destroying the register's contents. If the letters are (or rather were) equal, the automaton rejects the input. If they were different, the automaton saves the current letter in its register and proceeds forward. ◁

It is worth pointing out that a single-use automaton has an unrestricted (i.e. multiple-use) access to its input letters. This is illustrated by our final example:

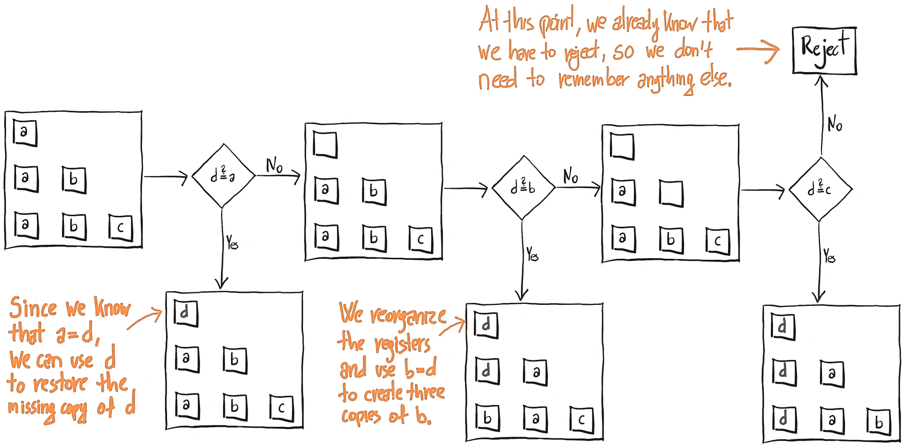
Example 9. In this example we show that the language:

$$\text{“There are at most 3 different letters in the input word”} \subseteq \mathbb{A}$$

is recognized by a single-use register automaton. First, let us point out that the standard (i.e. multiple-use) automaton presented in Section 1.1 violates the single-use restriction. Interestingly, a single-use construction is possible, but it requires six registers. Suppose that the automaton has already seen three different letters $a, b, c \in \mathbb{A}$. Then the automaton should store one of them in three copies, one of them in two copies and one of them in one copy:



Suppose that the automaton is about to process the next letter $d \in \mathbb{A}$, which may or may not be equal to a, b, or c. This transition is explained in the following (hopefully self-explanatory) diagram:



At this point, we already know that we have to reject, so we don't need to remember anything else.

Since we know that a=d, we can use d to restore the missing copy of d

We reorganize the registers and use b=d to create three copies of b.

Using a similar idea, it is easy to extend this construction to cases where the automaton has only seen two or fewer different letters so far. ◁

2.1.1 Single-use transition functions

In this section, we define *single-use transition functions* for register automata. This definition is specific to the functions of the type:

$$(Q \times (\mathbb{A} + \bot)^R) \times \mathbb{A} \to_{\text{eq}} (Q \times (\mathbb{A} + \bot)^R).$$

The syntactic definition of single-use transition functions is based on the syntactic definition of equivariant transition functions. Recall that in Section 1.1 we have defined equivariant transition functions using programs of the following shape:

```
if (condition and condition and ... and condition) then
    action; action; ...; action;
else if (condition and condition and ... and condition) then
    action; action; ...; action;
else if (condition and condition and ... and condition) then
    action; action; ...; action;
...
```

Examples of conditions include `state` $=$ q_7, $\mathtt{r}_1$ $=$ $\mathtt{r}_2$, or $\mathtt{r}_1$ $=$ `input`, and examples of actions include `state` $:=$ q_7, $\mathtt{r}_3$ $:=$ $\mathtt{r}_2$ or $\mathtt{r}_5$ $:=$ `input`. Notice that the if-statements are not allowed to branch. In the definition of *single-use transition functions,* we use a similar syntax but with a different semantics. In the single-use semantics, evaluating a condition has the side effect of destroying the content of each register that appears in the condition (by replacing it with $\bot$). For example, in the single-use semantics the following two programs are equivalent:

```
if $r_1$ = $r_3$ then
    state := $q_3$;
else
    state := $q_5$;
```

```
if $r_1$ = $r_3$ then
    $r_1$ := $\bot$;
    $r_3$ := $\bot$;
    state := $q_3$;
else
    $r_1$ := $\bot$;
    $r_3$ := $\bot$;
    state := $q_5$;
```

Because of those side effects, the order in which we evaluate the conditions might influence the outcome. To make this order clear, we modify the syntax of single-use transition functions, by (a) disallowing the use of **and** in the if-statements, and (b) allowing nested and branching if expressions.

As an example, we provide an implementation of the transition function for the register automaton described in Example 8. The automaton has 4 control states (q_0, q_1, q_2, q_3, q_{fail}) and 6 registers ($a_1, a_2, a_3, b_1, b_2, c_1$). For the sake of brevity, we limit the implementation to the case where the automaton has already seen 3 different letters:

```
if state = $q_3$ then
    if $a_1$ = input then
        $a_1$ := input;
    else if $b_1$ = input then
        $b_1$ := $a_2$;
        $b_2$ := $a_3$;
        $a_1$ := input;
        $a_2$ := input;
        $a_3$ := input;
    else if $c_1$ = input then
        $c_1$ := $b_2$;
        $b_1$ := $a_2$;
        $b_2$ := $a_3$;
        $a_1$ := input;
        $a_2$ := input;
        $a_3$ := input;
    else
        state := $q_{\text{fail}}$
else ...
```

This particular transition function could have been implemented using **and**'s instead of the branching if expressions, but this is not true for all single-use transition functions. Here is an example of a function that requires branching:

```
if $r_1$ = input then
    if $r_2 = r_3$
    then state := $q_{\text{ok}}$;
    else state := $q_{\text{fail}}$;
else
    if $r_2 = r_4$
    then state := $q_{\text{ok}}$;
    else state := $q_{\text{fail}}$;
```

2.1.1.1 Single-use acceptance function

Finally, let us discuss the way in which the single-use register automaton accepts its input. There are two possible approaches: In the first one, called *acceptance by state*, the automaton decides whether to accept its input by looking at its final control state – if it belongs to the accepting subset $F \subseteq Q$, it accepts; otherwise it rejects. In the second way, the automaton has an equivariant acceptance function:

$$(Q \times (\mathbb{A} + \bot)^R) \to_{\text{eq}} \{\text{Yes}, \text{No}\}.$$

Acceptance by control state is used by the standard (multiple-use) register automata, as defined in Chapter 1, and acceptance by configuration is used (implicitly) by the orbit-finite automaton. It follows, from Theorem 1, that the two acceptance models are equivalent for multiple-use automata. However, for the single-use register automaton, they can change the expressive power of the automaton[2]:

Example 10. The following language

$$\text{“The first letter is equal to the last one”} \subseteq \mathbb{A}^*$$

is recognized by a single-use register automaton that accepts by configuration, but not by one that accepts by control state.

We start by describing the automaton that accepts by configuration. The automaton has two registers: in the first one, it stores a copy of the first letter, and in the second one, it stores a copy of the previous letter. In the final configuration, the first register will contain a copy of the first letter, and the second register will contain a copy of the last letter, so the acceptance function can check if the two values are equal.

On the other hand, it is not hard to see that a register automaton that accepts by control state and recognizes the language “The first letter is equal to the last one”, has to compare each input letter with the first one (because every letter could be the last one). This would violate the single-use restriction. ◃

The language from Example 10 is recognized by an orbit-finite monoid. Since we would like single-use register automata to be equivalent to orbit-finite monoids, we choose acceptance by configuration as the standard acceptance model for the single-use register automaton. One could also argue that acceptance by configuration is more natural than acceptance by control state, because it naturally appears in orbit-finite automata.

Finally, let us mention that we could also consider a model, which requires the acceptance function to be single-use. (With the syntax for a single-use acceptance functions defined analogously to the one for single-use transition functions.) Fortunately, it turns out that this does not influence the expressive power of the automaton. We prove this later in the chapter as Lemma 38.

2.2 Single-use functions

In this section, we take a detour from our discussion of automata theory to introduce an abstract concept of *single-use functions*. Then, in the next section, we will use these functions as a tool for analysing single-use automata. The step from single-use transition functions to general single-use functions can be compared to the step from equivariant transition functions (described in Section 1.1) to the general equivariant functions (described in Section 1.2).

[2] We are going to briefly revisit this distinction in Chapter 3, while discussing the output function (λ) of a local monoid transduction (see Definition 19).

Before we define single-use functions, let us discuss a few of their properties. The defining feature of the class is that it does not contain the following function:

$$\texttt{copy} : \mathbb{A} \to \mathbb{A} \times \mathbb{A}.$$

This motivates the notation $X \multimap Y$ for the set of all single-use functions between X and Y. The notation comes from linear logic ([Gir87]) and linear type systems[3] ([Wad90]). It is important to point out that while single-use functions are not allowed to copy atoms, they are allowed to discard them, which makes them actually closer to affine logic, and affine type systems [Asp98] (they use the symbol $\multimap$ as well).

The proof that single-use automata are not stronger than orbit-finite monoids relies on two key properties of single-use functions: The first one is that single-use function spaces preserve orbit finiteness: if X and Y are orbit-finite, then so is $X \multimap Y$. Note that this is not true for finitely supported functions, where already $\mathbb{A} \to_{\mathrm{fs}} \{\mathrm{Yes}, \mathrm{No}\}$ is orbit-infinite (see Section 1.2.2 for details). The second important property is that single-use functions are closed under compositions: i.e. if f belongs $X \multimap Y$, and g belongs to $Y \multimap Z$, then $(g \circ f)$ belongs to $X \multimap Z$. Thanks to those two properties, we know that if Q is orbit-finite, then $Q \multimap Q$ is an orbit-finite monoid. We can use it to recognize the language of a single-use automaton whose set of states is Q.

Unfortunately, as we are going to see, the definition of single-use functions is (at least for now) rather syntactic in its nature. It limits their scope to a very specific subclass of orbit-finite sets called *polynomial orbit-finite sets*. The questions of finding a semantic definition of the class and extending its scope remain open.

2.2.1 Polynomial orbit-finite sets

We start by defining *polynomial orbit-finite sets*[4], which are the domains and codomains of single-use functions:

Definition 6. The class of *polynomial orbit-finite sets* is the smallest subclass of sets with atoms that:

1. contains the atomless singleton (1);
2. contains the set of all atoms ($\mathbb{A}$);
3. is closed under products ($P_1 \times P_2$); and
4. is closed under disjoint sums ($P_1 + P_2$).

[3] For more connections with linear type systems see [Nê21, Claim 1.4.11].

[4] The motivation of the name is as follows: The word *orbit-finite* is used, because every polynomial orbit-finite set is orbit-finite. The word *polynomial* is used because the class of polynomial orbit-finite sets is closed under × and + (i.e. products and coproducts).

◁

Example 11. Every finite S can be represented as polynomial orbit-finite, because it is isomorphic to:

$$\underbrace{1+1+\ldots+1}_{|S| \text{ times}}.$$

◁

Example 12. The set of all possible memory configurations of a register automaton – i.e. the set $Q \times (\mathbb{A}+\perp)^R$, for some finite R and Q – can be represented as a polynomial orbit finite set, because it is isomorphic to:

$$(\underbrace{1+1+\ldots+1}_{|Q| \text{ times}}) \times \underbrace{(\mathbb{A}+1) \times (\mathbb{A}+1) \times \ldots \times (\mathbb{A}+1)}_{|R| \text{ times}}$$

◁

It is easy to see that all polynomial orbit-finite sets are orbit-finite, and that all polynomial orbit-finite sets are straight (as defined in Definition 4). It is worth pointing out that the other inclusion does not hold – there are sets that are straight and orbit-finite but not polynomial orbit finite. An example of such a set is $\mathbb{A}^{(2)}$. In general, thanks to the distributivity of × over +, it is not hard to see that every polynomial orbit-finite set is isomorphic with a set of the following form (in Lemma 22 we are going to show that this isomorphism is a single-use function):

$$\mathbb{A}^{k_1} + \ldots + \mathbb{A}^{k_n}$$

2.2.2 Single-use functions

We are now ready to define the single-use functions:

Definition 7. The class of *single-use functions* is the smallest subclass of functions between polynomial orbit-finite that is closed under the following combinators, and contains all the following basic functions:

Combinators		
$\dfrac{X \xrightarrow{f} Y \quad Y \xrightarrow{g} Z}{X \xrightarrow{g \circ f} Z}$	$\dfrac{X_1 \xrightarrow{f} Y_1 \quad X_1 \xrightarrow{g} Y_1}{X_1 \times X_2 \xrightarrow{f \times g} Y_1 \times Y_2}$	$\dfrac{X_1 \xrightarrow{f} Y_1 \quad X_2 \xrightarrow{g} Y_2}{X_1 + X_2 \xrightarrow{f+g} Y_1 + Y_2}$

Functions about $\mathbb{A}$	
$\texttt{eq}$:	$\mathbb{A} \times \mathbb{A} \to 1 + 1$
$\texttt{const}_{a \in \mathbb{A}}$:	$1 \to \mathbb{A}$
$\texttt{id}$:	$\mathbb{A} \to \mathbb{A}$

Functions about $\times$	
$\texttt{proj}_1$:	$X \times Y \to X$
$\texttt{proj}_2$:	$X \times Y \to Y$
$\texttt{sym}$:	$X \times Y \to Y \times X$
$\texttt{assoc}$:	$(X \times Y) \times Z \to X \times (Y \times Z)$
$\texttt{leftI}$:	$X \to 1 \times X$

Functions about $+$	
$\texttt{coproj}_1$	$X \to X + Y$
$\texttt{coproj}_2$	$Y \to X + Y$
$\texttt{cosym}$:	$X + Y \to Y + X$
$\texttt{coassoc}$:	$(X + Y) + Z \to X + (Y + Z)$
$\texttt{merge}$:	$X + X \to X$

Distributivity	
$\texttt{distr}$:	$X \times (Y + Z) \to X \times Y + X \times Z$

◁

The class of single-use functions is a restriction of finitely supported functions. To illustrate that, we start with a negative example:

Example 13. The following function is finitely supported, but not single-use:

$$f : \mathbb{A} \to \underbrace{(1+1)}_{\substack{\text{represents}\\ \text{true or false}}} \qquad f(a) = \begin{cases} \text{true} & \text{if } a = 3 \vee a = 5 \\ \text{false} & \text{otherwise} \end{cases}$$

This is because f needs to compare its input with two different constants, which requires two copies of the input value. (A formal proof follows from the decision tree representation, presented later in this section.) ◁

Let us now present a few positive examples:

Example 14. The function $\texttt{constI} : X \to 1$ is a single-use function, because it can be constructed as the following composition:

$$X \xrightarrow{\texttt{leftI}} 1 \times X \xrightarrow{\texttt{proj}_1} 1$$

◁

Example 15. The function $\texttt{rightDistr} : (X + Y) \times Z \to (X \times Z) + (Y \times Z)$ is a single-use function. It can be constructed as the following composition:

$$(X + Y) \times Z \xrightarrow{\texttt{sym}} Z \times (X + Y) \xrightarrow{\texttt{distr}} Z \times X + Z \times Y \xrightarrow{\texttt{sym}+\texttt{sym}} X \times Z + Y \times Z$$

Using a similar idea, we can show that all the following functions are single-use:

$$\texttt{rightI} : X \to X \times 1 \quad \texttt{assoc}^{-1} : X \times (Y \times Z) \to (X \times Y) \times Z$$

$$\texttt{coassoc}^{-1} : X + (Y + Z) \to (X + Y) + Z$$

◁

Example 16. Single-use functions are closed under the following combinator:

$$\frac{X \xrightarrow{f} Z \quad Y \xrightarrow{g} Z}{X + Y \xrightarrow{[f,g]} Z}$$

Function $[f, g]$ is constructed as follows:

$$X + Y \xrightarrow{f+g} Z + Z \xrightarrow{\texttt{merge}} Z$$

This combinator can easily be generalized to any number of functions:

$$[f_1, \ldots, f_n] : X_1 + \ldots + X_n \to Y$$

◁

Example 17. The function $\texttt{distr}^{-1} : X \times Y + X \times Z \to X \times (Y + Z)$ is a single-use function. It can be constructed as follows:

$$X \times Y + X \times Z \xrightarrow{[\texttt{proj}_1 \times \texttt{coproj}_1, \texttt{proj}_1 \times \texttt{coproj}_2]} X \times (Y + Z)$$

◁

In order to simplify the notation, we declare both × and + to be right associative. This means that:

$$X_1 + X_2 + \ldots + X_n = X_1 + (X_2 + \ldots + X_n).$$

(and analogously for ×). In a similar manner, we define X^n to denote

$$\underbrace{X \times X \times \ldots \times X}_{n \text{ times}} = X \times (X \times \ldots \times X)$$

Example 18. The following function is single-use:

$$\texttt{proj}_i : X_1 \times \ldots \times X_n \to X_i$$

It can be constructed as follows:

$$X_1 \times \ldots \times X_n \xrightarrow{\texttt{proj}_2} X_2 \times \ldots \times X_n \xrightarrow{\texttt{proj}_2} \ldots \xrightarrow{\texttt{proj}_2} X_i \times \ldots \times X_n \xrightarrow{\texttt{proj}_1} X_i$$

Similarly, we show that $\texttt{coproj}_i : X_i \to X_1 + \ldots + X_n$ is a single-use function. ◁

Example 19. The following function is single-use:

$$\texttt{assoc}^* : (X_1 \times \ldots \times X_n) \times (Y_1 \times \ldots \times Y_m) \to X_1 \times \ldots \times X_n \times Y_1 \ldots \times Y_m$$

It can be constructed inductively on n:

$$(X_1 \times (X_2 \ldots \times X_n)) \times (Y_1 \times \ldots \times Y_m) \xrightarrow{\texttt{assoc}} X_1 \times ((X_2 \ldots X_n) \times (Y_1 \times \ldots \times Y_m)) \xrightarrow{\texttt{id} \times \texttt{assoc}^*}$$

$$\longrightarrow X_1 \times (X_2 \ldots \times X_n \times Y_1 \ldots \times Y_m)$$

Using a similar idea, we can extend $\texttt{assoc}^*$ to map between any two bracketings of $X_1 \times \ldots \times X_n$. In an analogous manner, we can show that the following functions are single-use:

$$\texttt{coassoc}^* : (X_1 + \ldots + X_n) + (Y_1 + \ldots + Y_m) \to X_1 + \ldots + X_n + \ldots Y_1 + \ldots Y_n$$

$$\texttt{distr}^* : (X_1 \times \ldots \times X_n) \times (Y_1 \times \ldots \times Y_m) \to X_1 \times Y_1 + X_1 \times Y_2 + \ldots + X_n \times Y_m$$

◁

Example 20. For every permutation $p : \{1, \ldots, k\} \to \{1, \ldots, k\}$, the following function is single-use:

$$\texttt{shuffle}_p : X_1 \times \ldots \times X_k \to X_{p(1)} \times \ldots \times X_{p(k)}$$

In order to see that, we notice that every $\texttt{shuffle}_p$ can from the following to functions:

$$\texttt{swap} : X_1 \times X_2 \ldots \times X_k \to X_2 \times X_1 \times \ldots \times X_k$$

$$\texttt{shift} : X_1 \times X_2 \ldots \times X_k \to X_2 \times X_3 \times \ldots \times X_k \times X_1$$

This leaves us with constructing `swap` and `shift`. Here is the construction for `swap`:

$$\mathbb{A}^k \xrightarrow{\texttt{assoc}^*} \mathbb{A}^2 \times \mathbb{A}^{k-2} \xrightarrow{\texttt{sym} \times \texttt{id}} \mathbb{A}^2 \times \mathbb{A}^{k-2} \xrightarrow{\texttt{assoc}^*} \mathbb{A}^k,$$

and here is the construction for `shift`:

$$\mathbb{A}^k \xrightarrow{\texttt{sym}} \mathbb{A}^{k-1} \times \mathbb{A} \xrightarrow{\texttt{assoc}^*} \mathbb{A}^k.$$

◁

Finally, let us show that we can use single-use bijections to present polynomial orbit-finite sets in a normal form:

Lemma 22. *For every polynomial orbit-finite X, there exists a single-use bijection:*

$$\tau_X : X \multimap \mathbb{A}^{k_1} + \mathbb{A}^{k_2} + \ldots + \mathbb{A}^{k_n},$$

such that τ_X^{-1} is a single-use function as well.

Proof. We construct the isomorphism inductively on X. First, we notice that both τ_1 and $\tau_{\mathbb{A}}$ are both equal to `id`. Then we construct $\tau_{X_1+X_2}$:

$$X_1 + X_2 \xrightarrow{\tau_1+\tau_2} (\mathbb{A}^{k_1} + \ldots + \mathbb{A}^{k_n}) + (\mathbb{A}^{l_1} + \ldots + \mathbb{A}^{l_m}) \xrightarrow{\texttt{coassoc}^*} \mathbb{A}^{k_1} + \ldots + \mathbb{A}^{l_m}$$

Finally, we construct $\tau_{X_1 \times X_2}$:

$$X_1 \times X_2 \xrightarrow{\tau_{X_1}+\tau_{X_2}} (\mathbb{A}^{k_1} + \ldots + \mathbb{A}^{k_n}) \times (\mathbb{A}^{l_1} + \ldots + \mathbb{A}^{l_m}) \xrightarrow{\texttt{distr}^*}$$

$$\mathbb{A}^{k_1}\times\mathbb{A}^{l_1}+\mathbb{A}^{k_1}\times\mathbb{A}^{l_2}+\ldots+\mathbb{A}^{k_n}\times\mathbb{A}^{l_m} \xrightarrow{\texttt{assoc}^*+\ldots+\texttt{assoc}^*} \mathbb{A}^{k_1+l_1}+\mathbb{A}^{k_1+l_2}+\ldots+\mathbb{A}^{k_n+l_m}$$

This finishes the first part of the proof. To prove that τ_X^{-1} exists and is a single-use function, we use the following claim:

Claim 14. *If a single-use function $f : X \multimap Y$ can be constructed using only the following basic functions (and all three combinators):* `sym`*,* `assoc`*,* `leftI`*,* `cosym`*,* `coassoc`*,* `id`*,* `distr`*,* $\texttt{proj}_1$ *limited to $X \times 1 \to X$, and* $\texttt{proj}_2$ *limited to $1 \times X \to X$, then $f^{-1} : Y \to X$ exists and is a single-use function.*

Proof. The proof goes by induction on the derivation of f as a single-use function. First, let us notice that all three combinators preserve reversibility:

$$(f \circ g)^{-1} = g^{-1} \circ f^{-1} \qquad (f \times g)^{-1} = f^{-1} \times g^{-1} \qquad (f + g)^{-1} = f^{-1} + g^{-1}.$$

This leaves us with showing that all the basic functions listed in the claim have single-use inverses: Each of `sym`, `cosym`, and `id` is its own inverse. Thanks to Examples 15 and 17, we know that $\texttt{distr}^{-1}$, $\texttt{assoc}^{-1}$ and $\texttt{coassoc}^{-1}$ are single-use functions. Finally, we notice that `leftI` and $\texttt{proj}_2 : 1 \times X \to X$ are each other's inverses, and the inverse of $\texttt{proj}_1 : X \times 1 \to X$ is `rightI` defined in Example 15. □

□

2.2.3 Single-use functions + `copy`

In this section we prove the following lemma, which justifies the intuition that:

$$\text{(single-use functions)} + \texttt{copy} = \text{(multiple-use functions)}$$

Lemma 23. *If we extend the class of single-use functions, by including*

$$\texttt{copy} \; : \; \mathbb{A} \; \to \; \mathbb{A} \; \times \; \mathbb{A}$$

as a basic function, we obtain the class of all finitely supported functions between polynomial orbit-finite sets.

We introduce the name *definable function*, for a function that belongs to the class of single-use functions extended with `copy`.

2.2.3.1 Definable ⇒ Finitely supported

We start the proof of Lemma 23 with the simpler inclusion:

Lemma 24. *Every definable function is finitely supported.*

Proof. Notice that for every $a \in \mathbb{A}$, the function $\mathtt{const}_a$ is supported by $\{a\}$, and that all other basic functions equivariant. To finish the proof, we need to show that all the combinators preserve finite supports:

$$\mathrm{supp}(f \circ g) \subseteq \mathrm{supp}(f) \cup \mathrm{supp}(g), \quad \mathrm{supp}(f \times g) \subseteq \mathrm{supp}(f) \cup \mathrm{supp}(g),$$

$$\text{and } \mathrm{supp}(f + g) \subseteq \mathrm{supp}(f) \cup \mathrm{supp}(g)$$

This follows from Lemma 3, because all $\circ$, $\times$, and $+$ are equivariant (higher-order) functions. □

Before we proceed with the proof of Lemma 23, we state a corollary of Lemma 24:

Lemma 25. *Every single-use function is finitely supported by the set of all constants used in its derivation. In particular, if a single-use function can be constructed without the use of any* $\mathtt{const}_a$*, then it is equivariant.*

2.2.3.2 Finitely-supported ⇒ Definable

This section is dedicated to proving the remaining inclusion:

Lemma 26. *If X and Y are polynomial orbit-finite, then every $f : X \to_{fs} Y$ is definable.*

The proof is going to be similar to the proof from Section 1.1, that every semantically equivariant transition function is also syntactically equivariant. We start by defining α-orbits, which are orbits of α-permutations:

Definition 8. Let α be a finite subset of atoms. For every element x, of a set with atoms X, we define the α-orbit of x to be the following set:

$$\{\pi(x) \mid \pi \text{ is an } \alpha\text{-permutation}\}$$

◁

Similarly as it was the case for equivariant orbits, every two α-orbits of X are either equal or disjoint. It follows that being in the same α-orbit is an equivalence relation on X, which means that every set with atoms X is partitioned into its orbits. By [Boj19, Theorem 3.16], we know that every orbit-finite set is also α-orbit-finite:

Lemma 27. *If a set X is orbit-finite, then it has finitely many α-orbits, for every $\alpha \subseteq_{fin} \mathbb{A}$.*

We start the proof of Lemma 27, by proving it in a very special case:

Claim 15. *Let α be a finite subset of atoms, and let $O \subseteq \mathbb{A}^k$ be an α-orbit of $\mathbb{A}^k$. For every α-supported $f : O \to_{fs} \mathbb{A}$, there exists a definable function $f' : \mathbb{A}^k \to \mathbb{A}$, such that f' restricted to O is equal to f.*

Proof. Take some $\bar{x}$ from O. It follows from Lemma 3 that $f(\bar{x})$ is either equal to some $a \in \alpha$ or to some $\bar{x}_i$. In the first we define $f' := \texttt{const}_a$, and in the second case we define $f' := \texttt{proj}_i$. Let us show that f' matches with f on O: Take some $\bar{y} \in O$. Since O is a single α-orbit, we know that $\bar{y} = \pi(\bar{x})$, for some α-permutation π. If $f(\bar{x}) = a$, for some $a \in \alpha$, then:

$$f(\bar{y}) = f(\pi(\bar{x})) = \pi(f(\bar{x})) = \pi(a) = a = \texttt{const}_a(\bar{x}).$$

If $f(\bar{x}) = \bar{x}_i$, then:

$$f(\bar{y}) = f(\pi(\bar{x})) = \pi(f(\bar{x})) = \pi(\bar{x}_i) = \bar{y}_i = \texttt{proj}_i(\bar{y}).$$

□

In the next step, we would like to extend Claim 15 to functions of type $O \to \mathbb{A}^k$. Before we do that, we need to define a couple of helper functions:

Example 21. The function `copy` can be extended from $\mathbb{A}$ to all polynomial orbit-finite sets. We construct $\texttt{copy} : X \to X^2$ inductively on X: If $X = 1$, then `copy` is equal to `rightI`. If $X = \mathbb{A}$, then `copy` is a basic function. If $X = X_1 \times X_2$, then `copy` can be defined as:

$$X_1 \times X_2 \xrightarrow{\texttt{copy}\times\texttt{copy}} X_1^2 \times X_2^2 \xrightarrow{\texttt{assoc}^* \circ \texttt{shuffle} \circ \texttt{assoc}^*} (X_1 \times X_2)^2$$

Finally, if $X = X_1 + X_2$, then `copy` can be defined as:

$$X_1 + X_2 \xrightarrow{\texttt{copy}+\texttt{copy}} X_1^2 + X_2^2 \xrightarrow{[\texttt{proj}_1\times\texttt{proj}_1,\texttt{proj}_2\times\texttt{proj}_2]} (X_1 + X_2)^2$$

◁

Example 22. Definable functions are closed under the following combinator:

$$\frac{X \xrightarrow{f} Y_1 \quad X \xrightarrow{g} Y_2}{X \xrightarrow{\langle f,g\rangle} Y_1 \times Y_2}$$

The function $\langle f, g\rangle$ can be constructed as:

$$X \xrightarrow{\texttt{copy}} X \times X \xrightarrow{f\times g} Y_1 \times Y_2$$

Using a similar construction, we can generalize this combinator to any number of functions:

$$\langle f_1, \ldots, f_n\rangle : X \to Y_1 \times \ldots \times Y_n$$

◁

Using the $\langle f_1, \ldots, f_k \rangle$ combinator, we can easily extend Claim 15 to functions of type $O \to_{\text{fs}} \mathbb{A}^k$. Now, let us notice that every function of type $O \to_{\text{fs}} Y$ (where Y can be any polynomial orbit-finite set) can be decomposed as:

$$O \xrightarrow{f'} \mathbb{A}^k \xrightarrow{\texttt{coproj}_i} \mathbb{A}^{l_1} + \ldots + \mathbb{A}^{l_n} \xrightarrow{\tau_Y^{-1}} Y$$

Thanks to this observation, we can extend Claim 15 to all functions of the type $O \to_{\text{fs}} Y$. In order to further extend it to $\mathbb{A}^k \to_{\text{fs}} Y$, we need to show that the definable functions can calculate the α-orbit of their argument. (By Lemma 27, we know that the set of α-orbits of $\mathbb{A}^k$ is finite, which means that it can be represented as $1 + \ldots + 1$.) Before that, we define two more helper functions:

Example 23. For every X, and for every $x \in X$, the function $\texttt{const}_x : 1 \to X$ is single-use. Let us start by defining $\texttt{const}$ for $X = \mathbb{A}^k$. For a $\bar{x} \in \mathbb{A}^k$, we construct $\texttt{const}_{\bar{x}}$ as:

$$1 \xrightarrow{\texttt{rightI}} 1 \times 1 \xrightarrow{\texttt{id} \times \texttt{rightI}} 1 \times 1 \times 1 \longrightarrow \cdots \longrightarrow 1^k \xrightarrow{\texttt{const}_{\bar{x}_1} \times \ldots \times \texttt{const}_{\bar{x}_k}} \mathbb{A}^k$$

Now, we use Lemma 22 to extend this construction to an arbitrary X: We take some $x \in X$, and we define $\bar{x} := \tau_X(x)$, then we construct $\texttt{const}_x$ as:

$$1 \xrightarrow{\texttt{const}_{\bar{x}}} \mathbb{A}^{k_j} \xrightarrow{\texttt{coproj}_j} \mathbb{A}^{k_1} + \ldots + \mathbb{A}^{k_n} \xrightarrow{\tau_X^{-1}} X$$

◁

Example 24. If X is *finite*, then every function $f : X \to Y$ is single-use. In order to see that, we first notice that since X is finite, then τ_X (from Lemma 22) has to be of the form $\tau_X : X \to 1 + \ldots + 1$. It follows that we can construct f as:

$$X \xrightarrow{\tau_X} 1 + \ldots + 1 \xrightarrow{\texttt{const}_{f(x_1)} \times \ldots \times \texttt{const}_{f(x_k)}} Y,$$

where $x_i = \tau_X^{-1}(\texttt{coproj}_i(1))$. ◁

We are now ready to show how to compute α-orbits:

Claim 16. *The following function, which computes the α-orbit of its input is definable for every $\alpha \subseteq_{fin} \mathbb{A}$:*

$$\texttt{orbit}_\alpha : \mathbb{A}^k \to 1 + \ldots 1.$$

Proof. The orbit of every $\bar{x} \in \mathbb{A}^k$ depends only on whether $\bar{x}_i = \bar{x}_j$ for every $i, j \in \{1, \ldots, k\}$, and on whether $\bar{x}_i = a$ for every $a \in \alpha$ and $i \in \{1, \ldots, k\}$. The first type of check can be performed by $\texttt{cmp}_{i,j}$, defined as:

$$\mathbb{A}^k \xrightarrow{\texttt{copy}} \mathbb{A}^k \times \mathbb{A}^k \xrightarrow{\texttt{proj}_i \times \texttt{proj}_j} \mathbb{A} \times \mathbb{A} \xrightarrow{\texttt{eq}} (1 + 1)$$

The second type of check can be performed by $\texttt{cmp}_{i,a}$, defined as:

$$\mathbb{A}^k \xrightarrow{\texttt{rightI}} \mathbb{A}^k \times 1 \xrightarrow{\texttt{proj}_i \times \texttt{const}_a} \mathbb{A} \times \mathbb{A} \xrightarrow{\texttt{eq}} (1 + 1)$$

Let $c : (1+1)^m \to 1 + \ldots + 1$ be the function that consolidates the results of all those checks, and computes the α-orbit. Thanks to Example 24, we know that c is definable. This means that we can construct $\texttt{orbit}_\alpha$ as:

$$\mathbb{A}^k \xrightarrow{\langle \texttt{cmp}_{1,1}, \ldots, \texttt{cmp}_{k,a} \rangle} (1+1)^m \xrightarrow{c} 1 + \ldots + 1$$

□

In order to extend Claim 15 from $O \to Y$ to $\mathbb{A}^k \to Y$, we combine Claim 16 with the following combinator:

Example 25. The class of single-use functions is closed under the following if-then-else combinator:

$$\frac{X \xrightarrow{f} Y \quad X \xrightarrow{g} Y}{(1+1) \times X \xrightarrow{f?g} Y}$$

The combinator is defined as follows:

$$(1+1) \times X \xrightarrow{\texttt{rightDistr}} X + X \xrightarrow{[f,g]} Y$$

Using a similar technique, we can generalize the combinator to take more than two functions:

$$(f_1 ? \ldots ? f_n) : (1 + \ldots + 1) \times X \to Y$$

◁

Finally, we use the $[f_1, \ldots, f_n]$ combinator and τ_X function, to extend Claim 16 to all functions of the type $X \to Y$. This finishes the proof of Lemma 26.

2.2.3.3 k-fold use functions

In this section we define and briefly discuss the class of *k-fold-use functions*. It is situated between single-use functions which can use only one copy of their input, and finitely supported functions which can use any number of copies.

Definition 9. For every $k \in \mathbb{N}$, we say that a function is k-fold-use, if it can be constructed as a composition of the following form:

$$X \xrightarrow{\texttt{copy}^k} X^k \xrightarrow{f'} \!\!\circ\; Y,$$

where f' is some single-use function. We denote the set of all k-fold use functions between polynomial orbit-finite X and Y as $X \multimap_k Y$. ◁

It is easy to see that:

$$(X \multimap Y) = (X \multimap_1 Y) \subseteq (X \multimap_2 Y) \subseteq (X \multimap_3 Y) \subseteq \ldots$$

It is also not hard to see that this hierarchy is strict:

Example 26. Consider the following function $f \in \mathbb{A} \to_{\mathrm{fs}} \{\mathrm{Yes}, \mathrm{No}\}$, which is supported by $\{4, 7\}$:

$$f(x) = \begin{cases} \mathrm{Yes} & \text{if } x \in \{4, 7\} \\ \mathrm{No} & \text{otherwise} \end{cases}$$

This function clearly belongs to $\mathbb{A} \multimap_2 \{\mathrm{Yes}, \mathrm{No}\}$, but not to $\mathbb{A} \multimap \{\mathrm{Yes}, \mathrm{No}\}$. (The formal proof that $f \notin \mathbb{A} \multimap \{\mathrm{Yes}, \mathrm{No}\}$ follows from the decision-tree representation of single-use functions, presented later in this chapter.) ◁

This example can easily be generalized, to show that:

$$(X \multimap_1 Y) \subsetneq (X \multimap_2 Y) \subsetneq (X \multimap_3 Y) \subsetneq \ldots$$

In the limit, this sequence reaches $X \to_{\mathrm{fs}} Y$:

Lemma 28. *For every polynomial orbit-finite X and Y:*

$$X \to_{fs} Y \;=\; \bigcup_{k \in \mathbb{N}} X \multimap_k Y$$

Proof. The proof that every k-fold use function is finitely supported is almost the same as the proof of Lemma 25. Now, let us prove the other inclusion: all the basic functions from Definition 7 are 1-fold, and `copy` is easily seen to be 2-fold, so thanks to Lemma 26, it suffices to show that all three combinators $+$, $\times$, and $\circ$ preserve belonging to $\bigcup_{k \in \mathbb{N}} X \multimap_k Y$:

We start with $\circ$. Let us show that we can present the following composition as an m-fold use function, for some $m \in \mathbb{N}$:

$$X \xrightarrow{\mathtt{copy}^k} X^k \overset{f}{\multimap} Y \xrightarrow{\mathtt{copy}^l} Y^l \overset{g}{\multimap} Z$$

We do this in the following way, with $m = k \cdot l$:

$$X \xrightarrow{\mathtt{copy}^{k \cdot l}} X^{k \cdot l} \overset{\mathtt{assoc}^*}{\multimap} \left((X^k)^l\right) \overset{f \times \ldots \times f}{\multimap} Y^l \overset{g}{\multimap} Z$$

For $\times$ and $+$, it is not hard to see that if f is a k-fold-use function, and l is a l-fold-use function, then both $f + l$ and $f \times l$ are $\max(k, l)$-fold-use functions. □

2.2.4 Single-use decision trees

In this section we prove the most important property of single-use functions:

Theorem 5. *The set of all single-use functions $X \multimap Y$ is orbit-finite, for all polynomial orbit-finite X and Y.*

We prove Theorem 5 by introducing the *single-use decision tree* representation of single-use functions – for every polynomial X and Y, we define the set $\mathrm{Trees}(X, Y)$, such that there is an equivariant surjection[5]:

$$\mathrm{Trees}(X, Y) \to_{\mathrm{eq}} (X \multimap Y)$$

[5] This tree representation is not bijective – many different trees can describe the same function.

Then, we show that the set Trees(X, Y) is polynomial orbit-finite for every X and Y. This is enough to proof Theorem 5, because by [Boj19, Lemma 3.24] images of orbit-finite sets under equivariant functions remain orbit-finite.

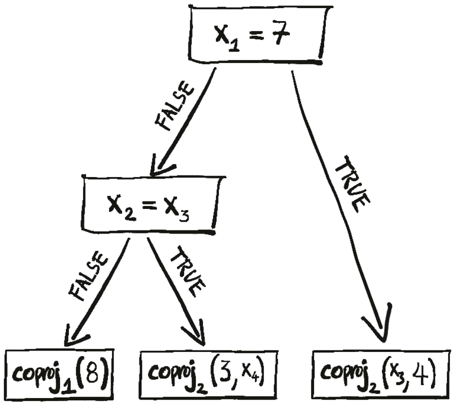
Definition 10. We start defining the *single-use decision trees* with the most interesting case, i.e. trees for functions of the following type:

$$\mathbb{A}^k \multimap \mathbb{A}^{l_1} + \ldots + \mathbb{A}^{l_n}.$$

A single-use decision tree of this type is a tree whose nodes contain *quieries* and whose nodes contain *constructors.* Each constructor is of the following form:

$$\texttt{coproj}_i(v_1, \ldots, v_{l_i}),$$

where each v_j is either an input variable $\texttt{x}_\texttt{v}$ or an atomic constant $a \in \mathbb{A}$. Each query is either of the form $\texttt{x}_\texttt{i} = \texttt{x}_\texttt{j}$ or $\texttt{x}_\texttt{i} = \texttt{a}$, where $i, j \in \{1, \ldots, k\}$. Moreover, every single-use tree has to satisfy the *single-use restriction*, which says that on every path from the root to a leaf, each variable $\texttt{x}_\texttt{i}$ may appear at most once (in queries or constructors). Here is an example of a single-use decision tree of type $\mathbb{A}^2 \multimap \mathbb{A} + \mathbb{A}^2$:



Each such tree naturally represents a function $f_T : \mathbb{A}^k \multimap \mathbb{A}^{l_1} + \ldots + \mathbb{A}^{l_n}$. It is not hard to see that the single-use restriction on T guarantees that f_T is a single-use function.

The construction for trees of more general types is standard. The single-use decision trees of the type $\mathbb{A}^{k_1} + \ldots + \mathbb{A}^{k_n} \multimap \mathbb{A}^{l_1} + \ldots + \mathbb{A}^{l_m}$ are simply n-tuples of trees $T = (T_1, \ldots, T_n)$ such that

$$T_i \in \text{Trees}(\mathbb{A}^{k_1}, \mathbb{A}^{l_1} + \ldots + \mathbb{A}^{l_m}).$$

In this case, function f_T is defined using the combinator Example 16:

$$f_T : \mathbb{A}^{k_1} + \ldots + \mathbb{A}^{k_n} \xrightarrow{[f_{T_1}, \ldots, f_{T_n}]} \mathbb{A}^{l_1} + \ldots + \mathbb{A}^{l_n}$$

Finally, in order to define single-use decision trees of type $X \multimap Y$, where X and Y are arbitrary polynomial orbit-finite sets, we use the isomorphisms from Lemma 22:

$$\tau_X : X \to \mathbb{A}^{k_1} + \ldots + \mathbb{A}^{k_n} \qquad \tau_Y : Y \to \mathbb{A}^{l_1} + \ldots + \mathbb{A}^{l_m}.$$

A single-use decision tree T representing a function $X \multimap Y$ is a tree:

$$T' \in \text{Trees}(\mathbb{A}^{k_1} + \ldots + \mathbb{A}^{l_n}, \mathbb{A}^{k_1} + \ldots + \mathbb{A}^{l_m})$$

It represents the following function:

$$f_T : X \xrightarrow{\tau_X} \mathbb{A}^{k_1} + \ldots + \mathbb{A}^{k_n} \xrightarrow{f_{T'}} \mathbb{A}^{l_1} + \ldots + \mathbb{A}^{l_k} \xrightarrow{\tau_Y^{-1}} Y$$

◁

It is not hard to see that the mapping $T \mapsto f_T$ is equivariant, i.e. $f_{\pi(T)} = \pi(f_T)$ for all atom permutations.

2.2.4.1 Single-use function ⇒ Single-use decision trees

In this section we show that the single-use decision tree representation is injective:

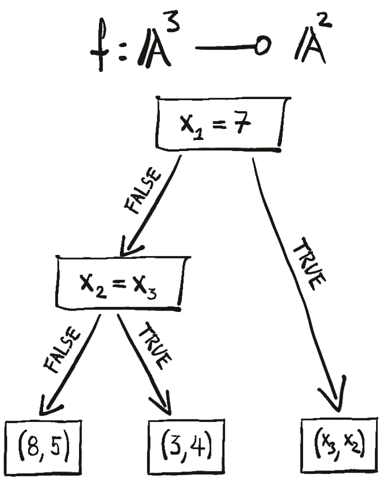
Lemma 29. *Every function from $X \multimap Y$ is represented by some $T \in$ Trees(X, Y).*

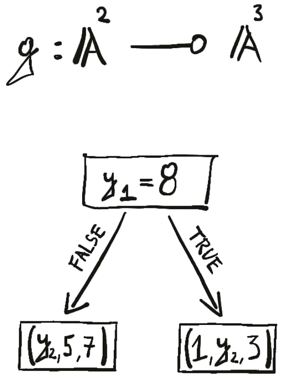
We prove the lemma, by showing that the class of functions recognized by single-use decision trees is closed under $\circ$, $+$, and $\times$; and contains all basic functions from Definition 7.

We start with the most interesting case, which is showing that single-use decision trees are closed under compositions. We first show it for trees on tuples:

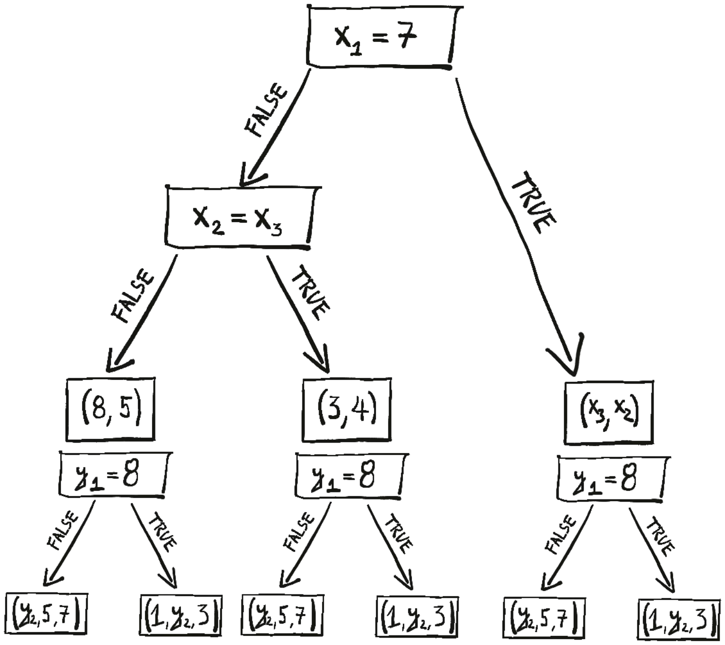
Claim 17. *If $g : \mathbb{A}^k \multimap \mathbb{A}^l$ and $f : \mathbb{A}^l \multimap \mathbb{A}^m$ are represented by single-use decision trees, then so is $(f \circ g) : \mathbb{A}^k \multimap \mathbb{A}^l$.*

Proof. Let us take two decision trees F, G that represent f and g, and let us show how to compose them, obtaining H that represent $f \circ g$. Here are some example F and G (for clarity we denote the variables in F as x_i, and the variables in G as y_i):

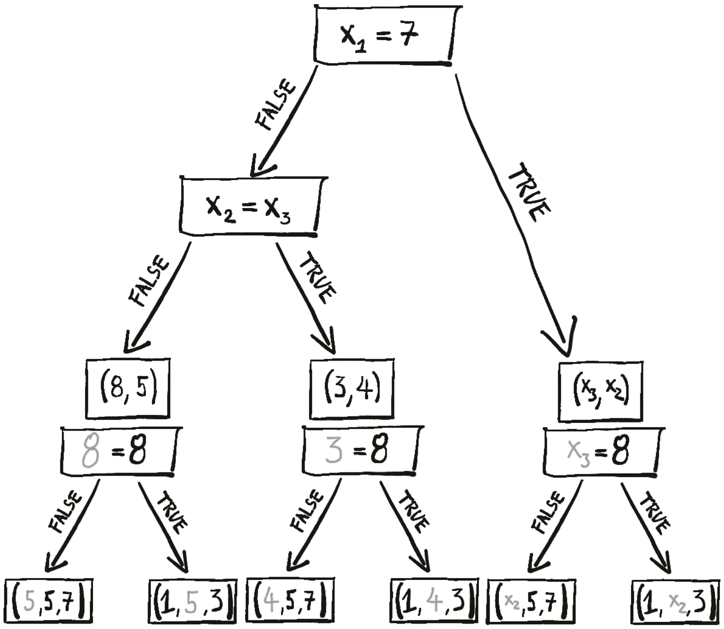




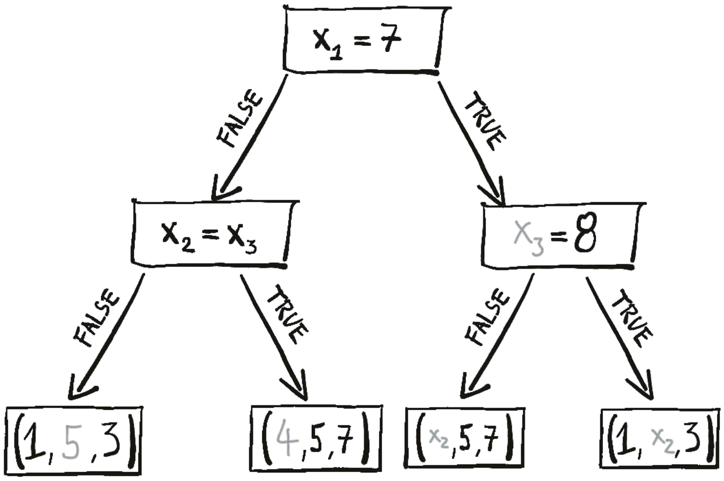
We start the construction by placing one copy of G under each leaf of F:



Then we replace every y_i in each copy of G, with the v_i from the leaf of F (each v_i is either som x_j or some $a \in \mathbb{A}$):



Finally, we resolve all the queries that can be resolved (and forget about F's leaves):



The constructed tree recognizes $f \circ g$ by design. It is also not hard to see that the construction preserves the single-use restriction. □

It is not hard to see that Claim 17 can be first generalized for tress Trees($\mathbb{A}^{k_1} + \ldots + \mathbb{A}^{k_n}, \mathbb{A}^{l_1} + \ldots + \mathbb{A}^{l_m}$), and then to all single-use decision trees.

We continue the proof of Lemma 29 by showing that functions recognized by single-use decision trees are closed under + and ×:

Lemma 30. *If $f : X_1 \to Y_1$ and $g : X_1 \to X_2$ are represented by single-use decision trees, then so is $(f + g) : X_1 + X_2 \to Y_1 + Y_2$.*

Proof. If f is represented by $(F_1, \ldots, F_n)$, and g is represented by $(G_1, \ldots, G_m)$, then $f+g$ is represented by $(F_1, \ldots, F_n, G'_1, \ldots, G'_m)$, where G'_i is a modification of G_i in which each $\mathtt{coproj}_j$ is replaced with $\mathtt{coproj}_{n+j}$. □

Lemma 31. *If $f : X_1 \to Y_1$ and $g : X_1 \to X_2$ are represented by single-use decision trees, then so is $(f \times g) : X_1 \times X_2 \to Y_1 \times Y_2$.*

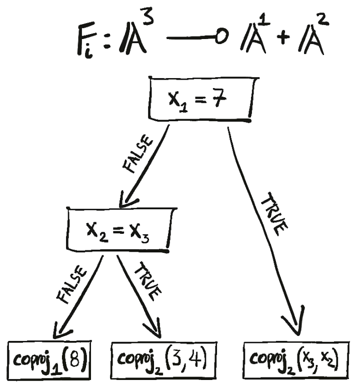
Proof. Let f be represented by $(F_1, \ldots, F_n)$ and g be represented by $(G_1, \ldots, G_m)$, and let us construct H that represents $f \times g$. First of all, notice that H should be a tuple of $n \cdot m$ trees $(H_{1,1}, \ldots, H_{i,j}, \ldots, H_{n,m})$. Let us show, how to construct $H_{i,j}$. First, we notice that if F_i belongs to $\text{Trees}(\mathbb{A}^{k_i}, \mathbb{A}^{l_1} + \ldots + \mathbb{A}^{l_m})$ and G_j belongs to $\text{Trees}(\mathbb{A}^{k'_j}, \mathbb{A}^{l'_1} + \ldots + \mathbb{A}^{l'_{m'}})$, then $H_{i,j}$ should belong to:

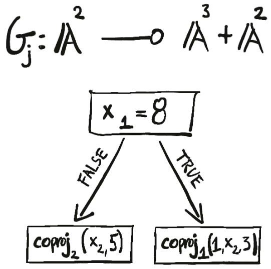
$$\text{Trees}(\mathbb{A}^{k_i + k'_j},\ \mathbb{A}^{l_1 + l'_1} + \ldots + \mathbb{A}^{l_i + l'_j} + \ldots + \mathbb{A}^{l_m + l'_{m'}}).$$

In order to construct $H_{i,j}$, we take F_i and replace all its leaves with a modified version of G_j, where each variable x_p has been replaced by x_{p+k}, and in which every leaf has been replaced with the following merge of the leaf from F_i with the leaf from G_j:

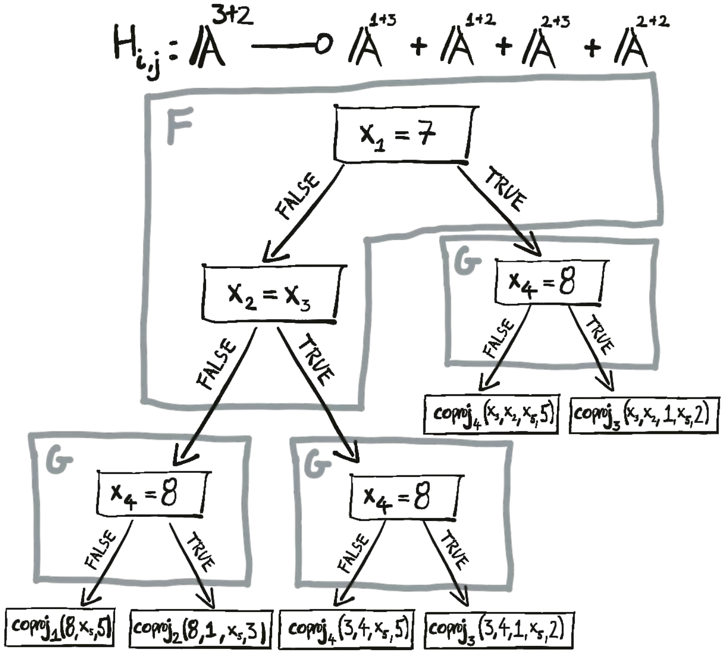
$$\mathtt{coproj}_a(v_1, \ldots, v_{l_a}),\ \mathtt{coproj}_b(w_1, \ldots, w_{l'_b}) \mapsto \mathtt{coproj}_c(v_1, \ldots, v_{l_a}, w_1, \ldots, w_{l'_b}),$$

where c is the index of $\mathbb{A}^{k'_i + l'_j}$ in $\mathbb{A}^{k'_1 + l'_1} + \ldots + \mathbb{A}^{k'_{n'} + l'_{m'}}$. For example, consider the following F_i, G_j:





For those F_i, G_j, we construct the following $H_{i,j}$:



$H_{i,j} : \mathbb{A}^{3+2} \multimap \mathbb{A}^{1+3} + \mathbb{A}^{1+2} + \mathbb{A}^{2+3} + \mathbb{A}^{2+2}$
F
$x_1 = 7$
FALSE
TRUE
$x_2 = x_3$
G
$x_4 = 8$
FALSE
TRUE
$\mathtt{coproj}_4(x_3, x_2, x_5, 5)$
$\mathtt{coproj}_3(x_3, x_2, 1, x_5, 2)$
FALSE
TRUE
G
$x_4 = 8$
FALSE
TRUE
G
$x_4 = 8$
FALSE
TRUE
$\mathtt{coproj}_1(8, x_5, 5)$
$\mathtt{coproj}_2(8, 1, x_5, 3)$
$\mathtt{coproj}_4(3, 4, x_5, 5)$
$\mathtt{coproj}_3(3, 4, 1, x_5, 2)$

□

Finally, we need to show that all basic functions from Definition 7 can be implemented as single-use decision trees. For the sake of conciseness, we only show how to implement $\mathtt{proj}_1$ (the implementations for other basic functions are analogous):

Lemma 32. *For every X and Y, the function $\mathtt{proj}_1 : X \times Y \to X$ is represented by some $T \in$ Trees$(X \times Y, X)$.*

Proof. Let τ_X and τ_Y (from Lemma 22) have the following types:

$$\tau_X : X \to \mathbb{A}^{k_1} + \ldots + \mathbb{A}^{k_n} \qquad \tau_Y : Y \to \mathbb{A}^{l_1} + \ldots + \mathbb{A}^{l_m}$$

This means that for every i, j, we need to define a single-use decision tree $T_{i,j} \in$ Trees$(\mathbb{A}^{k_i + l_j}, \mathbb{A}^{k_1} + \ldots + \mathbb{A}^{k_n})$, such that:

$$\mathtt{proj}_1 = \tau_X^{-1} \circ (T_{1,1} + \ldots + T_{i,j} + \ldots + T_{n,m}) \circ \tau_{X \times Y}$$

It is not hard to see that we can construct $T_{i,j}$ to be a single leaf with the constructor $\mathtt{coproj}_i(x_1, \ldots, x_{k_i})$. (Note that the constructor forgets about variables x_{k_i+1} to $x_{k_i+l_j}$.) □

This concludes the proof of Lemma 29.

2.2.4.2 Single-use decision trees are polynomial orbit finite

This section is dedicated to proving the following lemma:

Lemma 33. *For every polynomial orbit-finite X, Y, the set Trees(X, Y) is polynomial orbit-finite as well.*

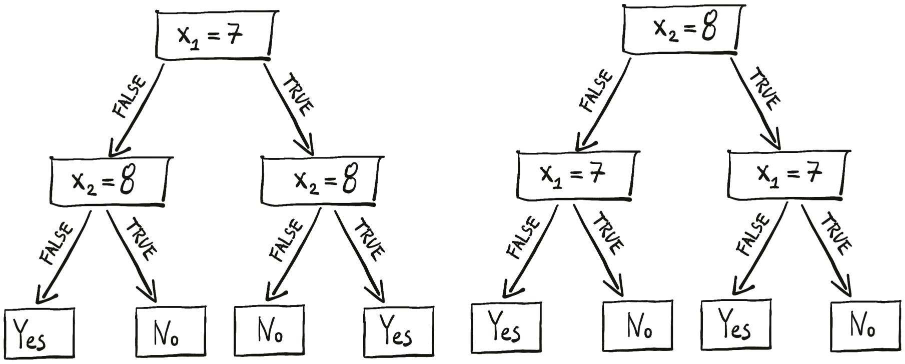
Proof. It suffices to prove the lemma only for the case where $X = \mathbb{A}^k$. The general version follows from Lemma 6. Notice that, thanks to the single-use restriction, the depth of the trees from Trees$(\mathbb{A}^k, Y)$ is bounded by $k+1$. Now, the key observation is that there is only finitely many possible *shapes* of binary trees of bounded height (a *shape* of a tree is a version of the tree where all atoms have been replaced by an atomless blank). Let $\{s_1, \ldots, s_k\}$ be the set of these shapes, and denote $|s_i|$ to be the number of blanks (i.e. atoms) in each tree of shape s_i. Then, the set of all from Trees$(\mathbb{A}^k, \mathbb{A}^l)$ can be represented as the following polynomial orbit-finite set:

$$\mathbb{A}^{|s_1|} + \ldots + \mathbb{A}^{|s_k|}$$

□

2.2.4.3 Canonical single-use decision trees

The definition of polynomial orbit-finite sets (Definition 6) does not include the constructor of single-use function spaces ($\multimap$). This means that if we want to treat $X \multimap Y$ as a polynomial orbit-finite set (for example to talk about higher-order single-use functions), we need to represent it as a polynomial orbit-finite set. For this purpose, we are going to use Trees(X, Y). Observe that, this tree representation of single-use functions is not injective – the following two decision trees represent the same function:



In this section we show that there is an equivariant way of choosing a canonical tree Trees(X, Y) for every function $X \multimap Y$ (this is not immediate, because sets with atoms do not always admit choice – see Claim 6):

Lemma 34. *For every polynomial orbit-finite X and Y there exists a function:*

$$\texttt{treeRepr} : (X \multimap Y) \to_{eq} \mathit{Trees}(X, Y),$$

such that $\texttt{treeRepr}(f)$ *represents* f*.*

As usual, we first prove the lemma for the special case where $X = \mathbb{A}^k$ and $Y = \mathbb{A}^{l_1} + \ldots + \mathbb{A}^{l_n}$: Define the height of a function $f \in (X \multimap Y)$ to be the smallest height of a tree from $\text{Trees}(X, Y)$ that recognizes f and define $\text{SU}_{\leq h}(X, Y) \subseteq X \multimap Y$ be the set of all single-use functions recognized by trees that are not taller than h. We prove by induction on h that for every h there is a function:

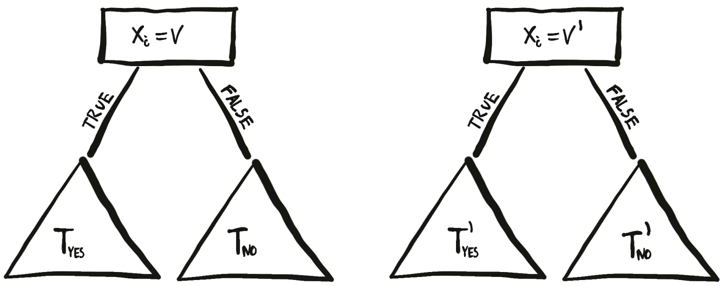
$$\texttt{treeRepr}_h : \text{SU}_{\leq h}(X, Y) \to_{\text{eq}} \text{Trees}(X, Y)$$

This is enough to prove the lemma: Thanks to the single-use restriction, the height of trees from $\text{Trees}(\mathbb{A}^k, Y)$ is bounded by $k + 1$, which means that $\texttt{treeRepr} = \texttt{treeRepr}\leq (k + 1)$.

We start the inductive proof with $h = 1$. If f has height 1, then it is recognized by some leaf. Moreover, it is not hard to see that if $\texttt{coproj}_j(v'_1, \ldots, v'_{l_j})$ and $\texttt{coproj}_{j'}(v'_1, \ldots, v'_{l_{j'}})$ recognize the same function, then $j = j'$, and for every i, $v_i = v'_i$. This means that we can define $\texttt{treeRepr}_1(f)$ to be the only leaf that recognizes f.

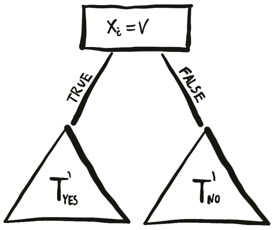
For the induction step we take a $h > 1$. Let f be a function in $\text{SU}_h(X, Y)$. We assume that the height of h is at least 2 – if not, we can simply repeat the construction from the induction base. We say that a tree is *minimal* if no tree of smaller height that represents the same function. We say that x_i is the leading variable of f, if i is the smallest index, for which there exists a minimal tree that represents f and queries x_i in its root. It is not hard to see that the function that maps f to its leading variable is equivariant. Let x_i be the leading variable of f, and let T be a minimal tree that recognizes f and queries x_i in its root. Let $\texttt{x}_\texttt{i} = \texttt{v}$ be the query from the root of T, and let T_{Yes} and T_{No} be the subtrees of T. The following claim states that v, $f_{T_{\text{Yes}}}$ and $f_{T_{\text{No}}}$ do not depend on the choice of T:

Claim 18. *Let T and T' be two minimal trees that recognize the same function, and query the same variable x_i in their roots:*



It follows that $v = v'$, $f_{T_{Yes}} = f_{T'_{Yes}}$, and $f_{T_{No}} = f_{T'_{No}}$.

Before we prove the claim, we finish the induction step. Notice that the heights of $f_{T_{\text{Yes}}}$ and $f_{T_{\text{No}}}$ are strictly lower than h. It follows that we can define $f_{\text{Yes}} = \texttt{treeRepr}(f_{T_{\text{Yes}}})$, $f_{\text{No}} = \texttt{treeRepr}(f_{T_{\text{No}}})$, and define $\texttt{treeRepr}(f)$ to be:



The construction does not depend on the choice of T, which makes it equivariant. This leaves us with proving Claim 18:

Proof of Claim 18: Let us take $T, T' \in \text{Trees}_h(X, Y)$ that recognize the same function $f \in \text{SU}_h(X, Y)$, and query the same x_i in their roots. Let $\texttt{x}_\texttt{i} = \texttt{v}$ and $\texttt{x}_\texttt{i} = \texttt{v}'$ be the root queries of T and T', and let T_{Yes}, T_{No}, T'_{Yes}, and T'_{No} be their subtrees. Notice that $f_{T_{\text{Yes}}} \neq f_{T_{\text{No}}}$, or otherwise f would have been recognized by T_{Yes}, and T would not be minimal. Similarly, $f_{T'_{\text{Yes}}} \neq f_{T'_{\text{No}}}$. Let us now show that $v = v'$. Since both v and v' might be equal to an input variable, or to an atomic constant, we need to consider four cases. They are all similar to each, so we only present the proof for $v = a \in \mathbb{A}$ and $v' = b \in \mathbb{A}$: Let us assume that $a \neq b$ and show that this leads to a contradiction. We start by noticing that thanks to the single-use restriction none of T_{Yes}, T_{No}, T'_{Yes}, T'_{No} uses x_i. It follows that, for every $\bar{x} \in \mathbb{A}^k$ and for every $c \in \mathbb{A}$:

$$T_{\text{Yes}}(\bar{x}[x_i := c]) = T_{\text{Yes}}(\bar{x}) \quad \text{and} \quad T_{\text{No}}(\bar{x}[x_i := c]) = T_{\text{No}}(\bar{x}).$$

Similarly for T'_{Yes} and T'_{No}. Moreover, since f is recognized by T and T', it follows that for every $\bar{x} \in \mathbb{A}^k$ and for every $c \in \mathbb{A}$:

$$f(\bar{x}[x_i := c]) = \begin{cases} T_{\text{Yes}}(\bar{x}) & \text{if } c = a \\ T_{\text{No}}(\bar{x}) & \text{if } c \neq a \end{cases} \qquad f(\bar{x}[x_i := c]) = \begin{cases} T'_{\text{Yes}}(\bar{x}) & \text{if } c = b \\ T'_{\text{No}}(\bar{x}) & \text{if } c \neq b \end{cases}$$

Since $f_{T_{\text{Yes}}} \neq f_{T_{\text{No}}}$, we know that there is a $\bar{x} \in \mathbb{A}^k$, such that $T_{\text{Yes}}(\bar{x}) \neq T_{\text{No}}(\bar{x})$. This leads to a contradiction because, if we take $c \in \mathbb{A}$, that is different from both a and b, we have that:

$$T_{\text{Yes}}(\bar{x}) = f(\bar{x}[x_i := a]) = T'_{\text{No}}(\bar{x}) = f(\bar{x}[x_i := c]) = T_{\text{No}}(\bar{x})$$

It follows that $v = v'$. Let us now show that $f_{T_{\text{Yes}}} = f_{T'_{\text{Yes}}}$ and $f_{T_{\text{No}}} = f_{T'_{\text{No}}}$. Since v, v' might both be equal to some $a \in \mathbb{A}$, or to some x_j, we have two cases to consider. Again, they are very similar, so we only present the proof for $v = v' = a \in \mathbb{A}$. In this case, for all $\bar{x} \in \mathbb{A}^k$:

$$T_{\text{Yes}}(\bar{x}) = f(\bar{x}[x_i := a]) = T'_{\text{Yes}}(\bar{x})$$

Similarly, if we take $b \in \mathbb{A}$, that is different from a, then for all $\bar{x}$:

$$T_{\text{No}}(\bar{x}) = f(\bar{x}[x_i := b]) = T'_{\text{No}}(\bar{x})$$

□

Let us now define `treeRepr` for functions $\mathbb{A}^{k_1} + \ldots + \mathbb{A}^{k_n} \multimap \mathbb{A}^{l_1} + \ldots + \mathbb{A}^{l_m}$:

$$\texttt{treeRepr}(f) := (\texttt{treeRepr}(f \circ \texttt{coproj}_1), \ldots, \texttt{treeRepr}(f \circ \texttt{coproj}_n))$$

Finally, we define the representation for functions $X \multimap Y$:

$$\texttt{treeRepr}(f) := \texttt{treeRepr}(\tau_Y \circ f \circ \tau_X^{-1})$$

2.3 Single-use automata

In this section we define the *single-use automaton* – a model which (slightly) generalizes the single-use register automaton. A single-use automaton uses a polynomial orbit-finite set of states Q to process words over a polynomial orbit-finite alphabet Σ. It has an equivariant initial state $q_0 \in Q$, and a single-use acceptance function $f : Q \multimap \{\text{Yes}, \text{No}\}$. The type of its transition function is slightly more complicated. We discuss it in the following section.

2.3.1 Single-use transition function

The first idea for the type of the transition function is:

$$(\Sigma \times Q) \multimap Q.$$

Such a transition function would only allow the automaton to use one copy of each input letter, which would make single-use automata weaker than both single-use register automata and orbit-finite monoids:

Example 27. The following language over $\mathbb{A}$ is easily seen to be recognized by an orbit-finite monoid, and by a single-use register automaton, but not by a single-use polynomial orbit-finite automaton of type $\Sigma \times Q \multimap Q$:

"The length of the word is 3 and all its letters are pairwise distinct".

◁

The main focus of this thesis are models equivalent to orbit-finite monoids, so we are not going to discuss single-use automata of type $\Sigma \times Q \multimap Q$ (which might be interesting in another context). Instead, we present two equivalent ways of typing the transition function that allow the automaton to use multiple copies of the input letters:

The first one is to let the automaton explicitly specify, how many copies of the input letters it requires. We add this information as a number $k \in \mathbb{N}$ to the automaton's specification, and then we type its transition function as:

$$\underbrace{\Sigma^k}_{\substack{k \text{ copies of} \\ \text{the same letter}}} \times \underbrace{Q}_{\substack{\text{one copy of} \\ \text{the state}}} \multimap Q$$

Let us stress that although, it is possible to feed a function of this type with k different letters from Σ, we are only going to use it with k identical copies of the input letter. Another way of typing the transition function is as follows:

$$\Sigma \to_{\text{eq}} (Q \multimap Q)$$

The function transforms every letter in a multiple-use, equivariant way into a single-use state transformation (which usually is not going to be equivariant, because it is going to use atoms from the input letter). This way, we give the automaton a multiple-use access to Σ and a single-use access to Q. A similar approach to typing the transition function can be found in *weighted automata*, where Σ is a finite set, Q is a finite-dimensional linear space, and the transition function maps every letter to a linear transformation:

$$\Sigma \to (Q \to_{\text{lin}} Q),$$

Before we show that the two ways of defining transition functions are equivalent, we need to discuss one more property of single-use functions:

2.3.1.1 Single-use currying

The following isomorphism called *currying* holds for many different classes of functions, such as all functions, finitely supported functions, linear transformations, ...

$$\begin{array}{ccc} A \to (B \to C) & \simeq & A \times B \to C \\ F(f)(a,b) = f(a)(b) & & F^{-1}(f)(a)(b) = f(a,b) \end{array}$$

In this section we would like to discuss currying for single-use functions, i.e. talk about the relationship between:

$$A \times B \multimap C \quad \text{and} \quad A \multimap \text{Trees}(B, C)$$

(As mentioned before, we need to represent $B \multimap C$ as $\text{Trees}(B, C)$).

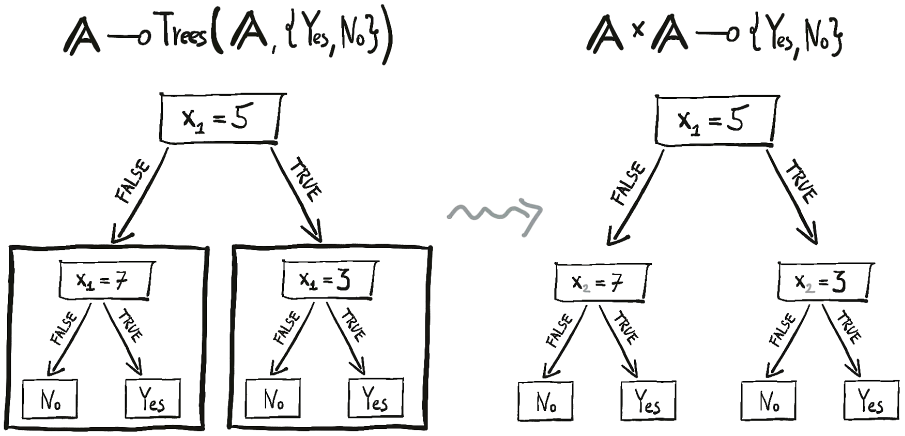
First, we notice that every single-use function $A \multimap \text{Trees}(B, C)$, can be transformed into $A \times B \multimap C$:

Lemma 35. *For every $f : A \multimap \text{Trees}(B, C)$, there exists an $f' : A \times B \multimap C$, such that for every $a \in A$ and $b \in B$:*

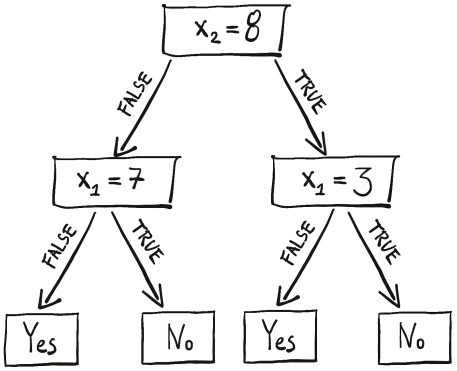
$$f(a)(b) = f'(a, b)$$

Moreover, the mapping $f \mapsto f'$ is an equivariant function.

In order to construct f', we simply have to unpack the leaves of f:



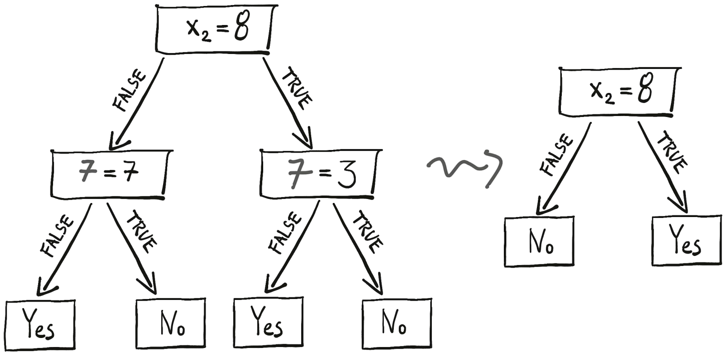
On the other hand, it might not be possible to translate a function $A \times B \multimap C$ into a function $A \multimap \text{Trees}(B, C)$. For example, consider the following function $f : \mathbb{A} \times \mathbb{A} \multimap \mathbb{A}$:



It is not hard to see that this function cannot be translated into a function $\mathbb{A} \multimap \text{Trees}(\mathbb{A}, \{\text{Yes}, \text{No}\})$ – this is because we need to compare the input argument with both 7 and 3. However, the variable x_1 appears only twice in the tree, which means that we can express f as $f' \in \mathbb{A} \multimap_2 \text{Trees}(\mathbb{A}, \{\text{Yes}, \text{No}\})$:

$$f'(a) = \text{resolve all resolvable queries } T[x_1 := a]$$

For example, let us compute $f'(7)$. We substitute both appearances of x_1 with the atomic constant 7, and resolve all resolvable queries:



This construction can be generalized:

Lemma 36. *For every A, B, and C, there is an $l \in \mathbb{N}$, such that for every f of type $A \times B \multimap C$, there exists an $f' : A \multimap_l$ Trees(B, C), such that for every $a \in A$, $b \in B$ and $c \in C$:*

$$f(a, b) = f'(a)(b)$$

Moreover, the mapping $f \mapsto f'$ is equivariant.

Proof. Note that thanks to the single-use restriction, the size of a tree in Trees$(A \times B, C)$ is bounded. This means that there is an $l \in \mathbb{N}$ such that every x_i appears at most l times in Trees$(A \times B, C)$. This means that we can construct f' in the same way as in the previous example. □

We are now ready to show that the two types of transition functions are equivalent:

Lemma 37. *For every $k \in \mathbb{N}$ and for every equivariant single-use function $f : \Sigma^k \times Q \multimap Q$, there is a function $f' : \Sigma \to_{eq} (Q \multimap Q)$, such that for every $a \in \Sigma$ and $q \in Q$:*

$$f'(a)(q) = f(\underbrace{a, a, \ldots, a}_{k \text{ times}}, q)$$

And conversely: for every $f \in \Sigma \to_{eq} (Q \multimap Q)$, there exists a $k \in \mathbb{N}$ and an equivariant $f' : \Sigma^k \times Q \multimap Q$, such that:

$$f'(\underbrace{a, a, \ldots, a}_{k \text{ times}}, q) = f(a)(q)$$

Proof. The first part of the proof follows from Lemma 36. To prove the second part, we transform $f : \Sigma \to_{\mathrm{eq}} (Q \multimap Q)$ into an equivalent $f' : \Sigma^k \times (Q \multimap Q)$, in the following way:

$$\boxed{\Sigma \to_{\mathrm{eq}} (Q \multimap Q)} \xrightarrow{\texttt{treeRepr} \circ \cdot} \boxed{\Sigma \to_{\mathrm{eq}} (\mathrm{Trees}(Q, Q))} \xrightarrow{\text{Lemma 28}}$$

$$\boxed{\Sigma \multimap_k \mathrm{Trees}(Q, Q)} \xrightarrow{\text{Definition 9}} \boxed{\Sigma^k \multimap \mathrm{Trees}(Q, Q)} \xrightarrow{\text{Lemma 35}} \boxed{\Sigma^k \times Q \multimap Q}$$

□

2.3.2 Single-use automaton

We are now ready to define the *single-use automaton*:

Definition 11. A *single-use automaton* consists of:

1. a polynomial orbit-finite alphabet Σ;
2. a polynomial orbit-finite set of states Q;
3. an equivariant initial state $q_0 \in Q$;
4. an equivariant single-use[6] acceptance function $f : Q \multimap \{\text{Yes}, \text{No}\}$;
5. a transition function $\delta : \Sigma \to_{\text{eq}} (Q \multimap Q)$.

◁

Observe that for $\Sigma = \mathbb{A}$, the single-use automaton model is equivalent to the single-use register automaton: Thanks to Example 12, we know that the set of memory configurations of every single-use register automaton can be represented as a polynomial orbit-finite set. Conversely, by Lemma 22, every polynomial-orbit-finite set is isomorphic to

$$\mathbb{A}^{k_1} + \mathbb{A}^{k_2} + \ldots + \mathbb{A}^{k_n},$$

whose elements can be stored in the memory of a single-use register automaton. The transition and acceptance functions are easily seen to be intertranslatable between the two models (thanks to the single-use decision tree representation).

As we have mentioned before, the expressive power of the single-use automaton does not change if we allow for the acceptance function to be multiple-use:

Lemma 38. *For every single-use automaton whose acceptance function has the type $Q \to_{eq} \{Yes, No\}$, there exists a standard single-use automaton (with a single-use accepting function $Q \multimap \{Yes, No\}$), which recognizes the same language.*

Proof. Let $\mathcal{A}$ be the automaton whose acceptance function f has the type $Q \to_{\text{eq}} \{\text{Yes, No}\}$. By Lemma 28, the function f can be transformed into an equivalent $f' : Q \multimap_k \{\text{Yes}, \text{No}\}$. The acceptance function is used only once, at the end of the computation, so we can simulate $\mathcal{A}$, using a single-use automaton that maintains k copies of the state of $\mathcal{A}$ (the automaton's set of states is Q^k). □

2.3.2.1 Single-use two-way automaton

Using single-use functions, we can easily define single-use versions of many classical automata models: In the next two chapters, we are going to see single-use Mealy machines, single-use two-way transducers, and single-use SSTs. For now, we define the *single-use two-way automaton*:

[6] See Lemma 38.

Definition 12. A *single-use two-way automaton* consists of:

1. a polynomial orbit-finite alphabet Σ;
2. a polynomial orbit-finite set of states Q;
3. an equivariant initial state $q_0 \in Q$;
4. a two-way transition function:

$$\delta \ : \ (\Sigma + \{\vdash, \dashv\}) \ \longrightarrow_{\text{eq}} \ (Q \ \multimap \ (Q \times \{\leftarrow, \rightarrow\}) \ + \ \{\text{Yes}, \text{No}\})$$

$\triangleleft$

The model is analogous to the orbit-finite two-way automaton from Section 1.4.1, but much weaker. An important difference between the two models is that the single-use two-way automaton has decidable emptiness. This is because, as we are going to show in Theorem 6, every two-way single-use automaton can be effectively translated into a one-way single-use automaton. Which is, in turn, a special case of a one-way orbit-finite automaton whose emptiness is decidable by [Boj19, Theorem 5.12].

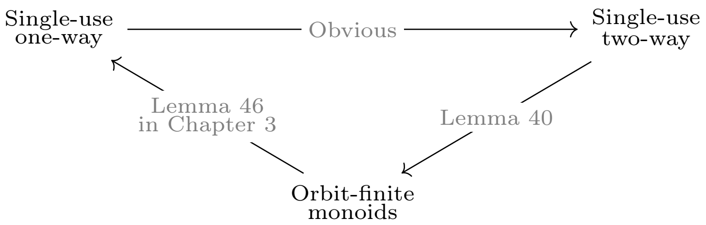
2.3.3 Single-use automata and orbit-finite monoids

The following theorem states the key property of the single-use automaton:

Theorem 6. *All the following models recognize the same class of languages:*

1. *One-way single-use automaton;*
2. *Two-way single-use automaton;*
3. *Orbit-finite monoids (limited to polynomial orbit-finite alphabets).*

We prove this theorem using the following strategy:



In this chapter we show how to translate a single-use two-way automaton into an orbit-finite monoid. In order to make the two-way construction easier to understand, we first present the construction for a one-way single-use automaton.

2.3.4 One-way single-use automata $\subseteq$ Orbit-finite monoids

This section is dedicated to proving the following lemma:

Lemma 39. *If a language L over a polynomial orbit alphabet Σ is recognized by a one-way single-use automaton, then it is also recognized by an orbit-finite monoid.*

The proof of the lemma is very similar to the classical translation of a finite automaton into a finite monoid (see Lemma 20). Take a single-use one-way automaton $\mathcal{A}$ over a polynomial orbit-finite alphabet Σ, and let Q be its polynomial orbit-finite set of states. For every word $w \in \Sigma^*$, define $\mathcal{A}$'s behaviour on w as the following function $Q \to_{\text{fs}} Q$:

$$b_w(q) = \substack{\text{In which state will } \mathcal{A} \text{ exit } w \text{ on the right,} \\ \text{if it enters } w \text{ on the left in the state } q?}$$

Note that behaviours are compositional: $b_{uv} = b_v \circ b_u$. Since single-use functions are (by definition) closed under compositions, it follows that, all possible behaviours of $\mathcal{A}$ are single-use functions. This means that the language of L is recognized by the monoid $Q \multimap Q$ with the following operation as $f \cdot g = g \circ f$, and with the following accepting set:

$$\{f \mid f(q_0) \text{ is an accepting state}\}$$

This finishes the proof of Lemma 39, because by Theorem 5, $Q \multimap Q$ is orbit-finite.

2.3.5 Two-way single-use automata $\subseteq$ Orbit-finite monoids

In this section we show how to translate a two-way single-use automaton into an orbit-finite monoid:

Lemma 40. *If a language L over a polynomial orbit alphabet Σ is recognized by a two-way single-use automaton, then it is also recognized by an orbit-finite monoid.*

In the proof we are going to use the theory of *compositional functions* (described, for example, in [Boj20, page 5]):

Definition 13. Let Σ and R be arbitrary sets (possibly infinite). We say that a function $h : \Sigma^* \to R$ is compositional if for every $u, w \in \Sigma^*$, the value $h(uw)$ is uniquely determined by the values of $h(u)$ and $h(w)$. ◁

For example, the function $f : \Sigma^* \to \mathbb{N}$ that maps every word to its length is compositional, and the function $g : \{a, b\}^* \to \{a, b, =\}$, defined as follows, is not compositional:

$$g(w) = \begin{cases} a & \text{if there are more } a\text{'s than } b\text{'s in } w \\ b & \text{if there are more } b\text{'s than } a\text{'s in } w \\ = & \text{if there is equally many } a\text{'s and } b\text{'s in } w \end{cases}$$

A compositional function $h : \Sigma^* \to R$ together with an accepting set $F \subseteq R$ can be used to recognize the language: $\{w \mid f(w) \in R\} \subseteq \Sigma^*$.

Lemma 41. *If a language $L \subseteq \Sigma^*$ is recognized by an equivariant, compositional function $\Sigma^* \to_{eq} R$, for an orbit-finite R, then L is also recognized by an orbit-finite monoid.*

Proof. Let M be the image of Σ under f, and let us define the following operation on M:

$$x \cdot y = f(uv) \quad \text{where } u, v \in \Sigma^* \text{ are such that } f(u) = x \text{ and } f(v) = y$$

Because $M = f(\Sigma)$, we know that such u and v always exist, and thanks to f's compositionality, we know that the value $x \cdot y$ does not depend on the choice of u and v. This operation is associative: for every x, y, z, if we pick some u, v, w such that $f(u) = x$, $f(v) = y$, $f(w) = z$, we have that:

$$(x \cdot y) \cdot z = f(uv) \cdot z = f((uv)w) = f(u(vw)) = x \cdot f(vw) = x \cdot (y \cdot z)$$

In a similar way, we can show that the image of the empty word ($f(\epsilon)$) is the operation's identity element. It follows that M is a monoid, and it is easy to see that M recognizes L. This leaves us with showing that M is an orbit-finite monoid. To see that M is an orbit-finite set, we notice that $M = h(R)$. This is enough, because orbit-finiteness is preserved by taking images under finitely supported functions ([Boj19, Lemma 3.24]). Finally, we show that the product operation is equivariant: for every x, y and π, if we pick some u, v such that $f(u) = x$, and $f(v) = y$, we have that:

$$\pi(x \cdot y) = \pi(f(uv)) = f(\pi(uv)) = f(\pi(u)\pi(v)) = \pi(x) \cdot \pi(y)$$

□

We are now ready to show how to translate two-way single-use automata into orbit-finite monoids. The construction is a single-use variant of [She59, Theorem 2]: Take a two-way automaton $\mathcal{A}$ over the alphabet Σ, and let Q be its polynomial orbit-finite set of states. We define $\mathcal{A}$'s behaviour on a word $w \in \Sigma^*$ to be a function:

$$b_w : \quad Q \times \{\leftarrow, \rightarrow\} \quad \longrightarrow \quad (Q \times \{\leftarrow, \rightarrow\}) + \{\text{Yes}, \text{No}\}$$

The function is analogous to the one-way behaviour, but this time $\mathcal{A}$ can enter w from the left or from the right, and it might leave w from the left or from the right. It might also never exit w, because it accepts, rejects, or starts to loop (which is considered as rejecting) inside w. Let us now show that all behaviours are single-use functions:

Claim 19. *For every $w \in \Sigma^*$, function b_w is single-use:*

$$b_w \quad \in \quad Q \times \{\leftarrow, \rightarrow\} \quad \multimap \quad (Q \times \{\leftarrow, \rightarrow\}) + \{\mathit{Yes}, \mathit{No}\}$$

Proof. We prove the claim by induction on the length of w. For the empty word, we have that $b_\epsilon = \texttt{coproj}_1$ (note that the symbol $\leftarrow$ means entering from the left and exiting from the right). Now, for the induction step let us take $w = va$. We know that b_a is equal to $\delta(a)$, which means that it is single-use. Thanks to the induction assumption, we also know that b_v is single-use. Now, let us use b_v and b_a to construct b_{va} as a single-use function. Our first (incorrect) approach is to define four mutually recursive functions, each of the type:

$$Q \quad \multimap \quad (Q \times \{\leftarrow, \to\}) + \{\text{Yes}, \text{No}\}$$

They correspond to four possible entry points to va: $(\to va)$, $(va \leftarrow)$, $(v \leftarrow a)$, and $(v \to a)$. Here are their definitions:

$$(\to va)(q) = \begin{cases} (v \to a)(q') & \text{if } b_v(q,\to)=(q',\to) \\ b_v(q, \to) & \text{otherwise} \end{cases} \qquad (va \leftarrow)(q) = \begin{cases} (v \leftarrow a)(q') & \text{if } b_a(q,\leftarrow)=(q',\leftarrow) \\ b_a(q, \leftarrow) & \text{otherwise} \end{cases}$$

$$(v \leftarrow a)(q) = \begin{cases} (v \to a)(q') & \text{if } b_v(q,\leftarrow)=(q',\to) \\ b_v(q, \leftarrow) & \text{otherwise} \end{cases} \qquad (v \to a)(q) = \begin{cases} (v \leftarrow a)(q') & \text{if } b_a(q,\to)=(q',\leftarrow) \\ b_a(q, \to) & \text{otherwise} \end{cases}$$

There are two problems with this approach: First, if $\mathcal{A}$ starts to loop in va, then this recursive definition will never terminate. Second, it is unclear if single-use functions are closed under taking fixpoints. We can deal with both of those problems by bounding the number of times that $\mathcal{A}$ can cross between v and a: Let $\alpha \subset_{\text{fin}} \mathbb{A}$ be the finite set of all atoms that appear in va (i.e. the support of va). Then, the subset $Q_\alpha \subseteq Q$ of all states supported by α is finite as well – this can be shown either by induction on the structure of Q, or by applying [Boj13, Lemma 5.2]. Now, let us notice that all the states that appear in $\mathcal{A}$'s run on va are supported by α (thanks to Lemma 3), which means that they belong to Q_α. It follows that if $\mathcal{A}$ crosses between v and a more than $2 \cdot |Q_\alpha|$ times, it will visit some state for the second time in the same position, which means that it is going to loop. In order to use this observation, we define $(\to va)_k$ to be the limited version of $(\to va)$, that only allows $\mathcal{A}$ to cross k times between v and a, before it rejects the input. And similarly, for $(va \leftarrow)_k$, $(v \leftarrow a)_k$, and $(v \to a)_k$. The definition of the functions is inductive on k. Here is the definition for$(\to va)_k$ (other definitions are analogous):

$$(\to va)_k(q) = \begin{cases} (v \to a)_{k-1}(q') & \text{if } b_v(q,\to)=(q',\to) \\ b_v(q, \to) & \text{otherwise} \end{cases}$$

For the induction base we pick $k = -1$, in which case all four functions are equal to $\texttt{const}_{\text{No}}$. Now, thanks to the if-then-else combinator from Example 25, it is not hard to use induction on k, to show that all those functions are single-use. To finish the proof of the claim, we notice that we can construct b_{va} as:

$$Q \times \{\leftarrow, \to\} \xrightarrow{\texttt{distr}} Q + Q \xrightarrow{[(va\leftarrow)_k, (\to va)_k]} Q \times \{\leftarrow, \to\} + \{\text{Yes}, \text{No}\},$$

for k equal to $2 \cdot |Q_\alpha| + 1$. □

It is easy to convince oneself that the behaviour b_{uv} depends entirely on the behaviours b_u and b_v (even though the exact formula for combining those behaviours might not be clear). Moreover, it is also easy to see that the behaviour b_w uniquely determines whether w is accepted by $\mathcal{A}$. This means that the language recognized by $\mathcal{A}$ is also recognized by the compositional function $w \mapsto f_w$, whose codomain is,

$$Q \times \{\leftarrow, \rightarrow\} \quad \multimap \quad (Q \times \{\leftarrow, \rightarrow\}) + \{\text{Yes}, \text{No}\}$$

By Theorem 5 it is orbit-finite, which means that we can finish the construction of an orbit-finite monoid, by applying Lemma 41.

2.4 Nondeterministic single-use automata

In this section, we explain why combining nondeterminsim with the single-use restriction is problematic. The key concept of nondeterminism is a relation, so in order to define a *nondeterministic single-use automaton*, it would be useful to define *single-use relations.* Unfortunately, the question of finding a well-behaved notion of a single-use relation remains open. Below, we present two possible definitions of single-use relations, and explain why they are not well behaved.

Definition 14. Let X and Y be polynomial orbit-finite sets. We say that a relation $R \subseteq X \times Y$ is *single-use*, if there is a single-use function $f : X \times Y \multimap \{\text{Yes}, \text{No}\}$ such that:

$$R = \{(x, y) \mid x \in X,\ y \in Y,\ \text{such that } f(x, y) = \text{Yes}\}.$$

We denote the set of all single-use relations between X and Y as $X \multimap_{\text{rel}} Y$. ◁

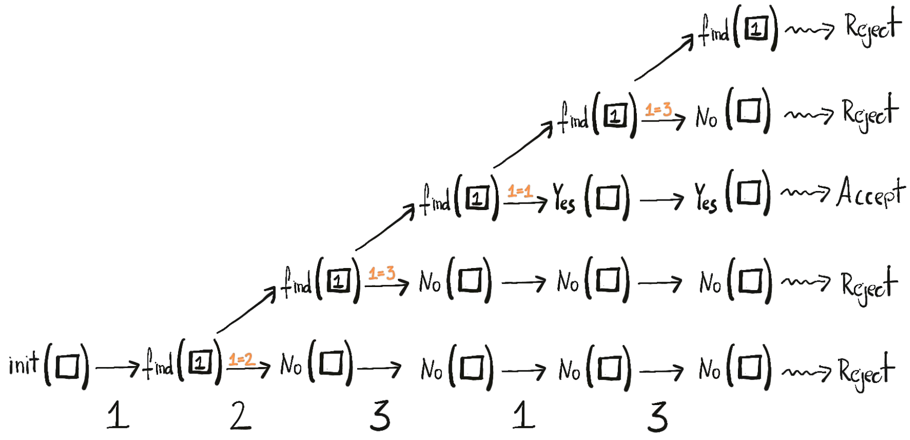
Now, we define *single-use nondeterministic automaton* to be just like the deterministic single-use automaton, except that its transition function, becomes a transition relation of the following type:

$$\Sigma \rightarrow_{\text{eq}} (Q \multimap_{\text{rel}} Q)$$

Observe that Definition 14 allows nondeterministic single-use automata to recognize the language:

“The first letter appears again”.

This is because a nondeterministic automaton can store the first letter in its state, nondeterministically guess the position where the first letter reappears, and verify its guess by comparing the two letters. Thanks to nondeterminism, this construction requires only one copy of the first letter. For example, here is a tree of all possible runs of the automaton on the word 1 2 3 1 3:


find(1) ⇝ Reject
find(1) 1=3 → No(□) ⇝ Reject
find(1) 1=1 → Yes(□) → Yes(□) ⇝ Accept
find(1) 1=3 → No(□) → No(□) → No(□) ⇝ Reject
init(□) → find(1) 1=2 → No(□) → No(□) → No(□) → No(□) ⇝ Reject
1
2
3
1
3

It follows that nondeterministic single-use automata are more expressive than orbit-finite monoids. The reason behind this is that $Q \multimap_{\mathrm{rel}} Q$ is not closed under compositions: Consider the following relation:

$$\texttt{check}_7 \in (\mathbb{A} + \top) \multimap_{\mathrm{rel}} (\mathbb{A} + \top)$$

$$\texttt{check}_7 = \{(\top, \top)\} \ \cup \ \{(a, a) \mid a \in \mathbb{A}\} \ \cup \ \{(7, \top)\}.$$

It is not hard to see that both $\texttt{check}_7$ and an analogous $\texttt{check}_8$ are single-use relations, but their composition $\texttt{check}_7 \circ \texttt{check}_8$ is not.

Our second approach to define single-use relations is based on the idea that every relation between X and Y can be expressed as a function:

$$X \to P(Y).$$

The problem, with lifting this definition to polynomial orbit-finite sets is that $P(Y)$ is neither polynomial nor orbit-finite. In order to make it at least polynomial, we can represent it as Y^*. To make it additionally orbit finite, we notice that for every function $f : X \to_{\mathrm{fs}} Y^*$, there exists a k_f, such that for all $x \in X$:

$$|f(x)| \leq k_f.$$

This is because X has only finitely many $\mathrm{supp}(f)$-orbits, and for each of those orbits the length of $f(x)$ is constant. This leads to the following (alternative) definition of a *single-use relation*:

Definition 15. Let X and Y be polynomial orbit-finite sets: We say that a relation $R \subseteq X \times Y$ is *single-use*, if there is a $k \in \mathbb{N}$, and a single-use function $f : X \multimap Y^{\leq k}$, such that:

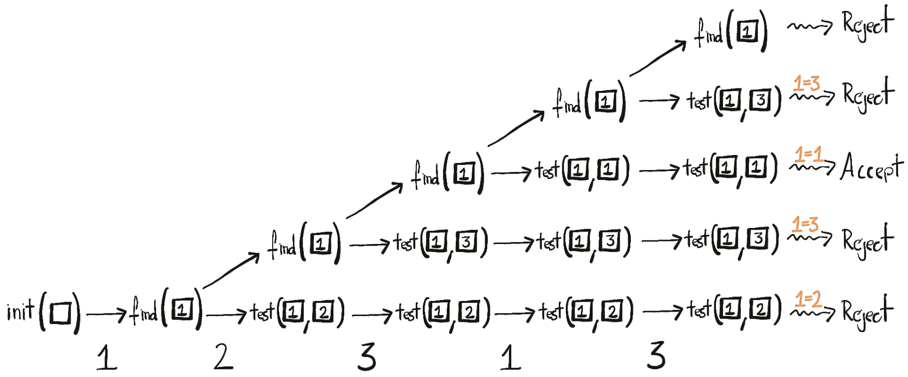
$$R = \{(x, y) \mid x \in X, \ y \in f(x)\}$$

◁

This time, it is not hard to see that single-use relations are closed under compositions. However, because k can be arbitrarily large, $Q \multimap_{\text{rel}} Q$ will usually be orbit-infinite. This can be exploited, to construct a nondeterministic single-use automaton that recognizes the language:

“The first letter appears again”.

The construction is slightly different from the previous one: This time the automaton saves the first letter in its state, then it nondeterministically picks another position, saves a second letter in its state, and keeps the two letters until the end of the word. For example, here is a tree of all possible runs of the automaton on the word 1 2 3 1 3:


find(1) ⇝ Reject
find(1) → test(1,3) 1=3 ⇝ Reject
find(1) → test(1,1) → test(1,1) 1=1 ⇝ Accept
find(1) → test(1,3) → test(1,3) → test(1,3) 1=3 ⇝ Reject
init(□) → find(1) → test(1,2) → test(1,2) → test(1,2) → test(1,2) 1=2 ⇝ Reject
1
2
3
1
3

Chapter 3

Single-use Mealy machines and their Krohn-Rhodes decompositions

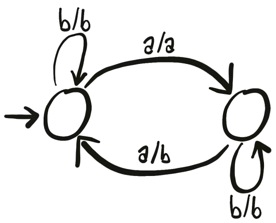
So far we have discussed languages (i.e. subsets of Σ^*) recognized by different automata models. In the next two chapters, we are going to discuss word transformations (i.e. functions $\Sigma^* \to \Gamma^*$) computed by *finite state transducers* (i.e. output-producing variants of automata). Before we introduce single-use models for infinite alphabets, let us discuss the classical theory of finite state transducers: Their classification is finer than the one of automata – two models that define the same class of languages, might define two different classes of word transformations. To illustrate this, we present three models of transducers that define three different classes of word transformations, but whose underlying automaton models recognize the same class of languages (i.e. regular languages).

1. **Mealy machine** This model (introduced in [Mea55, Section 2.1]) is a version of the deterministic register automaton, where every transition produces exactly one output letter. This is reflected in the type of transition function:

$$\underbrace{Q}_{\text{current state}} \times \underbrace{\Sigma}_{\text{input letter}} \to \underbrace{Q}_{\text{new state}} \times \underbrace{\Gamma}_{\text{output letter}}$$

   A Mealy machine produces output for every input word, which means that it does not have accepting (or rejecting) states. It follows that every Mealy machine computes a total, length-preserving function. Here is an example of a Mealy machine that recognizes the following transduction:

$$\text{“Change every other a to b”} \in \{\text{a, b}\}^* \to \{\text{a, b}\}^*$$

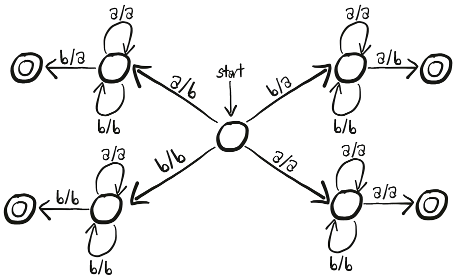


2. **Unambiguous Mealy machine** This model is a nondeterministic version of a Mealy machine. This means that the transition function becomes a transition relation:

$$\underbrace{Q}_{\text{current state}} \times \underbrace{\Sigma}_{\text{input letter}} \times \underbrace{Q}_{\text{new state}} \times \underbrace{\Gamma}_{\text{output letter}}$$

The nondeterministic version of a Mealy machine can have a number of runs on a single input word. To guarantee that it recognizes a total, function we reintroduce the accepting and rejecting states, and we require that for every input word there is exactly one accepting run (this is the unambiguity condition). Here is an example of an unambiguous Mealy machine that computes the following function:

"Swap the first and the last letters" $\in \{\text{a, b}\}^* \to \{\text{a, b}\}^*$



(Notice that the deterministic Mealy machine is not able to compute this language, as it has no way of guessing what the last letter is going to be). This class of transductions is known as *rational letter-to-letter functions.*[1] It is worth pointing out that unambiguous Mealy machines admit the

[1] The class was introduced in [Eil74]. For a more detailed bibliographical note, see the footnote in [BC18, Section 12.2].

following decomposition into two deterministic Mealy machines[2]:

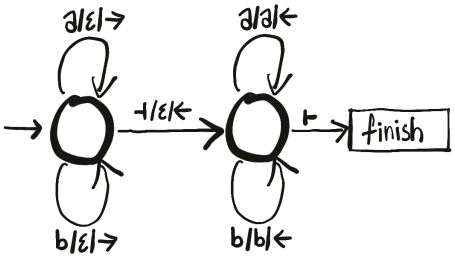
$$\begin{pmatrix}\text{unambigous}\\ \text{Mealy machine}\end{pmatrix} = \begin{pmatrix}\text{left-to-right}\\ \text{Mealy machine}\end{pmatrix} \circ \begin{pmatrix}\text{right-to-left}\\ \text{Mealy machine}\end{pmatrix}$$

3. Two-way transudcer This is a version of the two-way automaton whose every transition has an option of producing an output letter (it is a classical model discussed, for example, in [EH01]). The transition function of a two-way transducer has the following type:

$$\underbrace{Q}_{\substack{\text{current}\\ \text{state}}} \times (\underbrace{\Sigma}_{\substack{\text{current}\\ \text{letter}}} + \underbrace{\{\vdash, \dashv\}}_{\substack{\text{end of word}\\ \text{markers}}}) \longrightarrow \underbrace{Q}_{\substack{\text{new}\\ \text{state}}} \times \underbrace{\{\leftarrow, \rightarrow\}}_{\substack{\text{direction of}\\ \text{the next step}}} \times \underbrace{(\Gamma + \epsilon)}_{\substack{\text{output}\\ \text{letter}}} + \underbrace{\text{finish}}_{\substack{\text{finish}\\ \text{the run}}}$$

Here is an example of a two-way automaton that recognizes the function:

$$\text{“Reverse the input”} \subseteq \{a, b\}^* \to \{a, b\}^*$$



It can be shown that an unambiguous Mealy machine cannot compute the reverse function. Note that it is possible for a two-way automaton to loop. To guarantee that it does not loop we can add a semantic requirement that prohibits two-way automata from looping (an alternative approach would be to say that a looping run produces the empty word). The class of functions computed by the two-way transducer has many equivalent definitions, including streaming string transducers (see [AČe10, Section 3] or Section 4.1.2 in this thesis) and the logical model of MSO-transductions (see [Cou94, Section 2]). This class of transductions is known as the *regular functions.*

The three transducer models are well behaved: For example, they are closed under composition and their equivalence problem (as defined below) is decidable.

Input: Two transducers $\mathcal{A}$ and $\mathcal{B}$.
Output: Do $\mathcal{A}$ and $\mathcal{B}$ compute the same function?

[2] This is known as the Elgot-Mezei theorem. It was originally shown in [EM63, Theorem 7.8]. Since I was not able to access the full version of the original paper, I relied on [BC18, Theorem 12.1]. It is worth pointing out that the two papers prove the theorem for slightly different models (functional Mealy machines and unambiguous NFAs with output), but the proof from [BC18, Theorem 12.1] can be easily adapted to work with unambiguous Mealy machines.

In the next two chapters, we are going to use the single-use restriction to develop a similar theory for infinite alphabets. Chapter 3 covers single-use (deterministic) Mealy machines and Chapter 4 covers single-use two-way transducers. The theory of single-use unambiguous Mealy machines is still a work in progress, but it is briefly discussed in Chapter 4.

Finally, let us briefly mention the class of *polyregular functions*: This is a class over finite alphabets that extends *regular functions* while keeping many of their desirable properties (see [Boj22]). Unfortunately, the question of finding a notion of *single-use polyregular functions* that is a well-behaved class of functions remains open, so we do not discuss it in this thesis. However, we would like to note that this could be an interesting direction for further research.

3.1 Single-use mealy machines

Single-use Mealy machines are a transducer model that computes length-preserving functions $\Sigma^* \to \Gamma^*$. Here is its definition:

Definition 16. A *single-use Mealy machine* consists of:

1. a polynomial orbit-finite input alphabet Σ and a polynomial orbit-finite output alphabet Γ;
2. a polynomial orbit-finite set of states Q;
3. an initial state $q_0 \in Q$;
4. a single-use transition function:

$$\delta : \underbrace{\Sigma}_{\text{current letter}} \longrightarrow_{\text{eq}} \left(\underbrace{Q}_{\text{current state}} \multimap \left(\underbrace{Q}_{\text{new state}} \times \underbrace{\Gamma}_{\text{output letter}} \right) \right)$$

◁

Notice that if Σ, Γ and Q are finite (and not only orbit-finite), then this definition matches the classical definition of a Mealy machine – this follows from Example 24. Let us now consider some examples of Mealy machines:

Example 28 (Length-preserving single-use homomorphism)**.** For any $h : \Sigma \to_{\text{eq}} \Gamma$ (where Σ and Γ are polynomial orbit-finite sets), we define $h^* : \Sigma^* \to \Gamma^*$ to be the length-preserving transduction that applies h to every input letter. This h^* is can be computed by a one-state[3] single-use Mealy machine, with the following transition function:

$$\delta(a)(1) = (1, h(a))$$

◁

[3] in [Mea55] one-state machines are called *combinatorial circuits*

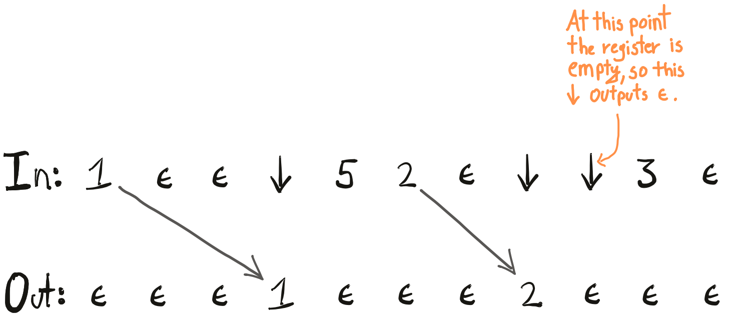
Example 29 (Single-use atom propagation)**.** *Single-use atom propagation* simulates single-use operations on one register. Its input alphabet is a set of instructions:

$$\underbrace{\mathbb{A}}_{\text{store an atom}} + \underbrace{\downarrow}_{\substack{\text{output the atom from the register}\\ \text{and destroy it}}} + \underbrace{\epsilon}_{\text{do nothing}}$$

Its output alphabet is:

$$\underbrace{\mathbb{A}}_{\text{outputs of }\downarrow} + \underbrace{\epsilon}_{\text{empty output}}$$

Here is an example input and output of the function (the grey arrows are only informative – they are not part of the input or output):


At this point the register is empty, so this ↓ outputs ϵ.
In: 1 ϵ ϵ ↓ 5 2 ϵ ↓ ↓ 3 ϵ
Out: ϵ ϵ ϵ 1 ϵ ϵ ϵ 2 ϵ ϵ ϵ

The semantic is rather intuitive, but to avoid confusion, we also define it formally: the i-th output letter is equal to $a \in \mathbb{A}$, if (a) i-th input letter is equal to $\downarrow$, and (b) there is $j < i$ such that the j-th input letter is equal to a and every input letter between i and j is equal to ϵ. Otherwise, the i-th input letter is equal to ϵ. Single-use atom propagation can be computed by a single-use Mealy machine: its states are $\mathbb{A} + \epsilon$ and its transition function is as follows:

$$\delta(l, \downarrow) = (\bot, l) \quad \delta(l, \epsilon) = (l, \epsilon) \quad \delta(l, a \in \Sigma) = (a, \epsilon)$$

◁

Notice that since the transition function of a Mealy machine is single-use, it has to forget every atom that it outputs. For this reason, the multiple-use version of Example 29 cannot be computed by a single-use Mealy machine. In order to prove this, we use a simple quantitative reasoning:

Definition 17. For every element x of a polynomial orbit-finite set X, we define the *multi-support* of x (denoted $\mathrm{msup}(x)$) to be the *multiset* of all atoms that appear in x, with repetitions. This definition can also be extended to work on words over polynomial orbit-finite sets. For example (if we take $X = \mathbb{A}^3$):

$$\mathrm{msup}(3, 2, 3) = \{2, 3, 3\}$$

◁

Lemma 42. *If f is a function computed by a single-use Mealy machine, then there exists a $k \in \mathbb{N}$, such that for every word w and every atom a:*

$$\begin{pmatrix}\textit{The number of times } a \\ \textit{appears in } f(w)\end{pmatrix} \leq k \cdot \begin{pmatrix}\textit{The number of times } a \\ \textit{appears in } w\end{pmatrix}$$

Proof. The lemma follows from Lemma 37 combined with the following claim (which can be easily shown using the tree representation):

Claim 20. *Let $f \in X \multimap_{eq} Y$, and for every $x \in X$:*

$$\mathit{msup}(f(x)) \subseteq \mathit{msup}(x)$$

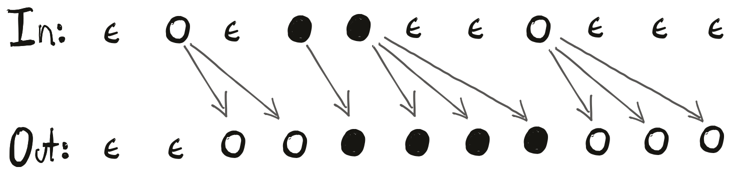
□

It is not hard to see that the multiple-use version of Example 29 does not satisfy the condition from Lemma 42, so it cannot be computed by a single-use Mealy machine. The following example shows that the single-use restriction does not apply to the finite information:

Example 30 (Multiple-use bit propagation)**.** The *multiple-use bit propagation* function can be seen as the classical variant of the single-use atom propagation from Example 29. It simulates operations on one multiple-use register that stores one bit of information, represented as one of two values – (● or ○). Its input alphabet is the following set of instructions:

$$\{ \underbrace{\bullet}_{\substack{\text{output the register value} \\ \text{and save } \bullet \text{ to the register}}}, \underbrace{\circ}_{\substack{\text{output the register value} \\ \text{and save } \circ \text{ to the register}}}, \underbrace{\epsilon}_{\substack{\text{output the register value} \\ \text{and keep its contents}}} \}$$

The output alphabet is the same: $\{\circ, \bullet, \epsilon\}$. (Value ϵ denotes empty register – the register cannot be emptied, but being empty is its initial value.) Here is an example input and output (again, the grey arrows are only informative – they are not part of the input or output):



The Mealy machine recognizing the bit propagation has three states: $Q = \{\circ, \bullet, \epsilon\}$. Its transition function is as follows:

$$\delta(q, \circ) = (\circ, q) \quad \delta(q, \bullet) = (\bullet, q) \quad \delta(q, \epsilon) = (q, q)$$

(To see that this is a single-use function, we notice that Q is finite and apply Example 24.) ◁

Here is an example of a more general class of functions recognized by Mealy machines:

Example 31 (Monoid prefixes)**.** For every finite monoid M, define the *M-prefix function* $M^* \to M^*$ to be the function that computes products of the input prefixes:

$$\begin{array}{ccccc} s_1 & s_2 & s_3 & \ldots & s_n \\ & & \mapsdown & & \\ s_1 & s_1 s_2 & s_1 s_2 s_3 & \ldots & s_1 s_2 \ldots s_n \end{array}$$

In particular, if we take a monoid $P = \{\bigcirc, \bullet, \epsilon\}$ whose multiplication is given by the following Cayley table, then the P-prefix function is the multiple-use bit propagation, but with the results shifted one position to the right:

$\cdot_P$	ϵ	○	●
ϵ	ϵ	○	●
○	○	○	●
●	●	○	●

It is not hard to see that for every finite M, the M-prefix function can be computed by a single-use Mealy machine, where $Q = M$, $q_0 = 1_M$ and whose transition function is defined as:

$$\delta(p, g) = (p \cdot g, p \cdot g)$$

(Again, Q is finite, so thanks to Example 24 we know that δ is single-use.) $\triangleleft$

It is worth pointing out that if M polynomial orbit-finite (and not finite), then the M-prefix function will not necessarily be computable by a single-use Mealy machine:

Example 32. Take $M = \mathbb{A}+1$ such that 1 is the identity element, and otherwise the operation is defined as follows:

$$a \cdot b = a$$

If a sequence $a_1, \ldots, a_n \in \mathbb{A}^*$ consists of n different atoms, then the M-prefix function looks as follows:

$$\begin{array}{ccccc} a_1 & a_2 & a_3 & \ldots & a_n \\ & & \mapsdown & & \\ a_1 & a_1 & a_1 & \ldots & a_1 \end{array}$$

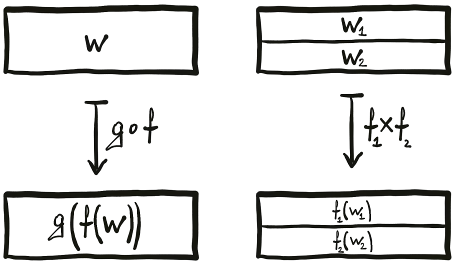
This function violates the condition from Lemma 42, which means that it cannot be computed by a single-use Mealy machine. $\triangleleft$

3.2 Krohn-Rhodes decomposition

The Krohn-Rhodes theorem [KR65, Equation 2.2] states that every function computed by a classical Mealy machine can be decomposed into certain prime function. It uses two types of composition:

$$\frac{\Sigma^* \xrightarrow{f} \Gamma^* \quad \Gamma^* \xrightarrow{g} \Delta^*}{\Sigma^* \xrightarrow{g\circ f} \Delta^*} \text{ sequential} \qquad \frac{\Sigma_1^* \xrightarrow{f_1} \Gamma_1^* \quad \Sigma_2^* \xrightarrow{f_2} \Gamma_2^*}{(\Sigma_1 \times \Sigma_2)^* \xrightarrow{f_1 \times f_2} (\Gamma_1 \times \Gamma_2)^*} \text{ parallel}$$

The sequential composition is the usual function composition, and the parallel composition, which only makes sense for length-preserving functions, applies one function to the first coordinate of the word, and the other function to the second coordinate of the word. Here is a schematic depiction of the two compositions:



Theorem 7 (Krohn-Rhodes)**.** *The class of functions computed by Mealy machines over finite alphabets is the equal to the smallest class of function that is closed under sequential and parallel compositions, and which contains the following* (classical) prime functions*:*

1. *the h^* function from Example 28, for every $h : \Sigma \to \Gamma$ (such that Σ and Γ are finite);*

2. *the multiple-use bit propagation function from Example 30;*

3. *the G-prefix function from Example 31, for every finite group G (note that a group is a special case of a monoid).*

The following theorem (proved by Bojańczyk and me in [BS20, Theorem 9]) shows that single-use Mealy machines for infinite alphabets admit a similar decomposition:

Theorem 8. *The class of functions computed by single-use Mealy machines (over polynomial orbit-finite alphabets) is the equal to the smallest class of functions that is closed under sequential and parallel compositions, and which contains the following* single-use prime functions*:*

1. *all functions recognized by Mealy machines over finite alphabets;*

2. *h^* for every equivariant $h : \Sigma \to_{eq} \Gamma$ from example 28*

3. *single-use propagation from Example 29*

(Note that thanks to Theorem 7 the first item can be further decomposed into classical prime functions.)

I would like to use this thesis to present a new and (hopefully) improved proof of Theorem 8. The new proof is also a joint work with Bojańczyk.

3.2.1 Compositions of primes $\subseteq$ Single-use Mealy machines

Translating compositions of single-use primes into single-use Mealy machines is the easy part – we have already seen that all prime functions can be computed by single-use Mealy machines, so it suffices to show that single-use Mealy machines are closed under both types of composition:

Lemma 43. *Single-use Mealy machines are closed under parallel composition.*

Proof. The proof is straightforward – the machine $\mathcal{A} \times \mathcal{B}$ can be constructed as a simple product construction: it keeps one copy of $\mathcal{A}$, one copy of $\mathcal{B}$ and whenever it receives a new letter $(a, b) \in \Sigma_{\mathcal{A}} \times \Sigma_{\mathcal{B}}$ it feeds a to $\mathcal{A}$ and b to $\mathcal{B}$, updates their states and outputs their outputs (as a pair). □

Lemma 44. *Single-use Mealy machines are closed under sequential compositions.*

Proof. We take any two single-use Mealy machines $\mathcal{A} : \Sigma^* \to \Delta^*$ and $\mathcal{B} : \Delta^* \to \Gamma^*$, and we construct a single-use Mealy machine $\mathcal{B} \circ \mathcal{A} : \Sigma^* \to \Gamma^*$. Let us start by restating the construction for classical (i.e. atomless) Mealy machines: the machine $\mathcal{B} \circ \mathcal{A}$ keeps a copy of $\mathcal{A}$ and a copy of $\mathcal{B}$. When it reads a new letter $a \in \Sigma$, it:

1. feeds $a \in \Sigma$ to $\mathcal{A}$;
2. updates $\mathcal{A}$'s state;
3. feeds $\mathcal{A}'s$ output letter to $\mathcal{B}$;
4. updates $\mathcal{B}$'s state;
5. outputs $\mathcal{B}$'s output letter.

The problem with using the same construction for single-use Mealy machines is that $\mathcal{A}$ produces only one copy of the output, whereas $\mathcal{B}$ might require a multiple-use access to its input. This is because the first arrow in the type of $\delta_{\mathcal{B}}$ is equivariant but not necessarily single-use:

$$\delta_{\mathcal{B}} : \Delta \to_{\text{eq}} (Q_{\mathcal{B}} \multimap (Q_{\mathcal{B}} \times \Gamma))$$

To deal with this problem, we use a similar reasoning as in the proof of Lemma 37, and show that that there is a $k \in \mathbb{N}$ such that $\delta_{\mathcal{B}}$ can be represented as:

$$\delta'_{\mathcal{B}} : \Delta^k \times Q_{\mathcal{B}} \multimap (Q_{\mathcal{B}} \times \Gamma)$$

It follows that $\mathcal{B}$ requires only k copies of its input (for some fixed number k). This means that we can repeat the classical construction for $\mathcal{B} \circ \mathcal{A}$, but we have to maintain one copy of $\mathcal{B}$ and k identical copies of $\mathcal{A}$. □

3.3 An algebraic model for single-use Mealy machines

In the proof of the remaining inclusion of Theorem 8, we would like to use the algebraic theory of single-use machines developed in Chapter 2. For this purpose, we will define an algebraic transducer model that is equivalent to single-use Mealy machines. This model is going to be based on *semigroups* (a version of monoids without the requirement of having an identity element). This shift from monoids is justified by Claim 22 (stated later in this section), which demonstrates that the identity elements would cause some technical problems in the theory of orbit-finite transducers. However, it is worth noting that semigroups and monoids are very similar algebraic structures. Trivially, every monoid is a semigroup, and, as shown by the following claim, every semigroup can be embedded into a monoid:

Claim 21. *For every semigroup S, there exists a monoid S^1 such that S is a subsemigroup of S^1.*

Proof. If S already happens to contain an identity element, we can set $S^1 = S$. Otherwise, we need to adjoin a formal identity element to S. This means that $S^1 = S + 1$, with the operation defined as follows (for all x and y from the original S):

$$x \cdot y = x \cdot_S y \quad 1 \cdot x = x \quad x \cdot 1 = x \quad 1 \cdot 1 = 1$$

(Observe that we could also unconditionally adjoin a formal identity element to S. This operation is usually denoted as S^I.) □

3.3.1 Semigroup transductions over finite alphabets

Before discussing the algebraic transducer model for orbit-finite alphabets, let us briefly discuss its classical version for finite alphabets. We start with a definition[4]:

Definition 18. A *semigroup transduction* of type $\Sigma^* \to \Gamma^*$ consists of

1. a finite semigroup S;
2. an input function $h : \Sigma \to S$; and
3. an output function $\lambda : S \to \Sigma$.

The semigroup transduction defines the following function $\Sigma^* \to \Gamma^*$:

$$\Sigma^* \xrightarrow{h^*} S^* \xrightarrow{S\text{-prefix function}} S^* \xrightarrow{\lambda^*} \Gamma^*,$$

[4] Although I was not able to find this definition in the literature, it is consistent with the commonly understood folklore in the field.

where the S-prefix function is defined in Example 31, and h^* and λ^* are length-preserving homomorphisms defined in Example 28. In other words, this means that a word $w_1 w_2 w_3 \ldots w_n \in \Sigma^*$, is transformed into the following word from Γ^*:

$$\lambda\left(h(w_1)\right) \quad \lambda\left(h(w_1) \cdot h(w_2)\right) \quad \ldots \quad \lambda\left(h(w_1) \cdot \ldots \cdot h(w_n)\right)$$

◁

As expected, this class of transductions is equivalent to finite Mealy machines[5]:

Lemma 45. *The class of functions computed by Mealy machines over finite alphabets is equivalent to the class of functions computed by semigroup transductions.*

Proof. $\supseteq$: Thanks to Examples 31 and 28, we know that the S-prefix function, h^* and λ^* can be computed by a classical Mealy machine. This finishes the proof, because by Lemma 44, Mealy machines are closed under compositions.

$\subseteq$: Let us take a classical Mealy machine $\mathcal{A}$, and let us construct an equivalent semigroup transduction. First, let us define the behaviour of a word $w \in \Sigma^*$ to be the following function $b_{\mathcal{A}}(w) : Q \to (Q \times \Gamma)$:

$$b_{\mathcal{A}}(w)(q) = (q', c) \quad \overset{\text{def}}{\Leftrightarrow} \quad \begin{array}{c}\text{If } \mathcal{A} \text{ enters } w \text{ from the left in the state } q, \\ \text{it exits } w \text{ from right in state } q', \text{ outputting the letter } c \in \Gamma.\end{array}$$

Similarly as it was in the case of finite automata, the set of all possible behaviours forms a semigroup. Its operation is defined as follows:

$$(f \cdot g) = g \circ \underbrace{\texttt{proj}_1}_{\substack{\text{projection} \\ Q \times \Gamma \to Q}} \circ f$$

It is not hard to see that this finite semigroup, together with the following h and λ, forms a semigroup transduction that is equivalent to $\mathcal{A}$:

$$h(a) = b_{\mathcal{A}}(a) \quad \lambda(f) = \texttt{proj}_2(f(q_0))$$

□

3.3.2 Semigroup transductions over orbit-finite alphabets

For infinite alphabets, things get more complicated. Even though orbit-finite monoids are equivalent to single-use automata, the orbit-finite semigroup transductions are stronger than single-use Mealy machines. We have already seen that in Example 32, but the problem persists even if Γ is finite:

[5] Again, this result seems to be a part of the field's folklore. Similar reasoning can be found in the literature – e.g. in [KR65, Section 4].

Example 33. Consider the function $f_{\text{cmp}} : \mathbb{A}^* \to \{=, \neq\}$ which replaces every atom that is equal to the first letter with $=$ and all other atoms with $\neq$. Here is an example:

$$\begin{array}{ccccccc} 1 & 2 & 1 & 5 & 6 & 1 & 1 \\ & & & \downarrow & & & \\ = & \neq & = & \neq & \neq & = & = \end{array}$$

This function is a semigroup transduction. We can implement it using $S = \mathbb{A}^2$ with the semigroup operation defined as follows

$$(a, b) \cdot (c, d) = (a, d),$$

and with the following λ and h:

$$h(a) = (a, a) \qquad \lambda(a, b) = \begin{cases} = & \text{if } a = b \\ \neq & \text{if } a \neq b \end{cases}$$

Let us now show that this function cannot be computed by a single-use Mealy machine. If we consider languages as functions $\Sigma^* \to \{\text{Yes}, \text{No}\}$, we can write that:

$$\binom{\text{The first letter}}{\text{appears again}} = \binom{\text{The letter = appears}}{\text{at least twice}} \circ f_{\text{first}}$$

By reasoning analogous to the one presented in the proof of Lemma 44, we obtain that:

$$\binom{\text{Single-use register}}{\text{automata}} \circ \binom{\text{Single-use}}{\text{Mealy machines}} = \binom{\text{Single-use register}}{\text{automata}}$$

The language "the letter '=' appears at least twice" is a regular (over a finite alphabet). Therefore, if f_{cmp} were recognized by a single-use Mealy machine, then the language "the first letter appears again" would also be recognized by a single-use Mealy machine. However, as shown by Example 6 and Lemma 39, this is not true. Therefore, f_{cmp} is not recognized by single-use Mealy machines. $\triangleleft$

It turns out that whether an orbit-finite semigroup transduction is equivalent to a single-use Mealy machine depends solely on the output function λ. As we will show, it depends on whether λ satisfies the following *locality equation* (which, to the best of my knowledge is an original contribution of this thesis, based on a joint work with Bojańczyk):

Definition 19. Let S be an orbit-finite semigroup and let Γ be an orbit-finite set. We say that a function $\lambda : S \to_{\text{eq}} \Gamma$ is *local*, if for every $a, e, b \in S$, such that e is an *idempotent* (i.e. $ee = e$) and b is a prefix of e (i.e. $e = bb'$ for some b' in S) and for every supp(e)-permutation π (i.e. a permutation π such that $\pi(a) = a$, for every $a \in \text{supp}(e)$), it holds that:

$$\lambda(aeb) = \lambda(\pi(a)eb).$$

We further say that a semigroup transduction (S, h, λ) is *local*, if its output function λ is local. In this thesis, we focus on local semigroup transductions that are also equivariant and orbit-finite. So, to simplify the notation, we additionally require that in a local semigroup transduction the components S, h and λ are equivariant and S is orbit-finite. ◁

To avoid interrupting the proof of Theorem 8 (which is already very long), we defer some of the discussion about single-use Mealy machines to Section 3.6. For now we only present some intuition and an example in the next couple of paragraphs. However, in order to provide context, it is worth mentioning that, later in this thesis we will show that local semigroup transductions recognize the same class of transductions as single-use Mealy machines (see Lemma 9). Furthermore, we will show that as long as S does not contain unreachable elements, a semigroup transduction (S, h, λ) is equivalent to a single-use Mealy machine if and only if λ is local (see Lemma 71 in Section 3.6).

Let us now offer some informal intuition behind the locality restriction: Consider a nonlocal semigroup transduction (S, h, λ). This implies that there exist $a, e, b \in S$ and a $\text{supp}(e)$-permutation π such that e is an idempotent and b is a prefix of e, for which the locality equation does not hold:

$$\lambda(aeb) \neq \lambda(\pi(a)eb)$$

Since b is a prefix of e, there exists $c \in S$ such that $bc = e$. Consider the following sequence:

$$a\underbrace{bc}_{e}\underbrace{bc}_{e}\cdots\underbrace{bc}_{e}$$

Observe that such a sequence contains arbitrarily many prefixes that evaluate to aeb. We know that $\lambda(aeb) \neq \lambda(\pi(a)eb)$, which means that the value of $\lambda(aeb)$ depends on an atom from $(\text{supp}(a) - \text{supp}(e))$. This means that every Mealy machine computing this semigroup transduction would have to use at least one copy of an atom from a while processing each bc part of the input. Since a appears only once, and the input sequence can be arbitrarily long, this would violate the single-use restriction.

Example 34. Let us construct a local semigroup transduction (S, λ, h) that computes the single-use atom propagation from Example 29. The semigroup S is defined as follows:

$$\underbrace{\epsilon}_{\text{do nothing}} + \underbrace{\mathbb{A}}_{\substack{\text{save an atom } a\\ \text{into the register}}} + \underbrace{\perp}_{\text{empty the register}} + \underbrace{\downarrow}_{\substack{\text{output and empty}\\ \text{the register}}} + \underbrace{\mathbb{A}\downarrow}_{\substack{\text{output } a \in \mathbb{A} \text{ and}\\ \text{empty the register}\\ (\text{denoted as } a\downarrow)}}$$

The operation in S is defined by the following table, for every $a, b \in \mathbb{A}$ (the general rule is that $x \cdot \epsilon = x = \epsilon \cdot x$ for every $x \in S$, and that $x \cdot y = y$ for every

$x, y \in S$, such that $y \neq \epsilon$. All exceptions to this rule are marked in blue):

$$\begin{array}{ccccc}
\epsilon \cdot \epsilon = \epsilon & \epsilon \cdot a = a & \epsilon \cdot \bot = \bot & \epsilon \cdot \downarrow = \downarrow & \epsilon \cdot a \downarrow = a \downarrow \\
a \cdot \epsilon = a & a \cdot b = b & a \cdot \bot = \bot & a \cdot \downarrow = a \downarrow & a \cdot b \downarrow = b \downarrow \\
\bot \cdot \epsilon = \bot & \bot \cdot a = a & \bot \cdot \bot = \bot & \bot \cdot \downarrow = \bot & \bot \cdot a \downarrow = a \downarrow \\
\downarrow \cdot \epsilon = \bot & \downarrow \cdot a = a & \downarrow \cdot \bot = \bot & \downarrow \cdot \downarrow = \bot & \downarrow \cdot a \downarrow = a \downarrow \\
a \downarrow \cdot \epsilon = \bot & a \downarrow \cdot b = b & a \downarrow \cdot \bot = \bot & a \downarrow \cdot \downarrow = \bot & a \downarrow \cdot b \downarrow = b \downarrow
\end{array}$$

Note that $\Sigma = \mathbb{A} + \downarrow + \epsilon$, which means that $\Sigma \subseteq S$, so we can define h to be the natural injection. Finally, we define λ as follows:

$$\lambda(x) = \begin{cases} a & \text{if } x = a \downarrow \\ \epsilon & \text{otherwise} \end{cases}$$

This semigroup transduction (S, h, λ) defines the single-use atom propagation function from Example 29. Let us now show that it satisfies the locality equation. We take $x, e, y \in S$, and a supp(e)-permutation π, such that e is an idempotent and y are a prefix of e, and we show that $\lambda(xey) = \lambda(\pi(x)ey)$. First, let us notice that unless $y = \downarrow$ or $y = a \downarrow$, we know that $\lambda(xey) = \epsilon = \lambda(\pi(x)ey)$. Moreover, if $y = a \downarrow$ we know that:

$$xey = a \downarrow = \pi(x)ey.$$

It follows that $\lambda(xey) = a = \lambda(\pi(x)ey)$. This leaves us with the case where $y = \downarrow$. We need to show that:

$$\lambda(xe \downarrow) = \lambda(\pi(x)e \downarrow)$$

Observe that e cannot be equal to $\downarrow$, because $\downarrow$ is not idempotent. Moreover e also cannot be equal to ϵ, because $\downarrow$ is not a prefix of ϵ. This means that e is either equal to a, $a \downarrow$ or $\bot$. It follows that $z \cdot e = e$, for every $z \in S$. This means that $xe = e = \pi(x)e$, which in turn means that $\lambda(xey) = \lambda(\pi(x)ey)$. ◁

Finally, let us mention that the locality restriction is very limiting if the underlying semigroup happens to be a monoid – this is the reason why we use semigroup-based models for algebraic transductions rather than monoid-based models:

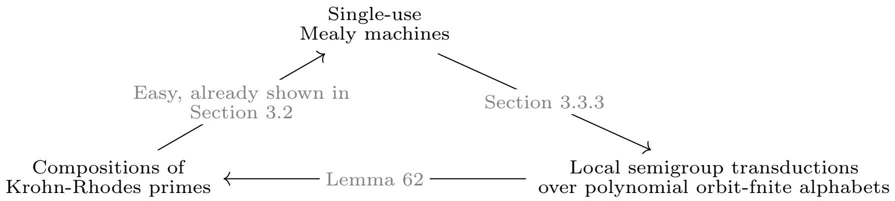
Claim 22. *Let (S, h, λ) be a local semigroup transduction. If S contains an identity element, then λ can only output equivariant (i.e. atomless) values.*

Proof. Every semigroup S contains at most one identity element (see [Pin10, Section II.1.1] for details). This means that, if it exists, the identity element of S can be computed from S in an equivariant way. Since the semigroup S is equivariant as a whole, it follows from Lemma 3 that the identity element $1 \in S$ has to be equivariant as well. This means that every atom permutation π is a supp(1)-permutation. Observe that 1 is idempotent and that it is its own prefix. Since λ is equivariant, it follows that for every $x \in S$:

$$\pi(\lambda(x)) = \lambda(\pi(x)) = \lambda(\pi(x) \cdot 1 \cdot 1) \overset{\text{locality}}{=} \lambda(x \cdot 1 \cdot 1) = \lambda(x)$$

This means that for every $x \in S$, the output value $\lambda(x)$ is equivariant (i.e. atomless). □

As mentioned before, we are going to use local semigroup transductions in the proof of Theorem 8. Here is the plan:



The hardest part of the proof is Lemma 62 (translating local semigroup transductions into compositions of primes). For this reason, we devote a significant portion of this chapter to explain it.

Notice, that as a byproduct of the proof strategy, we will obtain the following theorem:

Theorem 9. *Local semigroup transductions over polynomial orbit-finite alphabets*[6] *compute the same class of functions as single-use Mealy machines*

Before we proceed with the proof of Theorem 8, let us show how to use it to prove the missing implication from Theorem 6:

Lemma 46. *Every language that can be recognized by an orbit-finite monoid can also be recognized by a one-way single-use automaton.*

Proof. Let $L \subseteq \Sigma^*$ be recognized by an orbit-finite monoid M. We define a transduction $f_L : (\Sigma + \dashv)^* \to \{\epsilon, \texttt{Yes}, \texttt{No}\}^*$ such that the i-th letter of $f_L(w)$ is equal to:

- `Yes` if w_i the first $\dashv$ in w and $w_1, w_2, \ldots, w_{i-1} \in L$;
- `No` if w_i the first $\dashv$ in w and $w_1, w_2, \ldots, w_{i-1} \notin L$;
- ϵ if w_i is not the first $\dashv$ in w.

Since L is recognized by M, we can use the following semigroup $M^{\dashv}$ to define f_L as a local semigroup monoid transduction.

$$M^{\dashv} = \underbrace{M}_{\text{words without } \dashv} + \underbrace{M \dashv}_{\substack{\text{words that end with } \dashv \\ \text{and otherwise do not contain } \dashv}} + \underbrace{\bot}_{\text{all other words}}$$

[6] Notice that semigroup transductions work with all orbit-finite Γ and Σ (even if they are not polynomial). This general case could be an interesting topic for future work. See also Footnote 21 on page 155.

The operation on $M^{\dashv}$ is defined as follows for every $a, b \in M$ (note that there are only two cases where the result is not $\bot$):

$$\begin{array}{lll}
a \cdot b = a \cdot_M b & (a \dashv) \cdot b = \bot & \bot \cdot a = \bot \\
a \cdot (b \dashv) = (a \cdot_M b) \dashv & (a \dashv) \cdot (b \dashv) = \bot & \bot \cdot (a \dashv) = \bot \\
a \cdot \bot = \bot & (a \dashv) \cdot \bot = \bot & \bot \cdot \bot = \bot
\end{array}$$

The h function is the same as the one used to recognize L, and λ is defined as follows:

$$\lambda(a) = \epsilon \quad \lambda(a \dashv) = \begin{cases} \texttt{Yes} & \text{if } s \text{ is accepting} \\ \texttt{No} & \text{otheriwise} \end{cases} \quad \lambda(\bot) = \epsilon$$

It is easy to see that $(M^{\dashv}, h, \lambda)$ recognize f_L. Let us show that λ satisfies the locality equation: Take x, e, y and a supp(e)-permutation π such that e is an idempotent and b is a prefix of e. The only interesting case is where $y = a \dashv$, or otherwise $\lambda(xey) = \square = \lambda(\pi(x)ey)$. The only idempotent that contains $a \dashv$ as a prefix is $\bot$, which means that $e = \bot$. It follows that $aeb = \bot = \pi(a)eb$, which in particular means that $\lambda(aeb) = \lambda(\pi(a)eb)$.

It follows by Lemma 9, that there exists a single-use Mealy machine $\mathcal{A}$ that computes f_L. If we ignore the output of $\mathcal{A}$ and equip it with the following acceptance function, we obtain a single-use automaton for the language L.

$$f_{\text{acc}}(q) = \begin{cases} \texttt{Yes} & \text{if the output letter of } \delta(q, \dashv) \text{ is } \texttt{Yes} \\ \texttt{No} & \text{otherwise} \end{cases}$$

□

3.3.3 Single-use Mealy machines $\subseteq$ Local semigroup transductions

In this section, we show how to translate a single-use Mealy machine into a local semigroup transduction. The construction is a single-use version of the classical construction presented in the $\subseteq$-inclusion of Lemma 45. Given a single-use Mealy machine $\mathcal{A}$ of type $\Sigma^* \to \Gamma^*$, we can translate every non-empty word $w \in \Sigma^*$ into a behaviour $b(w)$, which is of the following type:

$$\underbrace{Q}_{\substack{\text{The state in} \\ \text{which } \mathcal{A} \text{ enters} \\ w \text{ from the left.}}} \multimap \underbrace{Q}_{\substack{\text{The state in} \\ \text{which } \mathcal{A} \text{ exits} \\ w \text{ from the right.}}} \times \underbrace{\Gamma}_{\substack{\text{The letter outputted} \\ \text{by } \mathcal{A} \text{ as it exits} \\ w \text{ from the right.}}}$$

To see that every behaviour is a single-use function, we notice that the behaviours of one-letter words as single-use functions (because the transition functions of Mealy machines are single-use), and that the behaviours can be composed according to the following formula:

$$b_{\mathcal{A}}(w_1 w_2) = b_{\mathcal{A}}(w_2) \circ \texttt{proj}_1 \circ b_{\mathcal{A}}(w_1)$$

Similarly as in the proof of Lemma 45, it is not hard to see that $\mathcal{A}$ is equivalent to the following semigroup transduction:

$$S = Q \multimap Q \times \Gamma \quad h(w) = b_{\mathcal{A}}(w) \quad \lambda(f) = \texttt{proj}_2(f(q_0)),$$

where q_0 is the initial state of $\mathcal{A}$. This leaves us with showing that λ is local.

We take $x, e, y \in S$ and a supp(e)-permutation such that e is an idempotent and y is a prefix of e, and we show that they satisfy the locality equation, i.e.:

$$\lambda(xey) = \lambda(\pi(x)ey)$$

By definition of λ this is equivalent to showing that:

$$\texttt{proj}_2(q_0\, xey) = \texttt{proj}_2(q_0\, \pi(x)ey),$$

where $q_0\, xey$ is a notation for $(xey)(q_0)$ – remember that q_0 is a function. Denote $q := \texttt{proj}_1(q_0 xe)$, and notice that since π is a supp(e)-permutation and since q_0 is equivariant, we know that $\pi(q) = \texttt{proj}_1(q_0\pi(x)e)$. This means that we need to show that:

$$\texttt{proj}_2(q\, y) = \texttt{proj}_2(\pi(q)\, y)$$

Let us pick some single-use decision tree T, that represents the function y, and let us consider the trees $\texttt{proj}_1 \circ T$ and $\texttt{proj}_2 \circ T$ (obtained using the construction from Claim 17). We are going to show that the leaf of $\texttt{proj}_1 \circ T$, that is reached when computing $(\texttt{proj}_1 \circ T)(q)$, contains all the input variables x_i, such that $\pi(x_i) \neq x_i$. This is enough to prove that $\texttt{proj}_2(q\, y) = \texttt{proj}_2(\pi(q)\, y)$, because thanks to the single-use restriction, we know that if all variables that are modified by π appear in the leaf of $\texttt{proj}_1 \circ T(q)$, then they cannot appear in the queries or in the leaf while computing $(\texttt{proj}_2 \circ T)(q)$. (This is because the queries for computing $(\texttt{proj}_2 \circ T)(q)$ and $(\texttt{proj}_1 \circ T)(q)$ are equal, and the output variables that appear in the leaf of $T(q)$ are partitioned between the leaves for $(\texttt{proj}_1 \circ T)(q)$ and $(\texttt{proj}_2 \circ T)(q)$.)

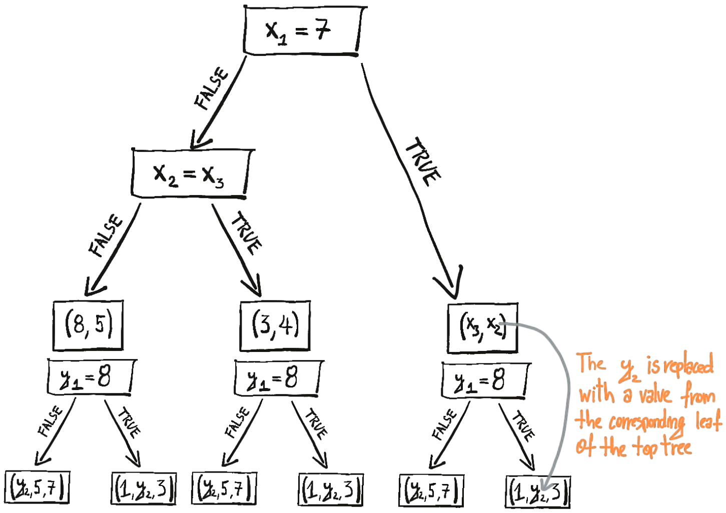
This leaves us with showing that the leaf of $(\texttt{proj}_1 \circ T)(q)$ contains all input variables for which $\pi(x_i) \neq x_i$. For this, we notice that since y is a prefix of e, there exists a y' such that $yy' = e$. Notice that q is a fixpoint of $y' \circ \texttt{proj}_1 \circ y$:

$$y'(\texttt{proj}_1(y(q))) = (yy')(q) = e(q) = e(q_0\, xe) \overset{\text{notation}}{=} q_0\, xee = q_0\, xe = q$$

Let us pick some tree T' that corresponds to y' and let us consider the tree

$$T' \circ \texttt{proj}_1 \circ T$$

By construction (from Claim 17), we know that the leaf corresponding to $(T' \circ \texttt{proj}_1 \circ T)(q)$, can only contain those input variables that were present in the leaf of $(\texttt{proj}_1 \circ T)(q)$:



It follows that all input atoms of q that are not present in $(\mathtt{proj}_1 \circ T)(q)$ as variables have to appear in the leaf of $(T' \circ \mathtt{proj}_1 \circ T)(q)$ as atomic constants. The tree $T' \circ \mathtt{proj}_1 \circ T$ represents the function $y' \circ \mathtt{proj}_1 \circ y$, which (by the definition of the product in S) is equal to $y' \cdot y = e$. By Lemma 47 (stated below), it follows all atoms that appear in the leaves of $(T' \circ \mathtt{proj}_1 \circ T)$ have to appear in $\mathrm{supp}(e)$. It follows that all input variables that do not belong to $\mathrm{supp}(e)$ have to appear in the leaf of $(\mathtt{proj}_1 \circ T)(q)$. This finishes the proof, because π is a $\mathrm{supp}(e)$-permutation. We are now left with proving Lemma 47:

Lemma 47. *Let T be a single-use decision tree, and let f_T be the single-use function represented by T. If a leaf of T contains an atomic constant $a \in \mathbb{A}$, then $a \in supp(f_T)$.*

Proof. We consider the case where T is of type $\mathbb{A}^k \multimap Y$ (the proof can be easily extended to the general case). We pick an a that appears in a leaf of T and we show how to use f_T and atomic constants other than a to construct a value y such that $a \in \mathrm{supp}(y)$. As long as this construction is equivariant, it follows by Lemma 3 that $a \in \mathrm{supp}(f_T)$. The proof goes by induction on the depth of T:

If T is a leaf, we take a tuple of k atoms $(b_1, \ldots, b_k)$ other than a, and define y as $f_T(b_1, \ldots, b_k)$. Since T is a leaf, and a appears in T, we know that $a \in \mathrm{supp}(y)$.

For the induction step, we assume that the query in the root is of the type $x_i = b$; as the case of $x_i = x_j$ is analogous but simpler. We consider two subclasses: $b \neq a$ and $b = a$. First, let us deal with $b \neq a$. We assume that the leaf with a belongs to the `Yes`-subtree (the case for the `No`-subtree is analogous), and we construct $y \in \mathbb{A}^k \multimap Y$ as the following function:

$$(x_1, \ldots, x_k) \mapsto f_T(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_k)$$

Such y is equal to the function defined by the `Yes`-subtree, and since the `Yes`-subtree contains a in its leaf, it follows by the induction assumption that $a \in \mathrm{supp}(y)$.

The case where $b = a$ is harder: We can assume that the `Yes`-subtree and the `No`-subtree define two different functions (or otherwise we can directly apply the induction assumption to the subtree with a) and we define $y \in \mathbb{A}$ as *the only atom $c \in \mathbb{A}$ such that for every $d \in \mathbb{A}$, such that $d \neq c$, the following functions are different*:

$$\begin{aligned}(x_1, \ldots, x_k) &\mapsto f_T(x_1, \ldots, x_{i-1}, c, x_{i+1}, \ldots, x_k)\\ (x_1, \ldots, x_k) &\mapsto f_T(x_1, \ldots, x_{i-1}, d, x_{i+1}, \ldots, x_k)\end{aligned}$$

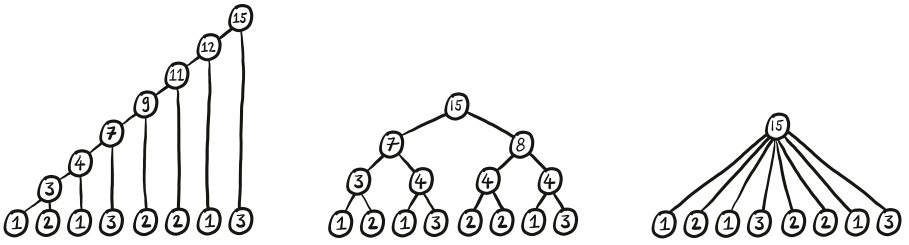
It is not hard to see that the only such c is equal to a, which means that $a \in \mathrm{supp}(y)$. □

3.4 Factorization forest theorem

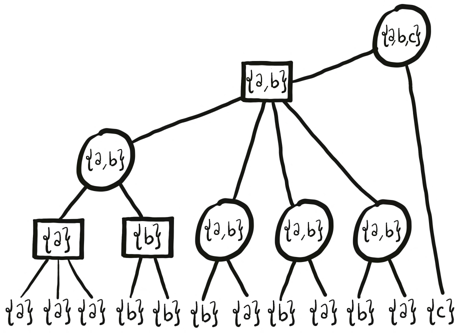
As mentioned before, the translation from local semigroup transductions to compositions of primes is the hardest part of the proof of Theorem 8. We split it into two sections: In this section, we define *factorization trees* and show how to construct them using compositions of primes. In the next section, we use factorization trees to construct the output of local semigroup transductions.

A *factorization tree* for a sequence $s_1, s_2, \ldots, s_n$ over a semigroup S is a tree labelled by elements of S. It has n leaves that correspond to the input positions – the i-th leaf is labelled with s_i. The inner nodes of a factorization tree correspond to infixes of the input sequence and are labelled by the product of that infix. Here are three examples of factorization trees for the infinite semigroup $(\mathbb{N}, +)$, over the following sequence:

1 2 1 3 2 2 1 3



An important parameter of a factorization tree is its height. Notice that if nodes are allowed to have an unbounded number of children, then every sequence admits a decomposition tree of height 1. On the other hand, if every node can only have at most two children, then the height of a decomposition tree over $s_1, \ldots, s_n$ cannot be lower than $\log_2(n)$. A compromise between those two approaches is to require that nodes with more than two children respect the structure of S. An example of this approach is the following *idempotency condition*: We say that a node is *idempotent* if all of its children are labelled by the same idempotent from S. A factorization tree satisfies the idempotency condition if all of its nodes are either binary or idempotent. Here is an example of an idempotent factorization tree for $S = (\, P(\{a, b, c\}),\ \cup\,)$:



Idempotent factorization trees are called *Simon's factorization trees* after Imre Simon, who has shown that finite semigroups admit idempotent factorization trees of bounded height[7]:

Theorem 10 ([Sim90, Theorem 3.3])**.** *For every finite semigroup S there exists an $h_S \in \mathbb{N}$, such that every sequence $s_1, s_2, \ldots, s_n \in S^*$ admits an idempotent factorization tree of height at most h_S.*

This theorem does not directly extend to orbit-finite semigroups:

Example 35. Consider $S = \mathbb{A}^2$, with the operation defined as follows:

$$(x_1, x_2) \cdot (y_1, y_2) = (x_1, y_2)$$

Let us show that S does not admit idempotent factorization trees of bounded height. Notice first, that the only idempotents in S are elements of the form (a, a). Consider the following family of sequences $l_n \in S^*$:

$$l_n = \quad (1,2) \quad (2,3) \quad (3,4) \quad \ldots \quad (n-1, n),$$

and notice that no infix of l_n evaluates to an idempotent. It follows that all nodes in an idempotent factorization tree over l_n have to be binary. This means that the height of an idempotent factorization tree over l_n is at least $\log_2(n)$, which is not bounded by any h_M. ◁

[7] The original theorem shows this for monoids, but the same proof can be applied to semigroups. It is also worth noting that in [Sim90] Simon's factorization trees are referred to as Ramseyan factorization trees.

It follows from Example 35 that if we want to have an equivalent of Theorem 10 that works for orbit-finite semigroups, we need to relax the idempotency condition. We are going to define this relaxation in terms of *smooth sequences*, which are in turn defined in terms of the *Green's infix relation:*[8]

Definition 20. Let S be a semigroup. We say that $x \in S$ is *an infix of* $y \in S$ if for some $a, b \in S^1$ (from Claim 21), it holds that $axb = y$. The infix relation is a preorder – i.e. it is reflexive and transitive. If x and y are each other's infixes, we say that they are *infix equivalent* (or $\mathcal{J}$*-equivalent*). This is an equivalence relation and its equivalence classes are called *infix classes* or $\mathcal{J}$*-classes.* ◁

Example 36. Consider the semigroup $S = \mathbb{A}^2 + \perp + 1$, where 1 is the identity element and otherwise, the operation is defined as follows (this semigroup is equal to the monoid from Example 4):

$$x \cdot \perp = \perp \quad \perp \cdot x = \perp \quad (x_1, x_2) \cdot (y_1, y_2) = \begin{cases} (x_1, y_2) & \text{if } x_2 \neq y_1 \\ \perp & \text{otherwise} \end{cases}$$

This semigroup has three infix classes: $\{1\}$, $\{\perp\}$, and $\{(x, y) \mid x, y \in \mathbb{A}\}$. ◁

We now are ready to define smooth sequences:

Definition 21. Let S be a semigroup. We say that a sequence $s_1, s_2, \ldots, s_n \in S^*$ is *smooth* if each s_i is $\mathcal{J}$-equivalent to the product of the sequence (i.e. $s_1 \cdot s_2 \cdot \ldots \cdot s_n$). (In particular, this means that all s_i's are pairwise $\mathcal{J}$-equivalent, but this is not a sufficient condition for a sequence to be smooth.) ◁

Example 37. Consider the semigroup from Example 36. Here is an example of a smooth sequence:

$$(1, 2)\ (3, 7)\ (4, 9)\ (7, 19)$$

Here are three examples of non-smooth sequences:

$$(1, 2)\ (3, 7)\ (7, 9)\ (1, 3); \quad (1, 3) \perp (2, 9)\ (7, 3); \quad (7, 3)\ 1\ (4, 8)$$

◁

We are now ready to define *smooth factorization trees*: we say that a node of a factorization tree is *smooth* if the labels of its children form a smooth sequence. We say that a factorization tree is *smooth* if all of its nodes are either binary or smooth. Notice that the smoothness condition is a relaxation of the idempotency condition – every idempotent factorization tree is also a smooth factorization tree. Thanks to this relaxation, we can extend Theorem 10 to work with all orbit-finite semigroups [9]:

Theorem 11. *For every orbit-finite semigroup S, there exists an $h_S \in \mathbb{N}$, such that every sequence $s_1, s_2, \ldots, s_n \in S^*$ admits an idempotent factorization tree of height at most h_S.*

[8] Green's relations were introduced and studied by James Alexander Green in [Gre51].

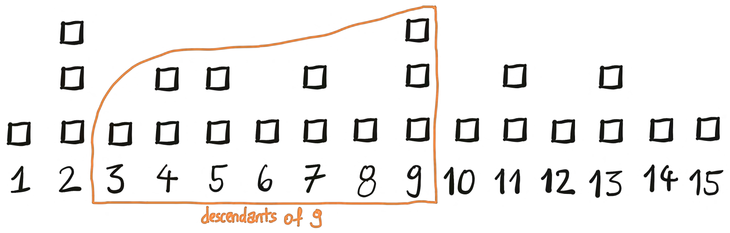
[9] Although a proof of this theorem can be found in [BS20, Lemma16], we include it in this thesis due to its central role in the proof of Theorem 8.

In fact, to complete the proof of Theorem 8, we require a slightly stronger version of Theorem 11: in addition to proving the existence of bounded smooth factorization trees, we need to show that they can be constructed using compositions of primes. This means that we need a way of representing factorization trees as words. For this we use a version of *splits* defined by Colcombet in [Col07, Section 2.3]:

Definition 22. Let S be a semigroup. A split of height h over a sequence $s_1, s_2, \ldots s_n \in S^*$ is a function $t : \{1, \ldots, n\} \to \{1, \ldots, h\}$, which assigns a height to every position of the input sequence. A split defines the following forest structure on the positions of the sequence – a position i is a *descendant* of j if:

1. the position i is to the left of (or equal to) j, i.e. $i \leq j$; and
2. the position i is visible from the position j, which means that the heights of all positions between i and j (including i, but excluding j) are strictly lower than the height of j.

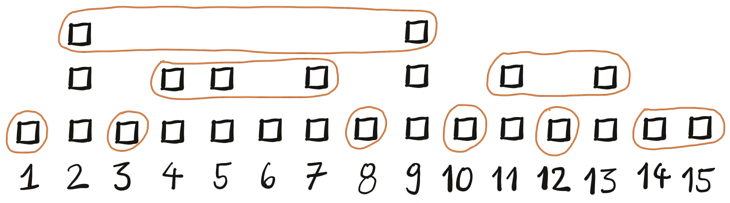
Note that the descendants of every position form a contiguous infix of the input sequence. Here is an example split, together with an example set of descendants:



We say that two positions i and j are *siblings* if:

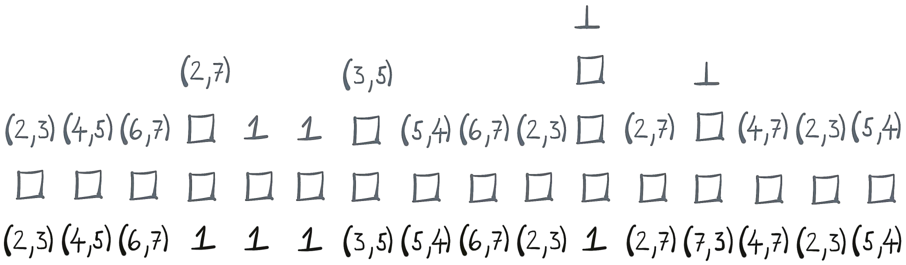
1. they have equal heights; and
2. all positions between i and j (excluding both i and j) are strictly lower than i and j.

Being siblings is an equivalence relation. Here is an example split partitioned into sibling equivalence classes:

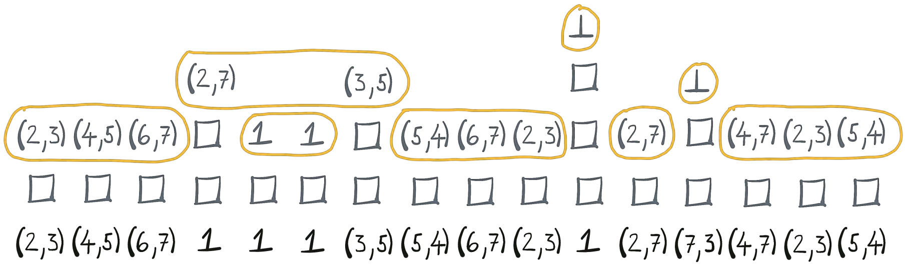


To avoid a potential confusion, let us clarify that being siblings is different from being descendants of the same position. For instance, positions 3 and 4 are both descendants of position 2, but they are not siblings. Conversely, positions 2 and 9 are siblings, but they are not descendants of any position.

The *split value* of a position is the semigroup product of the s_i-values of all of its descendants (this includes the position itself). Here is an example split over a sequence of elements of the semigroup defined in Example 37, where every position is annotated with its split value:

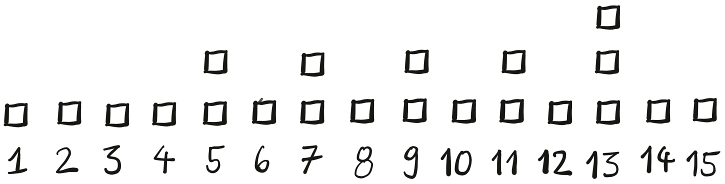


A *sibling subsequence* is a sequence containing all positions from a sibling equivalence class, where each position is labelled by its split value. We say that a split is *smooth* if all of its sibling subsequences are smooth. The split presented in the previous example is smooth. Here is a picture of the same split where all sibling subsequences are marked in orange:

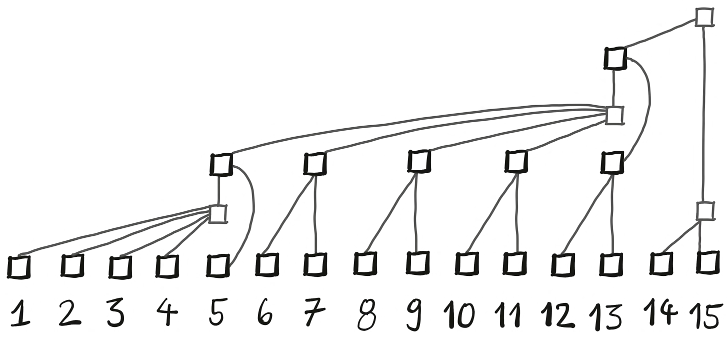


◁

Observe that every smooth split of height h can be transformed into a smooth factorization tree of height $2h + 1$. For example, consider the following split:



It can be transformed into the following factorization tree:



It is not hard to see that a similar construction works for every smooth split. This means that we can prove Theorem 11 by showing how to construct smooth splits of bounded heights. Before we do this, we need to briefly discuss polynomial orbit-finite representations of orbit-finite sets (that are not necessarily polynomial):

Definition 23. Let X be an orbit-finite set. We say that a polynomial orbit-finite set Σ together with a partial function $r : \Sigma \to_{\text{eq}} X + \perp$ is a *polynomial orbit-finite representation* of X if:

1. r is surjective – i.e. every element from X has a representation in Σ;
2. r preserves least supports – i.e. if $r(x)$ is defined, then $\text{supp}(x) = \text{supp}(r(x))$;

◁

Lemma 48. *Every orbit-finite set X has a polynomial orbit-finite representation.*

Proof. Thanks to Lemma 10, we know that there is a surjective total function that preserves least supports:

$$r' : \mathbb{A}^{(k_1)} + \mathbb{A}^{(k_2)} + \ldots + \mathbb{A}^{(k_n)} \to X$$

To obtain a polynomial orbit-finite representation, we choose the following Σ:

$$\Sigma = \mathbb{A}^{k_1} + \mathbb{A}^{k_2} + \ldots + \mathbb{A}^{k_n},$$

and we define r as follows:

$$r(x) = \begin{cases} r'(x) & \text{if } x \in \mathbb{A}^{(k_1)} + \mathbb{A}^{(k_2)} + \ldots + \mathbb{A}^{(k_n)} \\ \perp & \text{otherwise} \end{cases}$$

The function f is surjective and support-preserving supports because r' is surjective and support-preserving. It is worth noting that $r(x)$ is undefined for those tuples that contain repeating atoms, as those tuples may not contain enough distinct atoms to construct elements of X. □

We are now ready to formulate the lemma about constructing smooth splits with compositions of primes. This is the main technical lemma of this section.

Lemma 49. *For every orbit-finite semigroup S and its polynomial orbit-finite representation $r : \Sigma \to_{eq} S$, there exists a natural number h and a function*

$$f : \Sigma^* \to (\underbrace{\{1, 2, \dots, h\}}_{\textit{split height}} \times \underbrace{\Sigma}_{\textit{split value}})^*,$$

such that f can be constructed as a composition of primes and such that f outputs a smooth split over the input sequence, annotated with the split values. Note that the type of f guarantees that the height of the split is at most h.

The rest of this section is devoted to proving Lemma 49. The proof uses induction on the *$\mathcal{J}$-height* of S:

Definition 24. Let S be a semigroup. Define a $\mathcal{J}$-chain to be a sequence of elements:

$$s_1, s_2, \dots, s_n,$$

such that every s_i is a *proper infix* of s_{i+1} (i.e. s_i is an infix of s_{i+1} , but s_{i+1} is not an infix of s_i). The $\mathcal{J}$-height of S is the length of its longest $\mathcal{J}$-chain (or ∞ if S has arbitrarily long $\mathcal{J}$ chains). Similarly, the $\mathcal{J}$-height of an element $x \in S$, is the length of the longest $\mathcal{J}$-chain that starts with x. $\triangleleft$

In order to use $\mathcal{J}$-height as the inductive parameter, we need to know that it is finite:

Lemma 50 ([Boj13, Lemma 9.3])**.** *If S is orbit-finite then it has a finite $\mathcal{J}$ height.*

Proof. We are going to show that if two elements $x, y \in S$ belong to the same orbit, then they are either $\mathcal{J}$-equivalent or $\mathcal{J}$-incomparable. This is enough to prove the lemma because it means that the length of every $\mathcal{J}$-chain is limited by the number of orbits in S. Let us take x and y from the same orbit and show that if x is an infix of y, then also y is an infix of x: Since x and y are in the same orbit, then $y = \pi(x)$ for some atom permutation π. The following claim, which follows from [Pit13, Lemma 1.14], lets us assume that π touches only finitely many elements[10][11] :

Claim 23. *For every x and y that belong to the same orbit, there exists a permutation π such that $\pi(x) = y$, with only finitely many atoms a for which $\pi(a) \neq a$.*

[10] In fact, in [Pit13] Pitts defines sets with atoms (*nominal sets*) using only this type of atom permutations (called finite permutations).

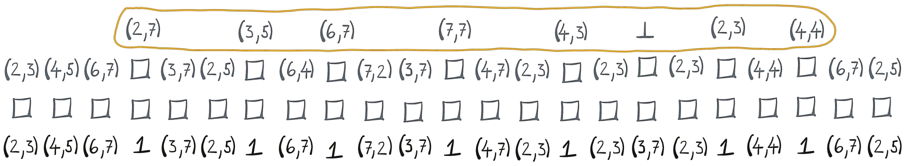
[11] It is worth pointing out that Claim 23 is not true for some other types of atoms that are sometimes studied in the literature. One example of such atoms are rational numbers with comparison $(\mathbb{Q}, \leq)$, called *total-order* atoms (see [Boj19] or [Boj13] for more details). For this reason, the single-use theory for total order atoms is different from the one for equality atoms. We are currently working on it together with Nathan Lhote.

We have assumed that x is an infix of $\pi(x)$ (i.e. y) and we need to show that $\pi(x)$ is an infix of x. The infix order is an equivariant relation, so $\pi(x)$ is an infix of $\pi^2(x)$, and by induction on k we can show that $\pi^k(x)$ is an infix of $\pi^{k+1}(x)$. Since π touches only finitely many atoms, there exists a k such that π^k is the identity permutation. It follows by transitivity that $\pi(x)$ is an infix of x, because x is an infix of $\pi(x)$, which is an infix of $\pi^2(x)$, $\ldots$, which is an infix of $\pi^{k-1}(x)$, which is an infix of $\pi^k(x)$, which is equal to x. □

We are now ready to start proving Lemma 49. We slightly strengthen its formulation, to make it compatible with the inductive structure of the proof:

Definition 25. The *output sequence* of a split is its sibling subsequence of maximal height (note that it is uniquely defined for every split). We say that a split is *semi-smooth* if all of its sibling subsequences are smooth, with the possible exception of the output subsequence. ◁

Here is an example of a semi-smooth split for the semigroup from Example 37. Its output sequence is marked in orange:



Lemma 51. *Fix an orbit-finite semigroup S, let H be its $\mathcal{J}$-height, and let $r : \Sigma \to_{eq} S + \perp$ be its polynomial orbit-finite representation. For every $h \le H$, there exists a function:*

$$f : \Sigma^* \to (\underbrace{\{1, 2, \ldots, h\}}_{\textit{split height}} \times \underbrace{\Sigma}_{\textit{split value}})^*,$$

that can be constructed as a composition of primes, which constructs semi-smooth *splits of height $\le h$ over the input sequence, and annotates each position with its split value. Furthermore, the output sequence of every split constructed by f consists only of elements whose $\mathcal{J}$-height is at most $H + 1 - h$.*

Before we prove Lemma 51, let us show that it implies Lemma 49. We start with the following claim:

Claim 24. *All elements of $\mathcal{J}$-height one are pairwise $\mathcal{J}$-equivalent.*

Proof. Let x and y be elements with $\mathcal{J}$-heights equal to 1. Assume towards a contradiction that they belong to two different $\mathcal{J}$-classes. It follows that either x or y is a proper infix of $x \cdot y$, so either $\{x, xy\}$ or $\{y, xy\}$ forms a $\mathcal{J}$-chain. This contradicts the assumption. □

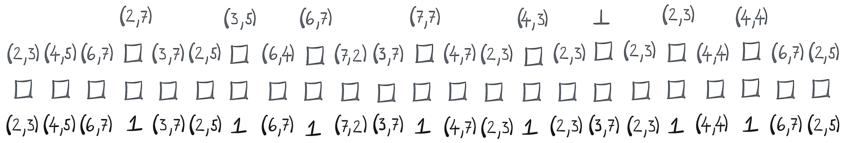
Thanks to Claim 24, we know that every sequence of elements with $\mathcal{J}$-heights equal to 1 has to be smooth. It follows that for $h = H$ Lemma 51 produces a smooth split.

Let us now proceed with the proof of Lemma 51. The induction base is very simple: For $k = 1$, it suffices to set the height of every position to 1, and set the split values to s_i's. This can be expressed as a homomorphism f^*, where f is the following function:

$$f(s) = (1, s)$$

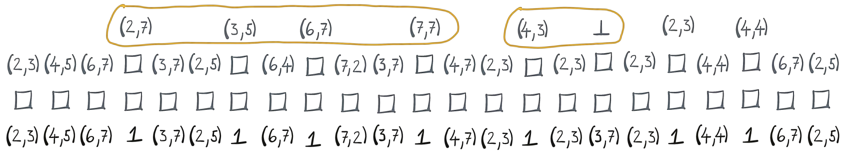
This leaves us with proving the induction step. We construct the split function for $h + 1$ in the following four steps:

1. We apply the induction assumption, constructing an almost smooth split of height h. Here an example for the monoid from Example 37 and $h = 2$:

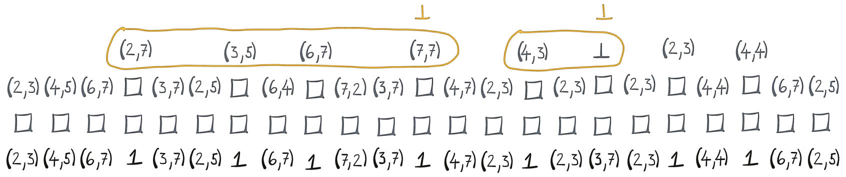


2. Then, we divide the output of the constructed split into *almost smooth blocks*, which are blocks that are smooth sequences except of their last element (this step is explained in detail in Section 3.4.1):

$$\overbrace{\underbrace{s_1, s_2, s_3, \ldots, s_n}_{\text{smooth}}, s_{n+1}}^{\text{non-smooth}}$$

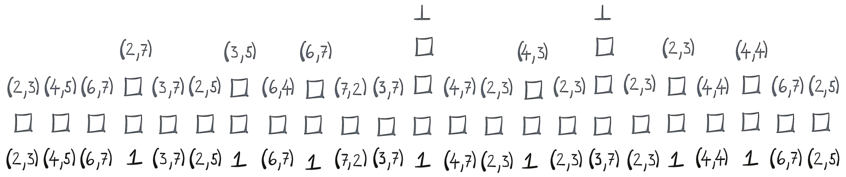


3. Then, we compute the product of each almost smooth block and show that all those products have $\mathcal{J}$-height of at most $(H + 1 - h) - 1$.



4. Finally, we construct a partially smooth split of height $h + 1$, by increasing the height of every last position in an almost smooth block by 1, and by setting the split value of each such position to the product

of its almost smooth block (this can be done using a homomorphism):



3.4.1 Breaking up the sequence

In this section, we construct a function that divides its input sequence into *almost smooth blocks*, which are blocks that are not smooth, but would be smooth if we removed their last element:

$$\overbrace{\underbrace{s_1, s_2, s_3, \ldots, s_n}_{\text{smooth}}, s_{n+1}}^{\text{non-smooth}}$$

We encode this as a function that underlines the last letter of each block:

$$\Sigma^* \to (\underbrace{\Sigma}_{\text{underlined}} + \underbrace{\Sigma}_{\text{not underlined}})^*$$

For example, consider the semigroup $S = \binom{\mathbb{A}}{\leq 3} + \bot$, with the following operation (see Example 5):

$$x \cdot y = \begin{cases} x \cup y & \text{if } x \neq \bot,\ y \neq \bot, \text{ and } |x \cup y| < 3 \\ \bot & \text{otherwise} \end{cases}$$

For the following input sequence:

$$\{1\},\ \{1\},\ \{1\},\ \{5\},\ \{7,8\},\ \{2\},\ \{9\},\ \{7\},\ \{4\},\ \{4\},\ \{4\}$$

The function should return:

$$\{1\},\ \{1\},\ \{1\},\ \underline{\{5\}},\ \{7,8\},\ \underline{\{2\}},\ \{9\},\ \{9\},\ \underline{\{7\}},\ \{4\},\ \{4\},\ \{4\}$$

Which corresponds to the division into the following blocks:

$$\underbrace{\{1\},\ \{1\},\ \{1\},\ \underline{\{5\}}}_{\text{1st block}},\ \underbrace{\{7,8\},\ \underline{\{2\}}}_{\text{2nd block}},\ \underbrace{\{9\},\ \{9\},\ \underline{\{7\}}}_{\text{3rd block}},\ \{4\},\ \{4\},\ \{4\}$$

Note that the last three elements do not belong to any block. This is because their almost smooth block is under construction – they form a smooth sequence and they have not seen an element that would break their smoothness. This is only allowed at the end of the input sequence:

Lemma 52. *Let S be a semigroup and let $r : \Sigma \to S + \bot$ be its polynomial orbit-finite representation. The following function f_{divide} can be constructed as a composition of primes:*

$$f_{divide} : \Sigma^* \to (\Sigma + \Sigma)^*$$

The function f_{divide} divides the input word into almost smooth blocks. Additionally, the rightmost part of the input that does not belong to any block must be smooth. (The input should only contain letters for which r is defined; if this is not the case, the output of f_{divide} is unspecified).

Proof. The main idea of the proof is noticing that being a smooth sequence is a local property:

Claim 25. *A sequence $s_1, \ldots, s_n$ over a semigroup S is smooth, if and only if all of its pairs of consecutive elements are smooth sequences.*

Proof. ($\Rightarrow$): If $s_i,\ s_{i+1}$ is not smooth for some i, then $s_i \cdot s_{i+1}$ is not an infix of either s_i or s_{i+1}. It follows that $s_1 \cdot s_i \cdot s_{i+1} \ldots \cdot s_n$ is not an infix of either s_i or s_{i+1}, which means that the entire sequence is not smooth.

($\Leftarrow$:) We prove this by induction on n. For $n \leq 2$, the claim is trivially true. For the induction step, we assume that both $s_1, \ldots, s_n$ and s_n, s_{n+1} are smooth and show that $s_1, \ldots, s_{n+1}$ is smooth. Every s_i is clearly an infix of $s_1, \ldots, s_{n+1}$. It suffices to show that $s_1 \cdot \ldots \cdot s_{n+1}$ is an infix of every s_i. We already know that $s_1 \ldots s_n$ is an infix of every s_i (for $i \leq n$), so we just need to show that $s_1 \cdot \ldots \cdot s_{n+1}$ is an infix of both of s_{n+1} and of $s_1 \ldots s_n$. For this, we are going to use the following orbit-finite version of a well-known lemma about Green's relation (the orbit-finite version was proved in [Boj13, Lemma7.1 and Theorem5.1]):

Lemma 53. *Let S be an orbit-finite semigroup, and let $x, y \in S$. If xy is an infix of x, then xy is a* prefix *of x. In other words, there exists an $x' \in S^1$ (as defined in Claim 21) such that $x = xyx'$. Analogously, if xy is an infix of y, then xy is a* suffix *of y.*

It follows from the lemma that there exists an s'_n such that $s'_n \cdot s_1 \cdot \ldots \cdot s_n = s_n$. Since s_n, s_{n+1} is a smooth sequence, there are a, b such that $as_n s_{n+1} b = s_{n+1}$. It follows that $a \cdot s'_n \cdot s_1 \cdot \ldots s_{n+1} \cdot b = s_{n+1}$, so $s_1 \cdot \ldots \cdot s_{n+1}$ is an infix of s_{n+1}. To prove that $s_1 \cdot \ldots \cdot s_{n+1}$ is an infix of $s_1 \cdot \ldots \cdot s_n$, it suffices to see that by Lemma 53, there exists $x \in S$, such that $s_n s_{n+1} x = s_n$. It follows that $s_1 \cdot \ldots \cdot s_{n+1} \cdot x = s_1 \cdot \ldots \cdot s_n$, which finishes the proof. □

Before we show how to construct f_{divide}, we define a couple of auxiliary functions:

Claim 26. *For every polynomial orbit-finite set Σ, the following* single-use letter propagation *function can be constructed as a composition of primes:*

$$f_{\Sigma\text{-}prop} : (\Sigma + \downarrow + \epsilon)^* \to (\Sigma + \epsilon)^*$$

The function works in the same way as the single-use letter propagation from Example 29, but it propagates elements of Σ instead of $\mathbb{A}$.

Proof. The claim can be shown by a straightforward induction on the construction of Σ as a polynomial orbit-finite set. □

Claim 27 ([BS20, Lemma 36])**.** *For every polynomial orbit-finite alphabet Σ, the following f_{delay} function can be constructed as a composition of primes:*

$$\begin{array}{ccccccc} a_1 & a_2 & a_3 & a_4 & a_5 & \dots & a_n \\ & & & \downarrow & & & \\ \vdash & a_1 & a_2 & a_3 & a_4 & \dots & a_{n-1} \end{array}$$

Proof. The function can be implemented in three steps: First, we use a classical Mealy machine to mark every position as odd or even (thanks to the classical Krohn-Rhodes theorem, we know that the classical Mealy machine decomposes into prime functions – see the last paragraph of the proof Lemma 52 for details). In the next step, we use a homomorphism together with the single-use letter propagation, to propagate all letters in even positions one position to the right. Finally, we do the same for letters in odd positions. □

We are now ready to construct f_{divide}. First, we apply the *delay* function, while keeping the original input. This is a common pattern, that is possible thanks to the × combinator:

$$\Sigma^* \xrightarrow{\texttt{copy}^*} (\Sigma \times \Sigma)^* \xrightarrow{f_{\text{delay}} \times \texttt{id}} ((\Sigma + \vdash) \times \Sigma)$$

This brings us to the following situation:

$$\begin{array}{cccccccccc} \vdash & s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 & s_8 & s_9 \\ s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 & s_8 & s_9 & s_{10} \end{array}$$

Now, we use a homomorphism to underline every pair s_{i-1}, s_i which is not smooth. We can do this using a homomorphism, because the function $\Sigma^2 \to \Sigma^2 + \Sigma^2$ that underlines non-smooth pairs is equivariant.

$$\begin{array}{cccccccccc} s_1 & s_2 & s_3 & \underline{s_4} & \underline{s_5} & \underline{s_6} & s_7 & \underline{s_8} & s_9 & s_{10} \\ \vdash & s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 & s_8 & s_9 \end{array}$$

We use one more homomorphism to project away the delayed letters:

$$\begin{array}{cccccccccc} s_1 & s_2 & s_3 & \underline{s_4} & \underline{s_5} & \underline{s_6} & s_7 & \underline{s_8} & s_9 & s_{10} \end{array}$$

Finally, we need to get rid of blocks of size 1 (they are always smooth, so they cannot be almost smooth). We do this by removing every other underline in a contiguous block of underlined letters:

$$\begin{array}{cccccccccc} s_1 & s_2 & s_3 & \underline{s_4} & s_5 & \underline{s_6} & s_7 & \underline{s_8} & s_9 & s_{10} \end{array}$$

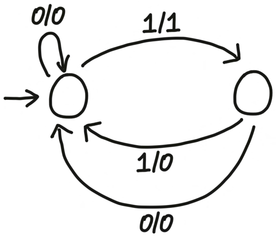
Before we show how to implement this step, let us notice that it finishes the construction of f_{divide}– it follows from Claim 25 that after this step o almost smooth blocks (and possibly one smooth block in the end). This leaves us with showing how to remove every other underline in each contiguous block of underlines. First, we use a homomorphism to apply the following isomorphism to every letter:

$$\underbrace{\Sigma}_{\text{unerline}} + \underbrace{\Sigma}_{\text{no underline}} \quad \simeq \quad \Sigma \times \{ \underbrace{1}_{\text{underline}}, \underbrace{0}_{\text{no-underline}} \}$$

This transformation extracts the finite information about underlines into a separate coordinate:

$$\begin{matrix} 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 & s_8 & s_9 & s_{10} \end{matrix}$$

Now, we use the parallel composition to apply the following classical (i.e. finite) Mealy machine to the $\{0,1\}$-coordinate (thanks to the classical Krohn-Rhodes theorem, we know that it further decomposes into prime functions):



Finally, we use the same isomorphism to go back to the $\Sigma + \Sigma$ alphabet:

$$s_1 \quad s_2 \quad s_3 \quad \underline{s_4} \quad s_5 \quad \underline{s_6} \quad s_7 \quad \underline{s_8} \quad s_9 \quad s_{10}$$

This finishes the proof of Lemma 52. □

Remember that in the proof of Lemma 51, we need to apply f_{divide} to the output of a semi-smooth split. This can be done using the following combinator:

Lemma 54. *Compositions of primes are closed under the following* subsequence *combinator:*

$$\frac{\Sigma^* \xrightarrow{f} \Gamma^*}{(\Sigma + \Box)^* \xrightarrow{(f+\Box)} (\Gamma + \Box)^*} ,$$

The function $(f + \Box)$ applies f to the word composed of Σ-letters of the input, and leaves the $\Box$'s unchanged.

Proof. We start by noticing that:

$$((f \circ g) + \Box) = (f + \Box) \circ (g + \Box) \quad \text{and} \quad (f \times g) + \Box = (f + \Box) \times (g + \Box)$$

It is not hard to see that for every prime function f, the function $(f + \Box)$ is a composition of primes. This finishes the proof. □

3.4.2 Computing almost smooth products

In this section, we show how to use compositions of primes to compute the products of almost smooth blocks. For example, we want to transform the following input:

$$s_1 \quad s_2 \quad s_3 \quad \underline{s_4} \quad s_5 \quad \underline{s_6} \quad s_7 \quad \underline{s_8} \quad s_9 \quad s_{10}$$

Into the following output:

$$\begin{array}{cccccccccc} \square & \square & \square & (s_1 \cdot s_2 \cdot s_3 \cdot s_4) & \square & (s_5 \cdot s_6) & \square & (s_7 \cdot s_8) & \square & \square \\ s_1 & s_2 & s_3 & \underline{s_4} & s_5 & \underline{s_6} & s_7 & \underline{s_8} & s_9 & s_{10} \end{array}$$

This step is formalized as the following lemma:

Lemma 55. *Let S be an orbit-finite semigroup and let $h : \Sigma \to S + \bot$ be its polynomial orbit-finite representation. Then the following function f_{blocks} is a composition of primes:*

$$f_{blocks}(\Sigma + \underline{\Sigma})^* \to (\Sigma + \square)^*$$

The function f_{blocks} inputs a sequence of elements divided into almost smooth blocks (for all other inputs its behaviour is undefined) and computes the product of each of the block: If the i-th letter of the input is underlined, then the i-th letter of the output is a representation *of the product of the block that ends in that letter, i.e. a representation of the value:*

$$h(s_{j+1}) \cdot h(s_{j+1}) \cdot \ldots \cdot h(s_i)$$

where j is the first underlined position to the left of i (or 0 if i is the first underlined position). If the i-th position is not underlined, then the i-th letter of the output is equal to $\square$.

We can reduce the general construction to the case a single block using the following *map* combinator:

Lemma 56. *If a function $f : \Sigma \to \Gamma$ is a composition of primes, then the function*

$$\texttt{map}\, f : (\Sigma + \underline{\Sigma})^* \to (\Gamma + \underline{\Gamma})^*,$$

which applies f independently to every block that ends with an underlined letter (or with the end of the word) is a composition of primes as well.

Proof. Since $\texttt{map}(f \circ g) = (\texttt{map}\, f) \circ (\texttt{map}\, g)$ and $\texttt{map}(f \times g) = (\texttt{map}\, f) \times (\texttt{map}\, g)$, it suffices to show that the `map` versions of all the prime functions can be expressed as compositions of primes. We deal only with the hardest case, i.e. group prefixes. Since it works over finite alphabets, its map version can be implemented as a classical Mealy machine, which, by the classical Krohn-Rhodes theorem, can be decomposed into prime functions. □

This leaves us with showing how to use compositions of primes to compute a product of a single almost smooth block, i.e. a version of Lemma 55, with the extra assumption that the last letter is the only underlined letter in the input:

$$s_1 \quad s_2 \quad s_3 \quad s_4 \quad s_5 \quad \underline{s_6}$$

We compute the product in two steps: In the first step, we compute the product of the smooth sequence of all letters except the last one (this step is formulated below as Lemma 57):

$$\begin{array}{cccccc} \square & \square & \square & \square & \square & (s_1 \cdot s_2 \cdot s_3 \cdot s_4 \cdot s_5) \\ s_1 & s_2 & s_3 & s_4 & s_5 & \underline{s_6} \end{array}$$

Then we include the last element to the product by using a homomorphism to apply the binary product function (see Claim 28 below) to the underlined position. This finishes the construction:

$$\begin{array}{ccccc} \square & \square & \square & \square & (s_1 \cdot s_2 \cdot s_3 \cdot s_4) \cdot s_5 \\ s_1 & s_2 & s_3 & s_4 & \underline{s_5} \end{array}$$

Claim 28. *Let S be an orbit-finite semigroup and $r : \Sigma \to S + \perp$ its polynomial orbit-finite representation. There exists an equivariant function*

$$f : \Sigma \times \Sigma \to_{eq} \Sigma + \perp$$

such that for all $s_1, s_2 \in \Sigma$ that represent elements of S it holds that:

$$f(s_1, s_2) \textit{ is a representation of } s_1 \cdot s_2$$

Proof. We obtain f by applying Lemma 9 to the following relation $R \subseteq \Sigma^2 \times \Sigma$: If elements $x, y \in \Sigma$ both represent elements of S, then the pair (x, y) is R-related with every representation of $r(x) \cdot r(y)$. If either x or y is not a valid representation, then (x, y) is only R-related with $\perp$.

In order to use Lemma 9, we need to show that for every $s_1, s_2 \in \Sigma$, there exists $s_3 \in \Sigma$ such that:

$$(s_1, s_2) \ R \ s_3 \quad \text{and} \quad \mathrm{supp}(s_3) \subseteq \mathrm{supp}(s_1, s_2).$$

If both s_1 and s_2 represent elements from S, we can pick any s_3 that represents $r(s_1) \cdot r(s_2)$ – by Definition 23 combined with Lemma 3, we know that $\mathrm{supp}(s_3) \subseteq \mathrm{supp}(s_1) \cup \mathrm{supp}(s_2)$. If either $r(s_1)$ or $r(s_2)$ is undefined, then we know that they are related to $\perp$, which is equivariant. □

It is worth pointing out that Σ with $f_{\Sigma-\mathrm{prop}}$ is not a semigroup, because $f_{\Sigma-\mathrm{prop}}$ does not have to be commutative: $(a, f(b, c))$ and $f(f(a, b), c)$ might be different representations of the same element.

This leaves us with showing how to use compositions of primes to compute smooth products, we do this in the following lemma, which is the main technical result of this section:

Lemma 57. *For every orbit-finite semigroup S and its polynomial orbit-finite representation $r : \Sigma \to S + \bot$, the following function can be constructed as a composition of primes:*

$$f_{smooth} : (\Sigma + \dashv)^* \to (\Sigma + \Sigma)^*$$

1. *The main case is when the input is a smooth sequence followed by the letter $\dashv$, which has to be the last letter of the input. In this case the function should compute a representation of the product of all its input letters:*

$$\begin{array}{rccccc} \texttt{Input :} & s_1 & s_2 & \ldots & s_n & \dashv \\ \texttt{Output :} & \square & \square & \ldots & \square & s1 \cdot \ldots \cdot s_n \end{array}$$

2. *The secondary case is when the input does not contain the $\dashv$ letter. In this case, f_{smooth} should only output $\square$'s. (This case is useful for handling the last, unfinished block in Lemma 55.)*

In all other cases, the output of f_{smooth} is unspecified.

The remainder of this section is dedicated to proving Lemma 57 In the proof, we use a similar approach as in Section 1.3.3. First, we show how to construct f_{smooth} as a composition of *finitely supported primes*, which are the single-use prime functions extended with homomorphisms based on finitely supported (and not just equivariant) functions. Then in Lemma 60, we show how to eliminate all atomic constants from the construction.

The construction of f_{smooth} as a composition of finitely supported primes consists of the following six steps (we assume that $n \geq 2$ – otherwise we can construct the product using the delay function):

1. In the first step, we fix a tuple $\bar{a}$ of $2\text{dim}(S)$ different atoms (remember that $\dim(S) = \max\{|\text{supp}(s)| : s \in S\}$) and we equip s_1 with representation of an idempotent e_1, that satisfies the following two conditions (for the purpose of this step we say that such e_1 is *good* for s_1):

   (a) e_1 and s_1 are $\mathcal{J}$-equivalent; and

   (b) $\text{supp}(e_1) \subseteq \text{supp}(\bar{a}) \cup \text{supp}(J(s_1))$, where $J(s_1)$ is the $\mathcal{J}$-class of s_1. Notice that the set of all $\mathcal{J}$-classes is a set with atoms itself, which means that $\text{supp}(J(s_1))$ is well-defined.

   The result of this step should look as follows:

$$\begin{array}{ccccccc} s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & \dashv \\ e_1 & \square & \square & \square & \square & \square & \square \end{array}$$

   Before we show how to construct such e_1, let us show that it exists:

   Claim 29. *There exists an idempotent e that is good for s_1.*

Proof. The sequence s_1, s_2 is smooth. It follows that s_1, s_2, and $s_1 \cdot s_2$ all belong to the same $\mathcal{J}$-class. By [Pin10, Corollary 2.25], this means that there is some idempotent e' that belongs to $J(s_1)$. Let us show how to transform this e' into an e that is good for s_1. Define π to be a permutation that swaps every atom from $(\mathrm{supp}(e') - \mathrm{supp}(J(x)))$ with a fresh atom from $(\bar{a} - \mathrm{supp}(J(x)))$. For such π to exist, $\bar{a}$ has to be large enough. This is not hard to see after noticing that function $x \mapsto J(x)$ is equivariant, which means that:

$$|\mathrm{supp}(J(x))| \overset{\text{Lemma 3}}{\leq} |\mathrm{supp}(x)| \leq \dim(S)$$

Define $e := \pi(e')$ and let us show that it an idempotent that is good for s_1:

- $\pi(e')$ is idempotent, because e' is idempotent and the product in S is equivariant;
- the inclusion $\mathrm{supp}(\pi(e')) \subseteq \mathrm{supp}(J(x)) \cup \mathrm{supp}(\bar{a})$ follows from the choice of π;
- $\pi(e') \in J(x)$ because π is a $\mathrm{supp}(J(x))$-permutation.

□

Now let us show how to construct e_1 in the first position. First, we use a classical Mealy machine to underline the first letter. Then, we use a homomorphism to apply a uniformization of the following relation (Lemma 8) to the underlined element of the input word:

$$R(x, y) \Leftrightarrow \begin{cases} y \text{ represents an idempotent that is good for } x \\ y = \bot \text{ and there is no idempotent that is good for } x \end{cases}$$

2. In the second step, we want to propagate e_1 throughout the word:

$$\begin{array}{ccccccc} s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & \dashv \\ e_1 & e_1 & e_1 & e_1 & e_1 & e_1 & \square \end{array}$$

We do this in two substeps: First, we use an $\bar{a}$-supported homomorphism based on the function $x \mapsto (x, \bar{a})$ to equip every input position with $\bar{a}$:

$$\begin{array}{ccccccc} s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & \dashv \\ e_1 & \bar{a} & \bar{a} & \bar{a} & \bar{a} & \bar{a} & \square \end{array}$$

Then, we propagate e_1 to every position. Notice that, since $J(s_i) = J(s_1)$, and $\mathrm{supp}(J(s_i)) \subseteq \mathrm{supp}(s_i)$, it follows that all the atoms from e_1 are already present in each position:

$$\mathrm{supp}(e_1) \subseteq \mathrm{supp}(s_i, \bar{a}).$$

The following lemma says that this is enough to perform a multiple-use propagation of e_1:

Lemma 58. *For every polynomial orbit-finite X, the following* conditional multiple-use propagation *function can be constructed as a composition of single-use primes:*

$$f_{prop} : X^* \to X^*$$

Given an input word $w = x_1, \ldots, x_n$, the function $f_{prop}(w)$ replaces each x_i with x_1, provided that $supp(x_1) \subseteq supp(x_i)$ for every input position i. If there is at least one i, for which $supp(x_1) \not\subseteq supp(x_i)$, then the output is unspecified. Here is an example (for $X = \mathbb{A}^2$):

$$\begin{array}{ccccc} (4,7) & (7,4) & (7,4) & (4,7) & (7,4) \\ & & \downarrow & & \\ (4,7) & (4,7) & (4,7) & (4,7) & (4,7) \end{array}$$

It is worth pointing out that the lemma crucially depends on the support-inclusion condition. We have already seen in Example 32 that the unrestricted version of f_{prop} cannot be implemented as a single-use Mealy machine. According to Section 3.2.1, this implies that it cannot be implemented as a composition of primes either.

Proof. First, let us notice that by a simple induction on the structure of X, we can show that it is possible to extract the supports of elements of X in form of a tuple (the crucial assumption is that X is polynomial):

Claim 30. *For every polynomial orbit-finite X ,there is an equivariant function:*

$$\overline{supp} : X \to_{eq} \mathbb{A}^{\leq \dim X},$$

such that for every x, the tuple $\overline{supp}(x)$ contains the least support of x and all atoms in $\overline{supp}(x)$ are distinct.

Now, let us show how to equip every position with $\overline{\text{supp}}(x_1)$. We start with a homomorphism that equips every position i with $\overline{\text{supp}}(x_i)$. Then, we use the delay function (Lemma 27) and a homomorphism to compute the following relation in every position:

$$r_i \in P(\{1, \ldots, \dim(X)\} \times \{1, \ldots, \dim(X)\})$$

$$n \; r_i \; m \;\Leftrightarrow\; \substack{n\text{-th atom in supp}(x_{i-1}) \text{ and} \\ m\text{-th atom in supp}(x_i) \text{ are equal}}$$

Notice that every r_i is an element of a finite set, which means that we can use a classical Mealy machine to compute the composition of r_i's on each prefix. This way, in every position we obtain the following $\overrightarrow{r}_i$:

$$n \; \overrightarrow{r_i} \; m \;\Leftrightarrow\; \substack{n\text{-th atom in supp}(x_1) \text{ and} \\ m\text{-th atom in supp}(x_i) \text{ are equal}}$$

Thanks to the values $\overrightarrow{r_i}$, we locate each atom from $\overline{\text{supp}}(x_1)$ in $\overline{\text{supp}}(x_i)$, so we can use a homomorphism to compute $\overline{\text{supp}}(x_1)$ in every position.

In the second phase of the construction, we use the values $\overline{\text{supp}}(x_1)$ to compute x_1 in every position. The general idea is as follows:

(a) encode every atom in x_1 as its position in $\text{supp}(x_1)$, obtaining an atomless value x_1';

(b) use a classical Mealy machine to propagate x_1' throughout the word;

(c) in every position, repopulate x_1' with values from $\overline{\text{supp}}(x_1)$.

Formally, we define x_1' using name abstraction and atom placeholders (from Section 1.3.3). Remember that $[\mathbb{A}]X$ denotes the set of elements of X, where one atom might have been replaced by an atomless placeholder. It comes with two operations:

$$\underbrace{\langle a\rangle(x)}_{\substack{\text{replace all } a\text{'s in } x\\ \text{with the placeholder}}} \qquad \underbrace{x@a}_{\substack{\text{replace the placeholder in } x\in[\mathbb{A}]X\\ \text{with the atom } a}}$$

Let us prove the name abstraction preserves polynomial orbit-finite sets:

Claim 31. *If X is a polynomial orbit-finite set, then so is $[\mathbb{A}]X$.*

Proof. By Lemma 12, we know that $[\mathbb{A}](X + Y) \simeq [\mathbb{A}]X + [\mathbb{A}]Y$ and $[\mathbb{A}](X \times Y) = [\mathbb{A}]X \times \mathbb{A}[Y]$, so it suffices to notice that:

$$[\mathbb{A}]1 \simeq 1 \quad \text{and} \quad [\mathbb{A}]\mathbb{A} \simeq \underbrace{\mathbb{A}}_{\text{real atom}} + \underbrace{1}_{\text{placeholder}}$$

□

Let us define $[\mathbb{A}^k]X$ and $[\mathbb{A}^{\leq k}]X$ as follows:

$$[\mathbb{A}^k]X = \underbrace{[\mathbb{A}](\ldots([\mathbb{A}]X)\ldots)}_{k \text{ times } [\mathbb{A}]} \qquad [\mathbb{A}^{\leq k}] = X + [\mathbb{A}]X + \ldots + [\mathbb{A}^k]X$$

We define $x_1' \in [\mathbb{A}^{\leq \dim(X)}]X$ as $\langle\overline{\text{supp}}(x_1)\rangle(x_1)$. By Lemma 13, we know that x_1' is equivariant (i.e. atomless). This means that we can propagate x_1' throughout the word using a classical Mealy machine. Now every position is equipped with both x_1' and $\overline{\text{supp}}(x_1)$. This means that can use a homomorphism to reconstruct x_1 in every position as $x_1'@(\text{supp}(x_1))$. □

3. In this step, we decompose every s_i into $x_i \cdot y_i$ such that e_1 is a suffix of x_1 and a prefix of y_1. We do this by using a homomorphism based on the following function:

   Claim 32. *Let S be an orbit-finite semigroup, and $h : \Sigma \to S + \bot$ be its polynomial orbit-finite representation. There exists a finitely supported function that $f_{decompose} : \Sigma^2 \to_{fs} \Sigma^2$, that does the following:*

- ***Input:*** *$s, e \in \Sigma$, such that e represents an idempotent from S, and s represents an element from the same $\mathcal{J}$-class as e;*
- ***Output:*** *$x, y \in \Sigma$, such that $h(e)$ is a suffix of $h(x)$, $h(e)$ is a prefix of $h(y)$, and $h(x) \cdot h(y) = h(s)$.*

Proof. Thanks to the Lemma 8, it suffices to show that for all $s, e \in S$ there exists at least one such decomposition. Since e is an infix of s, we know that $aeb = s$ for some $a, b \in S$. Define $x := ae$ and $y := eb$. It follows that:

$$xy = aeeb = aeb = s$$

Since e is clearly a suffix of x and a prefix of y, we can pick $s = x \cdot y$ as the desired decomposition.

□

4. Next, we apply the delay function to the y-coordinates and use homomorphism to compute $g_i := y_i \cdot x_{i+1}$:

$$\begin{array}{ccccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & \dashv \\ \vdash & y_1 & y_2 & y_3 & y_4 & y_5 & y_6 \\ \vdash & g_1 & g_2 & g_3 & g_4 & g_5 & \dashv \end{array}$$

We say that two elements of a semigroup are *$\mathcal{H}$-equivalent* if they are both prefix equivalent and suffix equivalent. The important property of the g_i values is that they are all $\mathcal{H}$-equivalent to e_1:

Claim 33. *All g_i are prefix and suffix equivalent to e_1.*

Proof. Let us show that e_1 is prefix equivalent to g_i (the proof for suffix equivalence is similar). Thanks to Lemma 53, it suffices to show that e_1 is a prefix of g_i and that g_i and *infix* of e_1. First, observe that e_1 is a prefix of y_i, which in turn is a prefix of g_i. Thus, e_1 is a prefix of g_i. Next, to see that g_i is an infix of e_1, observe that $x_i g_i y_{i+1} = s_i s_{i+1}$. This means that g_i is an infix of $s_i s_{i+1}$, which, by smoothness, is an infix of s_i, and s_i an infix of e_1. Thus g_i is an infix of s_i. □

5. In this step, we equip the last position (i.e. the position with $\dashv$) with a representation of the product of all g_i's:

$$\overrightarrow{g} := g_1 \cdot \ldots \cdot g_{n-1}$$

Recall that, by Claim 33, every g_i is $\mathcal{H}$-equivalent to e_i. It follows that the support of each g_i is equal to the support of e_i:

Lemma 59. *Let $e \in S$ be an idempotent and let $H(e)$ be its $\mathcal{H}$-class. Then, for every $x \in H(e)$, it holds that supp$(x) =$ supp(e).*

Proof. We start the proof by citing a few results:

- [Pin10, Proposition 1.13] states that in every semigroup S (possibly infinite), if an $\mathcal{H}$-class contains an idempotent, then it forms a subgroup. Moreover, thanks to [Pin10, Proposition 1.4] we know that the idempotent is the identity element of this subgroup. (In particular, this means that no $\mathcal{H}$-class contains more than one idempotent.)
- [CLP15, Corollary 2.17] says that in an orbit-finite semigroup S all $\mathcal{H}$-classes are finite[12].

It follows that $H(e)$ is a finite subgroup of S and that e is the identity element of this group. First, let us show that $\mathrm{supp}(e) \subseteq \mathrm{supp}(x)$: Since $H(e)$ is a finite group, there exists an n such that $x^n = e$. The function $x \mapsto x^n$ is equivariant, so by Lemma 3, $\mathrm{supp}(e) \subseteq \mathrm{supp}(x)$. Now, let us show that $\mathrm{supp}(x) \subseteq \mathrm{supp}(e)$: By Lemma 3, $\mathrm{supp}(H(e)) \subseteq \mathrm{supp}(e)$. Now, since $H(e)$ is finite, $\mathrm{supp}(x) \subseteq H(e)$ for every $x \in H(x)$ (or otherwise the $\mathrm{supp}(H(e))$-orbit of x would be infinite). It follows that $\mathrm{supp}(x) \subseteq \mathrm{supp}(H(e)) \subseteq \mathrm{supp}(e)$. □

Now, we equip every position with $\overline{\mathrm{supp}}(e_1)$ – the easiest way to do it is to simply not forget those values from Step 2, but we can also use Lemma 58. Then, we use a homomorphism to compute $g'_i = \langle\overline{\mathrm{supp}}(e_1)\rangle(g_i)$. By Lemmas 59 and 13, we know that all g'_i's are atomless. It follows that we can use a classical Mealy machine to compute $\overrightarrow{g'} = g'_1 \cdot \ldots g'_n$ and save it in the last letter. The binary product used to compute g' is defined using functoriality of $\langle\overline{\mathrm{supp}}(e_1)\rangle$ as $\langle\overline{\mathrm{supp}}(e_1)\rangle(\_ \cdot \_)$. By Lemma 16, we know that $\overrightarrow{g}'@\overline{\mathrm{supp}}(e_1) = \overrightarrow{g}$. Since, at this point, the last letter contains both $\overrightarrow{g}'$ and $\mathrm{supp}(e_q)$, it follows that we can use a homomorphism to construct $\overrightarrow{g}$ in the last letter.

6. Finally, we notice that $s_1 \cdot \ldots \cdot s_n = x_1 \cdot \overrightarrow{g} \cdot y_n$. The value $\overrightarrow{g}$ is already present in the last position. This means that in order to compute $s_1 \cdot \ldots \cdot s_n$, we can use the generalized single-use propagation (Claim 26) to send x_1 and y_n to the last position (which we recognize by $\dashv$), and then apply a homomorphism that multiplies values x_1, $\overrightarrow{g}$ and y_i in the last position.

This leaves us with showing how to get rid of the unnecessary atoms:

Lemma 60. *If $f : \Sigma^* \to \Gamma^*$ can be constructed as a composition of finitely supported primes, then it can also be constructed as a composition of supp(f)-supported primes. In particular, if f is equivariant, then it can be constructed as a composition of (equivariant) primes.*

Proof. Given a function $f : \Sigma^* \to \Gamma^*$, which can be constructed as a composition of α-supported primes, and an atom $a \notin \mathrm{supp}(f)$, we show how to construct f as a composition of $(\alpha - a)$-supported primes. This is enough to proof the

[12] It is also worth pointing out [CLP15, Lemma 2.14] which says that all orbit-finite groups are finite. This can be seen as the reason why Krohn-Rhodes decompositions of single-use Mealy machines use only finite (i.e. atomless) groups.

lemma, because we can repeat this process for every atom in $\alpha - \text{supp}(f)$.

The construction uses name abstraction (see Section 1.3.3 for details). Consider the function:

$$\langle a \rangle f : [\mathbb{A}](\Sigma^*) \to [\mathbb{A}](\Gamma^*)$$

Thanks to the isomorphism $W_X : \langle a \rangle (X^*) \simeq (\langle a \rangle X)^*$ from Lemma 12, we can treat $\langle a \rangle f$ as a function on words:

$$\langle a \rangle f : ([\mathbb{A}](\Sigma))^* \to ([\mathbb{A}](\Gamma))^*$$

Let us show that $\langle a \rangle f$ can be constructed as a composition of $(\alpha - \{a\})$-primes:

Lemma 61. *If $f : \Sigma^* \to \Gamma^*$ can be constructed as a composition of primes supported α, then $\langle a \rangle f : (\langle a \rangle \Sigma)^* \to (\langle a \rangle \Gamma)^*$ can be constructed as a composition of primes supported by $\alpha - \{a\}$.*

Proof. The proof goes by induction by on the construction of f. For the induction base we assume that f is a prime function. The only prime function, that might not me equivariant, is a homomorphism h^*. In this case, it is not hard to see that $\langle a \rangle (h^*) = (\langle a \rangle h)^*$ (thanks to the isomorphism W from Lemma 12). This is enough, because Lemma 13 we know that if h is supported by α, then $\langle a \rangle h$ is supported by $\alpha - \{a\}$.

For the induction step, we first notice that thanks to Lemma 15, we have that:

$$\langle a \rangle (f \circ g) = \langle a \rangle f \circ \langle a \rangle g$$

This is enough to handle the case of $\circ$-composition. For the $\times$-composition, we would like to show that:

$$\langle a \rangle (f \times g) = (\langle a \rangle f) \times (\langle a \rangle g)$$

The main problem is type inconsistency:

$$\langle a \rangle (f \times g) : \quad ([\mathbb{A}](\Sigma_1 \times \Sigma_2))^* \to ([\mathbb{A}](\Gamma_1 \times \Gamma_2))^*$$

$$\langle a \rangle f \times \langle a \rangle g : \quad ([\mathbb{A}]\Sigma_1 \times [\mathbb{A}]\Sigma_2)^* \to ([\mathbb{A}]\Gamma_1 \times [\mathbb{A}]\Gamma_2)^*$$

In order to solve it, we use the isomorphism from Lemma 12, which is defined as follows:

$$P : ([\mathbb{A}](X \times Y)) \simeq ([\mathbb{A}]X \times [\mathbb{A}]Y)$$

$$P(x) = (\langle a \rangle (\mathtt{proj}_1(x@a)), \langle a \rangle (\mathtt{proj}_2(x@a)))$$

Using P^*, we can (implicitly) cast between $([\mathbb{A}](\Sigma_1 \times \Sigma_2))^*$ and $([\mathbb{A}]\Sigma_1 \times [\mathbb{A}]\Sigma_2)^*$. Then it is not hard to see that:

$$\langle a \rangle (f \times g) = (\langle a \rangle f) \times (\langle a \rangle g)$$

□

Now, let us use $\langle a \rangle f$ to reconstruct the original f without using a. Remember that in Claim 11, we have defined an embedding: $\iota_X : X \to [\mathbb{A}]X$ such that:

$$\iota_\Sigma(x) = \langle a \rangle x \quad \text{where } a \notin \mathrm{supp}(x)$$

This is an embedding, so it has a partial inverse $\iota_X^{-1} : [\mathbb{A}]X \to X + \bot$. We use it, to construct f' in the following way and show that $f = f'$:

$$f' : \Sigma^* \xrightarrow{\iota^*} ([\mathbb{A}]\Sigma) \xrightarrow{\langle a \rangle f} ([\mathbb{A}]\Gamma)^* \xrightarrow{\iota^{-1}} (\Gamma + \bot)^*$$

Note that there is a slight type mismatch: the type of f is $\Sigma^* \to \Gamma^*$, and the type of f' is $\Sigma^* \to (\Gamma + \bot)^*$. We can ignore this mismatch, because (as we are going to show) f' never returns $\bot$. Let us now proceed with the proof that $f' = f$. It is enough to show that the following diagram commutes:

$$\begin{tikzcd}
\Sigma^* \arrow[r, "f"] \arrow[d, "\iota_\Sigma"] & \Gamma^* \arrow[d, "\iota_\Gamma"] \\
([\mathbb{A}]\Sigma)^* \arrow[r, "\langle a \rangle f"] & ([\mathbb{A}]\Gamma)^*
\end{tikzcd}$$

For that, we notice that (by assumption) $a \notin \mathrm{supp}(f)$, so $\langle a \rangle f = \iota_{\Sigma^* \to \Gamma^*}(f)$. It follows that we can finish the proof by applying the following claim:

Claim 34. *For every X, Y, and $f : X \to_{fs} Y$, the following diagram commutes:*

$$\begin{tikzcd}
\Sigma^* \arrow[r, "f"] \arrow[d, "\iota_\Sigma"] & \Gamma^* \arrow[d, "\iota_\Gamma"] \\
([\mathbb{A}]\Sigma)^* \arrow[r, "\iota(f)"] & ([\mathbb{A}]\Gamma)^*
\end{tikzcd}$$

Proof. We take a $x \in X$, and show that $\iota(f(x)) = (\iota f)(\iota x)$. Let b be an atom such that $b \notin \mathrm{supp}(x) \cup \mathrm{supp}(f)$. It follows by definition of ι that $\iota(f) = \langle b \rangle f$, $\iota(x) = \langle b \rangle x$, by and (by Lemma 3) $\iota(f(x)) = \langle b \rangle (f(x))$. Using the isomorphism from Lemma 14, we get that:

$$(\iota f)(\iota x) = (\langle b \rangle f)(\langle b \rangle x) = \langle b \rangle \left(f((\langle b \rangle x) \mathbin{@} b) \right) = \langle b \rangle (f(x)) = \iota(f(x))$$

□

This completes the proof of Lemma 57. □

3.5 Local semigroup transductions $\subseteq$ Compositions of primes

In this section, we finish the proof of Theorem 8 by proving the following lemma:

Lemma 62. *If Σ and Γ are polynomial orbit-finite sets and $f : \Sigma^* \to \Gamma^*$ is a local semigroup transduction, then f can be constructed as a composition of primes.*

Take any local semigroup transduction $f : \Sigma^* \to \Gamma^*$ given by (S, h, λ). All semigroup transductions are equivariant, so by Lemma 60, it suffices to show that f can be constructed as a composition of finitely supported primes. Moreover, we can assume that $h : \Sigma \to_{\mathrm{eq}} S$ is a polynomial orbit-finite representation[13] of S. This is because if the original function $h : \Sigma \to S$ is not a polynomial orbit-finite representation, we can pick a polynomial orbit-finite representation $h' : \Sigma' \to_{\mathrm{eq}} S$ and start the construction by applying to every letter the uniformization (Lemma 8) of the following relation (using a homomorphism prime function):

$$x \, R \, y \quad \Leftrightarrow \quad h(x) = h'(y)$$

This leaves us with implementing the transduction (S, h', λ), which satisfies the condition that h' is a polynomial orbit-finite representation. From now on, we assume that h is a polynomial orbit-finite representation of S. Thanks to this assumption, we can use Lemma 49 to construct a smooth split over the input sequence. In the remainder of this section, we show how to transform the split into the output of the local semigroup transduction. First, let us define some additional structure on the split:

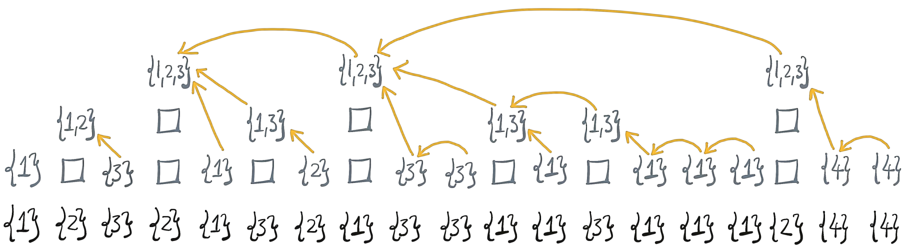
Definition 26. For every position i of a split, we define its *left ancestor* to be the rightmost j to the left of i, that is higher or equal than i. Note that every position has at most one left ancestor. The *ancestor sequence* of i is the smallest subsequence of the split positions, that contains i and is closed under ancestors – i.e. the sequence that contains i, i's ancestor, the ancestor of i's ancestor, and so on ... $\triangleleft$

Example 38. Let us consider the semigroup $S = \binom{\mathbb{A}}{\leq 3} + \bot$ with the following operation:

$$a \cdot b = \begin{cases} a \cup b & \text{if } a \neq \bot, \; b \neq \bot, \text{ and } |a \cup b| \leq 3 \\ \bot & \text{otherwise} \end{cases}$$

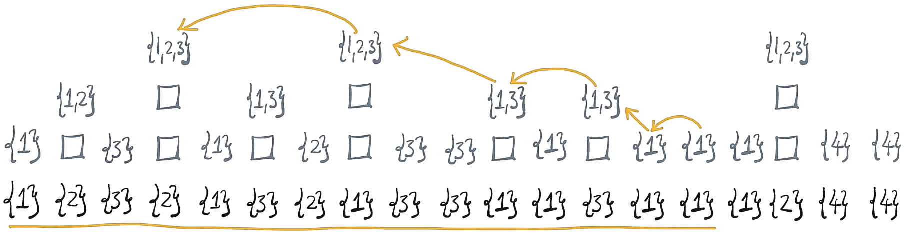
Here is a sequence over this semigroup (black) and an example split over the sequence (grey). The orange arrows point to the left ancestor of every position:

[13] There is a slight type mismatch here: a polynomial orbit-finite representations is a partial function $\Sigma \to S + \bot$, and h is a total function $h : \Sigma \to S$. We can deal with this type mismatch by defining a semigroup $S' = S + \bot$, where $\bot$ is an all-absorbing error element, i.e. $x \cdot \bot = \bot = \bot \cdot x$. Alternatively, we can assume that input does not contain any letters for which h is undefined.



◁

Importantly, the product of the split values in the ancestor sequence of a position i is equal to the product of the ith prefix of the input sequence. (This property follows immediately from a definition of a split value.) For example, consider the following split:



We can calculate the product for the underlined prefix as the product of the highlighted ancestor sequence:

$$\{1,2,3\} \cdot \{1,2,3\} \cdot \{1,3\} \cdot \{1,3\} \cdot \{1\} \cdot \{1\}$$

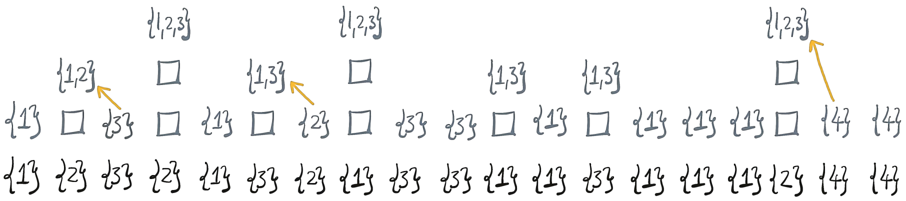
Observe that the product of every two consecutive values in this sequence is $\mathcal{J}$-equivalent to the element on the left (i.e. $a_i a_{i+1}$ is $\mathcal{J}$-equivalent to a_i). We say that a split is *monotone* if all ancestor sequences (labelled with split values) satisfy this condition:

Definition 27. Let $s_1, \ldots, s_n \in S^*$ be a sequence over a semigroup equipped with a smooth split and let $v_1, \ldots, v_n \in S^*$ be the split values of this split. We say that the split is *monotone* if for every j, i such that j is i's left ancestor it holds that:

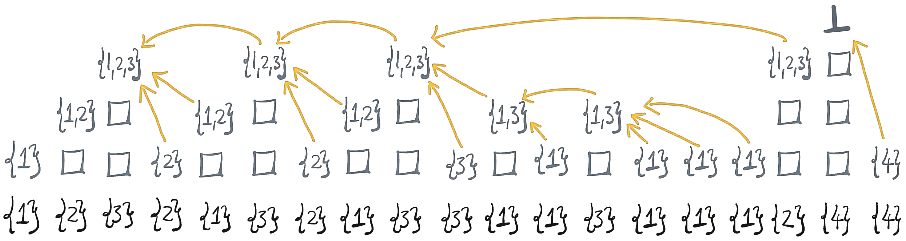
$$v_i \cdot v_j \text{ is infix equivalent to } v_i$$

◁

The split form Example 38 is not monotone, because the following pairs do not satisfy the monotonicity condition:



It is, however, possible to construct another smooth split over this sequence that is monotone (orange arrows point to ancestors):



At the end of this section, in Lemma 66, we will show how to transform smooth splits of bounded height into monotone smooth splits of bounded height. For now, let us assume that the input split is monotone.

In order to construct the output of the local semigroup transduction, it would be enough to equip every position with its ancestor sequence. Unfortunately, this cannot be achieved with compositions of primes, because the ancestor sequences can have unbounded lengths. The general idea of the proof of Lemma 62 is to compress the ancestor sequences so that they can be handled using compositions of primes, while preserving enough information to determine the output of the local semigroup transduction.

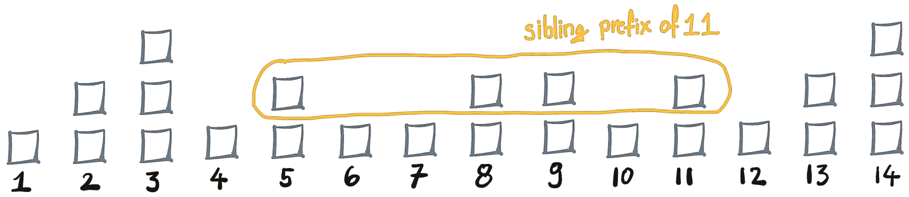
Before we continue with the proof, we need to define a few more notions: Remember that positions i and j are called *siblings* if they have equal heights and there is no higher position between them. Being siblings in an equivalence relation, and its equivalence classes are called *sibling subsequences*. The leftmost position in every sibling subsequence is called the *eldest sibling*. The *sibling prefix* of a position i is the prefix of the sibling subsequence that ends in this position. Finally, we say that a position of a split is *regular* if its split value is $\mathcal{J}$-equivalent to some idempotent in S. Note that if a position of a smooth split has at least one sibling, then by a reasoning similar to the one in Claim 29, it has to be regular.

The following lemma says how to equip all sibling subsequences with values x_i, y_i, and g_i analogous to the ones constructed in the proof of Lemma 57:

Claim 35. *Let Σ be a polynomial orbit-finite representation of S. There is a composition of primes that inputs a smooth split and equips every regular position i of the split with (representations of) values $g_i \in S$, $y_i \in S$, and if i is an eldest sibling $x_i \in S$ such that for every regular i and its eldest sibling j:*

1. *$x_j g_i y_i$ is equal to the product of the split values of i's sibling prefix;*
2. *g_j is an idempotent, and g_i is $\mathcal{H}$-equivalent to g_j;*
3. *x_j is prefix equivalent to v_j (i.e. the split value of j) and suffix equivalent to g_j;*
4. *y_i is suffix equivalent to v_i and prefix equivalent to g_i.*

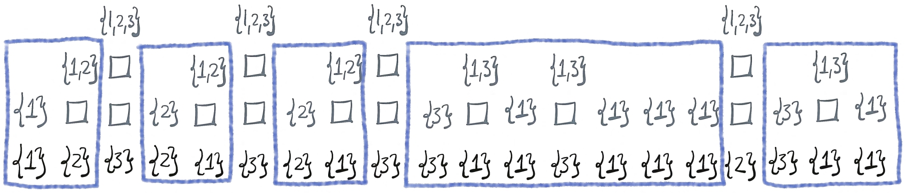
For example, consider the following split:



Highlighted is the sibling prefix of 11. *The eldest sibling of* 11 *is* 5. *According to the lemma, this means that $x_5 \cdot g_{11} \cdot y_{11}$ is equal to the product $v_5 \cdot v_8 \cdot v_9 \cdot v_{11}$, where v_i denotes the split value of the position i. Note that v_i differs from s_i, which is the i-th element of the split's underlying sequence. Furthermore, the lemma requires that g_5 is idempotent and $\mathcal{H}$-equivalent to g_{11}. Also, x_5 is prefix equivalent to v_5 and suffix equivalent to g_5. Lastly, y_{11} is prefix equivalent to g_{11} and suffix equivalent to v_5.*

Proof. The proof goes by induction on the split's height. If it is equal to 1, then the input is a smooth product and we can use a construction similar to the one in the proof of Lemma 57. The difference is that in Step 5, we use a classical (i.e. atomless) Mealy machine to equip *every* position i with the atomless value $\overrightarrow{g_i}' = g_1' \cdot \ldots \cdot g_i'$. Then we can use a use homomorphism to repopulate every $\overrightarrow{g_i}'$ with atoms from $\overline{supp}(e_1)$ (available in every position), obtaining $\overrightarrow{g}_i = g_1 \cdot \ldots \cdot g_i$. We conclude by keeping the x_i and y_i values constructed in Step 3 and defining g_i as $\overrightarrow{g}_i$ (this is not going to cause a name clash, because we are not going to use the original g_i's any more).

For the induction step, notice that the positions of the maximal height form a smooth subsequence. This means we can use the construction from the induction base (combined with the subsequence combinator from Lemma 54), to compute the values x, y and g in the positions of the maximal height. Then, we observe that the positions of the maximal height divide the input split into smooth splits of lower heights. For example:



Thanks to this observation, we can finish the construction by combining the induction assumption with the map combinator (Lemma 56). □

Now, let us use the values x_i, g_i, y_i to define the *compact ancestor sequence* for every position of the split:

Definition 28. For every position i of a monotone smooth split, we define its *compact ancestor sequence* (cas(i)) in the following way:

1. If i a regular position, then cas(i) is defined as the following tuple:

$$\text{cas}(i) = (\text{cas}(p), x_j, g_i, y_i),$$

   where j is i's eldest sibling, and p is j's left ancestor. Note that j always exists, but it might happen that $i = j$. It is, however, possible that j might not have a left ancestor – in this case, we simply omit the cas(p) part.

2. If i is not regular, then cas(i) is defined as:

$$\text{cas}(i) = (\text{cas}(p), v_i),$$

   where p is the left ancestor of i. Again, if i does not have a left ancestor, we omit the cas(p) part.

◁

Notice that by Claim 35 the product of the compact ancestor sequence is equal to the product of the split values in the full ancestor sequence. It follows that the product of the compact ancestor sequence of i is equal to the product of the ith prefix of the input sequence (i.e. $s_1 \cdot \ldots \cdot s_i$). Importantly, the length of the compact ancestor sequence is bounded.

Claim 36. *The length of every compact ancestor sequence is bounded by* $3h$*, where* h *is the height of the split.*

Proof. It suffices to see that in Definition 28 the position p is always higher than i. First, let us observe that, by construction, the position p is always either higher than i, or a proper sibling of i (i.e. $i \neq p$). This leaves us with showing that p is never a sibling of i: If i is regular, then we know that p cannot be i's sibling because it is to the left if i's eldest sibling j. And if i is not regular, then p cannot be i's sibling, because the nodes that are not regular do not have any siblings other than themselves (see the proof of Claim 29). □

Unfortunately, the compact ancestor sequence still contains too much information to be computed by compositions of primes – if there were a composition of primes that equips every position i with its compact ancestor sequence, then there would also be a composition of primes that computes the semigroup product of every prefix (it suffices to use a homomorphism to compute the product of each $\mathrm{cas}(i)$). This is a contradiction, because by Example 33, we know that compositions of primes are not capable of computing products of prefixes.

In order to make the compact ancestor subsequences manageable for the composition of primes, we need to forget some of their atoms. This is expressed in the language of α-orbits. Remember that, if α is a finite subset of atoms, then the α-orbit of an element x is defined as:

$$\mathrm{orb}_\alpha(x) = \{\pi(x) \mid \text{where } \alpha \text{ is an } \alpha\text{-permutation } \}$$

One can look at $\mathrm{orb}_\alpha(x)$ as an operation, that it forgets all the atoms from x that do not belong to α. Note that $\mathrm{orb}_\alpha(x)$ is usually not polynomial orbit-finite. However, as the following lemma shows, we can use polynomial orbit-finite sets to represent orbits of polynomial orbit-finite sets.

Lemma 63. *For every $k \in \mathbb{N}$ and every polynomial orbit-finite Σ, there is a polynomial orbit-finite set $orb_k(\Sigma)$ and a function:*

$$\overline{orb} : \mathbb{A}^{\leq k} \times \Sigma \to orb_k(\Sigma),$$

such that:

1. *if $\overline{orb}(\bar{a}, x) = \overline{orb}(\bar{a}, y)$, then x and y belong to the same $\bar{a}$-orbit; and*
2. *$supp\left(\overline{orb}(\bar{a}, x)\right) \subseteq supp(\bar{a})$.*

The intuition behind this lemma is that $\overline{orb}(\bar{\alpha}, x)$ is a representation of $orb_\alpha(x)$, where $\bar{\alpha}$ is any of the tuples that contains all atoms from α.

Proof. We start by defining $\mathrm{orb}_k(\Sigma)$:

$$\mathrm{orb}_k(\Sigma) = [\mathbb{A}^{\leq k}]\Sigma = \Sigma + [\mathbb{A}]\Sigma + [\mathbb{A}^2]\Sigma + \ldots + [\mathbb{A}^k]\Sigma$$

Now, in order to compute $\overline{\mathrm{orb}}(\bar{a}, x)$, we abstract away all the atoms that do not belong to $\bar{a}$:

$$\overline{\mathrm{orb}}(\bar{a}, x) = \langle \overline{\mathrm{supp}}(x) - \bar{a} \rangle (x).$$

(Note that $\overline{\mathrm{supp}}(x)$ is the support-extracting function from Claim 30).

Now let us show that this $\overline{\mathrm{orb}}$ satisfies the two required properties. The second one (i.e. $\mathrm{supp}(\overline{\mathrm{orb}}(\bar{a}, x)) \subseteq \mathrm{supp}(\bar{a})$) follows immediately from Lemma 13. This leaves us with showing the first property: We take some $\bar{a}$, x, y, such that $\overline{\mathrm{orb}}(\bar{a}, x) = \overline{\mathrm{orb}}(\bar{a}, y)$ and show that $x = \pi(y)$, for some $\bar{a}$-permutation π. Since $\overline{\mathrm{orb}}(\bar{a}, x) = \overline{\mathrm{orb}}(\bar{a}, y)$, we know that:

$$|\overline{\mathrm{supp}}(x) - \bar{a}| = |\overline{\mathrm{supp}}(y) - \bar{a}|$$

Clearly, neither $(\overline{\mathrm{supp}}(x) - \bar{a})$ nor $(\overline{\mathrm{supp}}(y) - \bar{a})$ contains repeating atoms or atoms from $\bar{a}$. This means that there is an $\bar{a}$-permutation π, that transforms $\overline{\mathrm{supp}}(y) - \bar{a}$ into $\overline{\mathrm{supp}}(x - \bar{a})$. Let us show that this π transforms y into x:

$$\pi(y) = \pi(\overline{\mathrm{orb}}(a, y) @ (\overline{\mathrm{supp}}(y) - \bar{a})) = (\pi(\overline{\mathrm{orb}}(a, y)) @ (\pi(\overline{\mathrm{supp}}(y) - \bar{a}))) =$$

$$= (\overline{\mathrm{orb}}(\bar{a}, y)) @ (\overline{\mathrm{supp}}(x) - \bar{a}) = (\overline{\mathrm{orb}}(\bar{a}, x)) @ (\overline{\mathrm{supp}}(x) - \bar{a}) = x$$

□

Let us now use $\overline{\mathrm{orb}}$ to define the *core ancestor sequence*, which is a more compressed version of the *compact ancestor sequence*:

Definition 29. For every position i, we define *core ancestor sequence* $(\mathrm{core}(i))$ in the following way:

1. If i a regular position, then $\mathrm{cas}(i)$ is defined as the following tuple:

$$\mathrm{core}(i) = (\underbrace{\overline{\mathrm{orb}}\,(\overline{\mathrm{supp}}(g_j),\ (\mathrm{core}(p),\ x_j))}_{\substack{\text{A polynomial orbit-finite representation} \\ \text{of the } \mathrm{supp}(g_j)\text{-orbit of the pair } (\mathrm{core}(p), x_j)}} ,\ g_i,\ y_i)$$

   where j is i's eldest sibling, and p is j's left ancestor. Similarly as in Definition 28, let us note that j always exists, but it might happen that $i = j$. It is, however, possible that j might not have a left ancestor – in this case, we simply skip the $\mathrm{core}(p)$ part.

2. If i is not regular, then:

$$\mathrm{core}(i) = (\mathrm{core}(p), v_i)$$

   where p is its left ancestor of i. Again, if i does not have a left ancestor, we skip the $\mathrm{core}(p)$ part.

◁

The intuition behind this definition is that the more distant an ancestor is, the more of its atoms are forgotten in $\mathrm{core}(i)$. Since $\mathrm{core}(i)$ does not keep all relevant atoms, we can no longer use it to reconstruct the product of the ith prefix. However, if the output function $\lambda : S \to \Gamma$ satisfies the locality equation, we can use $\mathrm{core}(i)$ to reconstruct the λ-value of the ith prefix, i.e. $\lambda(s_1, \ldots, s_i)$:

Lemma 64. *For every two sequences $s_1, \ldots, s_n$, and $s'_1, \ldots, s'_{n'}$ over an orbit-finite semigroup S that are equipped with* monotone *smooth splits, for every i and i' that are positions in s and s', and for every* local *$\lambda : S \to \Gamma$, it holds that:*

$$\mathit{core}(i) = \mathit{core}(i') \quad \Rightarrow \quad \lambda(s_1 \cdot \ldots \cdot s_i) = \lambda(s'_1 \cdot \ldots \cdot s'_{i'})$$

Proof. The proof goes by induction on the position i, but it requires a slight strengthening of the induction hypothesis:

Claim 37. *Let S, i, j, λ, $s_1, \ldots, s_n$ and $s'_1, \ldots, s'_{n'}$ be as in Lemma 64 and let $z \in S$ be a value that is monotone with both v_i and $v'_{i'}$, i.e.:*

$$v_i z \textit{ is infix equivalent to } v_i \quad \textit{and} \quad v'_{j'} z \textit{ is infix equivalent to } v'_{i'}.$$

It follows that:

$$\mathit{core}(i) = \mathit{core}(i') \quad \Rightarrow \quad \lambda(s_1 \cdot \ldots \cdot s_i \cdot z) = \lambda(s'_1 \cdot \ldots \cdot s'_{i'} \cdot z)$$

It is easy to see that the lemma follows from the claim – it suffices to take $S = S^1$ (from Claim 21) and $z = 1$. Let us now proceed with the proof of the claim. First, let us consider the case where i is regular. Since core(i) = core(i') it follows that i' is regular as well, and that:

$$\left(\overline{\text{orb}}\left(\overline{\text{supp}}(g_j), (\text{core}(p), x_j)\right), g_i, y_i\right) = \left(\overline{\text{orb}}\left(\overline{\text{supp}}(g'_{j'}), (\text{core}(p'), x'_{j'})\right), g'_{i'}, y'_{i'}\right)$$

This means that $g_i = g'_{i'}$, $y_i = y'_{i'}$ and

$$\overline{\text{orb}}\left(\overline{\text{supp}}(g_j), (\text{core}(p), x_j)\right) = \overline{\text{orb}}\left(\overline{\text{supp}}(g'_{j'}), (\text{core}(p), x'_{j'})\right)$$

By Claim 35, we know that $g'_{j'}$ is $\mathcal{H}$-equivalent to $g'_{i'}$, and that g_j is $\mathcal{H}$-equivalent to g_i. Since $g_i = g'_{i'}$, it follows that g_j is $\mathcal{H}$-equivalent to $g'_{j'}$. Moreover, by Claim 35 we know that both g_j and $g'_{j'}$ are idempotent. As explained in the proof of Lemma 59, each $\mathcal{H}$-class contains at most one idempotent, which means that $g_j = g'_{j'}$. To simplify the notation, let us use e to represent both of those values. We know that:

$$\overline{\text{orb}}\left(\overline{\text{supp}}(e), (\text{core}(p), x_j)\right) = \overline{\text{orb}}\left(\overline{\text{supp}}(e), (\text{core}(p'), x'_{j'})\right)$$

Thanks to Lemma 63, it follows that there is a supp(e)-permutation π such that $\pi(\text{core}(p')) = \text{core}(p)$ and $\pi(x'_{j'}) = x_j$.

We are now ready to prove that $\lambda(s_1 \cdot \ldots \cdot s_n \cdot z) = \lambda(s'_1 \cdot \ldots \cdot s'_{n'} \cdot z)$. First, notice that $s_1 \cdot \ldots \cdot s_n = s_1 \cdot \ldots \cdot s_p \cdot x_j \cdot g_i \cdot y_i$ (and similarly for s'). This means that we can proof the claim by showing that:

$$\lambda(\overrightarrow{s_p} \cdot x_j \cdot g_i \cdot y_i \cdot z) = \lambda(\overrightarrow{s'_{p'}} \cdot {x'}_{j'} \cdot g'_{i'} \cdot y'_{i'} \cdot z),$$

where $\overrightarrow{s_p} = s_1 \ldots s_p$ (and similarly for $s_{p'}$). We show this in two steps:

$$\lambda(\overrightarrow{s'_{p'}} \cdot {x'}_{j'} \cdot g'_{i'} \cdot y'_{i'} \cdot z) \overset{(1)}{=} \lambda(\pi(\overrightarrow{s'_{p'}}) \cdot \pi({x'}_{j'}) \cdot g'_{i'} \cdot y'_{i'} \cdot z) \overset{(2)}{=} \lambda(\overrightarrow{s_p} \cdot x_j \cdot g_i \cdot y_i \cdot z)$$

Let us first use the locality of λ to show that equation (1) holds. We know that $g'_{i'}$ is $\mathcal{H}$-equivalent to e, which means that $g'_{i'} = e \cdot g'_{i'}$. This means that in order to show (1), it suffices to show that:

$$\lambda(\overrightarrow{s'_{p'}} \cdot {x'}_{j'} \cdot e \cdot g'_{i'} \cdot y'_{i'} \cdot z) = \lambda(\pi(\overrightarrow{s'_{p'}}) \cdot \pi({x'}_{j'}) \cdot e \cdot g'_{i'} \cdot y'_{i'} \cdot z)$$

This is an instance of the locality equation (Definition 19), so since π was chosen to be a supp(e)-permutation, this leaves us with showing that $g'_{i'} \cdot y'_{i'} \cdot z$ is an infix of e. This follows from Lemma 53 and the following facts:

1. $v'_{i'} \cdot z$ is an infix of $v'_{i'}$ (by assumption);

2. $y'_{i'}$ is suffix equivalent to $v'_{i'}$ (by Claim 35);

3. $y'_{i'}$ is prefix equivalent to e (by Claim 35, because $e = g'_{j'}$);

4. $g'_{i'}$ is $\mathcal{H}$-equivalent to e (by Claim 35).

This leaves us with showing (2): Since $\pi(x'_{j'}) = x_j$, $g'_{i'} = g_i = e$, and $y'_{i'} = y_i$, it suffices to proof the following:

$$\lambda(\pi(\overrightarrow{s'_{p'}}) \cdot x_j \cdot e \cdot y_i \cdot z) = \lambda(\overrightarrow{s_p} \cdot x_j \cdot e \cdot y_i \cdot z)$$

Since core$(i) = core(i')$ we know that either both j and j' have a left ancestor, or neither of them has one. If they do not have left ancestors, then the equality is trivially true (as the factors $\pi(\overrightarrow{s'_{p'}})$ and $\overrightarrow{s_p}$ are omitted). When p and p' exist, we prove the equality by applying the induction assumption to:

$$s = s_1, \ldots, s_p, \quad s'' = \pi(s'_1), \ldots, \pi(s'_{p'}), \text{ and } \quad z' = x_j \cdot g_i \cdot y_i \cdot z.$$

This leaves us with showing that:

1. core(p'') = core(p), where p'' is a position in $s'' = \pi(s')$ and p is a position in s, and

2. z' is monotone with both v_p and $v''_{p''} = \pi(v'_{p'})$.

First, it is not hard to see that the function core is equivariant with respect to the input sequence, so since π was chosen to make $\pi(\text{core}(p'))$ equal to core(p), it follows that:

$$\text{core}(p'') = \pi(\text{core}(p')) = \text{core}(p)$$

This leaves us with showing that $v_p \cdot z'$ is infix equivalent to v_p (and similarly for $v''_{p''}$). Since $z' = x_j \cdot g_i \cdot y_i \cdot z$, this follows from the following facts:

1. $y_i \cdot z$ is a prefix of y_i: Thanks to Lemma 53, this can be shown by proving that $y_i \cdot z$ is an infix of y_i, which follows from the assumption that $v_i \cdot z$ is an infix of v_i, combined with Claim 35, which states that y_i is suffix equivalent to v_i.

2. $x_j \cdot g_i \cdot y_i$ is a prefix of x_i: This is because by Claim 35 combined with the smoothness of the split, we know that the product $x_j \cdot g_i \cdot y_i$ is smooth.

3. $v_p \cdot x_j$ is an infix of v_p: Thanks to the monotonicity of the split, we know that $v_p \cdot v_j$ is an infix of v_p and by Claim 35 x_j is prefix equivalent to v_j.

4. $v''_{p''} \cdot x_j$ is an infix of $v''_{p''}$: This is because $v''_{p''} = \pi(v'_{p'})$ and $x_j = \pi(x'_{j'})$, which means that it suffices to show that $v'_{p'} \cdot x'_{j'}$ is an infix of $v'_{p'}$, which can be shown using an analogous argument as in Item 3.

This finishes the proof in the case where i and i' are regular. In the case where i and i' are irregular (since $\text{core}(i) = \text{core}(i)$ we know that i and i' have to be either both regular or both irregular), we can directly apply the induction assumption. □

It follows almost immediately from Lemma 64 that we can transform core ancestor sequences into the output letter output letters:

Claim 38. *There is an equivariant function that transforms core(i) into the ith letter of the output of the local semigroup transduction.*

Proof. The existence of some function that maps core(i) into the ith letter of the output follows immediately from Lemma 64. To see that this function is equivariant, we can construct it as a composition of the following two equivariant relations. The first relation maps core(i) to all splits with an underlined position such that the core of the underlined position is equal to core(i). The second relation is a function that maps a split with an underlined position, into the output letter on that position. Thanks to Lemma 64, we know that this composition results in a function. □

This leaves us with showing how to equip every i with its core(i). First of all, let us notice that thanks to a reasoning similar to the one in Claim 36 combined with Lemma 63, we know that in a split of bounded height, the values of core(i) belong to a polynomial orbit-finite set. Now let us show how to use compositions of primes to compute all core(i):

Lemma 65. *For every height h, there is a composition of primes that inputs a smooth split of height at most h and equips every position i with core(i).*

Proof. We start the construction by applying Claim 35 to equip the input positions with their x-, y-, and g-values. Then, we construct the core-values, level by level top-down: First for the positions of height h, then $h-1, \ldots$, and all the way down to 1. Let us assume that we have already constructed all core-values for the positions that are strictly higher than k (for $k = h$ this is trivially true), and show how to construct core-values for the positions of height k:

First, we construct the core-values for all eldest siblings of height k. For this, let us notice that if i is an eldest sibling, then core(i) depends only on:

1. core(p), where p is the left ancestor of i (if it exists); and

2. the values x_i, g_i, y_i if i is regular, or on v_i if i is not regular.

The values v_i, and x_i, g_i, y_i (if applicable) are already present in all i's, so it suffices to equip every eldest sibling with the core-value of its left ancestor (if it exists). We do this in one round of a (generalized) single-use propagation: Every node higher than k transmits its core-value, and every node of height k tries to receive a value. This construction works, because for every position p there is at most one i of height k such that p is the left ancestor of i. After this propagation, we can use a homomorphism to construct core(j) in every eldest sibling j.

We are left with constructing the core-values in the non-eldest siblings. Notice that if i a non-eldest sibling, then core(i) depends only on:

$$g_i, \quad y_i, \quad \overline{\text{orb}}\,(\overline{\text{supp}}(g_j), (\text{core}(p), x_j)),$$

where j is i's eldest sibling, and p is the left ancestor of j (if it exists). Since the values x_i, g_i are already present in i, we only need to equip every non-eldest sibling with $\overline{\text{orb}}\,(g_j, (\text{core}(p), x_j))$. We start by using a homomorphism to construct this value in every eldest sibling j of height k (this is possible because values x_j and core(p) are already present each such eldest sibling j). Then, we notice that by Lemma 59, Claim 35, and Lemma 63, for every i whose eldest sibling is j, it holds that:

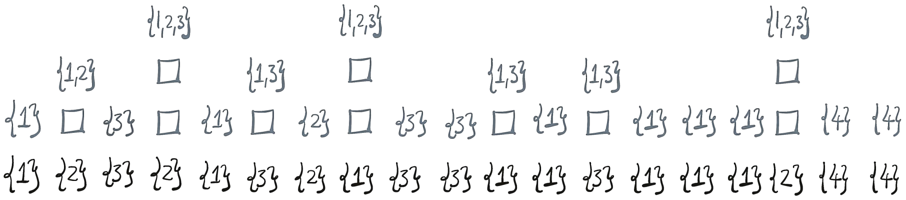
$$\text{supp}\left(\overline{\text{orb}}\,(g_j, (x_j, \text{core}(p)))\right) \quad \subseteq \quad \text{supp}(g_j) \quad = \quad \text{supp}(g_i)$$

This means that we can use multiple use propagation (Lemma 58) together with the map and subsequence combinators to equip every i of height k with $\overline{\text{orb}}\,(g_j, (x_j, \text{core}(p)))$. After this step we have enough information in every non-eldest node i of height k to use a homomorphism and compute its core(i). □

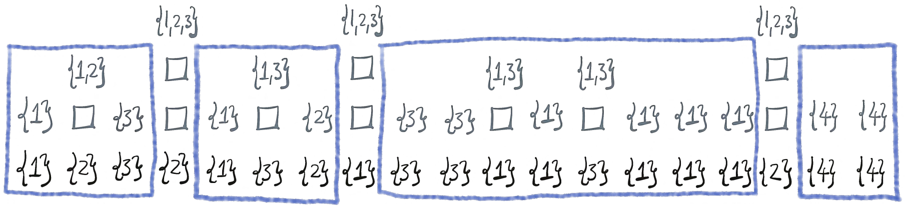
This almost finishes the proof of Lemma 62. The last thing to show is how to transform smooth splits of bounded height, into *monotone* smooth splits of bounded height:

Lemma 66. *For every semigroup S (represented by Σ), and for every height h, there is a composition of primes that inputs a smooth split of height at most h and transforms it into a monotone smooth split (over the same sequence) of height at most $h \cdot (h_{\mathcal{J}}(S) + 1)$, where $h_{\mathcal{J}}(S)$ is the $\mathcal{J}$-height of S.*

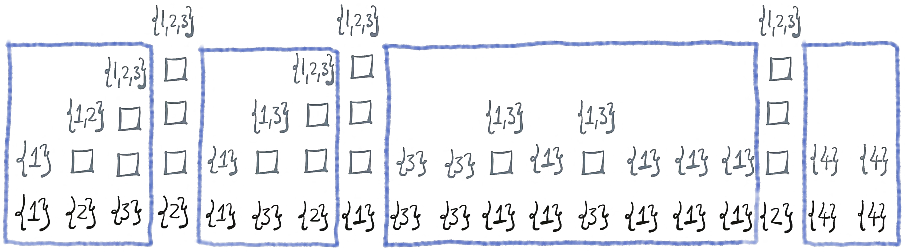
Proof. The proof is an induction on h. The induction base is trivial – splits of height 1 are necessarily smooth, so every smooth split of height 1 is already monotone. For the induction step, we assume that we have a construction for h, and we derive a construction for $h+1$. We are going to illustrate the construction on the following example:



Similarly as in the proof of Claim 35, we notice that the positions of the maximal height divide the input sequence into subsplits of heights not greater than h:



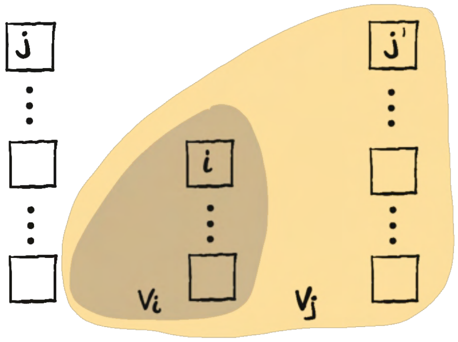
This means that using the map combinator (Lemma 56), we can apply induction assumption to each of those subsplits. This might increase the height of those subsplits from h to $h' := h \cdot (h_{\mathcal{J}}(S) + 1)$. It follows that in order to preserve the structure of the split, we need to increase the heights of the dividing positions from $h + 1$ to $h' + 1$. (We can do this using a homomorphism.) In our example, this looks as follows (for the sake of clarity, we lower h' to 3, which is still high enough to preserve the structure of the split):



Now let us investigate all possible remaining *non-monotone positions*, i.e. all positions i, such that $v_j \cdot v_i$ is not infix equivalent to v_j (where j is the left ancestor of i). Note that it follows from the induction assumption that if (j, i) is such a non-monotone pair, then j is of the maximal height (i.e. $h' + 1$) and i is not of the maximal height. Moreover, this can only happen in the last subsplit (i.e. j has to be the rightmost position of maximal height):

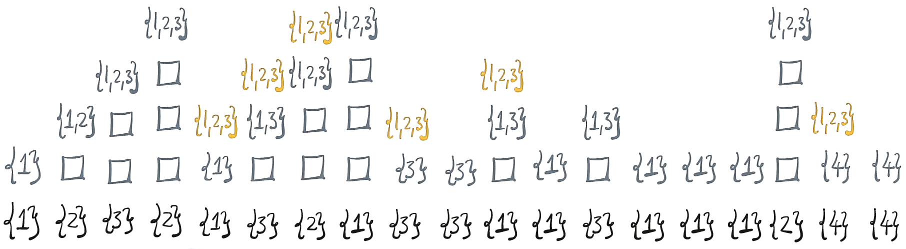
Claim 39. *If the left ancestor of i is not the* youngest sibling *(i.e. it has a sibling to the right), then i is monotone (i.e. $v_j v_i$ is infix equivalent to v_j).*

Proof. Let j be i's left ancestor, and let j' be the first sibling of j (to the right):

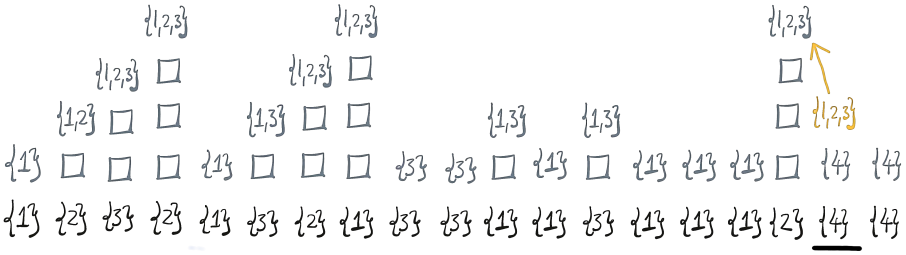


Because the split is smooth, we know that $v_j v_{j'}$ is an infix of v_j. By Lemma 53, it follows that $v_j v_{j'}$ is also a prefix of v_j. By definition of the split values, we see that v_i is a prefix of $v_{j'}$. It follows that $v_j v_i$ is a prefix of $v_j v_j'$. By transitivity, this means that $v_j v_i$ is a prefix of v_j. □

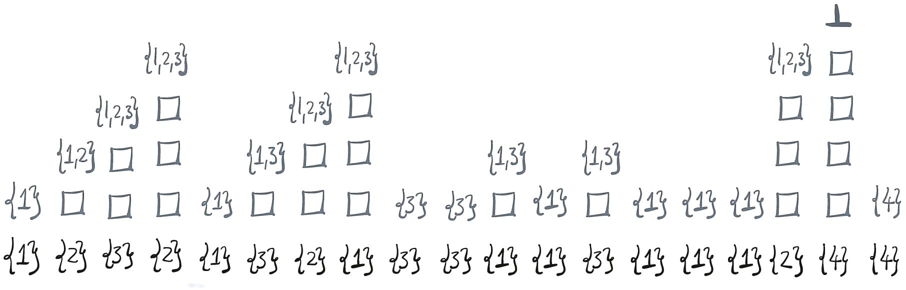
It is easy to see that for every height k and for every position j, there is at most one i of height h such that j is the left ancestor of i. It follows that there are at most h' non-monotone positions. Let us show how to detect them: We start by equipping every position i whose left ancestor j is of the maximal height (i.e. $h' + 1$) with the value v_j. This can be done in h' rounds of the generalized single-use propagation – in the kth round, nodes of height $h' + 1$ transmit their split value and nodes of height k try to receive. This brings us to the following situation (the ancestor split values are marked in orange):



Now we can use a homomorphism to underline all those nodes that have received a v_j; and for which $v_j \cdot v_i$ is not infix equivalent to v_j. In our example there is only one such position:



Then, we use a classical Mealy machine (or a multiple-use bit propagation) to detect the first underlined node, and using a homomorphism, we set its height to $h' + 2$ and its split value to $v_j \cdot v_i$:



It is not hard to see that this operation preserves the smoothness of the split. In our example the split is already monotone, but in general there might still be some non-monotone nodes left. Using a similar reasoning as before, we can see that if a node is non-monotone then its left ancestor has to be the newly elevated position of height $h' + 2$. This means that we apply a similar construction:

1. Equip every position i whose left ancestor j is of the maximal height with the value v_j;
2. Detect the first non-monotone position (i.e. a position for which $v_j \cdot v_i$ is not infix equivalent to v_j);
3. Increment the position's height, and set its split value to $v_j \cdot v_i$.

It is not hard to see that each such operation decreases the $\mathcal{J}$-height of the split value in the position of the maximal height. This means that after $h_{\mathcal{J}}(S)$ iterations, we will obtain a monotone split of height at most $h' + h_{\mathcal{J}}(S)$. As $h' = h \cdot h_{\mathcal{J}}(S) + 1$, this is smaller than $(h + 1) \cdot (h_{\mathcal{J}}(S) + 1)$. □

This finishes the proof of Lemma 62 and, in turn, the proof of Theorem 8.

3.6 Local semigroup transductions revisited

Although local semigroup transductions were introduced as an intermediate model in the proof of Theorem 8, we believe that they are of independent interest. For this reason, we devote this (brief) section to studying them for their own sake. The main result of this section is that the containment problem between general orbit-finite semigroup transductions and local semigroup transduction is decidable. However, this main result is mostly a pretext to delve deeper into the properties of local semigroup transductions.

Theorem 12. *The following problem is decidable:*

`Input :` *A (possibly non-local) orbit-finite semigroup transduction* (S, h, λ).
`Output :` *Is there a* local *semigroup transduction* (S', h', λ')
that defines the same function?

For the sake of brevity, we do not discuss finite representations of orbit-finite set (which is technically required to talk about the decidability of this problem). Instead, we will focus on the intuitive understanding of decidability for sets with atoms. The algorithm presented in the proof is relatively simple and we hope that it will be intuitively clear, that it can work with any reasonable representation[14].

Before presenting the algorithm, let us analyse the structure of semigroup transductions. The main result of this analysis is going to be Lemma 70, which says that if a function $f : \Sigma^* \to \Gamma^*$ is defined by *some* local orbit-finite semigroup transduction, then *all* orbit-finite semigroup transductions that define f are local (as long as their semigroups do not contain unnecessary elements).

3.6.1 λ-morphisms

We start by defining a suitable notion of a morphism between two semigroups with outputs:

Definition 30. A Γ-*coloured semigroup* is a pair (S, λ), where S is a semigroup, and lambda is a function $S \to \Gamma$. A λ-*morphism* between two Γ-coloured morphisms $(S_1, \lambda_1) \to (S_2, \lambda_2)$ is a function $f : S_1 \to S_2$ such that:

1. f is a semigroup morphism, i.e. $f(x \cdot y) = f(x) \cdot f(y)$ for every $x, y \in S_1$; and

2. f preserves colours, i.e. $\lambda_1(x) = \lambda_2(f(x))$ for every $x \in S_1$.

◁

The interesting fact about equivariant λ-morphisms is that they preserve the locality of λ both backwards and (as long as the λ-morphism is a surjection) forwards:

[14] For more information on representing orbit-finite sets see [Boj19, Chapter 4]. For more information on computations on sets with atoms see [Boj19, Chapter 5 and Part III].

Lemma 67. *Let (S_1, λ_1) and (S_2, λ_2) be two orbit-finite, equivariant Γ-coloured semigroups, such that there exists an equivariant λ-epimorphism (i.e. a surjective λ-morphism):*

$$f : (S_1, \lambda_1) \rightarrow_{eq} (S_2, \lambda_2).$$

It follows that λ_1 is local if and only if λ_2 is local.

Proof. We start with the easier ($\Leftarrow$)-implication, which holds even if f not surjective. We assume that λ_2 is local, and show that λ_1 is local as well. For this, we take $x, e, y \in S_1$ and a supp(e)-permutation π such that e is an idempotent and y is a prefix of e, and we show that $\lambda_1(xey) = \lambda_1(\pi(x)ey)$. We start by converting the left-hand side to λ_2:

$$\lambda_1(xey) = \lambda_2(f(xey)) = \lambda_2(f(x) \cdot f(e) \cdot f(y))$$

At this point, we would like to apply the locality of λ_2. For this we observe the following:

1. $f(e)$ is idempotent: This is because $f(e) \cdot f(e) = f(e \cdot e) = f(e)$;
2. $f(y)$ is a prefix of $f(e)$: Since y is a prefix of e, there is a z such that $yz = e$. It follows that $f(y) \cdot f(z) = f(yz) = f(e)$.
3. π is a supp($f(e)$)-permutation: f is equivariant, so by Lemma 3, we know that supp($f(e)$) $\subseteq$ supp(e). It follows that every supp(e)-permutation is also a supp($f(e)$)-permutation.

This means that we can use the locality of λ_2:

$$\lambda_2(f(x) \cdot f(e) \cdot f(y)) = \lambda_2(\pi(f(x)) \cdot f(e) \cdot f(y))$$

Since f is equivariant, we have that:

$$\lambda_2(\pi(f(x)) \cdot f(e) \cdot f(y)) = \lambda_2(f(\pi(x)) \cdot f(e) \cdot f(y))$$

We finish the proof by going back to λ_1:

$$\lambda_2(f(\pi(x)) \cdot f(e) \cdot f(y)) = \lambda_2(f(\pi(x)ey)) = \lambda_1(\pi(x)ey)$$

This finishes the proof of the ($\Leftarrow$)-implication.

The proof of the ($\Rightarrow$)-implication is more complicated and it requires f to be surjective. This time we assume that λ_1 is local and show that λ_2 is local as well: Take $x, e, y \in S_2$ and a supp(e)-permutation π such that e is an idempotent and y is a prefix of e. We need to show that $\lambda_2(xey) = \lambda_2(\pi(x)ey)$. The immediate approach would be to take some x', e' and y' that belong accordingly to $f^{-1}(x)$, $f^{-1}(e)$ and $f^{-1}(y)$ and apply the locality for $\lambda_1(x'e'y')$. There are, however, three problems with this approach: e' might not be idempotent, y' might not be a prefix of x', and π might not be a supp(e') permutation (as e' might contain more atoms than e). In the next paragraphs, we present a solution that deals with all three of those problems.

It would be useful to assume that π touches only finitely many atoms – but, since π is an arbitrary supp(e)-permutation, this might not be true. However, by Claim 23, we can assume that there is another supp(e)-permutation π' that only touches finitely many atoms such that $\pi'(x) = \pi(x)$. For such a π' we have that $\lambda_2(\pi(x)ey) = \lambda_2(\pi'(x)ey)$. This leaves us with proving that $\lambda_2(x_2ey) = \lambda_2(\pi'(x)_2ey)$. We start by picking an appropriate e':

Claim 40. *There is an element $e' \in S_1$, such that:*

1. $f(e') = e$*;*
2. *every* $s \in f^{-1}(e)$ *is an infix of* e'*;*
3. e' *is idempotent;*
4. π' *is a supp*(e')*-permutation.*

Proof. Let us consider $E = f^{-1}(e)$. As long as we pick $e' \in E$, it will satisfy the first condition of the claim. Observe that E is a subsemigroup of S_1: This is because $\{e\}$ is a subsemigroup of S_2, and inverse images of semigroup morphisms preserve subsemigroups. Let E' be the heaviest $\mathcal{J}$-class of E, i.e. the set of all elements whose $\mathcal{J}$-height is equal to 1. (By Claim 24, we know that this is indeed a $\mathcal{J}$-class). Observe that E' is a subsemigroup – this is because the $\mathcal{J}$-height of $x \cdot y$ cannot be higher than the $\mathcal{J}$-height of x or y. Moreover, observe that every element of E is an infix of every element of E': if $x \in E$ and $y \in E'$, then it is not hard to see that $xy \in E'$, which means that xy is $\mathcal{J}$-equivalent to y. This means that as long as we pick e' from E', it will satisfy the second condition. Since E' is a subsemigroup of S_1, it contains at least one idempotent: Take some $x \in E'$ and let X be the subsemigroup generated by x, i.e. $X = \{x, x^2, x^3 \ldots\}$. By [Boj13, Theorem 5.1], we know that X is finite and every finite (sub)semigroup contains an idempotent. Finally, let us notice that E' is supported by supp(e) (this follows from Lemma 3, as E' can be computed from E, and E can be computed from e). Since E' contains an idempotent, it follows that there is at least one supp(e)-orbit of idempotents in E'. Moreover, we know that π' is a supp(e)-permutation that only touches finitely many atoms, which means that this orbit contains at least one (or, in fact, infinitely many) e', such that π' is a supp(e')-permutation. Such e' satisfies the conditions of the claim. □

We are now ready to show that $\lambda_2(x_2ey) = \lambda_2(\pi'(x)_2ey)$. Let us pick e' as in Claim 40 and some x' and y' such that $f(x') = x$ and $f(y') = y$. Observe that:

$$\lambda_2(xey) = \lambda_2(xe(ey)) = \lambda_2(f(x'e'(e'y'))) = \lambda_1(x'e'(ey'))$$

In the next step, we apply the locality of λ_1 for x', e', ey' and π'. We already know that e' is an idempotent (in S_1) and that π' is a supp(e')-permutation. Let us show that $e'y'$ is a prefix of e': By Lemma 53, it suffices to show that

$e'y'$ is an infix of e'. Since y is a prefix of e, we know that there exists a $z \in S_2$ such that $yz = e$. Let z' be any element such that $f(z') = z$. It follows that $f(e'y'z') = eyz = ee = e$, so by Claim 40, we know that $e'y'z'$ is an infix of e'. It follows that $e'y'$ is an infix of e. This means that we can apply the locality equation:

$$\lambda_1(x'e'(ey')) = \lambda_1(\pi'(x') \cdot e' \cdot (ey'))$$

We finish the proof with the following transformations:

$$\lambda_1(\pi'(x') \cdot e' \cdot (ey')) = \lambda_1(\pi'(x') \cdot e' \cdot y') = \lambda_2(f(\pi(x') \cdot e' \cdot y')) = \lambda_2(\pi'(x)ey)$$

□

3.6.2 Syntactic semigroup transduction

Syntactic semigroups are a well-established tool for studying formal languages – they were introduced[15] in [Sch55, Chapter 2], and they are discussed for example in [Pin10, Section IV.4] or in [Boj13, Theorem 1.7]. In this section, we discuss their generalization for word-to-word functions. We start with a definition:

Definition 31. We say that a length-preserving function $f : \Sigma^* \to \Gamma^*$ is *future independent* if for all $w, v_1, v_2 \in \Sigma^*$:

$$\text{the } |w|\text{-th letter of } f(wv_1) = \text{the } |w|\text{-th letter of } f(wv_2)$$

◁

It is not hard to see that the class of word-preserving, future-independent functions is the equal to the class of functions recognized by possibly infinite semigroup transductions (as we can always pick the free semigroup $S = \Sigma^*$ together with λ that maps a word w into the last letter of $f(w)$). For this reason, we are only going to define syntactic semigroup transduction for functions that are length-preserving and future independent.

When talking about syntactic semigroups, it is important to assume that the underlying semigroup of a semigroup transduction does not contain unreachable elements:

Definition 32. We say that a semigroup transduction (S, h, λ) of type $\Sigma^* \to \Gamma^*$ is *full*[16] if h^* is a surjective homomorphism: i.e. if every element of S corresponds to some word from Σ^*, i.e. for every $s \in S$, there is a $w \in \Sigma^*$, such that:

$$s = h(w_1) \cdot \ldots \cdot h(w_n)$$

◁

[15] It might be worth noting that [Sch55] points further to [Dub41]. However, as I was unable to access [Dub41], I keep [Sch55] as the reference.

[16] An alternative for the name *full* semigroup transduction might be *surjective* semigroup transduction. However, the name *surjective semigroup transduction* might erroneously suggest that the word-to-word function $\Sigma^* \to \Gamma^*$ is surjective. For this reason, we stick with *full semigroup transduction*.

Note that every semigroup transduction can be transformed into an equivalent full transduction:

Claim 41. *For every semigroup transduction (S, h, λ), there is an equivalent full-semigroup transduction (S', h', λ'). (Moreover, if (S, h, λ) is equivariant, then so is (S', h', λ).)*

Proof. Define $S' \subseteq S$ to be the set of those elements that correspond to words from Σ^*. It is not hard to see that S' is a subsemigroup of S. If we define h' and λ' to be restrictions of h and λ, we obtain an equivalent semigroup transduction. (It is easy to see that this construction preserves equivariance.) □

We are now ready to define the syntactic semigroup transduction for a function f, which intuitively is the minimal semigroup transduction that computes f:

Lemma 68. *For every length-preserving and future-independent function $f : \Sigma^* \to \Gamma^*$, there exists* syntactic semigroup transduction *(S_f, h_f, λ_f), such that:*

1. *(S_f, h_f, λ_f) is full;*
2. *(S_f, h_f, λ_f) defines f; and*
3. *for every* full *(S', h', λ') that computes f there exists a λ-epimorphism $g : (S', \lambda') \to (S_f, \lambda_f)$. (Note that since g is a λ-morphism, it follows that $(g \circ h', S_f, \lambda_f)$ is equivalent to (S_f, h_f, λ_f), which means that $(g \circ h', S_f, \lambda_f)$ is an implementation of f.)*

(It is worth pointing out that the syntactic semigroup transduction does not have to be finite or orbit-finite.)

Proof. The proof is analogous to the proof of [Boj20, Theorem 1.7]: For every f, we define the two-sided Myhill–Nerode equivalence relation $\sim_f$, which identifies two words $w_1, w_2 \in \Sigma^*$ if:

$$\text{the last letter of } f(uw_1v) = \text{the last letter of } f(uw_2v), \quad \text{for all } u, v \in \Sigma^*$$

The syntactic semigroup of f is constructed as follows: $S_f = (\Sigma^+)_{/\sim_f}$ (i.e. the set of non-empty words over Σ divided by $\sim_f$), with the following operation:

$$[u]_f \cdot [v]_f = [uv]_f,$$

where $[u]_f$ denotes the abstraction class of u. Functions h_f and λ_f are defined as follows:

$$h_f(w) = [w]_f \qquad \lambda_f([w]_f) = \text{the last letter of } f(w)$$

It is not hard to see that definitions of λ and of the semigroup operation do depend on the choice of the representants. It is also not hard to see that (S_f, h_f, λ_f) implements f. For every full (S', h', λ') that implements f, we define the following g:

$$g(x) = [w_x]_f \quad \text{where } w_x \text{ is any word whose } h'\text{-image's product is equal to } x$$

To see that g is a well-defined, surjective, and a monoid morphism, we can use the same argument that is used in the classical construction of a semantic monoid of a language (see, for example, [Boj20, Theorem 1.7]). This leaves us with showing that g preserves λ-values. Since both (S', h', λ') and (S_f, h_f, λ_f) are implementations of f, it follows that:

$$\lambda'(s) = \text{the last letter of } f(w_s) = \lambda_f([w_s]_f) = \lambda_f(g(s))$$

□

It can be shown that the syntactic semigroup transduction of f is unique (with respect to λ-isomorphisms), which means that we can define it abstractly using only the statement of Lemma 68. However, we stick with the concrete definition using the construction from the proof of Lemma 68 (i.e. $S_f = \Sigma^+_{/\sim_f}$) as it is sometimes easier to reason about it. For example, by analysing the construction, it is not hard to see that it preserves equivariance:

Claim 42. *If $f : \Sigma^* \to \Gamma^*$ is equivariant, then so is its syntactic semigroup transduction (S, h, λ). Moreover for every equivariant full (S', h', λ') that corresponds to f, the λ-epimorphism $g : (S', \lambda') \to (S, \lambda)$ is equivariant.*

In the context of formal languages, the syntactic monoid often serves as a useful tool for checking if a language possesses certain properties (such as first-order definability). A similar approach can be used for syntactic semigroup transductions and the locality restriction:

Lemma 69. *An equivariant function $f : \Sigma^* \to_{eq} \Gamma^*$ is recognized by some local semigroup transduction if and only if its syntactic semigroup transduction is local. (Remember that by Definition 19 every local semigroup transduction is in particular orbit-finite.)*

Proof. The “only if” part is immediate. Let us focus on the “if” part: Let (S, h, λ) be the local semigroup transduction that recognizes f. As described in Claim 41, we can transform it into an equivalent full (S', h', λ'). It is not hard to see that (S', h', λ') is local as well. Let (S_f, h_f, λ_f) be the syntactic semigroup transduction of f. It follows by Lemma 68, and Claim 42 that there is an equivariant, surjective λ-morphism $k : (S'_1, \lambda'_1) \to_{\text{eq}} (S_f, \lambda_f)$. Observe that S' is orbit-finite (because it is local). Since k is surjective, it follows that S_f is orbit-finite as well. By Lemma 67, λ_f is local. □

Interestingly, instead of checking whether the syntactic semigroup transduction of a function f is local, it suffices to check whether *some* full semigroup transduction that recognizes f is local. This is formalized by the following lemma which follows directly from combining Lemma 69 with Lemmas 67 and 68:

Lemma 70. *If $f : \Sigma^* \to \Gamma^*$ is recognized by some local semigroup transduction, then all full semigroup transductions that recognize f are local.*

We are now ready to prove Theorem 12. As noted in the introduction to this section, in the proof we are going to use an informal intuition of what it means to be a computable function over sets with atoms. For a formal definition, see [Boj13, Chapter 8].

Proof of Theorem 12. We are given an orbit-finite semigroup transduction (S, h, λ) and we want to check if it can be implemented as a local semigroup transduction. Thanks to Lemma 70 we can do this in the following two steps:

1. In the first step we, we use the construction from Claim 32 to compute an equivalent full-semigroup transduction (S', h', λ'). This can be done using the following fixpoint algorithm. Initiate $S'_0 = h(\Sigma)$, and keep computing S'_{i+1} as follows:
$$S'_{i+1} = S'_i \cup \{x \cdot y \mid x, y \in S'_i\},$$
until $S'_i = S'_{i+1}$. When this process stabilizes, set $S' := S'_i$.

   To see that this procedure terminates, observe that if S_i and S_{i+1} have an equal number of orbits, then $S_i = S_{i+1}$. Consequently, the running time of this procedure is limited by the number of orbits in S.

2. In the second step, we check if (S', h', λ') is local: For every tuple $(x, x', y, z) \in S^4$ such that yz is an idempotent and x and x' belong to the same $\text{supp}(yz)$-orbit, we verify that:
$$\lambda(xyz) = \lambda(x'yz)$$
   Since λ and S are equivariant, it suffices to check this condition for only one representant of every orbit in S^4. By Lemma 6, S^4 is orbit-finite, which means that we can do this in finite time.

□

We conclude our discussion of local semigroup transition with the following lemma, which underlines the connection between local semigroup transduction and single-use Mealy machines:

Lemma 71. *A full semigroup transduction over polynomial orbit-finite alphabets is equivalent to some single-use Mealy machine, if and only if it is local.*

Proof. The "if" part follows directly from Lemma 9. Let us focus on the "only if" part: Take a full (S, h, λ) that is equivalent to some single-use Mealy machine. It follows by Lemma 9 that (S, h, λ) it is equivalent to some local semigroup transduction. By Lemma 70 it follows that (S, h, λ) is local itself. □

A corollary of Theorem 12 and Lemma 71 is that it is decidable whether an orbit-finite semigroup transduction can be translated into a single-use Mealy machine.

3.7 Rational transductions with atoms and their Krohn-Rhodes decompositions

In this section, we discuss rational transduction (i.e. the class of transductions defined by unambiguous Mealy machines – see the introduction to this chapter for details) and their possible extension to polynomial orbit-finite alphabets. The main result of this section is a Krohn-Rhodes-like decomposition theorem for this extension. Apart from being of its own significance, this result plays an important role in the next chapter.

One possible approach to define rational transductions over polynomial orbit-finite alphabets would be to use *unambiguous single-use Mealy machines*. However, unambiguity is a form of nondeterminism, and so far we do not have a good notion of nondeterminism compatible with the single-use restriction (see Section 2.4 for details[17]). For this reason, we define rational transductions with atoms using an algebraic approach. Before discussing the definition for infinite alphabets, we start by recalling the classic algebraic definition[18] for rational transductions over *finite* alphabets:

Definition 33. A *rational semigroup transduction* of type $\Sigma^* \to \Gamma^*$ (for finite Σ and Γ) consists of a finite semigroup S, an input function $h : \Sigma \to S$, and an output function λ:

$$\lambda : \quad \underbrace{(S+\vdash)}_{\substack{\text{prefix product}\\ \vdash \text{ represents empty prefix}}} \quad \times \quad \underbrace{\Sigma}_{\text{current letter}} \quad \times \quad \underbrace{(S+\dashv)}_{\substack{\text{suffix product}\\ \dashv \text{ represents empty suffix}}} \quad \to \quad \underbrace{\Gamma}_{\text{output letter}}$$

The rational monoid transduction defines the function $f : \Sigma^* \to \Gamma^*$, where the ith letter of $f(w)$ is equal to:

$$\lambda(h(w_1) \cdot \ldots \cdot h(w_{i-1}),\ w_i,\ h(w_{i+1}) \cdot \ldots \cdot h(w_n))$$

◁

Example 39. For example, let us construct a rational semigroup transduction of type $\Sigma^* \to \Sigma^*$ that defines the function:

"Swap the first and the last letter"

[17] It is worth noting that both examples from Section 2.4, which demonstrate that nondeterministic single-use automata are stronger than deterministic ones, use automata that are ambiguous (which means that some accepted words will always have more than one accepting run). It follows that those examples cannot be used to show that unambiguous automata are stronger than deterministic ones. In fact, the question of whether unambiguous nondeterministic automata are equivalent to deterministic single-use automata remains open. If the two models turned out to be equivalent, it would open a path to a machine-based definition of single-use rational transductions.

[18] I was unable to find this definition in the literature. However, it is consistent with the field's folklore, as it can be viewed as a semigroup version of Eilenberg's bimachine [Eil74, Section XI.7].

The transduction is based on the semigroup $S = \Sigma^2$, with the following operation:

$$(x_1, y_1) \cdot (x_2, y_2) = (x_1, y_2)$$

The input and output functions of the transduction are as follows:

$$h(a) = (a, a) \qquad \lambda(p, a, s) = \begin{cases} y_s & \text{if } p = \dashv \text{ and } s = (x_s, y_s) \\ x_p & \text{if } s = \vdash \text{ and } p = (x_p, y_p) \\ a & \text{otherwise} \end{cases}$$

$\triangleleft$

For finite alphabets, rational semigroup transductions define the class of rational transductions:

Lemma 72. *Rational semigroup transductions define the same class of functions as unambiguous Mealy machines.*

Proof. $\subseteq$: Observe that a rational semigroup transduction can be computed by a composition of three deterministic right-to-left and left-to-right Mealy machines – the first one (right-to-left) computes the semigroup product of each suffix, the second one computes the product of each prefix (left-to-right), and the third one (right-to-left or left-to-right) computes the output letters. This finishes the proof, because by reasoning similar as in Lemma 44 unambiguous Mealy machines are closed under compositions with both left-to-right and right-to-left Mealy machines.

$\supseteq$: In order to translate an unambiguous Mealy $\mathcal{A}$, we can use the following semigroup of behaviours. The behaviour of a word Σ^* is the following relation $b_w \subseteq Q \times Q$:

$$(q_1, q_2) \in b_w \quad \Leftrightarrow \quad \begin{array}{c}\mathcal{A} \text{ has a run over } w \\ \text{that enters in } q_1 \text{ on the left,} \\ \text{and exits in } q_2 \text{ on the right}\end{array}$$

Similarly, as it was for the behaviour functions, the behaviour of concatenation is a composition of behaviours: $b_{uv} = b_v \circ b_u$. This means that the set of all behaviours forms a semigroup. It is not hard to see that the i-th output letter can be computed based on the behaviour of the $(i-1)$-th prefix, the behaviour of the $(i+1)$-th suffix, and the i-th input letter (the unambiguity restriction guarantees that there is only one possible output letter). □

Now let us define *local rational semigroup transductions*, which is an extension of rational monoid transductions to orbit-finite alphabets. To the best of my knowledge, this definition is an original contribution of this thesis. Similarly as in Definition 19, the key idea is to restrict the power of λ with a locality equation:

Definition 34. A *local rational semigroup transduction* is a version of rational semigroup transduction, where Σ, Γ, and S are orbit-finite, h and λ are equivariant, and λ satisfies the following locality equation for all $x_1, x_2, y_1, y_2, e \in S$ such that e is an idempotent, for all $a \in \Sigma$, and for all supp(e)-permutations π:

$$e = y_1 h(a) y_2 \quad \Rightarrow \quad \lambda(x_1 \cdot y_1,\ a,\ y_2 \cdot e \cdot y_2) = \lambda(\pi(x_1) \cdot e \cdot y_1,\ a,\ y_2 \cdot e \cdot \pi(y_2))$$

A local rational monoid transduction defines a function $\Sigma^* \to_{\mathrm{eq}} \Gamma^*$ (defined in the same way as in Definition 33). It is worth pointing out that the locality restriction does not restrict the values $\lambda(\vdash, \cdot, \cdot)$ or $\lambda(\cdot, \cdot, \dashv)$. The intuitive reason for this is that those values are computed only once, and only repetitive behaviours create obstacles for the single-use restriction. $\triangleleft$

Example 40. The transduction "swap the first and the last letter" from Example 39 is a local rational semigroup transduction for every orbit-finite alphabet Σ. Since the locality restriction only talks about situations where both prefix and suffix are real semigroup elements (and not $\dashv$ or $\vdash$), the locality restriction is trivially satisfied:

$$\lambda(x_1 \cdot e \cdot y_1, a, x_2 \cdot e \cdot y_2) = a = \lambda(\pi(x_1) \cdot e \cdot y_1, a, x_2 \cdot e \cdot \pi(y_2)).$$

$\triangleleft$

Finally, let us explore the relationship between semigroup transductions and local rational semigroup transductions:

Claim 43. *The class of local rational semigroup transductions that are future independent (see Definition 31) is equal to the class of local semigroup transactions.*

Proof. We start the proof by observing that a rational transduction is future independent if and only if for every x, a, y, y', it holds that:

$$\lambda(x, a, y) = \lambda(x, a, y')$$

The $\subseteq$ inclusion is easy: In order to transform a local monoid transduction into a local rational semigroup transduction, we can use the following output function:

$$\lambda'(x, a, y) = \lambda(x \cdot h(a))$$

It is easy to see that λ' is future independent. Moreover, locality of λ implies the locality of λ'. This is because if $y_1 \cdot h(a) \cdot y_2 = e$, then $y_1 \cdot h(a)$ is a prefix of e.

The proof of $\supseteq$ is similar. The main difference is that we need to define another semigroup S' that keeps track of the last letter of a word. We define it as $S' = S \times \Sigma$ with the following operation:

$$(x_1,\ a_1) \cdot (x_2,\ a_2) = (x_1 \cdot h(a_1) \cdot x_2,\ a_2)$$

Now, we define $h'(a) = (1, a)$ and $\lambda'((x, a)) = \lambda(x, a, 1)$. Thanks to the future independence of λ, we know that the (S, h', λ') defines the same function as (M, h, λ). Moreover, using a similar idea as in the proof of $\subseteq$, we can show that λ' satisfies the locality equation. $\square$

3.7.1 Rational Krohn-Rhodes decompositions

In this section, we formulate and prove a version of the Krohn-Rhodes theorem for local semigroup transductions. Observe that all classical prime functions (see Theorem 7) and single-use prime functions (see Theorem 8) except for the homomorphisms are left-to-right oriented. The rational version of Krohn-Rhodes theorem extends the set of prime functions with their right-to-left counterparts. Let us start with the classical version of the theorem for finite alphabets[19]:

Theorem 13. *The class of rational transductions (over finite alphabets) is equal to the smallest class closed under* $\circ$ *and* $\times$*, that contains the following* rational prime functions*:*

1. *Length-preserving homomorphism* $h^* : \Sigma^* \to \Gamma^*$*, for every* $h : \Sigma \to \Gamma$*, where* Σ *and* Γ *are finite.*

2. *Left-to-right multiple-use bit propagation (from Example 30) and right-to-left multiple-use bit propagation (analogous);*

3. *The G-prefix function (from Example 31) and the G-suffix*[20] *function (analogous) for every finite group G.*

The following Krohn-Rhodes theorem for local rational semigroup transductions is the main result of this section. To the best of my knowledge, it is an original contribution of this thesis.

Theorem 14. *The class of local rational semigroup transductions over polynomial orbit-finite alphabets*[21] *is equal to the smallest class closed under* $\circ$ *and* $\times$*, that contains the following* single-use rational prime functions*:*

1. *All rational prime functions over finite alphabets (from Theorem 13);*

2. *Length-preserving equivariant homomorphism* $h^* : \Sigma^* \to \Gamma^*$*, for every equivariant* $h : \Sigma \to_{eq} \Gamma$*, where* Σ *and* Γ *are polynomial orbit-finite;*

[19] The theorem belongs to folklore. It follows immediately from the Elgot-Mezei theorem ([EM63, Theorem 7.8], see introduction to this chapter for details) combined with the Krohn-Rhodes theorem (Theorem 7).

[20] Actually, it can be shown that we do not need the G-suffix function, as we can derive it from the other rational primes. However, the G-suffixes function does not cause any problems later on, and keeping it makes the formulation more symmetrical.

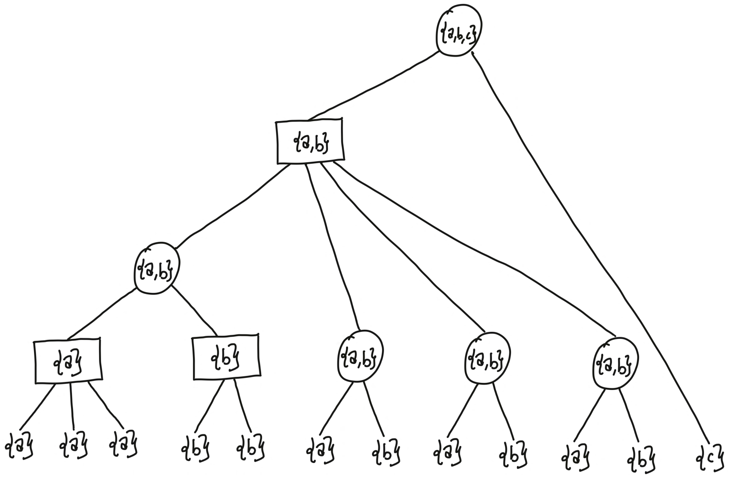
[21] Note that local rational semigroup transductions are defined for all orbit-finite alphabets, but the theorem only holds for polynomial orbit-finite alphabets. A counterexample is the single-use propagation of $\binom{\mathbb{A}}{2}$, which can be constructed as a local rational semigroup transduction but not as a composition of single-use rational primes. One way to address this issue would be to extend the set of single-use rational primes with the generalized single-use propagation for every orbit-finite Σ (i.e. an extended version of the function from Claim 26). However, the current proof of the theorem only works for polynomial orbit-finite alphabets, leaving the question of whether compositions of these generalized primes are equivalent to local rational semigroup transductions over orbit-finite alphabets open. A similar (but simpler) open question can also be asked about Krohn-Rhodes decomposition of local semigroup transductions.

3. *Left-to-right single-use atom propagation (Example 29) and right-to-left single-use atom propagation (analogous).*

The remainder of this section is dedicated to the proof of Theorem 14.

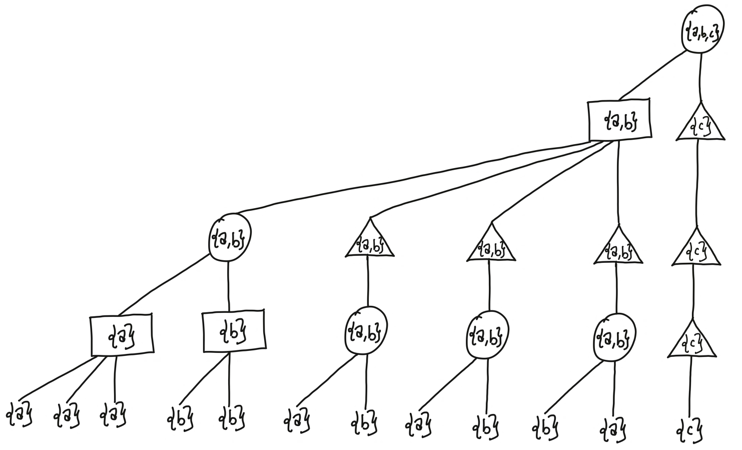
3.7.1.1 Local rational transductions $\subseteq$ Compositions of rational primes

We begin the proof of Theorem 14 by showing that all local rational semigroup transductions can be constructed as compositions of (single-use rational) primes. Thanks to Theorem 8, it is enough to show that we can construct every local rational monoid transduction as a composition of left-to-right and right-to-left single-use Mealy machines[22]. The proof follows the approach of Lemma 62: First, we construct a smooth factorization of the input sequence, and then we show how to transform it into the output of the local rational semigroup transduction. This time, leveraging the more powerful computation model, we do not construct smooth splits. Instead, we directly construct smooth factorization trees using the following *right-aligned encoding* of trees. We illustrate the encoding using the following example:



The key idea of the encoding is to write every node in the position of its rightmost descendant. To make things cleaner, we also assume that all leaves are on the same depth – for this reason, we introduce unary nodes. Here is a right-aligned version of the example above (with the unary nodes marked as triangles):

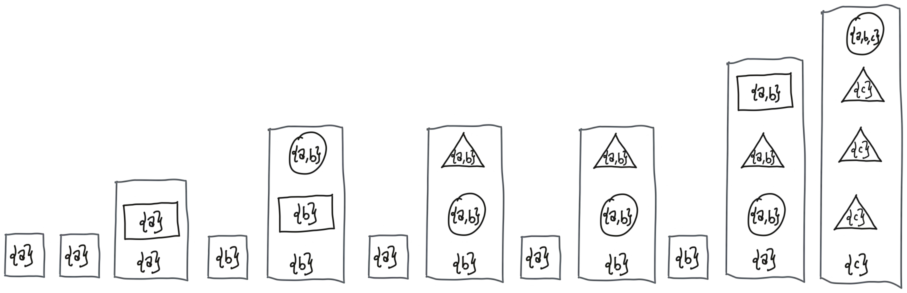
[22] For finite alphabets, it was enough to use one left-to-right and one right-to-left Mealy machine. In contrast, in the proof of Theorem 14, we are going to use multiple left-to-right and multiple right-to-left single-use Mealy machines. It remains an open question whether every local rational semigroup transduction can be constructed using one left-to-right and one right-to-left single-use Mealy machine.



Observe that in a right-aligned tree of height h, each position contains at most h nodes. It follows that we can encode right-aligned trees as words over the following alphabet (where Σ is a polynomial orbit-finite representation of the semigroup):

$$\left(\underbrace{\Sigma}_{\text{leaf}} + \underbrace{\Sigma}_{\text{binary node}} + \underbrace{\Sigma}_{\text{smooth node}} + \underbrace{\Sigma}_{\text{unary node}}\right)^{\leq k}$$

Our example tree corresponds to the following word:



Such a word contains enough information to recreate the original tree: the parent of a node q is the first node to the right of q whose height is $h_q + 1$ (where h_q is q's height).

Let us now prove that we can use compositions of rational primes to construct smooth factorization trees:

Lemma 73. *Let S be an orbit-finite semigroup, and let Σ be its polynomial orbit-finite representation. There exists h and a function f_{tree} of the following* *type that can be constructed as a composition of rational single-use primes:*

$$f_{tree} : \Sigma^* \to (\underbrace{(\Sigma + \Sigma + \Sigma + \Sigma)^{\leq k}}_{\substack{\textit{The alphabet for representing} \\ \textit{right-aligned smooth factorications} \\ \textit{(described earlier)}}})^*,$$

such that f_{tree} outputs a right-aligned smooth factorization tree for the input sequence.

Proof. The proof is analogous to the proof of Lemma 49. It is an induction on the maximal $\mathcal{J}$-height of the input elements. If all elements have $\mathcal{J}$ heights equal to 1 then, by Claim 24, we know that the input sequence is smooth. This means that we can use Lemma 57 to compute its product, and construct a smooth factorization tree by inserting a smooth root in the last position. Note that we can detect the last node using a right-to-left Mealy machine – this is not possible using only left-to-right Mealy machines.

This leaves us with the induction step: We start by partitioning the input sequence into maximal smooth blocks, i.e. smooth blocks that would lose their smoothness if extended by one element left or right. The construction is similar to Lemma 52, but without the "keep every other underline" phase. Instead, there is an additional step, where a right-to-left Mealy machine shifts all underlines one position to the left. Next, we use a construction similar to Lemma 57 to compute the product of each of those blocks. Then, for each block, we insert a smooth node that groups its elements together – with right-aligned encoding, this means inserting a smooth node in the last position of each smooth block (for blocks of length one we use a unary node instead of a smooth one). Next, we insert binary nodes to group the new nodes in pairs (if their number is odd, we use two binary nodes to group the last three nodes together). We can do this with a single-use Mealy machine that keeps track of the parity and remembers enough copies of the previous value to compute binary products (this is possible thanks to Lemma 28). Now, observe that those newly inserted binary nodes contain values whose $\mathcal{J}$-heights are strictly lower than the maximal $\mathcal{J}$-height of the input sequence. This means that we can finish the construction by combining the induction assumption with the subsequence combinator from Lemma 54. □

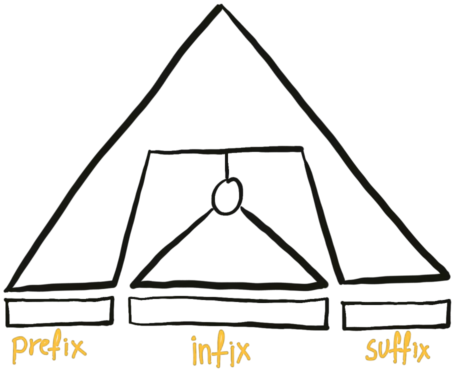
We are now ready to prove the $\subseteq$-inclusion of Theorem 14. For this we fix a local rational semigroup transduction $f = (S, h, \lambda)$ of type $\Sigma^* \to \Gamma^*$ and we show how to construct it as a composition of left-to-right and right-to-left single-use Mealy machines. Observe first, that by a similar reasoning as in the proof of Lemma 62 combined with the reasoning from the proof of the $\supseteq$-inclusion in Lemma 43, we can assume that $h : \Sigma \to S$ is a polynomial orbit-finite representation of S. This means that we can use Lemma 73 to construct a smooth factorization tree over the input sequence. In the remainder of this section, we show how to transform the smooth factorization tree into the output of the transduction.

Before we continue with the proof, let us present a few definitions: First, we define a *pointed word* (denoted as $\underline{w}$) to be a word with one underlined letter. We define the output of a pointed word $\underline{w}$ (with respect to the fixed local rational semigroup transduction f) to be the i-th letter of $f(w)$, where i is the index of the underlined letter in $\underline{w}$, and w is the word $\underline{w}$, but without the underline. Finally, we define the *profile* of $\underline{w}$ to be the following element of $S^1 \times \Sigma \times S^1$ (remember that S^1 is defined in Claim 21 – we use it to handle the case where the prefix or the suffix is an empty word):

$$(h(w_1) \cdot \ldots \cdot h(w_{i-1}),\ w_i,\ h(w_{i+1}) \cdot \ldots \cdot h(w_n))$$

It is easy to see that the output of a pointed word depends entirely on its profile.

Let us now consider an input sequence $s_1, \ldots, s_n \in \Sigma^*$ equipped with a smooth factorization tree. Notice that every node of this tree splits the input sequence into a suffix, an infix, and a prefix:



We say the *context* of a node is the following pair from $S^1 \times S^1$:

$$(\text{product of the prefix},\ \text{product of the suffix})$$

Observe that if q is a node that contains the i-th position of the input, then the i-th position of the output depends entirely on the context of q, and on the profile of q's infix with i as the underlined position. Moreover, observe that if q is a leaf, then its infix consists of a single letter s_i, whose profile is equal to $(1,\ s_i,\ 1)$. This means that the i-th output letter depends only on s_i and on the context of the i-th leaf. This means that if we were able to use compositions of primes to compute the context of every node (or even every leaf), then we would also be able to compute the output word. Unfortunately, compositions of rational primes are unable to compute all contexts. The reason for this is analogous to the reason why compositions of left-to-right primes cannot compute the products of all prefixes (see Example 33). Instead, we define the *reduced context* of a node which is an analogue of the core ancestor subsequences from Definition 29. We start the definition by formulating an analogue of Claim 35. (We skip the proof, as it is analogous to Claim 35.)

Claim 44. *There is a composition of rational primes that equips every smooth sequence $s_1, \ldots, s_n \in \Sigma^*$ with the (representations) of $x_i, y_u, \overrightarrow{g_i}, \overleftarrow{g_i}$, such that:*

1. *the values x_i, y_i are a decomposition of s_i, i.e. $x_i \cdot y_i = s_i$;*
2. *the values $x_1, \overrightarrow{g_i}, y_i$ can be used to compute the i-th prefix, i.e.*
$$x_1 \cdot \overrightarrow{g_i} \cdot y_i = s_1 \cdot \ldots s_i$$
3. *the values $y_i, \overleftarrow{g_i}, x_n$ can be used to compute the i-th suffix, i.e.*
$$x_i \cdot \overleftarrow{g_i} \cdot y_n = s_i \cdot \ldots s_n$$
4. *all $\overrightarrow{g_i}$s and $\overleftarrow{g_i}$s are pairwise $\mathcal{H}$-equivalent;*
5. *$\overrightarrow{g_1} = \overleftarrow{g_n}$ and they are both idempotent;*
6. *all x_i's are suffix-equivalent to $\overrightarrow{g_1} = \overleftarrow{g_n}$, and all y_i's are prefix-equivalent to $\overrightarrow{g_1} = \overleftarrow{g_n}$.*
7. *for all $i > 1$, it holds that $\overrightarrow{g_i} = \overrightarrow{g_{i-1}} \cdot (y_{i-1} \cdot x_i)$, and for all $i < n$, it holds that $\overleftarrow{g_i} = (y_i \cdot x_{i+1}) \cdot \overleftarrow{g_{i+1}}$.*

Note the claim talks about finitely supported primes and not about equivariant primes. However, we do not have to worry about that, because in the end we will be able to remove the unnecessary atoms from our construction, using Lemma 61 (which can be easily extended to compositions of rational primes). We are now ready to define *reduced contexts*:

Definition 35. The *reduced context* (denoted as $\mathrm{ctx}(q)$) of a node q is defined as follows:

1. If q is the root of the tree, then its reduced context is empty.
2. If q is a child of a unary node r, then its reduced context is equal to the reduced context of r.
3. If q is a child of a binary node r, and q' is its sibling then the reduced context of q is equal to $(\mathrm{ctx}(r), v_{q'})$, where $v_{q'}$ is the value of the node q'.
4. If q is the first child of a smooth node r, whose children are $q_1, \ldots, q_n$ (i.e. $q = q_1$), then its reduced context t contains the reduced context of r, and the (smooth) product of the rest of its sibling values, i.e.:
$$(\mathrm{ctx}(r), v_{q_2} \cdot \ldots \cdot v_{q_n})$$
If q is the last child of a smooth node r, then the context of q is defined analogously.

5. If q is an inner child of a smooth node r, whose children are $q_1, \ldots, q_n$, i.e. $q = q_i$ for some $1 < i < n$, then its reduced context is equal to the following tuple:

   $$\overline{\mathrm{orb}}_e(x_1, y_n, \mathrm{ctx}(r)),\ \overrightarrow{g_{i-1}}, y_{i-1}, \overleftarrow{g_{i+1}}, x_{i+1},$$

   where $x, y, \overrightarrow{g}, \overleftarrow{g}$ are the values defined by Claim 44 for the smooth sequence $v_{q_1}, \ldots, v_{q_n}$, and e is the only idempotent that is $\mathcal{H}$-equivalent to $\overrightarrow{g_i}$. (As explained in the proof of Lemma 59, each $\mathcal{H}$-class has at most one idempotent, which means that e is well-defined.)

Moreover, the reduced context remembers to which one of those cases q belongs. This means that we can represent the reduced context of a node at depth k, using the polynomial orbit-finite set C_k, defined recursively: C_0 is the singleton set and C_{k+1} is the following disjoint sum:

$$\underbrace{C_k}_{\substack{\text{child of a}\\ \text{unary node}}} + \underbrace{C_k \times \Sigma}_{\substack{\text{left child of a}\\ \text{binary node}}} + \underbrace{C_k \times \Sigma}_{\substack{\text{right child of a}\\ \text{binary node}}} + \underbrace{C_k \times \Sigma}_{\substack{\text{first child of a}\\ \text{smooth node}}} + \underbrace{C_k \times \Sigma}_{\substack{\text{last child of a}\\ \text{smooth node}}} + \underbrace{\mathrm{orb}_d(\Sigma^2 \times C_k) \times \Sigma^4}_{\substack{\text{inner child of a}\\ \text{smooth node}\\ \text{(where } d\text{=dim}(\Sigma))}}$$

◁

Let us now show that the reduced context of a node and the profile of the infix is enough information to calculate the output letter (this is an analogue of Lemma 64):

Lemma 74. *Let $\underline{w}$ and $\underline{w'}$ be two pointed words, let T and T' be smooth factorization trees for w and w' (without the underlines), and let q and q' be nodes of T and T', such that:*

1. *the underlined positions in $\underline{w}$ and $\underline{w'}$ belong respectively to the infix of q and to the infix of q';*
2. *the infix profile of q is equal to the infix profile of q';*
3. *the reduced context of q is equal to the reduced context of q'.*

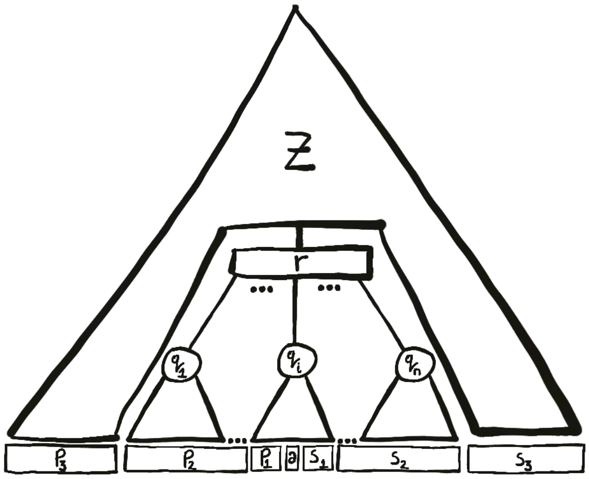
Then the output letter of $\underline{w}$ is equal to the output letter of $\underline{w'}$.

Proof. The proof is very similar to the proof of Lemma 64. It is an induction on the depth of q and q' (note that if $\mathrm{ctx}(q) = \mathrm{ctx}(q')$, then q and q' have equal depths). The induction base is trivial: We know that q and q are roots of T and T'. It follows that the infix profile of q is equal to the profile of $\underline{w}$ and the infix profile of q' is equal to the profile of $\underline{w'}$. By assumption, the infix profiles of q and q' are equal, which means that the profiles $\underline{w}$ and $\underline{w'}$ are equal as well. This finishes the proof, because the output letter of a pointed word depends entirely on its profile.

For the induction step, we only deal with the most interesting case, which is when q is an inner node of a smooth node (other cases are immediate). We start by introducing some notation: Let r, $q_1, \dots, q_i, \dots q_n$, and e be as in the definition of $\mathrm{ctx}(q)$ (in particular this means that $q = q_i$ and that r is the parent of q). Let $(p_1,\ a,\ s_1)$ be the infix profile of q, let (s_3, p_3) be the context of r, and let p_2 and s_2 be defined as follows:

$$p_2 = v_1 \cdot \ldots \cdot v_{i-1} \qquad s_2 = v_{i+1} \cdot \ldots \cdot v_n,$$

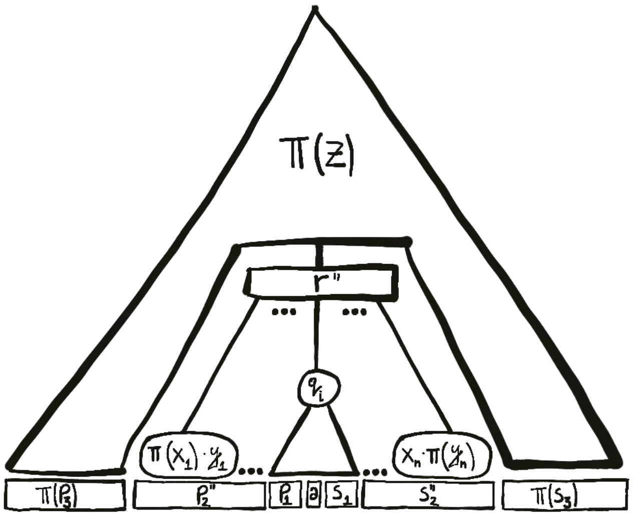
where v_i denotes the tree value of the node q_i. Finally, let Z be an incomplete tree obtained by cutting out r's subtree from T. Here is a sketch:



Observe that q' is an inner node of a smooth node as well – this is because $\mathrm{ctx}(q) = \mathrm{ctx}(q')$. It follows that we can introduce the same notation for T', i.e. we define r', $q'_1, \dots, q'_{n'}$, $p'_1, p'_2, p'_3, a', s'_3, s'_2, s'_1$ and Z' analogously as for T. By assumption, we know that infix profiles of q and q' are equal, which means that $p_1 = p'_1$, $a = a'$, and $s_1 = s'_1$. Moreover, since $\mathrm{ctx}(q) = \mathrm{ctx}(q')$, we also know that:

1. $\overrightarrow{g_{i-1}} = \overrightarrow{g_{i'-1}}'$ and $y_{i-1} = y'_{i'-1}$;
2. $\overleftarrow{g_{i+1}} = \overleftarrow{g_{i'+1}}'$ and $x_{i+1} = x'_{i'+1}$;
3. $\overline{\mathrm{orb}}_e(x_1, y_n, \mathrm{ctx}(r)) = \overline{\mathrm{orb}}_{e'}(x'_1, y'_{n'}, \mathrm{ctx}(r'))$.

Since $\overrightarrow{g_{i-1}} = \overrightarrow{g_{i'-1}}'$, we know that $e = e'$, which (by Item 3) means that there exist a $\mathrm{supp}(e)$-permutation π such that $\pi(x_1) = x'_1$, $\pi(y_n) = y'_{n'}$ and $\pi(\mathrm{ctx}(r)) = \mathrm{ctx}(r')$. Let us use this π to construct an intermediate tree T'' with an underlined element, and us prove that both the output of T and the output of T' are equal to the output of T''. We obtain T'' by taking T and replacing Z with $\pi(Z)$, replacing the node q_1 with a node whose value is equal to $\pi(x_1) \cdot y_1$ (it can be a leaf combined with unary nodes to keep the heights aligned), and replacing q_n with a node whose value is equal to $x_n \cdot \pi(y_n)$. Here is a sketch of T'':



As noted in the sketch, it holds by construction of T'' that $p_1'' = p_1$, $s_1'' = s_1$, $a'' = a$, $p_3'' = \pi(p_3)$, and $s_3'' = \pi(s_3)$. We are now left with showing that (a) T'' is a valid smooth decomposition tree, (b) the output of T' is equal to the output of T'' and that (c) the output of T is equal to the output of T''.

Let us first prove that T'' is a valid smooth factorization tree. It suffices to show that $v_{r''} = \pi(v_r)$, which means that it fits to $\pi(Z)$, and that r'' is a smooth node. Let us with $v_{r''} = \pi(v_r)$. Observe that, by construction of T'':

$$v_{r''} = \pi(x_1) \cdot y_1 \cdot v_2 \cdot \ldots \cdot v_{n-1} \cdot x_n \cdot \pi(y_n).$$

By Claim 44, it follows that:

$$v_{r''} = \pi(x_1) \cdot \overrightarrow{g_n} \cdot \pi(y_n)$$

By Claim 59, we know that $\mathrm{supp}(\overrightarrow{g_n}) = \mathrm{supp}(e)$, which means that $\pi(\overrightarrow{g_n}) = \overrightarrow{g_n}$ (as π is a $\mathrm{supp}(e)$-permutation). It follows that:

$$v_{r''} = \pi(x_1) \cdot \pi(\overrightarrow{g_n}) \cdot \pi(y_n) = \pi(x_1 \cdot \overrightarrow{g_n} \cdot y_n) = \pi(v_r)$$

Now, to show that r'' is a smooth node, it suffices to show that the following product is smooth:

$$\pi(x_1) \cdot y_1 \cdot v_2 \cdot \ldots \cdot v_{n-1} \cdot x_n \cdot \pi(y_n).$$

We have already shown that the value of this product is equal to $\pi(v_r)$, which, thanks to the smoothness of r, belongs to the $\mathcal{J}$-class of $\pi(e) = e$. According to Claim 44, $\pi(x_1)$, y_1, x_n and $\pi(y_n)$ also belong to the $\mathcal{J}$-class of $\pi(e) = e$. Thanks to the smoothness of r, we know that each v_j also belongs to this $\mathcal{J}$-class. It follows that the product is smooth.

Now, let show that the output T' is equal to the output of T''. For this we use the induction assumption for r' and r''. This means that we need to show that the infix profile of r' is equal to the infix profile of r'', and that $\mathrm{ctx}(r') = \mathrm{ctx}(r'')$. First, observe that $\mathrm{ctx}(r'') = \pi(\mathrm{ctx}(r))$. This is because the reduced context of both r'' and r depends entirely on the Z-part of the tree, and this dependence is easily seen to be equivariant. We have chosen π, so that $\pi(\mathrm{ctx}(r)) = \mathrm{ctx}(r')$, which means that $\mathrm{ctx}(r'') = \mathrm{ctx}(r')$. Now, let us show that the infix profiles of r' and r'' are equal, i.e. that:

$$(p''_2 \cdot p_1, a, s_1 \cdot s''_2) = (p'_2 \cdot p_1, a, s_1 \cdot s'_2)$$

It suffices to show that $p''_2 = p'_2$ (the proof for $s''_2 = s'_2$ is analogous). Notice that $p''_2 = \pi(x_1) \cdot y_1 \cdot v_{q_2} \cdot \ldots \cdot v_{q_{i-1}}$. By Claim 44 it follows that $p''_2 = \pi(x_1) \cdot \overrightarrow{g_{i-1}} \cdot y_{i-1}$. By assumption, we know that $\pi(x_1) = x'_1$, $\overrightarrow{g_{i-1}} = \overrightarrow{g_{i'-1}}'$, and $y_{i-1} = y'_{i'-1}$. It follows that $p''_2 = x'_1 \cdot \overrightarrow{g_{i'-1}}' \cdot y'_{i'-1}$, which by Claim 44 means that $p''_2 = p'_2$.

Finally, let us show that the output of T is equal to the output of T''. We do this by directly showing that:

$$\lambda(p_3 \cdot p_2 \cdot p_1,\ a,\ s_1 \cdot s_2 \cdot s_3) = \lambda(\pi(p_3) \cdot p''_2 \cdot p_1,\ a,\ s_1 \cdot s''_2 \cdot \pi(s_3))$$

We define $c_1 = p_3 \cdot x_1$ and $b_1 = y_1 \cdot v_{q_2} \cdot \ldots v_{q_{i-1}} \cdot p_1$ (and analogously for c_2 and b_2). This means that $p_3 \cdot p_2 \cdot p_1 = c_1 \cdot b_1$ and $\pi(p_3) \cdot p''_2 \cdot p_1 = \pi(c_1) \cdot b_1$ (and analogously for b_2, c_2 and the s-values). This leaves us with showing that:

$$\lambda(c_1 \cdot b_1,\ a,\ b_2 \cdot c_2) = \lambda(\pi(c_1) \cdot b_1,\ a,\ b_2 \cdot \pi(c_2))$$

Observe that that $c_1 \cdot e = c_1$ – this is because e is an idempotent that is a suffix of x_1, which is a suffix of c_1 – and that $\pi(c_1) \cdot e = \pi(c_1)$, as π is a $\mathrm{supp}(e)$-permutation. (And analogously for c_2.) This means that it is enough to show that:

$$\lambda(c_1 \cdot e \cdot b_1,\ a,\ b_2 \cdot e \cdot c_2) = \lambda(\pi(c_1) \cdot e \cdot b_1,\ a,\ b_2 \cdot e \cdot \pi(c_2))$$

This resembles the locality equation for λ, but we do not know if $c_1 a c_2 = e$ (in fact, this might not be true). However, it is not hard to see that $c_1 a c_2$ is $\mathcal{H}$-equivalent to e. We finish the proof, by showing that this is enough to apply the locality equation:

Claim 45. *If λ satisfies the locality equation, then for every idempotent e, for every supp(e)-permutation π, and for all a, b_1, b_2, c_1, c_2, a, such that $b_1 a b_2$ is $\mathcal{H}$-equivalent to e, it holds that:*

$$\lambda(c_1 \cdot e \cdot b_1,\ a,\ b_2 \cdot e \cdot c_2) = \lambda(\pi(c_1) \cdot e \cdot b_1,\ a,\ b_2 \cdot e \cdot \pi(c_2))$$

Proof. Let $g := b_1 a b_2$. By assumption, we know that g is $\mathcal{H}$-equivalent to e. By [Pin10, Proposition 1.13], we know that the $\mathcal{H}$-class of e is a subgroup of S, and that e is the identity element of this subgroup. This means that there exists a

g^{-1} (also $\mathcal{H}$-equivalent to e) such that $g^{-1} \cdot g = g \cdot g^{-1} = e$. Define $c_1' := c_1 g$, and $b_1' := g^{-1} b_1$. Observe that $c_1' \cdot e \cdot b_1' = c_1 \cdot e \cdot b_1$, which means that:

$$\lambda(c_1 \cdot e \cdot b_1,\ a,\ b_2 \cdot e \cdot c_2) = \lambda(c_1' \cdot e \cdot b_1',\ a,\ b_2 \cdot e \cdot c_2)$$

We know that $b_1' \cdot a \cdot b_2 = g^{-1} \cdot g = e$, which means that we can use the locality equation:

$$\lambda(c_1' \cdot e \cdot b_1',\ a,\ b_2 \cdot e \cdot c_2) = \lambda(\pi(c_1') \cdot e \cdot b_1',\ a,\ b_2 \cdot e \cdot \pi(c_2))$$

By Lemma 59, we know that $\mathrm{supp}(g) = \mathrm{supp}(e)$. It follows that $\pi(c_1') = \pi(c_1) \cdot g$, which leads to $\pi(c_1') \cdot e \cdot b_1' = \pi(c_1) \cdot e \cdot b_1$. Thus, we have:

$$\lambda(\pi(c_1') \cdot e \cdot b_1',\ a,\ b_2 \cdot e \cdot \pi(c_2)) = \lambda(\pi(c_1) \cdot e \cdot b_1,\ a,\ b_2 \cdot e \cdot \pi(c_2))$$

This completes the proof of the claim. □

□

At this point, it is not hard to show how to construct the output of the local rational semigroup transduction. First, we notice that, thanks to Lemma 74, it is not hard to show the following analogue of Claim 38:

Claim 46. *Let w_i be the ith input letter, and let l_i be the i-th leaf. There exists an equivariant function, that transforms every pair $(w_i, ctx(l_i))$ into the i-th output letter of the local rational semigroup transduction.*

Proof. Once, we notice that the infix profile of l_i is equal to $(1, w_i, 1)$, the claim follows from Lemma 74, in the same way as Claim 38 follows from Lemma 64. □

This finishes the construction, because by reasoning very similar to the proof of Lemma 65, we can show that we can use compositions of rational prime functions to construct $\mathrm{ctx}(q)$ in each node of a smooth factorization tree.

3.7.1.2 Compositions of rational primes $\subseteq$ Local rational transductions

In this section, we show that every function $f : \Sigma^* \to \Gamma^*$ that can be constructed as a composition of single-use rational primes, can be implemented as a rational semigroup transduction. The proof goes by induction on the construction of f as a composition of primes. However, in order to simplify the proof, we introduce an alternative way of constructing compositions of primes, that only uses sequential composition:

Claim 47. *Let P be a set of prime functions that contains all length-preserving homomorphisms – for example, P could be the set of single-use rational primes. It follows that every word-to-word function can be constructed as a $(\times, \circ)$-composition of primes from P, if and only if it can be constructed as a $(\circ)$-composition of letter-to-letter homomorphisms and functions of the form $p \times$ `id`, where $p \in P$.*

Proof. Let P' be the extended set of primes. The ($\Leftarrow$)-implication is easy: it suffices to notice that every function in P' is a $(\times, \circ)$-composition of primes. The proof of the ($\Rightarrow$)-implication uses the following equality to push down the ($\times$)-compositions:

$$(f \times g) = (f \times \mathtt{id}) \circ (\mathtt{id} \times f)$$

Formally, the proof goes by induction on the construction of f as a $(\circ, \times)$-composition of primes from P:

1. The induction base is trivial: if f is equal to a $p \in P$, it can be constructed as follows:

   $$\mathtt{leftI}^{-1^*} \circ (p \times \mathtt{id}) \circ \mathtt{leftI}^*,$$

   where $\mathtt{leftI}$ is the function $\Sigma \to \Sigma \times 1$.

2. The case where f is a composition $f = g \circ h$ is also simple. By induction, we know that:

   $$g = g_n \circ \ldots \circ g_1 \qquad h = h_m \circ \ldots \circ h_1,$$

   where all g_i's and h_i's belong to P'. This means that we can construct f in the following way:

   $$f = g_n \circ \ldots \circ g_1 \circ h_m \circ \ldots \circ h_1$$

3. Finally, the most interesting case is where $f = g \times h$. Again, by induction assumption, we know that:

   $$g = g_n \circ \ldots \circ g_1 \qquad h = h_m \circ \ldots \circ h_1,$$

   It follows that:

   $$f = (g_n \circ \ldots \circ g_1) \times (h_m \circ \ldots h_1) = (g_n \times \mathtt{id}) \circ \ldots \circ (g_1 \times \mathtt{id}) \circ (\mathtt{id} \times h_m) \circ \ldots \circ (\mathtt{id} \times h_1)$$

   This leaves us with showing that for every $p \in P'$ functions $p \times \mathtt{id}$ and $\mathtt{id} \times p$ are ($\circ$)-compositions of functions from P'. We only show this for $p \times \mathtt{id}$, as the proof for $\mathtt{id} \times p$ is analogous. There are two cases to consider: If p is a letter-to-letter homomorphism then, so is $p \times \mathtt{id}$. If, on the other hand, p is of the form $p' \times \mathtt{id}_{\Sigma^*}$, where $p' \in P$, then $p \times \mathtt{id} = (p' \times \mathtt{id}_\Sigma) \times \mathtt{id}_\Gamma$, which is almost the same as $p' \times \mathtt{id}_{\Sigma \times \Gamma} \in P'$. This finishes the proof, as we can fix this slight type mismatch using ($\circ$)-compositions with $\mathtt{assoc}^*$ and $\mathtt{assoc}^{-1^*}$.

□

This leaves us with translating ($\circ$)-compositions of the extended primes into local rational semigroup transductions:

Lemma 75. *Let $p_1, \ldots, p_n$ be a sequence of extended single-use rational primes (i.e. every p_i is either a letter-to-letter homomorphism or a $p' \times \mathtt{id}$, where p' is a single-use rational prime). It follows that the composition $p_n \circ \ldots \circ p_1$ can be expressed as a local rational semigroup transduction.*

The remaining part of this section is dedicated to proving Lemma 75: The proof goes by induction on n. The induction base is trivial – if $n = 0$ then the composition of primes is equal to $\texttt{id}^* : \Sigma^* \to \Sigma^*$. For the induction step, it suffices to show that local rational semigroup transductions are closed under *pre-compositions*[23] with extended single-use rational prime functions. We start with the length-preserving homomorphisms:

Claim 48. *Let $f : \Gamma^* \to \Delta^*$ be defined by some local rational semigroup transduction, and let $g : \Sigma \to_{eq} \Gamma$ be an equivariant function over orbit-finite sets. It follows that the following composition can also be defined by a local rational semigroup transduction:*

$$\Sigma^* \xrightarrow{g^*} \Gamma^* \xrightarrow{f} \Delta^*$$

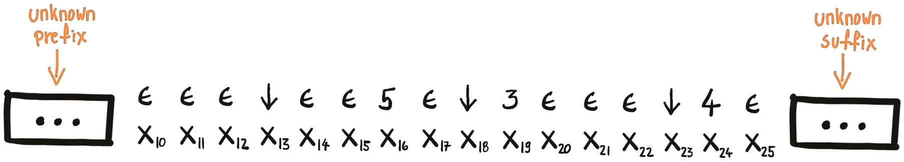
Proof. If f is recognized by (S, h, λ), then $f \circ g^*$ is recognized by $(S, h \circ g, \lambda)$. □

Next, we show that local rational semigroup transductions are closed under pre-composition with single-use left-to-right atom propagation – the right-to-left variant of the proof is analogous.

Claim 49. *Let $f : ((\mathbb{A} + \epsilon) \times \Sigma)^* \to \Gamma^*$ be defined by some local rational semigroup transduction. It follows that the following composition can also be defined by a local rational semigroup transduction:*

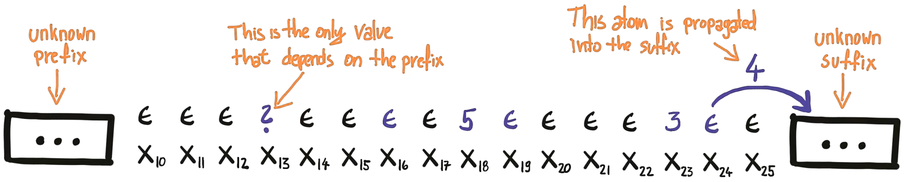
$$((\mathbb{A} + \{\downarrow, \epsilon\}) \times \Sigma)^* \xrightarrow{\overrightarrow{\textit{su-prop}} \times \texttt{id}} ((\mathbb{A} + \epsilon) \times \Sigma)^* \xrightarrow{f} \Gamma^*$$

Proof. Let (S, h, λ) be a local rational semigroup transduction that defines f. We show how to construct (S', h', λ') that defines $f \circ (f_{\overrightarrow{\text{su-prop}}} \times \texttt{id})$. First, let us analyse how $f_{\overrightarrow{\text{su-prop}}} \times \texttt{id}$ modifies an input infix. Here is an example:

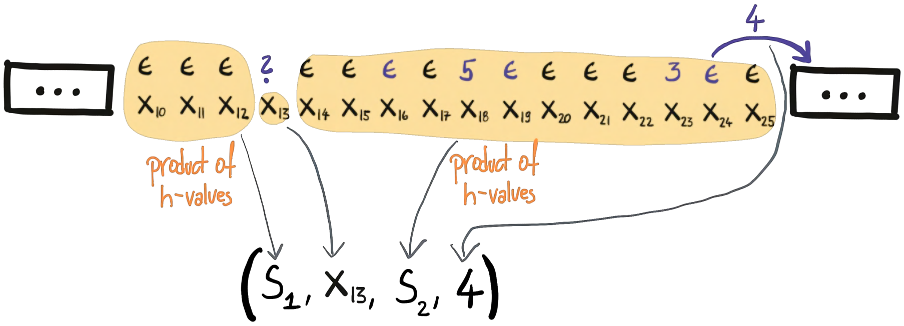

unknown prefix
unknown suffix
...
ε ε ε ↓ ε ε 5 ε ↓ 3 ε ε ε ↓ 4 ε
x10 x11 x12 x13 x14 x15 x16 x17 x18 x19 x20 x21 x22 x23 x24 x25
...

Since we do not know the prefix, we cannot exactly predict the output of $f_{\overrightarrow{\text{su-prop}}} \times \texttt{id}$ on the infix. (Observe that the unknown suffix does not influence the output on the infix.) We can, however, predict almost all of the output letters:

[23] Showing this for post-compositions would be an equally valid proof strategy, but pre-compositions seem to be more compatible with local rational semigroup transductions. We are going to see a similar reasoning in Chapter 4 where, depending on the model, we are going to use either pre- or post-compositions.


unknown prefix
This is the only value that depends on the prefix
This atom is propagated into the suffix
4
unknown suffix
ε ε ε ? ε ε ε ε 5 ε ε ε ε 3 ε ε
x_{10} x_{11} x_{12} x_{13} x_{14} x_{15} x_{16} x_{17} x_{18} x_{19} x_{20} x_{21} x_{22} x_{23} x_{24} x_{25}

Since we do not know all output letters, we cannot exactly compute the product of their h-values. We can, however, use the product of the h-values to compress the information about the infix:


4
ε ε ε ? ε ε ε ε 5 ε ε ε ε 3 ε ε
x_{10} x_{11} x_{12} x_{13} x_{14} x_{15} x_{16} x_{17} x_{18} x_{19} x_{20} x_{21} x_{22} x_{23} x_{24} x_{25}
Product of h-values
Product of h-values
$(S_1, x_{13}, S_2, 4)$

Observe that s_1 and s_2 come from S^1 (defined in Claim 21) – this is because both the prefix before the first $\downarrow$ and the suffix after the first $\downarrow$ might be empty. Moreover, in the general case, the last non-ϵ letter might be $\downarrow$ – in this case the value propagated into the suffix is going to be $\bot$. It is also possible that the first non-ϵ value is not a $\downarrow$, or that the input word contains only ϵ's. In order to account for all those cases, we represent the compressed information about an infix as an element of the following set:

$$S' = \underbrace{S^1 \times \Sigma \times S^1 \times (\mathbb{A} + \bot)}_{\substack{\text{An infix whose first non-}\epsilon \\ \text{is equal to } \downarrow}} \quad + \quad \underbrace{S \times (\mathbb{A} + \bot)}_{\substack{\text{An infix whose first non-}\epsilon \\ \text{is an element of } \mathbb{A}}} \quad + \quad \underbrace{S}_{\substack{\text{An infix that} \\ \text{only contains } \epsilon\text{'s}}}$$

It is not hard to see that this compression is compositional – i.e. the compressed information about $w_1 w_2$ depends only on the compressed information about w_1 and the compressed information about w_2. It follows by (the semigroup version of) the proof of Lemma 41 that this compression operation induces a semigroup structure on S'. Moreover, since both S and Σ are orbit-finite, it follows that S' is orbit-finite as well. It follows that we can use S' as the underlying semigroup for the rational semigroup transduction for $f_{\overrightarrow{\text{su-prop}}} \times \texttt{id}$. This leaves with defining h' and λ'. We define h' to be the compression function. For λ' we are going to need a couple of auxiliary functions. We start with `feed`:

$$\texttt{feed} : \underbrace{(\mathbb{A} + \bot)}_{\substack{\text{Value propagated} \\ \text{from the prefix}}} \times \underbrace{S'}_{\substack{\text{The compressed information} \\ \text{about the infix}}} \to \underbrace{S}_{\substack{\text{The product of the } h\text{-values} \\ \text{on the infix after applying } f_{\overrightarrow{\text{su-prop}}}}}$$

It is not hard to see that the function feed can be computed using the following formula:

$$\texttt{feed}(v, (s_1, x, s_2, a)) = s_1 h\tbinom{v}{x} s_2 \quad \texttt{feed}(v, (s, a)) = s \quad \texttt{feed}(v, s) = s$$

Now, we define the function get : $S' \to (\mathbb{A} + \bot)$ that computes the atom propagated by the infix:

$$\texttt{get}(s_1, x, s_2, a) = a \quad \texttt{get}(s, a) = a \quad \texttt{get}(s) = \bot$$

Now, we can use feed, get, and the original output function λ, to define λ' that computes the output letter for $w_1 \underline{\tbinom{y}{x}} w_2$ with respect to $f \circ (f_{\overrightarrow{\text{su-prop}}} \times \texttt{id})$:

$$\lambda'\left(p',\ \begin{pmatrix}\downarrow\\ x\end{pmatrix},\ s'\right) = \lambda\left(\texttt{feed}(\bot,\ p'),\ \begin{pmatrix}\texttt{get}(p')\\ x\end{pmatrix},\ \texttt{feed}(\bot,\ s')\right)$$

$$\lambda'\left(p',\ \begin{pmatrix}\epsilon\\ x\end{pmatrix},\ s'\right) = \lambda\left(\texttt{feed}(\bot,\ p'),\ \begin{pmatrix}\epsilon\\ x\end{pmatrix},\ \texttt{feed}(\texttt{get}(p'),\ s')\right)$$

$$\lambda'\left(p',\ \begin{pmatrix}a \in \mathbb{A}\\ x\end{pmatrix},\ s'\right) = \lambda\left(\texttt{feed}(\bot,\ p'),\ \begin{pmatrix}\epsilon\\ x\end{pmatrix},\ \texttt{feed}(a,\ s')\right)$$

It is not hard to see that this λ' is equivariant, and that (S', h', λ') defines $f \circ (f_{\overrightarrow{\text{su-prop}}} \times \texttt{id})$. This leaves us showing that λ' is local: Let us take $x_1, x_2, y_1, y_2, e \in S'$, $a \in (\mathbb{A} + \{\downarrow, \epsilon\}) \times \Sigma$, and a supp$(e)$-permutation π, such that e is an idempotent, and $y_1 h'(a) y_2 = e$, and let us show that:

$$\lambda'(x_1 e y_1,\ a,\ y_2 e x_2) = \lambda'(\pi(x_1)\, e y_1,\ a,\ y_2 e \pi(x_2))$$

For this, we consider two cases:

First, we consider the case where e contains only ϵ's, i.e. $e \in S$ (see the definition of S'). Since $y_1 h'(a) y_2 = e$, it follows that also y_1, a, and y_2 contain only ϵ's – i.e. $y_1, y_2 \in S$ and a is of the form $\tbinom{\epsilon}{a}$ (for some $a \in \Sigma$). By definition of S' and λ', it follows that:

$$\lambda'(x_1 e y_1, \begin{pmatrix}\epsilon\\ a\end{pmatrix}, y_2 e x_2) = \lambda(\texttt{feed}(\bot, x_1) \cdot e \cdot y_1, \begin{pmatrix}\epsilon\\ a\end{pmatrix}, y_2 \cdot e \cdot \texttt{feed}(\texttt{get}(x_1), x_2))$$

Observe that (by definition of S'), e is an idempotent as an element of S, and also in S it holds that $y_1 h\tbinom{\epsilon}{a} y_2 = e$. This means that we can use the locality of λ to transform the right-hand side of the equality further into:

$$\lambda(\pi(\texttt{feed}(\bot, x_1)) \cdot e \cdot y_1, \begin{pmatrix}\epsilon\\ a\end{pmatrix}, y_2 \cdot e \cdot \pi(\texttt{feed}(\texttt{get}(x_1), x_2)))$$

By equivariance of feed and set, we know that this is equal to:

$$\lambda(\texttt{feed}(\bot, \pi(x_1)) \cdot e \cdot y_1, \begin{pmatrix}\epsilon\\ a\end{pmatrix}, y_2 \cdot e \cdot \texttt{feed}(\texttt{get}(\pi(x_1)), \pi(x_2))),$$

which by definition of S' and λ' is equal to:

$$\lambda'(\pi(x_1)ey_1, \begin{pmatrix}\epsilon\\ a\end{pmatrix}, y_2e\pi(x_2))$$

This finishes the proof for the first case.

In the second case, we assume the complement of the first case, i.e. that e contains at least one $\downarrow$ or an element of $\mathbb{A}$. In this case a could be of any form: $\binom{\epsilon}{a}$, $\binom{\downarrow}{a}$, or $\binom{b\in\mathbb{A}}{a}$. All of those cases are analogous, so we only show how to deal with the most interesting case, which is $\binom{\downarrow}{a}$. First, we notice that since e is an idempotent, we have that $\lambda'(x_1ey_1, \binom{\downarrow}{a}, y_2ex_2) = \lambda'(x_1eey_1, \binom{\downarrow}{a}, y_2ex_2)$. Then, we use the definition of S' and λ' to unfold the right-hand side in the following way:

$$\lambda\left(\mathtt{f}(\bot, x_1)\cdot\mathtt{f}(x_1, e)\cdot\mathtt{f}(e, e)\cdot\mathtt{f}(e, y_1), \begin{pmatrix}\mathtt{get}(ey_1)\\ a\end{pmatrix}, \mathtt{f}(\bot, y_2)\cdot\mathtt{f}(y_2, e)\cdot\mathtt{f}(e, x_2)\right),$$

where $\mathtt{f}(x_1, e)$ is a notational shortcut for $\mathtt{feed}(\mathtt{get}(x_1), e)$. Now, let us observe that (by definition of S'), $\mathtt{feed}(e, e)$ is an idempotent in S:

$$\mathtt{feed}(e, e)\cdot\mathtt{feed}(e, e) = \mathtt{feed}(e, e\cdot e) = \mathtt{feed}(e, e)$$

Moreover, since $y_1h'\binom{\downarrow}{a}y_2 = e$, we know that $\mathtt{feed}(y_2, e) = \mathtt{feed}(e, e)$, and that

$$\mathtt{feed}(e, y_1)\cdot h\begin{pmatrix}\mathtt{get}(ey_1)\\ a\end{pmatrix}\cdot\mathtt{feed}(\bot, y_2) = \mathtt{feed}(e, e)$$

Finally, we observe that by Lemma 3, it holds that $\mathrm{supp}(\mathtt{feed}(e, e)) \subseteq (\mathrm{supp}(e))$, which means that π is a $\mathrm{supp}(\mathtt{feed}(e, e))$-permutation. It follows that we can apply the locality equation obtaining:

$$\lambda\left(\pi\left(\mathtt{f}(\bot, x_1)\cdot\mathtt{f}(x_1, e)\right)\cdot\mathtt{f}(e, e)\cdot\mathtt{f}(e, y_1), \begin{pmatrix}\mathtt{get}(ey_1)\\ a\end{pmatrix}, \mathtt{f}(\bot, y_2)\cdot\mathtt{f}(y_2, e)\cdot\pi(\mathtt{f}(e, x_2))\right)$$

By definition of S' and λ' this is equal to $\lambda(\pi(x_1)eey_1, \binom{\downarrow}{a}, y_2e\pi(x_2))$, which is in turn equal to $\lambda'(\pi(x_1)ey_1, \binom{\downarrow}{a}, y_2e\pi(x_2))$. It follows that λ' is local. □

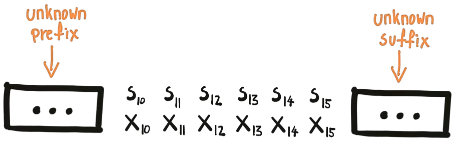
Finally, we show that local rational prime functions are closed under pre-compositions with left-to-right multiple-use bit propagation and with group-prefix functions (left-to-right propagation and group-suffix functions can be handled analogously). We can deal with both of those cases by proving the following claim:

Claim 50. *Let $\mathcal{A} : \Sigma_1^* \to \Sigma_2^*$ be a Mealy machine over* finite *(and not only orbit-finite) alphabets, let Γ, Δ be orbit-finite alphabets, and let $f : (\Sigma_2 \times \Gamma)^* \to \Delta^*$ be a local rational semigroup transduction. It follows that the following composition is a local rational semigroup transduction as well:*

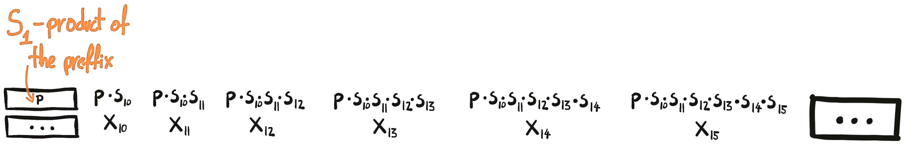
$$f \circ (\mathcal{A} \times \mathtt{id}) : (\Sigma_1 \times \Gamma)^* \to \Delta^*$$

Proof. By Lemma 45, we know that $\mathcal{A}$ is equivalent to some finite semigroup transduction (see Definition 18), which means that it can be expressed as $\lambda_1^* \circ f_{S_1\text{-pref}} \circ h_1^*$, where $f_{S_1\text{-pref}}$ is the semigroup prefix function for some *finite* (and not only orbit-finite) semigroup S_1. Since, by Claim 48, we already know that local rational semigroup transductions are closed under pre-compositions with homomorphisms, it suffices to show how to construct $f \circ (f_{S_1\text{-pref}} \times \texttt{id})$ as a local rational semigroup transduction:

Let (S_2, h_2, λ_2) be a local rational semigroup transduction for f. We show how construct a local rational semigroup construction for $f \circ (f_{S_1\text{-pref}} \times \texttt{id})$, using the classical wreath product construction for finite semigroups[24]. First, let us analyse how $f_{S_1\text{-pref}} \times \texttt{id}$ modifies an input infix. Here is an example:


unknown prefix
unknown suffix
...
s_{10} s_{11} s_{12} s_{13} s_{14} s_{15}
x_{10} x_{11} x_{12} x_{13} x_{14} x_{15}
...

In order to compute the output of $f_{S_1\text{-pref}} \times \texttt{id}$, it suffices to know the S_1-product of the prefix. For example:


S_1-product of the prefix
p
...
$p \cdot s_{10}$
x_{10}
$p \cdot s_{10} \cdot s_{11}$
x_{11}
$p \cdot s_{10} \cdot s_{11} \cdot s_{12}$
x_{12}
$p \cdot s_{10} \cdot s_{11} \cdot s_{12} \cdot s_{13}$
x_{13}
$p \cdot s_{10} \cdot s_{11} \cdot s_{12} \cdot s_{13} \cdot s_{14}$
x_{14}
$p \cdot s_{10} \cdot s_{11} \cdot s_{12} \cdot s_{13} \cdot s_{14} \cdot s_{15}$
x_{15}
...

At this point, we can apply h_2 to every letter, and compute the S_2 product of the infix. In general, this can be represented as a function of the following type:

$$\underbrace{S_1^1}_{\substack{\text{Given the } S_1\text{-value of a prefix} \\ \text{(which might be empty) } \ldots}} \quad \to \quad \underbrace{S_2}_{\substack{\ldots\text{what is the} \\ S_2\text{-value of the infix} \\ \text{after applying } f_{S_1-\text{pref}} \text{ and } h_2^*}}$$

In order to obtain compositionality, we also need to remember the S_1-value of the infix. In total, we compress the information about each infix into an element of the following set:

$$S_3 = \underbrace{S_1}_{\substack{S_1\text{-product} \\ \text{of the prefix}}} \times \underbrace{(S_1^1 \to S_2)}_{\substack{\text{The function} \\ \text{explained above}}}$$

The key observation is that since S_1 is finite (and not only orbit-finite), we know that $S_1 \to S_2$ is orbit-finite, as it is isomorphic to a finite power $S_2^{|S_1|}$. This

[24] See [KR65, Definition 1.7].

procedure of compressing is compositional, which (as explained in Lemma 41) imposes a semigroup product on S_3. The explicit formula for this product looks as follows:

$$(x, X) \cdot (y, Y) = (xy,\ p \mapsto X(p) \cdot Y(px))$$

The intuition behind this formula is that p is the S_1-product of the X's prefix, and px is the S_1-product of the Y's prefix (as Y's prefix also includes x). This is a well-studied construction called the wreath product of S_1 and S_2 sometimes denoted[25] as $S_3 = S_2 \wr S_1$. We use S_3 as the underlying semigroup for the local rational semigroup transduction for $f \circ (f_{S_1\text{-pref}} \times \mathtt{id})$. The input function $h_3 : S_1 \times \Gamma \to S_3$ is given by the following formula:

$$h_3(s, a) = (s,\ p \mapsto h_2(ps, a)),$$

and the output function $\lambda_3 : S_3^1 \times (S_1 \times \Gamma) \times S_3^1 \to \Delta$, is given by the following formula:

$$\lambda_3((x, X),\ (s, a),\ (y, Y)) = \lambda_2(X(1), (xs, a), Y(xs))$$

The intuition behind this formula for λ_3 is that after applying $f_{S_1-\text{pref}}$ to the input word, the S_2-product of the h_2-values of the prefix is equal to $X(1)$, the current letter is equal to (xs, a), and the S_2-product of the h_2-values of the suffix is equal to $Y(xs)$. Thanks to this intuition, it is not hard to see that (S_3, h_3, λ_3) implements $f \circ (f_{S_1\text{-pref}} \times \mathtt{id})$.

This leaves us with showing that λ_3 is local. For this we take (x_1, X_1), (x_2, X_2), (y_1, Y_1), (y_2, Y_2), $(e, E) \in S_3$, $(s, a) \in S_1 \times \Gamma$ and a supp$((e, E))$-permutation π such that (e, E) is an idempotent, and $(y_1, Y_1) \cdot h_3(s, a) \cdot (y_2, Y_2) = (e, E)$, and we show that:

$$\begin{gathered}\lambda_3((x_1, X_1)(e, E)(y_1, Y_1),\ (s, a),\ (y_1, Y_1)(e, E)(x_1, X_1)\\ =\\ \lambda_3(\pi((x_1, X_1))(e, E)(y_1, Y_1),\ (s, a),\ (y_2, Y_2)(e, E)\ \pi((x_2, X_2)\end{gathered}$$

First, since (e, E) is an idempotent, we can transform the initial expression into:

$$\lambda_3((x_1, X_1)(e, E)(e, E)(y_1, Y_1),\ (s, a),\ (y_1, Y_1)(e, E)(x_1, X_1))$$

Then, using the definition of λ_3 and of the product in S_3, we transform it into:

$$\lambda_2(X_1(1) \cdot E(x_1) \cdot E(x_1 e) \cdot Y_1(x_1 ee),\ (x_1 eey_1 s, a),\ Y_2(x_1 eey_1 s) \cdot E(x_1 eey_1 sy_2) \cdot X_2(x_1 eey_1 sy_2 e))$$

Now, we observe that since (e, E) is an idempotent in S_3, we know that e is an idempotent in s_1, and that since $(y_1, Y_1) \cdot h_3(s, a) \cdot (y_2, Y_2) = (e, E)$, we know that $y_1 sy_2 = e$. Thanks to this observation, we can transform our expression into:

$$\lambda_2(X_1(1) \cdot E(x_1) \cdot E(x_1 e) \cdot Y_1(x_1 e),\ (x_1 ey_1 s, a),\ Y_2(x_1 ey_1 s) \cdot E(x_1 e) \cdot X_2(x_1 e))$$

[25] In [KR65] this is denoted as $S_3 = S_2 w S_1$.

At this point, we would like to apply the locality equation for λ_2. For this, we need to show proof all of its assumptions. First, we observe that since (e, E) is an idempotent, we know that:

$$(e, E) = (e, E)(e, E) = (ee, p \mapsto E(p) \cdot E(pe))$$

In particular, this means that $E(p) = E(p) \cdot E(pe)$ for every $p \in S_1^1$. If we take $p = x_1 e$, we obtain that $E(x_1 e) = E(x_1 e)E(x_1 ee) = E(x_1 e)E(x_1 e)$, which means that $E(x_1 e)$ is an idempotent. Next, by a similar reasoning, we observe that since $(y_1, Y_1) \cdot h_3(s, a) \cdot (y_2, Y_2) = (e, E)$, we know that for all p, it holds that:

$$Y_1(p) \cdot h_2(py_1 s, a) \cdot Y_2(py_1 s) = E(p)$$

If we take $p = x_1 e$, we get that:

$$Y_1(x_1 e) \cdot h_2(x_1 ey_1 s, a) \cdot Y_2(x_1 ey_1 s) = E(x_1 e)$$

Finally, we notice that S_1 is atomless (this is because every equivariant set that is both finite and orbit-finite has to be atomless). This means that both x_1 and e have empty supports, which means that π is a $\mathrm{supp}(E(x_1 e))$-permutation, as by Lemma 3 it holds that $\mathrm{supp}(E(x_1 e)) \subseteq \mathrm{supp}(E)$. It follows that we can apply the locality equation, obtaining:

$$\lambda_2(\pi(X_1(1) \cdot E(x_1)) \cdot E(x_1 e) \cdot Y_1(x_1 e),\ (x_1 ey_1 s, a),\ Y_2(x_1 ey_1 s) \cdot E(x_1 e)\pi(X_2(x_1 e)))$$

Observe now that since x_1 is atomless (i.e. equivariant), we know that $\pi(x_1) = x_1$, and since π is a $\mathrm{supp}(e, E)$-permutation, we know that $\pi(E) = E$. This means that (after unfolding some of the e's back to $y_1 s y_2$ or to ee) we can fold the definitions of λ_3 and of the product in S_3, obtaining:

$$\lambda_3(\pi(x_1, X_1)(e, E)(e, E)(y_1, Y_1),\ (s, a),\ (y_2, Y_2)(e, E)\pi(x_2, X_2))$$

After folding $(e, E)(e, E)$ back to (e, E), we conclude that λ_3 is local. □

Chapter 4

Two-way transductions with atoms

In this chapter, we define and study the class of word-to-word functions recognized by *single-use two-way transducers*. Our main result is Theorem 15, which states that this class of transductions admits three more equivalent definitions: *copyless streaming string transducers with atoms*, *regular list functions with atoms*, and *compositions of single-use two-way primes*. Furthermore, we show that single-use two-way transducers (and their equivalent models) are closed under compositions and have decidable equivalence.

In my opinion, these results demonstrate that single-use models are better behaved than their multiple-use counterparts: Two-way multiple-use register automata lack decidable equivalence (see Theorem 3), and copyless streaming string transducers with multiple-use registers are not closed under composition (see [AČ11, Proposition 4]). I believe that for this reason, the class of function defined by single-use two-way transducers deserves the name of *regular transductions with atoms.*

This chapter, along with the four models it introduces, is based on [BS20] (specifically, on [BS20, Theorems 13 and 14]). However, the presentation of the models and the results is new and, hopefully, improved. The new approach builds upon the idea of single-use functions from Chapter 2, and uses Theorem 14 from Chapter 3.

4.1 Definitions

We use this section to formulate and briefly discuss the four equivalent definitions of regular transductions with atoms:

4.1.1 Single-use two-way transducers

A *single-use two-way transducer* is a single-use two-way automaton with output. We have already seen all the building blocks of the definition – we simply need to combine them:

Definition 36. A *single-use two-way transducer* consists of:

1. A polynomial orbit-finite input alphabet Σ and a polynomial orbit-finite output alphabet Γ;

2. A polynomial orbit-finite set of states Q;

3. An equivariant initial state $q_0 \in Q$;

4. A single-use transition function:

$$\delta : \underbrace{\Sigma + \{\vdash, \dashv\}}_{\substack{\text{current letter, or an}\\ \text{end-of-word marker}}} \to_{\text{eq}} \left(\underbrace{Q}_{\substack{\text{current}\\ \text{state}}} \multimap \underbrace{Q}_{\substack{\text{new}\\ \text{state}}} \times \underbrace{(\Gamma + \epsilon)}_{\substack{\text{output}\\ \text{letter}}} \times \underbrace{\{\leftarrow, \rightarrow\}}_{\substack{\text{which way}\\ \text{to go}}} + \underbrace{\texttt{finish}}_{\substack{\text{or finish}\\ \text{the run}}} \right)$$

A single-use two-way transducer defines the following function $\Sigma^* \to \Gamma^*$: Given an input word $w \in \Sigma^*$, we equip it with end-of-word markers obtaining $\vdash w \dashv$, and we place the transducer's head at the first letter of w in the initial state q_0. Then, we start applying the transition function, feeding it with the transducer's state and with the input letter seen by the transducer's head, and using its output to update the transducer's state and move the transducer's head. We continue this process until[1] the transition function returns `finish`, at which point we construct the output word as the concatenation of all output letters produced by the transition function (excluding ϵ's). $\triangleleft$

Example 41. Consider the following *map duplicate* function, which duplicates every #-separated block:

$$f_{\text{map-dup}} : (\mathbb{A} + \#)^* \to (\mathbb{A} + \#)^*$$

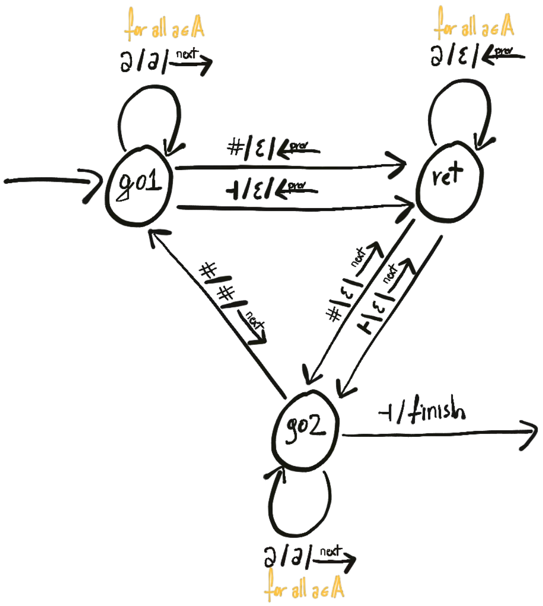
For example:

$$f_{\text{map-dup}}(1\ 2\ 3\ \#\ 5\ 7\ \#\ 1\ 2) = 1\ 2\ 3\ 1\ 2\ 3\ \#\ 5\ 7\ 5\ 7\ \#\ 1\ 2\ 1\ 2$$

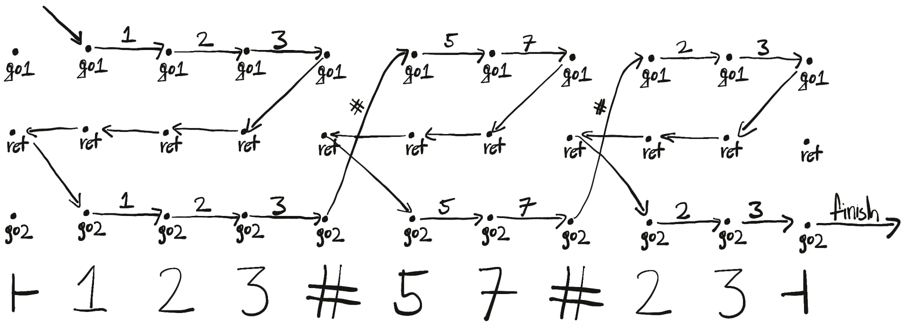
Here is an example[2] of a single-use two-way transducer that implements $f_{\text{map-dup}}$:

[1] There is the usual problem with looping – a looping transducer will never finish the run, which could mean that it defines a partial function. The simplest way to fix this is to require that the transducers do not loop. However, it would not make any difference if we assumed instead that the output of a looping run is the empty word, or even if we agreed that the two-way transductions are partial functions.

[2] Observe that this transducer only uses finitely many states. Since this is the only example of a single-use two-way transducer presented in this thesis, it is worth pointing out that single-use two-way are allowed to use polynomial orbit finite set of states.



And here is an example run of this transducer:



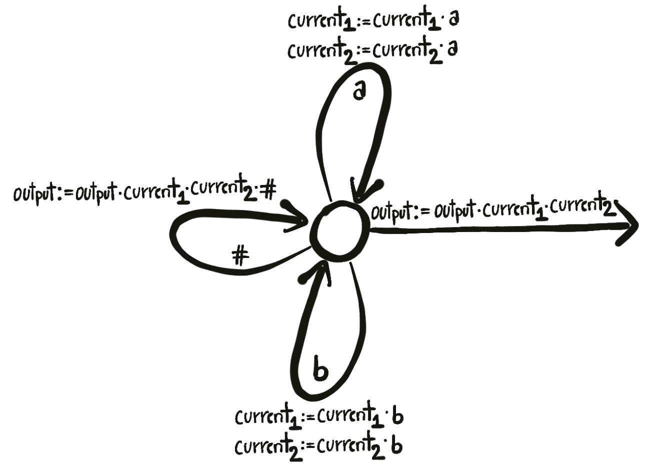
◁

4.1.2 Single-use streaming string transducers with atoms

Single-use streaming string transducers with atoms are a variant of *copyless streaming string transducers*. The original model was simultaneously defined in [AČ11, Section 2.2] and in [AČe10, Section 3] (the papers cite one another). Interestingly, the two papers provide different definitions: [AČ11] defines streaming string transducers as a model over infinite alphabets (using multiple-use atom registers, in style of [KF94]), while [AČe10] limits the definition to finite alphabets. One of the main results of [AČe10] is [AČe10, Theorem 3], which states that the finite-alphabet version of copyless streaming string transducers is equivalent to MSO-transductions. As MSO-transductions are equivalent to two-way transducers, it follows that copyless streaming string transducers benefit from the robustness of regular transductions over finite alphabets. In particular, they are closed under compositions and have decidable equivalence. On the other hand, the infinite-alphabet version from [AČ11] retains only some of those properties – it has decidable equivalence ([AČ11, Theorem 12]), but it is not closed under compositions ([AČ11, Proposition 4]). Possibly for this reason the finite-alphabet version from [AČe10] is much more prevalent in the literature.

Before defining single-use string streaming transducers with atoms, let us discuss its finite-alphabet version[3]. We start by an informal description (to be followed by a formal definition): The *copyless streaming string transducer* is a variant of a one-way automaton, which constructs its output using a finite number of string registers over the output alphabet. It can concatenate its registers (e.g. $\mathtt{r}_1 := \mathtt{r}_2 \cdot \mathtt{r}_3$), but the *copyless* restriction requires that this operation destroys the contents of the concatenated registers (i.e. $\mathtt{r}_2$ and $\mathtt{r}_3$ in the example), by overriding them with ϵ. Observe the analogy between the copyless and the single-use restrictions. A streaming string transducer is not allowed to query its registers (unlike the register automaton from Section 1.1).

Here is an example of a copyless streaming string transducer that implements the finite-alphabet version of $f_{\text{map-dup}}$ from Example 41 over the alphabet $\{a, b, \#\}$. It has only one state, and three registers: $\mathtt{current}_1$, $\mathtt{current}_2$, and $\mathtt{output}$ (due to the copyless restriction the transducer has to maintain two copies of the current block: $\mathtt{current}_1$ and $\mathtt{current}_2$):


current$_1$:=current$_1$·a
current$_2$:=current$_2$·a
a
output:=output·current$_1$·current$_2$·#
#
output:= output·current$_1$·current$_2$
b
current$_1$:=current$_1$·b
current$_2$:=current$_2$·b

[3] As mentioned before, the model was originally defined in [AČe10, Section 3]. However, our presentation differs slightly from the original one, as we want to maintain consistency with the previous chapters of this thesis.

For the formal definition, we are going to use a variant of the single-use function (from Definition 7). First, let us define the class of *polynomial* Γ^*-*register sets* (over a finite output alphabet Γ) to be the smallest class closed under $+$ and $\times$ that contains the following sets:

$$\underbrace{1}_{\text{Singleton set}} \qquad \underbrace{\Gamma^*}_{\substack{\text{Contents of a}\\ \text{string register}}}$$

Next, let us define the class of *single-use functions over polynomial* Γ^*-*register sets* to be the smallest class of functions closed under the combinators from Definition 7 (i.e. $\circ$, $\times$, and $+$), that contains all the basic functions from Definition 7 except the functions about $\mathbb{A}$, and all the following basic functions about Γ^*:

Functions about Γ^*	
$\texttt{concat}:$	$\Gamma^* \times \Gamma^* \to \Gamma^*$
$\texttt{singleton}:$	$\Gamma \to \Gamma^*$
$\texttt{const}_\epsilon:$	$1 \to \Gamma^*$

(In the type of `singleton`, we use the fact that since Γ is finite, it can be represented as the following polynomial Γ^*-register set: $1 + \ldots + 1$).

Example 42. Consider the following function $f : \{a, b\} \times (\Gamma^*)^2 \to \Gamma^*$:

$$f(a, \mathtt{r}_1, \mathtt{r}_2) = \mathtt{r}_1 \cdot \mathtt{r}_2 \quad f(b, \mathtt{r}_1, \mathtt{r}_2) = \mathtt{r}_2 \cdot \mathtt{r}_1$$

This is a single-use function over polynomial Γ^*-register sets, as it can be implemented in the following way (assuming that $\{a, b\}$ is represented as $1 + 1$):

$$(1+1) \times (\Gamma^*)^2 \xrightarrow{\texttt{distr}} (\Gamma^*)^2 + (\Gamma^*)^2 \xrightarrow{\texttt{id+sym}} (\Gamma^*)^2 + (\Gamma^*)^2 \xrightarrow{\texttt{merge}} (\Gamma^*)^2 \xrightarrow{\texttt{concat}} \Gamma^*$$

$\triangleleft$

We denote the set of all single-use functions between two polynomial Γ^*-register sets as $X \multimap Y$. Overloading the notation should not cause any confusion, as it is usually clear from the context whether X and Y are polynomial Γ^*-register sets or polynomial orbit-finite sets.

We are now ready to give a formal definition of copyless string streaming transducers for finite alphabets:

Definition 37. A *copyless streaming string transducer* of type $\Sigma^* \to \Gamma^*$ (where Σ and Γ are finite sets) consists of:

1. a polynomial Γ^*-register set Q of states;
2. an initial state $q_0 \in Q$;
3. a single-use transition function $\delta : \Sigma \to (Q \multimap Q)$; and

4. an output function $\lambda : Q \multimap \Gamma^*$.

Every coplyless string streaming transducer defines a function $\Sigma^* \to \Gamma^*$: In order to compute the output for a $w \in \Sigma^*$, the transducer processes w letter by letter, updating its state according to the transition function. After processing the entire w, it computes the output word by applying λ to its final state. $\triangleleft$

It is not hard to extend this definition to polynomial orbit-finite alphabets. First, we define the class of *polynomial orbit-finite* Γ^**-register sets* (where Γ is a polynomial orbit-finite output alphabet) to be the smallest class of sets closed under $\times$ and $+$, that contains the sets 1, $\mathbb{A}$, and Γ^*. Then, we define the class of *single-use functions over polynomial orbit-finite* Γ^**-register sets* to be the smallest class of functions that is closed under the combinators from Definition 7 (i.e. $\circ$, $\times$ and $+$), contains all basic functions from Definition 7, and contains the three basic functions about Γ^* (i.e. `concat`, $\texttt{const}_\epsilon$, and `singleton`). Now, we can define *single-use string streaming transducers* (for infinite alphabets) in the same way as in Definition 37:

Definition 38. A *single-use streaming string transducer* of type $\Sigma^* \to \Gamma^*$ (where Σ and Γ are polynomial orbit-finite sets) consists of:

1. a polynomial orbit-finite Γ^*-register set Q of states;
2. an initial state $q_0 \in Q$;
3. a single-use transition function $\delta : \Sigma \to (Q \multimap Q)$; and
4. an output function $\lambda : Q \multimap \Gamma^*$.

A single-use streaming string transducer defines a function $\Sigma^* \to_{\text{eq}} \Gamma^*$ in the same way as a finite streaming string transducer. $\triangleleft$

Finally, let us point out that if we extend single-use functions with

$$\texttt{copy}_{\mathbb{A}} : \mathbb{A} \to \mathbb{A} \times \mathbb{A},$$

we obtain a definition equivalent to the one from [AČ11, Section 2.2]. If we, instead, include the function:

$$\texttt{copy}_{\Gamma^*} : \Gamma^* \times \Gamma^* \to \Gamma^*,$$

we obtain *copyful streaming string transducers*, a model whose finite-alphabet version is studied in [FR17]. (The main result of the paper is that copyful streaming string transducers over finite alphabets have decidable equivalence, see [FR17, Section 3].) Of course, it is also possible to include both $\texttt{copy}_{\mathbb{A}}$ and $\texttt{copy}_{\Gamma^*}$ and obtain a version of the transducer that is both multiple use and copyful, but I was unable to find this variant in the literature.

4.1.3 Regular list functions with atoms

Regular list functions with atoms are based on *regular list functions* – a model for finite alphabets that was introduced in [BDK18]. One of the main results of the paper is [BDK18, Theorem 6.1], which proves that regular list functions are equivalent to MSO-transductions. This means that, similarly to two-way transducers or copyless string streaming transducers, regular list functions (over finite alphabets) exhibit the robustness of regular transductions.

Extending regular list functions to infinite alphabets is very natural. (In fact, the extension was already suggested in [BDK18, Section 7].) For this reason, we proceed directly to the definition of list functions *with atoms*. It is structurally similar[4] to the definition of single-use functions (i.e. Definition 7):

First, we define *polynomial sets with atoms* to be the smallest class of sets that contains 1 and $\mathbb{A}$, and that is closed under $\times$, $+$, and X^* (i.e. lists of finite length). Here is an example:

$$((\mathbb{A} \times \mathbb{A})^* + \mathbb{A})^*$$

It is worth pointing out that every polynomial orbit-finite Γ^*-register set (as defined in Section 4.1.2) is a polynomial set with atoms, but not vice versa. This is because polynomial sets with atoms treat X^* as an independent set constructor, whereas polynomial orbit-finite Γ^*-register sets only allow lists over one fixed polynomial orbit-finite Γ.

We are now ready to define regular list functions with atoms:

Definition 39. The class of *regular list functions with atoms* is the smallest class of functions that is closed under the combinators from Definition 7 (i.e. $+$, $\times$, and $\circ$), contains all basic functions from Definition 7, and additionally is closed under the following combinator `map` and contains the following basic

[4] In fact, the current shape of the definition of single-use functions was inspired by [BDK18, Definition 2.1].

functions:

The combinator `map`
$\dfrac{X \xrightarrow{f} Y}{X^* \xrightarrow{\texttt{map} f} Y^*}$

Functions about lists		
$\texttt{const}_\epsilon$:	$1 \to X^*$	Returns the empty lists
`cons` :	$X \times X^* \to X^*$	Adds an element to the front of a list.
`destruct` :	$X^* \to X \times X^* + 1$	Extracts the head and tail of a list (if possible).
`concat` :	$(X^*)^* \to X^*$	Flattens nested lists.
`reverse` :	$X^* \to X^*$	Reverses the list.
`blocks` :	$(X+1)^* \to (X^*)^*$	Groups elements of X into maximal blocks.
$\texttt{group}_G$:	$(X \times G)^* \to (X \times G)^*$	Computes group prefixes (see below).

Copying	
$\texttt{copy}_\mathbb{A}$	$\mathbb{A} \to \mathbb{A} \times \mathbb{A}$
$\texttt{copy}_{X^*}$	$X^* \to X^* \times X^*$

The function $\texttt{group}_G : (G \times X)^* \to (G \times X)^*$ is defined for every finite group G. It computes group prefixes on the first coordinate and leaves the second coordinate unchanged:

$$\texttt{group}_G([(g_1, x_1), (g_2, x_2), \ldots, (g_n, x_n)]) = [(g_1, x_1), (g_1 \cdot g_2, x_2), \ldots, (g_1 \cdot \ldots \cdot g_n, x_n)]$$

Finally, we define *regular list transductions* to be all regular list functions of the type $\Sigma^* \to \Gamma^*$, where Σ and Γ are polynomial orbit-finite.

◁

It is worth pointing out that if we remove $\mathbb{A}$ and related basic functions from Definition 39, we obtain the original regular list functions from [BDK18].

Observe that regular list functions are a copyful model: They include $\texttt{copy}_\mathbb{A}$ and $\texttt{copy}_{X^*}$ which allow for copying both atoms and lists. Moreover, using a construction similar to the one from Example 21, one can derive a general $\texttt{copy}_X$ function which works for every polynomial set with atoms X. Despite that, regular list functions are equivalent to the single-use versions of two-way transducers and string streaming transducers. Moreover, without the `copy` functions, regular list functions would become too weak: Without $\texttt{copy}_\mathbb{A}$, they would not be able to simulate multiple-use access to input letters, and without $\texttt{copy}_{X^*}$, they would not be able to implement the duplicate function (i.e. $w \mapsto ww$). For this reason, regular list functions are an interesting link between single-use and multiple-use models that could help us understand the connections between the two approaches. For finite alphabets, a related line of research has been recently explored in [Boj23].

4.1.4 Compositions of single-use two-way primes

In this section we define *compositions of single-use two-way primes*, which is a model based on the Krohn-Rhodes decomposition theorems.

Observe that the functions computed by two-way transducers might not preserve length, so if we want to define an equivalent class of compositions of primes, we need to include some primes that are not length-preserving. Those are going to be the $f_{\text{map-dup}}$ function (from Example 41) and *letter-to-word* homomorphisms, defined as follows:

Example 43. Let Σ, Γ be polynomial orbit-finite sets, and let $f : \Sigma \to_{\text{eq}} \Gamma^*$ be an equivariant function. (Observe that since Σ is orbit-finite and f is equivariant, it follows the length of words in $f(\Sigma)$ is bounded.) Define the *letter-to-word homomorphism based on* f to be the function $f^* : \Sigma^* \to \Gamma^*$, that applies f to every input letter and concatenates the results. For example, if we take $\Sigma = \mathbb{A} + \bot$, $\Gamma = \mathbb{A}$, and f defined as $f(a \in \mathbb{A}) = aa$ and $f(\bot) = \epsilon$, then:

$$f^*(123\bot 45\bot\bot) = 1122334455$$

$\triangleleft$

The parallel composition (i.e. ×) only makes sense for length-preserving functions, which means that we can no longer use it in the definition of composition of single-use two-way primes. Instead, we use a similar approach as in Claim 47, and define compositions of primes only in terms of ($\circ$)-compositions:

Definition 40. We define the *compositions of single-use two-way primes*, to be the smallest class of word-to-word functions that is closed under the sequential composition (i.e. $\circ$) and contains all the following prime functions:

1. *letter-to-word*[5] *homomorphisms*, i.e. functions of the form $f^* : \Sigma^* \to \Gamma^*$, for every $f : \Sigma \to_{\text{eq}} \Gamma^*$, where Σ and Γ are polynomial orbit-finite sets (the function f^* is defined as a function that applies f to every letter and concatenates the results);

2. functions of the form $p \times \mathtt{id}_{\Delta^*} : (\Sigma \times \Delta)^* \to (\Gamma \times \Delta)^*$, where $p : \Sigma^* \to \Gamma^*$ is one of the single-use prime functions from Theorem 8 (i.e. the Krohn-Rhodes decomposition theorem for single-use Mealy machines), and Δ is a polynomial-orbit-finite set;

3. *map duplicate*, i.e. function $f_{\text{map-dup}} : (\Sigma + \#)^* \to (\Sigma + \#)^*$, which is a generalization of the map duplicate function from Example 41 for an arbitrary polynomial orbit-finite alphabet Σ;

[5] In other words, a letter-to-words homomorphism $\Sigma^* \to \Gamma^*$ is simply a monoid morphism between the free monoids Σ^* and Γ^*. The phrase *letter-to-word* is used to distinguish this general class of homomorphism from the letter-to-letter homomorphisms used in the previous chapter.

4. *map reverse*, i.e. the function $f_{\text{map-rev}} : (\Sigma + \#)^* \to (\Sigma + \#)^*$ defined below in Example 44, for every polynomial orbit-finite Σ;

   Example 44. For every polynomial orbit-finite Σ, we define the *map reverse* function, which independently reverses every #-separated block:

   $$f_{\text{map-rev}} : (\Sigma + \#)^* \to (\Sigma + \#)^*$$

   For example:

   $$f_{\text{map-rev}}(1\ 2\ 3\ \#\ 5\ 7\ \#\ 1\ 2) = 3\ 2\ 1\ \#\ 7\ 5\ \#\ 2\ 1$$

   $\triangleleft$

5. *end of word marker*, i.e. the function $w \mapsto w \dashv$. This function is only[6] required to deal with the empty word, as for all other prime functions it holds that $p(\epsilon) = \epsilon$.

$\triangleleft$

4.2 Equivalence of the models

As mentioned in the introduction, all models defined in the previous section are equivalent:

Theorem 15. *All the following models recognize the same class of transductions over polynomial orbit-finite alphabets:*

1. *Single-use two-way transducers;*
2. *Single-use streaming string transducers;*
3. *Regular list transductions with atoms;*
4. *Compositions of single-use two-way primes;*

Before proving Theorem 15, let us point out its two important corollaries:

Claim 51. *Single-use two-way transducers and single-use streaming string transducers are closed under composition.*

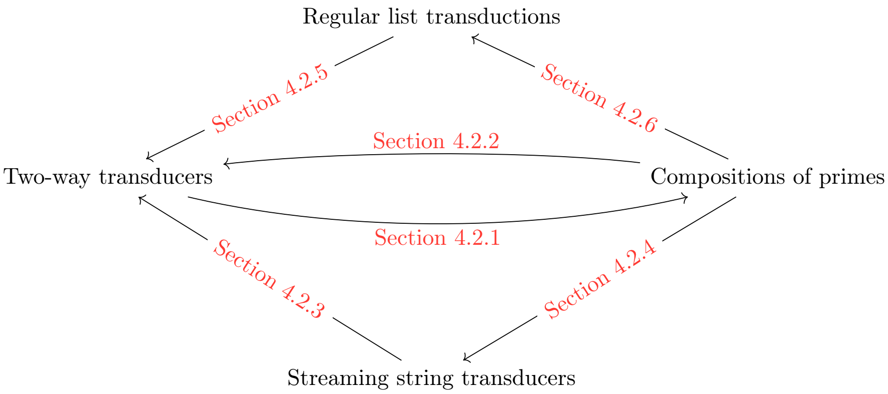
Proof. The claim is an immediate consequence of Theorem 15, as compositions of single-use two-way primes are trivially seen to be closed under compositions. □

[6] If we only want to consider non-empty words, we can skip this function. Or if we want to consider the empty word, we can replace it with the *start of word marker* (i.e. $w \mapsto\ \vdash w$). If we want a symmetric function, we can also use the *both ends marker* (i.e. $w \mapsto\ \vdash w \dashv$), or the *empty word selector*, i.e. the function which maps every word to itself, with the exception of the empty word which is mapped to a single letter ○. All of those functions result in an equivalent class of compositions of two-way single-use primes.

Claim 52. *The equivalence problem is decidable for the four models that appear in Theorem 15.*

Proof. By analysing the proof of Theorem 15 (presented later in this section), one can show that it is effective, i.e. translating between any two of the models is a computable function. This leaves us with showing that just one of the models has decidable equivalence. Let us focus on single-use string streaming string transducers: As mentioned in the last paragraph of Section 4.1.2, single-use streaming string transducers are a special case of streaming string transducers from [AČ11], which by [AČ11, Theorem 12] have decidable equivalence. It follows that single-use streaming string transducers have decidable equivalence as well. (Alternatively, we can use [BS20, Theorem 14] which is a direct proof of decidable equivalence for single-use streaming string transducers.) □

The rest of Section 4.2 is dedicated to proving Theorem 15, according to the following plan:



It is worth pointing out that Section 4.2.2 seems to be redundant. However, together with Section 4.2.1 it completes the proof of Claim 51, which is later used in Sections 4.2.3 and 4.2.5.

4.2.1 Two-way automata ⊆ Compositions of primes

In this section, we show how to translate two-way automata into compositions of single-use two-way primes. This is the most complicated part of the proof of Theorem 15. We start by showing that every non-looping single-use two-way transducer can visit every position only a bounded number of times. This is not obvious, because the number of states of a single-use two-way transducer is usually infinite – in particular, an analogous lemma does not hold for the multiple-use variant of two-way transducers.

Lemma 76. *For every non-looping single-use two-way transducer $\mathcal{A}$, there is a bound k, such that $\mathcal{A}$ visits every position in every input word at most k times.*

Proof. Let $\mathcal{A}$ be a transducer of type $\Sigma^* \to \Gamma^*$ and let Q be $\mathcal{A}$'s set of states. Consider the following semigroup $S_\mathcal{A}$ of $\mathcal{A}$'s behaviours (see the proof of Lemma 40 for details):

$$Q \times \{\leftarrow, \rightarrow\} \multimap Q \times \{\leftarrow, \rightarrow\} + \text{finish}$$

Thanks to Theorem 5, we know that $S_\mathcal{A}$ is orbit-finite. It follows that there is a bound p such that every element of $S_\mathcal{A}$ is supported by at most p atoms. Similarly there is a q such that every letter in Σ is supported by at most q atoms. Moreover, since Q is orbit-finite, it follows that every finite set of atoms supports only a finite number of elements in Q (this follows from [Boj13, Lemma 5.2] or alternatively, since Q is *polynomial* orbit-finte, it can also be shown by structural induction on Q). Additionally, it is not hard to see that the number of elements in Q that are supported by a subset of atoms depends only on the subset's size. It follows that there is a function f_Q, such that $f_Q(n)$ is the number of elements in Q that are supported by a subset of n atoms. We are going to show that $\mathcal{A}$ visits each position no more than $k := f_Q(2p + q)$ times.

Let us take a word $w \in \Sigma^*$, a position i in w, and let us show that $\mathcal{A}$ visits i at most k times. First, we split w into the prefix $w_{<i} \in \Sigma^*$, the letter $w_i \in \Sigma$, and the suffix $w_{>j} \in \Sigma^*$. Define $b_{<i}, b_{>j} \in S_\mathcal{A}$ to be the behaviour of $\mathcal{A}$ on $w_{<i}$ and $w_{>j}$. By Lemma 3 (applied multiple times), we know that every state in which $\mathcal{A}$ visits i, can contain only the atoms that appear in $b_{<i}$, $b_{>j}$, or w_i. It follows that there are at most $f_Q(2 \cdot p + q)$ different states in which $\mathcal{A}$ can visit i. Since $\mathcal{A}$ is non-looping, it follows that each of those states is visited at most once. This means that the number of visits in i is bounded by $k = f_Q(2 \cdot p + q)$. □

A useful abstraction for the construction of translating two-way automata into compositions of primes is the *shape of a run*, which is a record of all visits of $\mathcal{A}$ in each of the input position. A visit is represented as an element of the following set:

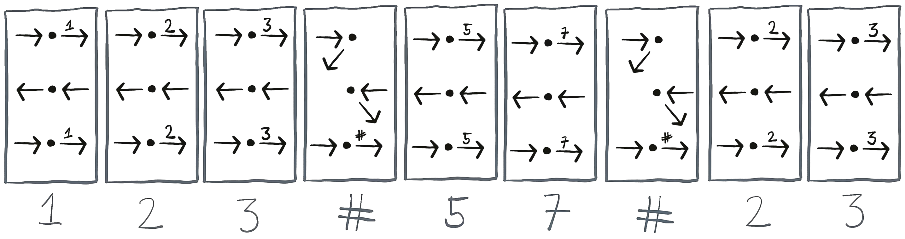
$$\underbrace{\{\leftarrow, \rightarrow\}}_{\substack{\text{wheather } \mathcal{A} \text{ entered} \\ \text{from left or right}}} \times \underbrace{\{\leftarrow, \rightarrow\}}_{\substack{\text{wheather } \mathcal{A} \text{ left} \\ \text{towards left or right}}} \times \underbrace{(\Gamma + \epsilon)}_{\substack{\text{the letter that } \mathcal{A} \text{ outputs} \\ \text{when leaving the position}}}$$

For the sake of simplicity, let us assume that $\mathcal{A}$ finishes all of its runs $\dashv$, and that it never outputs any letters in $\dashv$ or in $\vdash$. (It is not hard to see that every $\mathcal{A}$ can be transformed into an equivalent $\mathcal{A}'$ that satisfies this restriction[7].) By Lemma 76, we know that each position admits at most k visits. It follows that the shape of the run can be represented as a word over the following alphabet:

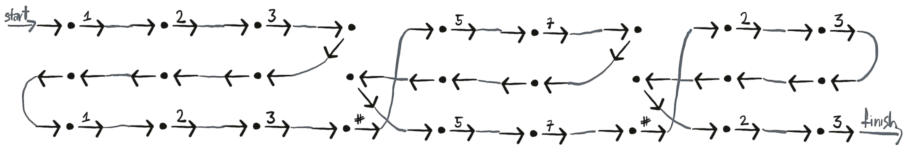
$$(\{\leftarrow, \rightarrow\} \times \{\leftarrow, \rightarrow\} \times (\Gamma + \epsilon))^{\leq k},$$

[7] The only problem arises when the input word is empty – in this case, the restriction prohibits the automaton from outputting any letters at all. One way to deal with this problem is to start the translation, by applying the end-of-word marker prime function. This, way we equip the input word with a copy of $\dashv$ (followed by the actual $\dashv$). Now $\mathcal{A}'$ can use this extra letter to construct its output for ϵ.

where each position stores its visits in chronological order. For example, here is the shape of the run from Example 41:



Observe that the shape of $\mathcal{A}$'s run uniquely determines its output – thanks to the chronological order of the events we can retrace the steps of the automaton:



We split the construction of $\mathcal{A}$ as a composition of primes into two steps. First, we show how to use compositions of primes to construct the shape of $\mathcal{A}$'s run over the input word. Then, we show how to use compositions of primes to transform the shape of a run into the output.

4.2.1.1 Constructing the shape of the run

In this section, we show that the following function, that outputs the shape of $\mathcal{A}$'s run over its input word, can be constructed as a composition of primes:

$$f_{\mathcal{A}\text{-shape}} : \Sigma^* \ \to \ \Big((\{\leftarrow, \rightarrow\} \times \{\leftarrow, \rightarrow\} \times (\Gamma + \epsilon))^{\leq k} \Big)^*$$

It is enough to show that $f_{\mathcal{A}\text{-shape}}$ is a local rational semigroup transduction. This is because, by Theorem 14, every local rational semigroup transduction can be decomposed into single-use rational primes, which can be further decomposed into single-use two-way primes:

Claim 53. *Every composition of single-use rational primes is also a composition of single-use two-way primes.*

Proof. Thanks to Claim 47, it is enough to show that for every rational single-use prime function p, the function $p \times \mathtt{id}_\Sigma$ is a composition of single-use two-way primes. For homomorphisms and left-to-right functions, this is trivial. For the right-to-left multiple-use bit propagation, we can use the following decomposition:

$$f_{\overleftarrow{\text{prop}}} \times \text{id}_\Sigma = \text{reverse} \circ (f_{\overrightarrow{\text{prop}}} \times \text{id}_\Sigma) \circ \text{reverse},$$

where the reverse function is easily seen to be a special case of $f_{\text{map-reverse}}$ with no #'s. We finish the proof, by observing that we can use the same approach for all other right-to-left rational prime functions. $\square$

This leaves us with showing that $f_{\mathcal{A}\text{-shape}}$ is a local rational semigroup transduction. For that we take S to be the semigroup of $\mathcal{A}$'s behaviours as described in Section 2.3.5:

$$\underbrace{Q}_{\substack{\text{In what state} \\ \text{does } \mathcal{A} \text{ enter} \\ \text{the word}}} \times \underbrace{\{\leftarrow,\rightarrow\}}_{\substack{\text{Does } \mathcal{A} \text{ enter} \\ \text{from the right} \\ \text{from the left}}} \multimap \underbrace{Q}_{\substack{\text{In what state} \\ \text{does } \mathcal{A} \text{ exit} \\ \text{the word}}} \times \underbrace{\{\leftarrow,\rightarrow\}}_{\substack{\text{Does } \mathcal{A} \text{ exit} \\ \text{from the right} \\ \text{from the left}}} \times \underbrace{(\Gamma+\epsilon)}_{\substack{\text{What letter} \\ \text{does } \mathcal{A} \text{ outputs} \\ \text{when it leaves} \\ \text{the word}}} + \text{finish}$$

It is not hard to see that for every $u, w, v \in \Sigma^*$, the w-part of $\mathcal{A}$'s run on uwv depends only on w and on $\mathcal{A}$'s behaviours of u and v. It follows that there is a function:

$$\lambda : S \times \Sigma \times S \to_{\text{eq}} (\{\leftarrow,\rightarrow\} \times \{\leftarrow,\rightarrow\} \times \Gamma)^{\leq k},$$

that computes the shape of the run on the single-letter infix. (It is not hard to see that this function is equivariant.) This means that $f_{\mathcal{A}\text{-shape}}$ can be implemented as (S, h, λ), where S and λ are as described above, and h is a function that maps single-letter words to their behaviours. The hard part of the proof is showing that λ satisfies the locality equation, i.e. that:

$$\lambda(x_1 e y_1, a, y_2 e x_2) = \lambda(\pi(x_1)\, e\, y_1, a, y_2\, e\, \pi(x_2)),$$

provided that e is idempotent, $y_1 \cdot h(a) \cdot y_2 = e$, and π is a supp(e)-permutation. We show this by using a slightly stronger result: Remember that the w-part of the shape of $\mathcal{A}$'s run on uwv depends only on w and on $\mathcal{A}$'s behaviours on u and v. It follows that we can extend λ to:

$$\lambda' : S \times \Sigma^* \times S \to_{\text{eq}} \left(\{\leftarrow,\rightarrow\} \times (\{\leftarrow,\rightarrow\} \times \Gamma)^{\leq k}\right)^*,$$

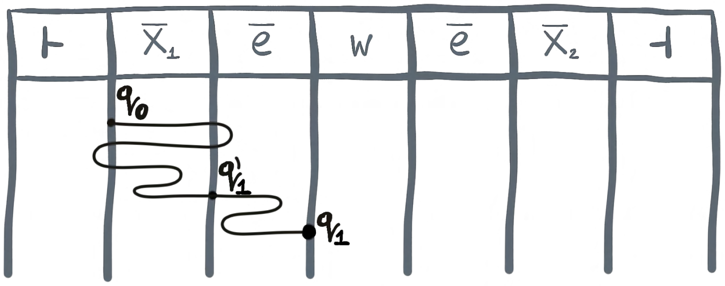
Now, in order to proof the locality of λ, we take some words $\bar{y}_2, \bar{y}_2 \in \Sigma^*$ whose $\mathcal{A}$-behaviours are equal to y_1 and y_2 (thanks to a reasoning similar to Claim 41, we can assume that S only contains behaviours correspond to actual words) and apply the following lemma for x_1, x_2, e and $w = \bar{y}_2 a \bar{y}_2$:

Lemma 77. *Let $e \in S$ be an idempotent behaviour, and let w be a word whose behaviour is equal to e. For all behaviours x_1, x_2 and for every supp(e)-permutation π, it holds that:*

$$\lambda'(x_1 e, w, e x_2) = \lambda'(\pi(x_1)\, e, w, e\, \pi(x_2))$$

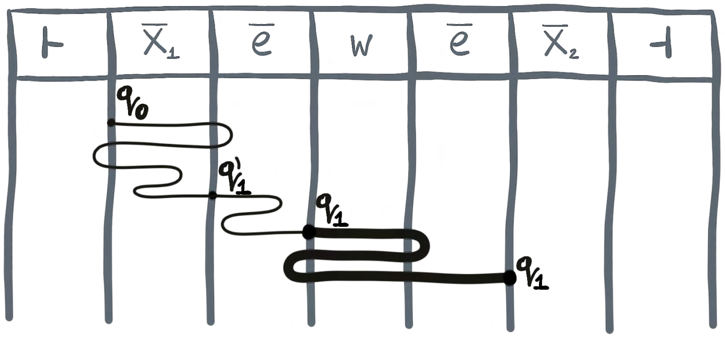
Proof. In order to prove the lemma, we pick some $\bar{x}_1, \bar{x}_2, \bar{e} \in \Sigma^*$ (again, thanks to a similar reasoning as in Claim 41, we can assume that those words exist), and we trace and compare the runs of $\mathcal{A}$ on $\vdash \bar{x}_1\, \bar{e}\, w\, \bar{e}\, \bar{x}_2 \dashv$ and on $\vdash \pi(\bar{x}_1)\, \bar{e}\, w\, \bar{e}\, \pi(\bar{x}_2) \dashv$.

Consider the first part of the run, that starts in the initial state q_0 and ends when $\mathcal{A}$ enters the w-part of its input:



First, let us consider the run on $\vdash \bar{x}_1\, \bar{e}\, w\, \bar{e}\, \bar{x}_2 \dashv$: We define q_1 be the state in which $\mathcal{A}$ first enters w and q_1' to be the last state in which $\mathcal{A}$ enters $\bar{e}$, before entering w (see the picture). Let us now notice that analogous states for $\vdash \pi(\bar{x}_1)\, \bar{e}\, w\, \bar{e}\, \pi(\bar{x}_2) \dashv$ are equal to $\pi(q_1)$ and $\pi(q_1')$. This is because those states depend equivariantly on the behaviour of the prefix up until w, which in this case is equal to $\pi(x_1) \cdot e = \pi(x_1) \cdot \pi(e) = \pi(x_1 \cdot e)$.

Consider now the second part of the run, which lasts until $\mathcal{A}$ leaves the $\bar{e}w\bar{e}$-part of the input. Let us show that this part of the run exits the $\bar{e}w\bar{e}$-part of the input on the right in q_1 (or $\pi(q_1)$):



Observe that the behaviours of both $\bar{e}$ and w and equal to e. Since e is idempotent, it follows that the behaviour of the word $\bar{e}w\bar{e}$ is also equal to e. By definition of q_1', we know that $e(q_1', \rightarrow) = (q_1, \rightarrow)$. It follows that the second part of the run exits $\bar{e}w\bar{e}$ from the right in state q_1:

$$(\bar{e}w\bar{e})(q_1', \rightarrow) = e(q_1', \rightarrow) = (q_1, \rightarrow)$$

Now, let us show that during this second part of the run (marked as a bold line), $\mathcal{A}$ has to preserve all atoms from q_1 that do not appear in $\operatorname{supp}(e)$ – in particular, this means that $\mathcal{A}$ cannot query or output those atoms. The proof is analogous to the one in Section 3.3.3. First, let us consider the following function, which describes the behaviour of $\mathcal{A}$ on $\bar{e}w\bar{e}$, when it enters w from the left:

$$(\bar{e}↳w\bar{e}) : \quad \underbrace{Q}_{\substack{\text{The state in which}\\ \mathcal{A} \text{ is placed in}\\ \text{the first letter of } w.}} \quad \multimap \quad \underbrace{Q \times \{\rightarrow, \leftarrow\}}_{\substack{\text{The state and the direction}\\ \text{in which } \mathcal{A} \text{ exits } \bar{e}w\bar{e}}}$$

By an argument similar to the one in Claim 19, we know that $(\bar{e}↳w\bar{e})$ is, indeed, a single-use function. Moreover, it is not hard to see that $(\bar{e}↳w\bar{e})$ depends (in an equivariant way) only on the behaviours on $\bar{e}$ and w, which are both equal to e. By Lemma 3 it follows that:

$$\mathrm{supp}(\bar{e}↳w\bar{e}) \subseteq \mathrm{supp}(e)$$

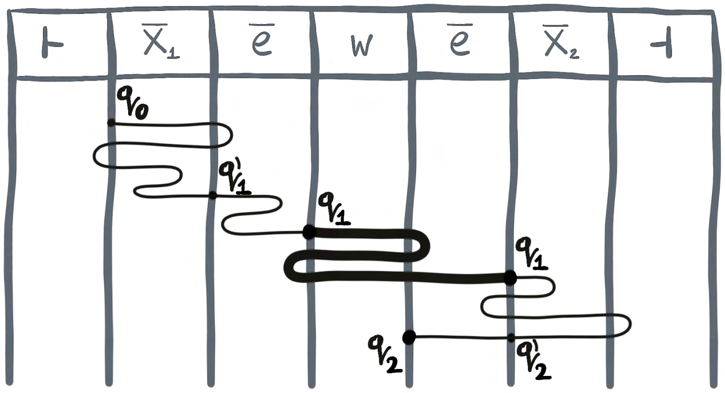
Moreover since $e(\rightarrow, q_1') = q_1$, we know that:

$$(\bar{e}↳w\bar{e})(q_1) = (ewe)(\rightarrow, q_1') = e(\rightarrow, q_1') = (\rightarrow, q_1)$$

As the second part of the run corresponds to $(\bar{e}↳w\bar{e})(q_1)$, it follows that it can only destroy only those atoms from q_1 that appear in $\mathrm{supp}(e)$. This is because each atom from q_1 that is destroyed during the second part of the run has to be restored before $\mathcal{A}$ exits $\bar{e}w\bar{e}$, as $(\bar{e}↳w\bar{e})(q_1) = q_1$. By a reasoning similar to the one from Section 3.3.3, we know that each such restored atom has to appear in $\mathrm{supp}(\bar{e}↳w\bar{e})$, and we know that $\mathrm{supp}(\bar{e}↳w\bar{e}) \subseteq \mathrm{supp}(e)$.

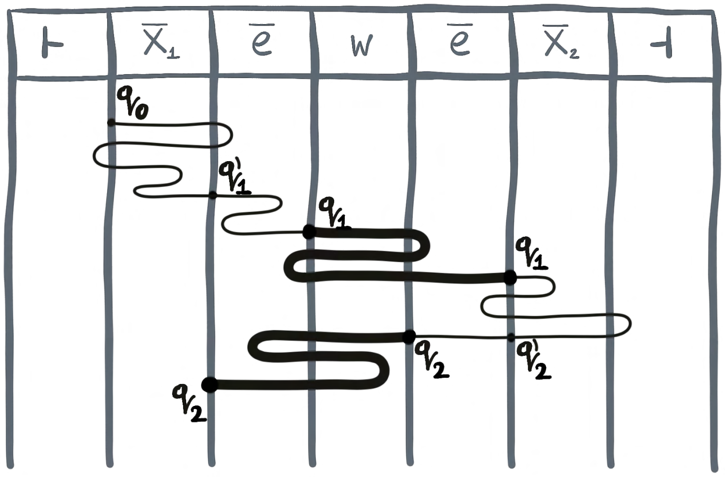
Now, let us consider the second part of $\mathcal{A}$'s run on $\vdash\ \pi(\bar{x}_1)\,\bar{e}w\bar{e}\,\pi(\bar{x}_2)\ \dashv$, i.e. the part that starts in $\pi(q_1)$ (in the first letter of w), and ends when $\mathcal{A}$ leaves $\bar{e}w\bar{e}$. Thanks to the same arguments as for $\vdash \bar{x}_1\bar{e}w\bar{e}\bar{x}_2 \dashv$, we know that the second part of the run also leaves $\bar{e}w\bar{e}$ from the right in $\pi(q_1)$, and during the second part of the run, $\mathcal{A}$ has to preserve all atoms from $\pi(q_1)$ that do not appear in $\mathrm{supp}(e)$. In particular, this means that $\mathcal{A}$ does not query or output any atoms from $\pi(q_1)$ that do not appear in $\mathrm{supp}(e)$. Since π is a $\mathrm{supp}(e)$-permutation, it follows that, in the second part of the run, $\mathcal{A}$ cannot distinguish between q_1 and $\pi(q_1)$, as the only difference between the two states are the atoms outside of $\mathrm{supp}(e)$. It follows that the second part of the run has the same shape when starting in q_1 and in $\pi(q_1)$. In particular, this means that the shape of the w-part in the second part of the run is the same on both $\bar{x}_1\bar{e}w\bar{e}\bar{x}_2$ and $\pi(\bar{x}_1)\bar{e}w\bar{e}\pi(\bar{x}_2)$.

Now, let us consider the third part of the run – from exiting $\bar{e}w\bar{e}$ in q_1 (or $\pi(q_1)$) until reentering w:



We define q_2 and q'_2 analogously to q_1 and q'_1, i.e. q_2 is the state in which $\mathcal{A}$ reenters w, and q'_2 is the last state in which $\mathcal{A}$ enters $\bar{e}$, before reentering w. Similarly as before, we observe that the analogous states for $\vdash \pi(\bar{x}_1)\, \bar{e}\, w\, \bar{e}\, \pi(\bar{x}_2) \dashv$ are equal to $\pi(q_2)$ and $\pi(q'_2)$.

Let us now consider the fourth part of the run, which starts in q_2 (or $\pi(q_2)$) and continues until $\mathcal{A}$ leaves $\bar{e}w\bar{e}$:



An analysis, similar to the one for the second part of the run, shows that $\mathcal{A}$ leaves $\bar{e}w\bar{e}$ from the left in q_2 (or in $\pi(q_2)$), and that both in the run on $\bar{x}_1\bar{e}w\bar{e}\bar{x}_2$ and on $\pi(\bar{x}_1)\bar{e}w\bar{e}\pi(\bar{x}_2)$, $\mathcal{A}$ cannot query atoms from q_2 (or $\pi(q_2)$) that do not appear in $\mathrm{supp}(e)$. It follows, by the same argument as before, that the fourth part of the run has the same shape for both $\vdash \bar{x}_1\bar{e}w\bar{e}\bar{x}_2 \dashv$ and $\vdash \pi(\bar{x}_1)\, \bar{e}\, w\, \bar{e}\, \pi(\bar{x}_2) \dashv$. In particular, this means that during the fourth part of the run, the w-parts of the two runs have equal shapes.

We finish the proof by observing that a similar reasoning can be continued until the end of the two runs. □

4.2.1.2 Untangling the run graphs

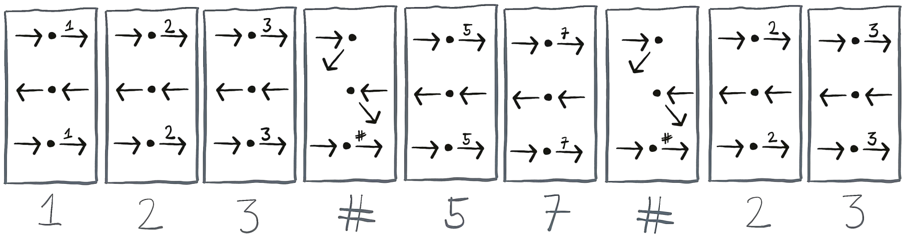
In this section, we show how to use compositions of single-use two-way primes to untangle the run graphs:

Lemma 78. *For every polynomial orbit-finite Γ, and for every $k \in \mathbb{N}$, the following function can be constructed as a composition of single-use two-way primes:*

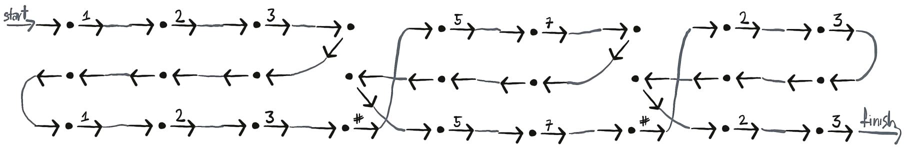
$$f_{untangle}: \underbrace{\left(\left(\{\leftarrow,\rightarrow\}^2 \times (\Gamma+\epsilon)\right)^{\leq k}\right)^*}_{\textit{encoding of a shape of the run}} \rightarrow \underbrace{\Gamma^*}_{\textit{the untangled string}}$$

We assume that the input shape is a single path, i.e. all nodes except the initial one have exactly one successor, and all nodes except the final one have exactly one predecessor.

For example, consider the following input:



It corresponds to the following *run graph*:



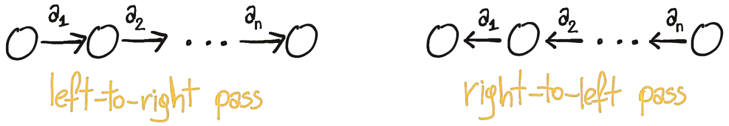
Which means that the function f_{untangle} should return the following word:

123123#5757#2323

This section is entirely dedicated to proving Lemma 78. We present the same proof as in [BS20, Section E.4.3]. Interestingly, the proof looks almost the same as it would for a finite[8] Γ.

[8] To the best of my knowledge, the proof for a finite Γ was first presented in [Boj18, Section 6.1]. We present the proof following the lines of [BS20, Section E.4].

The proof goes by induction on k, which represents the *width* of the run graph, i.e. the maximal number of times a position is visited. For the induction base, we notice that there are only two possible types of run graphs for $k = 1$:



Both of those cases are easy to untangle: The left-to-right pass is almost already untangled – it suffices to apply a homomorphism that extracts the letters. For the right-to-left pass, we can use the same homomorphism followed by the reverse function. Unexpectedly, the hardest part of the induction base is combining the two procedures into a single function. We do this using the following lemma:

Lemma 79. *Let $L \subseteq \Sigma^*$ be a language recognized by a single-use two-way automaton. If $f_1 : \Sigma^* \to \Gamma^*$ and $f_2 : \Sigma^* \to \Gamma^*$ are both compositions of two-way primes, then so is the following function:*

$$(\texttt{if } L \texttt{ then } f_1 \texttt{ else } f_2)(w) = \begin{cases} f_1(w) & \textit{if } w \in L \\ f_2(w) & \textit{if } w \notin L \end{cases}$$

Proof. The construction consists of the following six steps:

$$\Sigma^* \xrightarrow{w \mapsto w\dashv} (\Sigma + \dashv)^* \xrightarrow{f_L} (\Sigma + \{\texttt{Yes}, \texttt{No}\})^* \xrightarrow{f_{\text{colour}}} (\Sigma + \Sigma)^*$$

$$(\Sigma + \Sigma)^* \xrightarrow{f_1 + \texttt{id}} (\Gamma + \Sigma)^* \xrightarrow{\texttt{id} + f_2} (\Gamma + \Gamma)^* \xrightarrow{\texttt{merge}^*} \Gamma^*,$$

1. First, we equip the input word with the end-of-word marker. This function (i.e. $w \mapsto w\dashv$) is easily seen to be a composition of a rational transduction, that underlines the last letter of the input, with a letter-to-word homomorphism that inserts $\dashv$ after the underlined letters. (Thanks to Claim 53, the rational transduction can be further decomposed into two-way primes.)

2. Next, we transform the end of word marker $\dashv$ into either `Yes` or `No` depending on whether the input word belongs to L. This step is implemented as the following function $f_L : (\Sigma + \dashv)^* \to (\Sigma + \{\texttt{Yes}, \texttt{No}\})^*$:

   $$f_L(w \dashv) = \begin{cases} w\,\texttt{Yes} & \text{if } w \in L \\ w\,\texttt{No} & \text{if } w \notin L \end{cases}$$

   To see that f_L is a composition of primes, observe that thanks to Theorem 6, L can be recognized by a one-way single-use automaton. This automaton can be easily modified into a single-use Mealy machine that recognizes f_L. It follows by Theorem 8 and Claim 53 that f_L is a composition of single-use two-way primes.

3. Next, we propagate the output of f_L, by colouring each letter into either blue or yellow depending on whether the input word belongs to L. For this, we use the following function:

$$f_{\text{colour}} : (\Sigma + \{\texttt{Yes}, \texttt{No}\})^* \to (\underbrace{\Sigma}_{\text{blue copy}} + \underbrace{\Sigma}_{\text{yellow copy}})^*,$$

It is not hard to see that f_{colour} can be implemented by a right-to-left single-use Mealy machine. It follows, by (an analogue) of Theorem 8 and Claim 53 that f_{colour} is a composition of single-use two-way primes.

4. Next, we apply the function $(f_1 + \texttt{id})$, defined in the following claim:

Claim 54. *If $f : \Sigma^* \to \Gamma^*$ is a composition of single-use two-way primes then so is the following function $(f + \texttt{id}) : (\Sigma + \Delta)^* \to (\Gamma + \Delta)^*$:*

$$(f + \texttt{id})(w) = \begin{cases} f(w) & \textit{if all letters in } w \textit{ belong to } \Sigma \\ w & \textit{if all letters in } w \textit{ belong to } \Delta \\ \text{(unspecified)} & \textit{if } w \textit{ contains letters from both } \Sigma \textit{ and } \Delta \end{cases}$$

Proof. The proof goes by induction on the construction of f as a composition of single-use two-way primes. The induction step follows from the following observation:

$$(g \circ h) + \texttt{id} = (g + \texttt{id}) \circ (h + \texttt{id}),$$

This leaves us with the induction base, which states that for every single-use two-way prime p, the function $(p + \texttt{id})$ is a composition of single-use two-way primes. We only show it for $p = f_{\text{su-prop}} \times \texttt{id}_X$, and $p = f_{\text{map-rev}}$, as other cases are either trivial or analogous.

First, we show how to construct $f_{\text{su-prop}} \times \texttt{id}_X \; + \; \texttt{id}$. Here is its type:

$$((\mathbb{A} + \{\downarrow, \epsilon\}) \times X \; + \; \Delta)^* \quad \to \quad ((\mathbb{A} + \epsilon) \times X \; + \; \Delta)^*$$

We start the construction with a homomorphism that equips every letter from Δ with ϵ – i.e. the neutral letter of $f_{\text{su-prop}}$. This gives us a word over $(\mathbb{A} + \{\downarrow, \epsilon\}) \times (X + \Delta)$. Then, we apply $f_{\text{su-prop}} \times \texttt{id}_{X+\Delta}$, obtaining a word over $(\mathbb{A} + \epsilon) \times (X + \Delta)$. Finally, we use homomorphism to remove the ϵ's from Δ.

This leaves us with constructing $(f_{\text{map-rev}} + \texttt{id})$, whose type is:

$$((\Sigma + \#) + \Delta)^* \to ((\Sigma + \#) + \Delta)^*$$

We start the construction with a letter-to-word homomorphism that expands every a from Δ into $\#a\#$, and keeps elements of $(\Sigma + \#)$ unchanged.

Then, we apply $f_{\text{map-rev}}$. Finally, we remove all #'s that are adjacent to at least one letter from Δ. (This last step is a composition of a local rational semigroup transduction that underlines neighbours of Δ, and a letter-to-word homomorphism that removes underlined #'s.) □

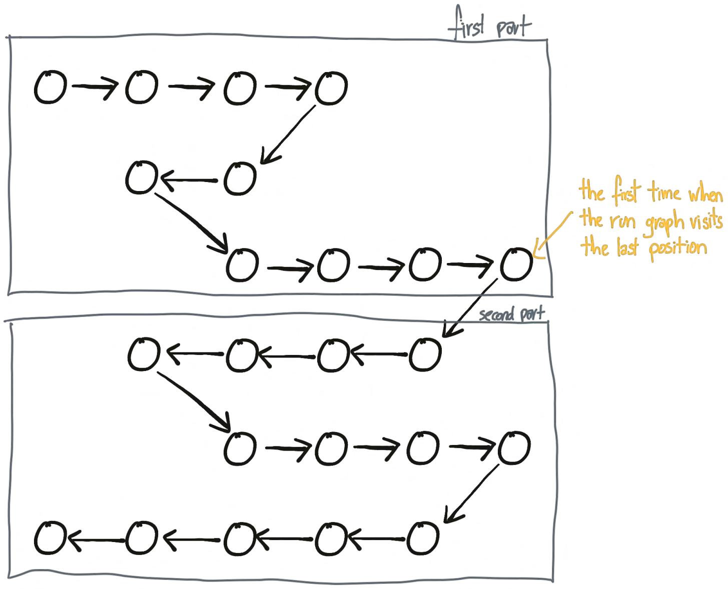
5. Next, we apply $\texttt{id} + f_2$, defined analogously to $f_1 + \texttt{id}$.

6. Finally, we forget about the colours by applying homomorphism:

$$\texttt{merge}^* : (\Gamma + \Gamma)^* \to \Gamma^*$$

□

This finishes the proof of the induction base for Lemma 78. We start the proof of the induction step, with the special case of *right loops*, which are those run graphs whose both initial and final nodes belong to the first position. In order to untangle a right loop, let us consider the following way of dividing it into two parts: the first one contains all the nodes up to (and including) the first visit in the last position, and the second one contains all other nodes. Here is an example:

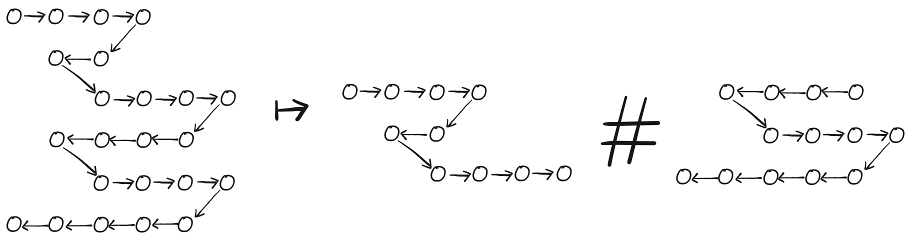


The idea behind this division is that the width of either part is smaller than the width of the whole run graph – this will enable us to apply the induction assumption. Moreover, the division can be constructed as a composition of primes:

Lemma 80. *The following function $f_{loop\text{-}div}$, which inputs a right loop and splits it into two #-separated parts (as described earlier), can be constructed as a composition of single-use two-way primes.*

$$f_{loop\text{-}div} : \left((\{\leftarrow,\rightarrow\}^2 \times (\Gamma + \epsilon))^{\leq k}\right)^* \rightarrow \left((\{\leftarrow,\rightarrow\}^2 \times (\Gamma + \epsilon))^{\leq k-1} + \#\right)^*,$$

Here is an example:



Proof. First, we show how to colour each input node into yellow or blue, depending on whether it belongs to the first or to the second part of the decomposition. We start with the special case where Γ is the singleton set, i.e. $\Gamma = 1$. In this case, both the input and the output alphabets are finite, so it is enough to construct the colouring as an unambiguous Mealy machine – thanks to Lemma 72, Theorem 13, and Lemma 53, we know that the unambiguous Mealy machine can be decomposed into single-use two-way primes.

The unambiguous Mealy machine uses nondeterminism to guess the colour of each node, and verifies that the colouring is correct, by checking a few local conditions:

1. the initial node is yellow;
2. the first node in the last position is yellow, and its successor is blue;
3. no yellow node is followed by a blue node.

The machine is unambiguous because there is only one correct colouring.

Now, let us go back to the general case: Since the output alphabet is now infinite, we cannot use an unambiguous Mealy machine. However, it is not hard to see the colouring does not depend on the Γ-values, which means that we can reduce the polynomial orbit-finite case to the finite case: First, we apply a homomorphism $f^*_{\Gamma-\text{extr}}$, where $f_{\Gamma-\text{extr}}$ is a function that splits each input letter into its Γ-free shape and its Γ-labels:

$$f_{\Gamma-\text{extr}} \quad : \quad \underbrace{(\{\leftarrow,\rightarrow\}^2 \times (\Gamma + \epsilon))^{\leq k}}_{\text{The input letter}} \quad \longrightarrow \quad \underbrace{(\{\leftarrow,\rightarrow\}^2)^{\leq k}}_{\substack{\text{The } \Gamma\text{-free shape part of} \\ \text{the input letter}}} \quad \times \quad \underbrace{(\Gamma + \epsilon)^{\leq k}}_{\substack{\text{The labels of} \\ \text{the input letter}}}$$

Next, we use an unambiguous Mealy machine to construct the colouring for the Γ-free version (using Claim 55 defined below) and finally, we use a homomorphism to transfer the Γ-labels back to the run graph.

Claim 55. *Let A and B be finite alphabets, and let $f : A^* \to B^*$ be a function recognized by an unambiguous Mealy machine. It follows that for every polynomial orbit-finite C, the following function is a composition of rational single-use primes (which by Lemma 53 means that it is also a composition of single-use two-way primes):*

$$f \times \mathtt{id}_C : (A \times C)^* \to (B \times C)^*$$

Proof. The claim is a direct consequence of Lemma 72 and Theorem 14. □

Once we have coloured the nodes, we can easily produce the output of $f_{\text{loop-div}}$: First, we duplicate the coloured run (using $f_{\text{map-dup}}$ with no separators), and then we apply a single-use Mealy machine that filters out all blue nodes from the first copy and all yellow nodes from the second copy. □

After dividing the right loop into the two parts, we can untangle it by independently untangling each of the parts (using f_{untangle} from the induction assumption), and concatenating the results:

$$\left((\{\leftarrow, \rightarrow\}^2 \times \Gamma)^{\leq k-1} + \#\right)^* \xrightarrow{\mathtt{map}\ f_{\text{untangle}}} (\Gamma + \#)^* \xrightarrow{\text{concat}} \Gamma^*$$

The list-flattening function concat : $(\Gamma + \#)^* \to \Gamma$ is a simple letter-to-word homomorphism that filters out all #'s, and the `map` combinator is defined by the following lemma:

Lemma 81. *If $f : A^* \to B^*$ is a composition of single-use two-way primes, then so is the following function* `map` f*, which applies f independently to each #-separated block:*

$$\mathtt{map}\ f : (A + \#)^* \to (B + \#)^*$$

Proof. We start the proof by noticing that:

$$\mathtt{map}(f \circ g) = (\mathtt{map}\ f) \circ (\mathtt{map}\ g)$$

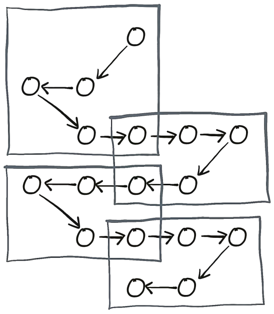
This leaves us with showing that for every prime function p, the function `map` p is a composition of primes. Most of the cases are either easy or handled analogously as in the proof of Lemma 56. The only interesting cases are $p = f_{\text{map-rev}}$ and $p = f_{\text{map-dup}}$. Moreover, the two cases are analogous, so we only show how to construct `map` $f_{\text{map-rev}}$. Observe that it uses two types of separators:

$$(\Sigma + \underbrace{\#_1}_{\substack{\text{separator}\\ \text{for } \mathtt{map}}} + \underbrace{\#_2}_{\substack{\text{separator}\\ \text{for } f_{\text{map-rev}}}})^* \to (\Sigma + \underbrace{\#_1}_{\substack{\text{separator}\\ \text{for } \mathtt{map}}} + \underbrace{\#_2}_{\substack{\text{separator}\\ \text{for } f_{\text{map-rev}}}})^*$$

However, as one can easily verify, both of those separators are treated in the same way: `map` $f_{\text{map-rev}}$ is a version of $f_{\text{map-rev}}$ that treats both $\#_1$ and $\#_2$ as its separator. It follows that we can implement `map` $f_{\text{map-rev}}$ by mapping both $\#_1$ and $\#_2$ to #, and applying $f_{\text{map-rev}}$. Finally, we have to restore the #'s back to $\#_1$ or $\#_2$. In order for this to be possible, we need to modify the first step: instead mapping both $\#_1$ and $\#_2$ to #, we map them respectively to

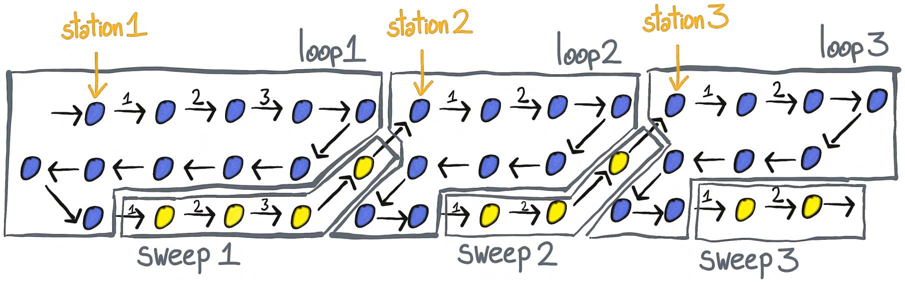
$a_1\#$ and $a_2\#$ (where a_1 and a_2 are letters that do not appear in Σ). After this modification, we can map the $\#$'s back to $\#_1$ and $\#_2$ using a single-use Mealy machine. □

Let us now deal with arbitrary *loops*, i.e. run graphs that start and finish in the same position. It is not hard to see that they can be split into at most k left loops and right loops, as illustrated by the following example:



Observe that, thanks to a reasoning similar to the one from Lemma 80, we can construct this decomposition as a composition of single-use two-way primes. This way, we reduce untangling an arbitrary loop into untangling right loops and left loops. This finishes the construction, as left loops can be untangled analogously to right loops.

Finally, let us show how to untangle arbitrary run graphs. Without loss of generality, we assume that the initial node is to the left of the final node – the other case can be handled analogously. First, let us inductively define *stations*, *sweeps* and *loops* of a run graph: The *first station* is the position that contains the initial node. The *i-th loop* is the part of the run between the first and the last visit in the i-th station. The $(i+1)$*-st station* is the first position to the right of the i-th station, that was not visited by the i-th loop. Finally, the *i-th sweep* is the part of the run graph between the last visit in the i-th station and the first visit in the $(i+1)$-th station. Here is an example:

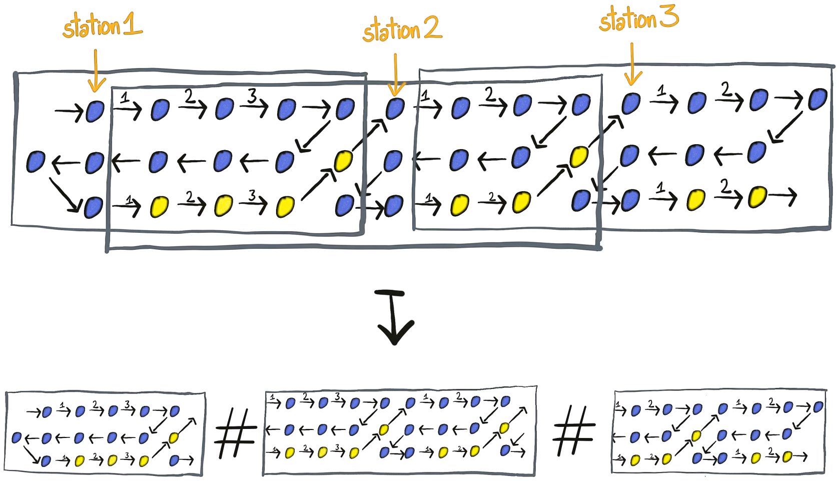


Notice that every position that is visited by the i-th sweep is also visited by the i-th loop, so the width of each sweep is smaller than the width of the original graph – this will allow us to untangle the sweeps using the induction assumption.

Here is the procedure for untangling a run graph:

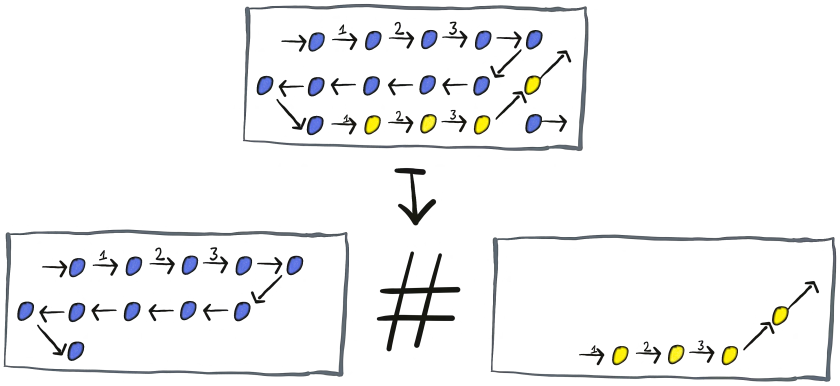
1. First, we underline all stations, and colour each node into either yellow or blue, depending on whether it belongs to a loop or to a sweep – the construction is analogous to the one for the right-loop decomposition from Lemma 80.

2. Next, we transform a run graph into a #-separated list of its *windows*, where the i-th window is defined to be the maximal interval that contains the i-th station and no other station. (Note that the windows are usually overlapping.) Here is an example:



   We do this by surrounding every station with #'s, applying the map duplicate function, and cleaning up the output with a single-use Mealy machine.

3. Observe that the i-th window contains the entire i-th loop and i-th sweep. In this step, we extract the loop and the sweep from each of the windows. For example, for the first window, this looks as follows:



We do this by duplicating every window and filtering out the spurious nodes. The filtering phase can be handled by an unambiguous Mealy machine combined with Claim 55. (To apply this construction to all windows, we use the `map` combinator from Lemma 56.)

4. Now, we untangle each loop and each sweep. For the loops, we use the construction described earlier in this section, and for the sweeps, we use the induction assumption (as mentioned before, the sweeps are always thinner than the whole run graph). We combine those constructions, using the `if` – `then` – `else` and `map` combinators (see Lemmas 79 and 81).

5. Finally, we obtain the untangled result by concatenating all outputs produced by the previous step. (As mentioned before, this simply means filtering out the #'s.)

4.2.2 Compositions of primes ⊆ Two-way automata

In this section, we show how to translate compositions of two-way primes into single-use two-way transducers (this construction is also described in [BS20, Section E.1]).

Lemma 82. *For every composition of primes $f = p_1 \circ \ldots \circ p_n$, there is an equivalent single-use two-way transducer.*

The proof goes by induction on n. For $n = 0$, the function f is the identity, which makes the lemma trivial. For the induction step, it suffices to show that single-use two-way automata are closed under pre-compositions with prime functions:

$$\binom{\text{2-way single-use}}{\text{transducers}} \circ \binom{\text{2-way single-use}}{\text{primes}} \subseteq \binom{\text{2-way single-use}}{\text{transducers}}$$

This is shown in the following claim:

Claim 56. *For every single-use two-way prime function p, and for every single-use two-way transducer $\mathcal{A}$, there is a two-way transducer $\mathcal{A}'$ that computes $\mathcal{A} \circ p$.*

Proof. We prove the claim by the case analysis of p. For the sake of conciseness, we only show the proof for multiple-use bit propagation, single-use atom propagation, and map-reverse (other cases are either simple or analogous).

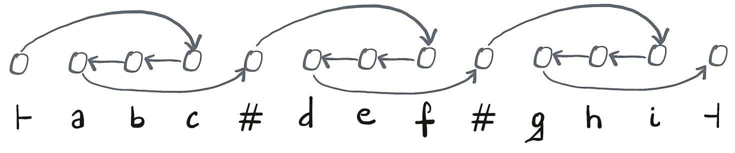
1. **Multiple-use bit propagation** To recognize $\mathcal{A} \circ (f_{\text{prop}} \times \mathtt{id})$, we use $\mathcal{A}'$ which simulates $\mathcal{A}$ and additionally keeps track of the current register value for f_{prop}. Every time it makes a transition, $\mathcal{A}'$ combines the input letter with the current register value and feeds this pair to $\mathcal{A}$. It outputs the same letter and moves in the same direction as $\mathcal{A}$. When $\mathcal{A}$ goes forward, then $\mathcal{A}'$ can easily update the register value. When $\mathcal{A}$ goes backwards, then $\mathcal{A}'$ checks the current register operation (in its input): if it is ϵ, then the register value stays the same; otherwise $\mathcal{A}'$ goes left to the first non-ϵ operation, updates its register value, and finds its way back by going forward to the first non-ϵ operation.

2. **Single-use atom propagation** The transducer $\mathcal{A}'$ for $\mathcal{A} \circ (f_{\text{su-prop}} \times \mathtt{id})$ resembles the transducer for f_{prop} from the previous item, but it does not keep track of the register value. Instead, it computes it on demand: every time $\mathcal{A}'$ enters a position with a $\downarrow$ (i.e. the *read* operation), it goes left to the first non-ϵ operation, saves enough copies (see Lemma 37) of the register value (or if the operation is $\downarrow$, remembers that the register is empty) and finds its way back by going forward to the first $\downarrow$. Then it is ready to simulate the transition of $\mathcal{A}$.

3. **Map-reverse** To recognize $\mathcal{A} \circ f_{\text{map-rev}}$ we use $\mathcal{A}'$ that simulates $\mathcal{A}$, but modifies the order of the input letters – every time $\mathcal{A}$ wants to go right, $\mathcal{A}'$ goes left, and every time $\mathcal{A}$ wants to go right, $\mathcal{A}'$ goes left. Moreover:

   - whenever $\mathcal{A}'$ enters # (or $\vdash$) from the right, it goes to the next # (or $\dashv$);
   - whenever $\mathcal{A}'$ enters # (or $\dashv$) from the left, it goes to the previous # (or $\vdash$);
   - whenever $\mathcal{A}'$ exits # (or $\vdash$) to the right, it goes to the rightmost element of the block to the right;
   - whenever $\mathcal{A}'$ exits # or ($\dashv$) to the left, it goes to the leftmost element of the block to the left.

   Here is an example equipped with a schematic illustration of this order:

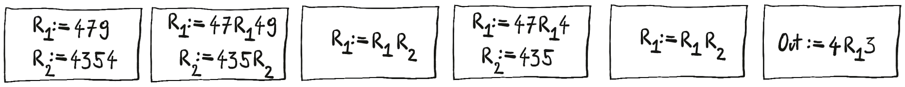


□

This finishes the proof that compositions of single-use two-way primes are included in single-use two-way automata. Together with Section 4.2.1, it follows that the two classes are equal. In particular, this means that we already have the proof that single-use two-way automata are closed under compositions (see Claim 51). We are going to use this fact in the remaining parts of this section.

4.2.3 Streaming string transducers $\subseteq$ Two-way automata

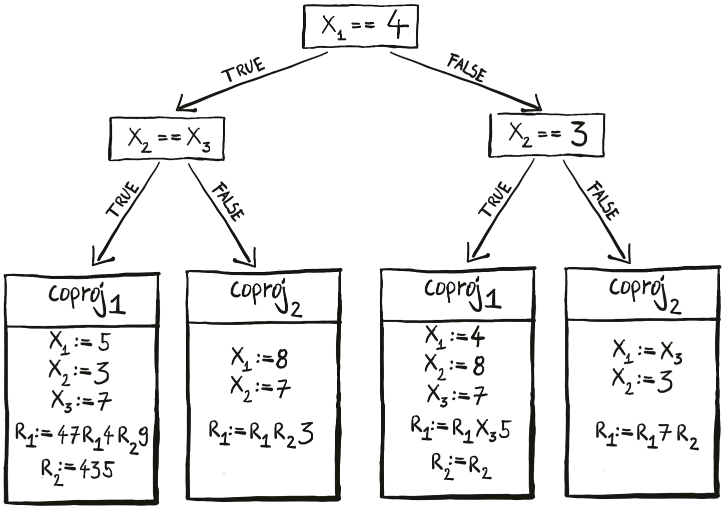
In this section, we show how to translate single-use streaming string transducers into single-use two-way transducers. The translation has two parts: First, we translate the streaming string transducer into a single-use two-way automaton that transforms its input into the sequence of operations on Γ^*-registers performed by the streaming string transducer while processing this input. Here is an example of such a sequence of operations:



In the second step, we construct a single-use two-way transducer that interprets the register operations and produces their output. This finishes the translation because single-use two-way transducers are closed under compositions (see Claim 51, and the last paragraph of Section 4.2.2).

Let us start by showing how to construct the sequence of register operations. Notice that the Γ^*-registers are write-only, i.e. they can be used to construct the output, but the transducer is not allowed to query their contents. It follows, by a reasoning similar to the one in Section 2.2.4, that all single-use functions over polynomial orbit-finite Γ^*-register sets, can be represented by single-use decision trees such as the following one (for $\Gamma = \mathbb{A}$):

$$\mathbb{A}^3 \times (\mathbb{A}^*)^2 \multimap \mathbb{A}^3 \times (\mathbb{A}^*)^2 + \mathbb{A}^2 \times \mathbb{A}^*$$



In this type of tree, the inner nodes look the same as in the usual single-use trees (from Section 2.2.4) – each inner node contains a query about the $\mathbb{A}$-variables. The difference is in the leaves, which can now include constructors for the Γ^*-variables. Each such constructor is a finite word over Γ^*-values, where each Γ^*-value is either a Γ^*-variable (R_i) or a Γ-literal. Each Γ-literal is, in turn, an expression of the following form:

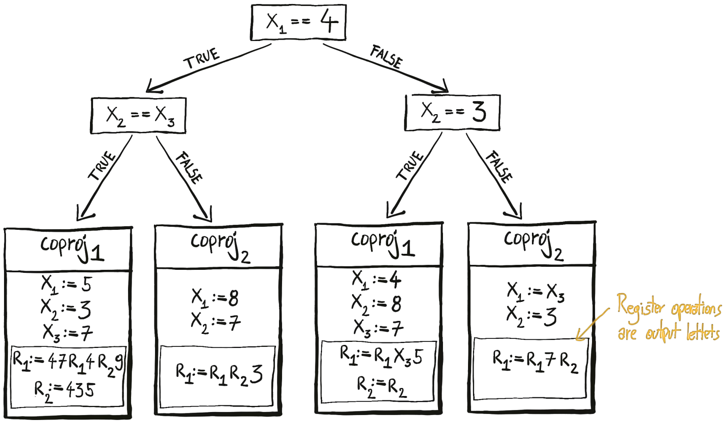
$$\texttt{coproj}_i(v_1, \ldots, v_{k_i}),$$

where each v_j is either an $\mathbb{A}$-variable (x_i) or an atomic constant. The single-use restriction says that every x_i and every R_i can appear at most once on each path from the root to a leaf.

Thanks to this tree representation, we can look on a transition function of a single-use streaming string transducer, as a function $\Sigma \to_{\text{eq}} (\text{Trees}(Q, Q))$. Notice that for every Q there is a limit n of how many Γ^*-values can be stored in the elements of Q. Moreover, since Σ is orbit-finite and the transition function is equivariant, we know that the length of Γ^* constructors (which are words over R_i's and Γ-constructors) is bounded by some $m \in \mathbb{N}$. This means that the set of all possible Γ^*-constructors that appear in the transition functions can be represented as the following polynomial orbit-finite set:

$$\overline{\Gamma^*} = (\underbrace{(1 + \ldots + 1)}_{\text{representing } R_i\text{'s}} + \underbrace{\Gamma}_{\text{single letters}})^{\leq m}$$

This means that we can think about $\text{Trees}(Q, Q)$ as a function that ignores the Γ^*-registers from the input, and instead of performing the Γ^*-operation, it outputs them:



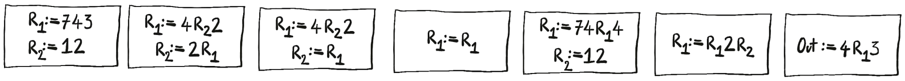
This means that we can translate Trees(Q, Q) into Trees$(Q', Q' \times \overline{\Gamma^*}^{\leq n})$, where Q' is the version of Q where every Γ^* has been replaced by 1. Observe that both Q' and $\overline{\Gamma^*}^{\leq n}$ are polynomial orbit-finite sets. It follows that we can translate the transition function of the streaming string transducer into a function of the following type:

$$\Sigma \to_{\text{eq}} (Q' \multimap Q' \times \overline{\Gamma^*}^{\leq n})$$

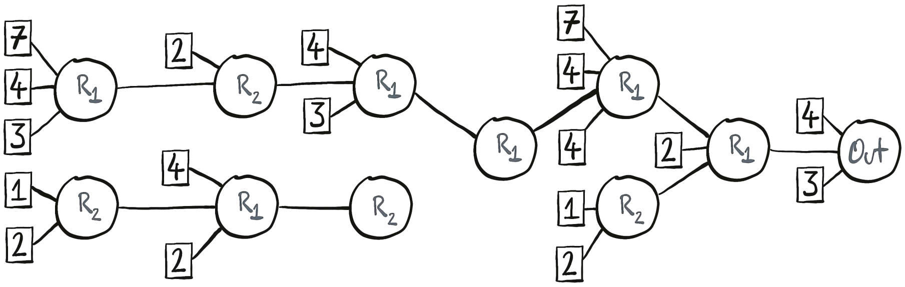
(Observe that this is a single-use function for polynomial orbit-finite sets, as defined in Definition 7). It follows that we can treat each single-use string streaming transducer as a single-use Mealy machine that outputs its register operations (instead of performing them). This almost finishes the first part of the construction. The last (technical) part is to produce the final register operations generated by the output function λ (see Definition 38). Since a Mealy machine has to finish its run as soon as it reaches $\dashv$, it it is too weak for this purpose. Instead, we use a single-use two-way automaton[9] that simulates the Mealy machine and outputs the λ-operations as soon as it reaches the $\dashv$ marker. (We use the same construction as for the transition function, to translate λ into $Q' \multimap \overline{\Gamma^*}^{\leq n}$.)

This leaves us with showing how to use a single-use two-way transducer to interpret the sequence of operations on Γ^*. We present a construction (also described in [BS20, Section E.6]) that is almost identical to the construction for finite alphabets (see [BC18, Lemma 13.4]). First, we notice that thanks to the single-use restrictions for Γ^*-registers, each sequence of register operations can be described as a forest. For example, consider the following sequence of operations:

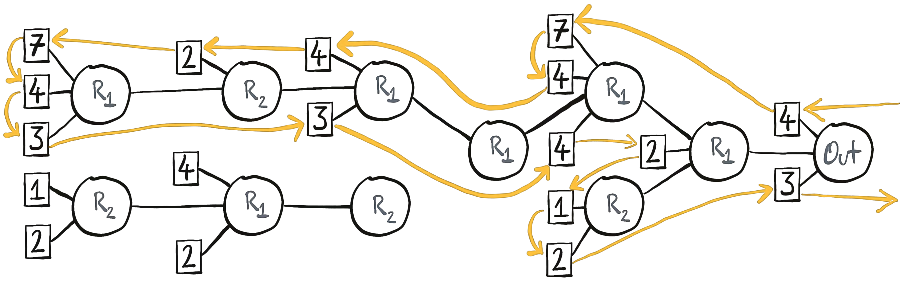
[9] A more adequate model would be a single-use variant of a one-way transducer (see last paragraph of [BS20, Section 2]). However, since we have not defined this variant in this thesis, we need to use the stronger model of a two-way transducer.



It corresponds to the following forest:



In order to execute the register operations, it suffices to perform a DFS traversal starting in the final node (i.e. `Out`), outputting the encountered letters as we visit them:



It is not hard to see that a single-use two-way transducer is capable of performing such a traversal.

4.2.4 Compositions of primes ⊆ Streaming string transducers

In this section, we show how to translate compositions of two-way primes into single-use streaming string transducers:

Lemma 83. *For every composition of primes $p_1 \circ \ldots \circ p_n$, there is an equivalent single-use streaming string transducer.*

The proof of the lemma uses a similar induction as the proof for two-way transducers (Lemma 82), but this time we show that single-use streaming string transducers are closed under post-compositions with single-use two-way primes:

$$\binom{\text{single-use}}{\text{2-way primes}} \circ \binom{\text{single-use string streaming}}{\text{transducers}} \subseteq \binom{\text{single-use string streaming}}{\text{transducers}}$$

(Note the difference with Lemma 82, where we have used pre-compositions).

Claim 57. *For every single-use two-way prime function p, and for every streaming string transducer $\mathcal{A}$, there is a two-way transducer $\mathcal{A}'$, that recognizes $\mathcal{A} \circ p$.*

Proof. We prove the claim, by case analysis of p. This time, we present the proof for all possible p's:

1. **Map reverse** Observe that since $f_{\text{map-rev}}$ is of the type $(\Gamma + \#)^* \to (\Gamma + \#)^*$, then both $\mathcal{A}$ and $\mathcal{A}'$ are of the type $\Sigma^* \to (\Gamma + \#)^*$. We construct $\mathcal{A}'$ in the following way: The set of states of $\mathcal{A}'$ is equal to the set of states of $\mathcal{A}$, where every $(\Gamma + \#)^*$ is replaced by:

$$\underbrace{\Gamma^*}_{\substack{\text{The initial part of}\\ \text{the register value in } \mathcal{A}\\ \text{up to the first } \#,\\ \text{reversed.}}} \times \underbrace{(\Gamma + \#)^*}_{\substack{\text{The middle part of}\\ \text{the register value in } \mathcal{A}\\ \text{from the first to the last } \#,\\ \text{map-reversed.}}} \times \underbrace{\Gamma^*}_{\substack{\text{The final part of}\\ \text{the register value in } \mathcal{A}\\ \text{after the last } \#,\\ \text{reversed.}}} \times \underbrace{\{\text{Yes}, \text{No}\}}_{\substack{\text{Does the register}\\ \text{value in } \mathcal{A} \text{ contain}\\ \text{at least one}\\ \text{separator?}}}$$

If the register in $\mathcal{A}$ contains no separator, then its entire content is stored in the first Γ^*. It is also worth pointing out that the Γ^*-registers are implemented as $(\Gamma + \#)^*$-registers (which happen to have the semantic property of never containing any $\#$'s). Here is an example of a register value in $\mathcal{A}$ and the corresponding value in $\mathcal{A}'$ (for $\Gamma = \mathbb{A}$):

$$\boxed{12\#345\#67\#89} \quad \rightsquigarrow \quad \left(\boxed{21}, \boxed{\#543\#76\#}, \boxed{98}, \text{Yes}\right)$$

The transition function of $\mathcal{A}'$ is a version of the transition function of $\mathcal{A}$, where `concat`, `singleton`, and $\texttt{const}_\epsilon$ are interpreted as follows:

$$(A_1, A_2, A_3, a_4) \cdot (B_1, B_2, B_3, b_4) = \begin{cases} (A_1,\ A_2 B_1 A_3 B_2,\ B_3,\ \text{Yes}), & \text{if } b_4 = \text{Yes} \\ (A_1,\ A_2,\ B_1 A_3,\ a_4), & \text{otherwise} \end{cases}$$

$$\texttt{singleton}'(a \in \Gamma) = (a, \epsilon, \epsilon, \text{No}) \quad \texttt{singleton}'(\#) = (\epsilon, \#, \epsilon, \text{Yes})$$

$$\texttt{const}_\epsilon = (\epsilon, \epsilon, \epsilon, \text{No})$$

(It is not hard to see that all of those functions are single-use.) Similarly, we define λ' (i.e. the output function of $\mathcal{A}'$) to be λ, where `concat`, `singleton` and $\texttt{const}_\epsilon$ are interpreted in the same way as for the transaction function. Since such λ' produces an element of

$$\Gamma^* \times (\Gamma + \#)^* \times \Gamma^* \times \{\text{Yes}, \text{No}\},$$

we need to compose it with the following *exit function*, which collapses this compound register type back to $(\Gamma + \#)^*$:

$$(R_1, R_2, R_3, r_4) \mapsto R_1 R_2 R_3$$

2. **Map duplicate** In this case, we use the same idea as for $f_{\text{map-rev}}$, but we keep two copies of the initial and final blocks. For example:

$$\boxed{12\#345\#67\#89} \quad \leftrightsquigarrow \quad \Big(\boxed{12}, \ \boxed{12}, \boxed{\#345345\#6767\#}, \boxed{89}, \boxed{89}, \text{Yes}\Big)$$

3. **Letter-to-word homomorphism** We show how to construct $\mathcal{A}'$ for $h^* \circ \mathcal{A}$, where h is a function of type $\Gamma \to_{\text{eq}} \Delta^*$. In this case, $\mathcal{A}$ is of type $\Sigma^* \to \Gamma^*$ and $\mathcal{A}'$ is of type $\Sigma^* \to \Delta^*$. We define $\mathcal{A}'$ to be a version of $\mathcal{A}$ where every Γ^*-register is replaced by a Δ^*-register, which keeps the h^*-image of the original Γ-register from $\mathcal{A}$. Simulating register concatenation is trivial – whenever $\mathcal{A}$ concatenates two Γ^*-registers, $\mathcal{A}'$ can simply concatenate the corresponding Δ^*-registers. Simulating `singleton` requires some explanation. The principle is easy – $\mathcal{A}'$ needs to interpret `singleton` as follows:
$$\texttt{singleton}'(a) = h(a)$$
The harder part is implementing h as a single-use function: Observe, first, that since Γ is orbit-finite, and h is equivariant, it follows that there exists a limit l on the length of the output of h. This means that we can translate h into an equivalent $h_1 : \Gamma \to_{\text{eq}} \Delta^{\leq l}$. Now, we can apply Lemma 28 to obtain an equivalent $h_2 : \Gamma \multimap_k \Delta^{\leq l}$, which (by composing it with at most l `concat`'s) can be easily transformed into an equivalent $h_3 : \Gamma \multimap_k \Delta^*$. Finally, by Definition 9, we can transform h_3 into an equivalent $h_4 : \Gamma^k \multimap \Delta^*$. This finishes the construction, as $\mathcal{A}'$ can be easily modified to maintain k copies of every $\mathbb{A}$-register from $\mathcal{A}$.

4. **Single-use atom propagation** For the sake of simplicity, we show how to construct $\mathcal{A}'$ for $f_{\text{su-prop}} \circ \mathcal{A}$. (The construction can be easily modified to construct the actual $(f_{\text{su-prop}} \times \texttt{id}) \circ \mathcal{A}$.) As $f_{\text{su-prop}}$ has the type $(\mathbb{A} + \downarrow + \epsilon)^* \to (\mathbb{A} + \epsilon)^*$, we know that $\mathcal{A}$ and $\mathcal{A}'$ have the following types:
$$\mathcal{A} : \Sigma^* \to (\mathbb{A} + \downarrow + \epsilon)^* \qquad \mathcal{A}' : \Sigma^* \to (\mathbb{A} + \epsilon)^*$$
We construct $\mathcal{A}'$ as a version of $\mathcal{A}$, where every $(\mathbb{A} + \downarrow + \epsilon)^*$-register is replaced by the following set (compare with the proof of Claim 49):
$$\underbrace{\epsilon^*}_{\substack{\text{Maximal}\\ \epsilon\text{-prefix.}}} \times \underbrace{\{\downarrow, \Box, \epsilon\}}_{\substack{\text{The first non-}\epsilon\\ \text{operation, or } \epsilon\\ \text{if there is none.}\\ (\Box \text{ represents elements of } \mathbb{A})}} \times \underbrace{(\mathbb{A} + \epsilon)^*}_{\substack{\text{The output for}\\ \text{the suffix after}\\ \text{the first non-}\epsilon.}} \times \underbrace{(\mathbb{A} + \downarrow + \epsilon)}_{\substack{\text{The final non-}\epsilon\\ \text{operation, or } \epsilon\\ \text{if there is none.}}}$$
(Again, the ϵ^*-registers are actually implemented as $(\epsilon + \mathbb{A})^*$-registers.) Observe that the output of the suffix after the first non-ϵ does not depend on the initial register value for $f_{\text{su-prop}}$.

Here is an example of a register in $\mathcal{A}$ and the corresponding value in $\mathcal{A}'$:
$$\boxed{\epsilon\epsilon\downarrow 12\epsilon\downarrow 3\epsilon\epsilon\epsilon\downarrow 4\epsilon\epsilon} \quad \leftrightsquigarrow \quad \Big(\boxed{\epsilon\epsilon}, \downarrow, \ \boxed{\epsilon\epsilon\epsilon 2\epsilon\epsilon\epsilon\epsilon 3\epsilon\epsilon\epsilon}, 4\Big)$$

This representation allows $\mathcal{A}'$ to simulate register concatenation. For example:

$$(A_1, a_2, A_3, a_4 \in \mathbb{A}) \cdot (B_1, \downarrow, B_3, b_4) = (A_1, a_2, A_3 B_1 a_4 B_3, b_4)$$

Other cases are handled analogously. Operations `singleton` and $\texttt{const}_\epsilon$ are trivial, and the exit function looks as follows:

$$(A_1, a_2, A_3, a_4) \quad \mapsto \quad \begin{cases} A_1 \epsilon A_3 & \text{if } a_2 \in \{\square, \downarrow\} \\ A_1 & \text{if } a_2 = \epsilon \end{cases}$$

5. **Multiple-use bit propagation** Again, for simplicity, we present the construction for $f_{\text{prop}} \circ \mathcal{A}$. The construction is similar to the one for $f_{\text{su-prop}}$. This time both $\mathcal{A}$ and $\mathcal{A}'$ have the type $\Sigma^* \to \{\bigcirc, \bullet, \epsilon\}^*$. We construct $\mathcal{A}'$ as a version of $\mathcal{A}$, where every $\{\bigcirc, \bullet, \epsilon\}^*$-register is replaced by:

$$\underbrace{(\{\bigcirc, \bullet, \epsilon\}^*)^{\{\bigcirc, \bullet, \epsilon\}}}_{\substack{f_{\text{prop}}\text{-output for the} \\ \text{maximal } \epsilon\text{-prefix,} \\ \text{depending on the} \\ \text{initial register value.}}} \times \underbrace{\{\bigcirc, \bullet, \epsilon\}^*}_{\substack{f_{\text{prop}}\text{-output for the suffix} \\ \text{that starts in the first non-}\epsilon. \\ \text{(Note, that this does not depend} \\ \text{on the initial register value of } f_{\text{prop}}.)}} \times \underbrace{\{\bigcirc, \bullet, \epsilon\}}_{\substack{\text{The final non-}\epsilon\text{ value,} \\ \text{or } \epsilon \text{ if there is none.}}}$$

We represent $(\{\bigcirc, \bullet, \epsilon\}^*)^{\{\bigcirc, \bullet, \epsilon\}}$ as $(\{\bigcirc, \bullet, \epsilon\}^*)^3$.

For example:

$$\boxed{\epsilon\epsilon\bullet\bigcirc\epsilon\epsilon\bullet\epsilon\epsilon} \quad \leftrightsquigarrow \quad \left(\begin{pmatrix} \epsilon \mapsto \boxed{\epsilon\epsilon} \\ \bigcirc \mapsto \boxed{\bigcirc\bigcirc} \\ \bullet \mapsto \boxed{\bullet\bullet} \end{pmatrix}, \ \boxed{\bullet\bigcirc\bigcirc\bigcirc\bullet\bullet\bullet}, \bullet \right)$$

The concatenation of registers is interpreted as follows:

$$(A_1, A_2, a_3) \cdot (B_1, B_2, b_3) = \begin{cases} (A_1,\ A_2 \cdot B_1(a_3) \cdot B_2,\ b_3) & \text{if } a_3 \neq \epsilon \\ (x \mapsto A_1(x) \cdot B_1(x),\ B_2,\ b_3) & \text{if } a_3 = \epsilon \end{cases}$$

Observe that this is a single-use function – in particular, each $A_1(x)$ and $B_1(x)$ is used at most once. Functions `singleton` and $\texttt{const}_\epsilon$ are trivial, and the exit function looks as follows:

$$(A_1, A_2, a_3) \quad \mapsto \quad A_1(\epsilon) \cdot A_2$$

6. **Group prefixes** Again, for simplicity, we present the construction for $f_{\text{G-pref}} \circ \mathcal{A}$. This time both $\mathcal{A}$ and $\mathcal{A}'$ are of the type $\Sigma \to G^*$ (where G is a finite group). We construct $\mathcal{A}'$ as a version of $\mathcal{A}$ where every G^* is replaced by:

$$\underbrace{(G^*)^G}_{\substack{\text{The output for the register,} \\ \text{depending on the initial } G\text{-value}}} \times \underbrace{G}_{\substack{\text{The } G\text{-product of} \\ \text{the entire register}}}$$

(Again, we represent $(G^*)^G$ as $(G^*)^{|G|}$).

For example, if we take $G = \mathbb{Z}_3$:

$$\boxed{001211} \quad \rightsquigarrow \quad \left(\begin{matrix} {\scriptstyle 0 \mapsto} \boxed{001012} \\ {\scriptstyle 1 \mapsto} \boxed{112120} \\ {\scriptstyle 2 \mapsto} \boxed{220201} \end{matrix}, \; 2 \right)$$

The register concatenation can be interpreted as follows (note the similarity to the wreath product):

$$(A_1, a_2) \cdot (B_1, b_2) = (g \mapsto A_1(g) \cdot B_1(g \cdot a_2),\ a_2 \cdot b_2)$$

Observe, that since G is a group, then $g \mapsto g \cdot a_2$ is a bijection on G. It follows that each $A_1(x)$ and each $B_1(x)$ is used at exactly once, which makes the function single-use. Operations `singleton` and $\texttt{const}_\epsilon$ are trivial, and the exit function looks as follows:

$$(A_1, a_2) \quad \mapsto \quad A_1(1)$$

7. **End of word marker** This is the simplest case. It can be simulated by $\mathcal{A}'$ that looks exactly of like $\mathcal{A}$ except of the output function, which for $\mathcal{A}'$ is defined in the following way:

$$\lambda'(q) = \lambda(w) \dashv$$

□

It might be worth mentioning that, as observed in the last paragraph of [BS20, Section E3], this proof of Claim 57 also works in the presence of both $\texttt{copy}_{\Gamma^*}$ and $\texttt{copy}_{\mathbb{A}}$. It follows that:

$$\left(\substack{\text{single-use}\\ \text{two-way primes}}\right) \circ \left(\substack{\text{multiple-use string streaming}\\ \text{transducers}}\right) \subseteq \left(\substack{\text{multiple-use string streaming}\\ \text{transducers}}\right)$$

The same proof also works in the finite case, which might be of independent interest. In particular, it follows that:

$$(\text{two-way transducers}) \circ \left(\substack{\text{copyful string streaming}\\ \text{transducers}}\right) \subseteq \left(\substack{\text{copyful string streaming}\\ \text{transducers}}\right)$$

4.2.5 Regular list transductions ⊆ Two-way transducers

In this section, we show how to translate regular list transductions with atoms into single-use two-way transducers (the construction, which is also presented in [BS20, Section E.5], is an adaptation of the left-to-right implication from [BDK18, Theorem 4.3]).

Remember that regular list functions with atoms work over the class of polynomial sets with atoms – i.e. the smallest class that contains 1 and $\mathbb{A}$ and is closed under ×, + and X^*. We start the construction by observing that every element of every polynomial set with atoms can be encoded as a word over the following alphabet:

$$\Sigma_{[\mathbb{A}]} = \{ \underbrace{\circ}_{\substack{\text{element}\\\text{of } 1}}, \underbrace{[\ ,\ ]}_{\substack{\text{used for}\\ X^*}}, \underbrace{(\ ,\ )}_{\substack{\text{used for}\\ X\times Y}}, \underbrace{\texttt{coproj}_1, \texttt{coproj}_2}_{\substack{\text{used for}\\ X+Y}}, \underbrace{\ ,\ }_{\substack{\text{separator for}\\ X\times Y \text{ and } X^*}} \} + \underbrace{\mathbb{A}}_{\substack{\text{elements}\\\text{of } \mathbb{A}}}$$

For example, here is an encoding of an element from $(\mathbb{A}^2 + 1)^*$:

$$[\texttt{coproj}_1\ (5, 8),\ \texttt{coproj}_2\ \circ,\ \texttt{coproj}_1\ (1, 2),\ \texttt{coproj}_1\ (7, 8),\ \texttt{coproj}_2\ \circ]$$

Thanks to this encoding, we can translate every regular list function with atoms into a two-way transducer:

Lemma 84. *For every regular list function with atoms $f : X \to Y$, there is a two-way transducer:*

$$\mathcal{F} : \Sigma^*_{[\mathbb{A}]} \to \Sigma^*_{[\mathbb{A}]},$$

such that $\mathcal{F}(w_x)$ outputs the encoding of $f(x)$ (where w_x denotes the $\Sigma_{[\mathbb{A}]}$-encoding of x).

Proof. The proof is a standard induction on the construction of f as a regular list function with atoms. The most interesting case is function composition but, as explained in the last paragraph of Section 4.2.2, we already know that single-use automata are closed under compositions. □

We finish the construction, by observing, that for every polynomial *orbit-finite* Γ, the following two functions can be implemented as single-use two-way transducers:

$$\mathcal{T}_\Gamma : \Gamma^* \to \Sigma^*_{[\mathbb{A}]} \quad \mathcal{T}_\Gamma^{-1} : \Sigma^*_{[\mathbb{A}]} \to \Gamma^*,$$

where $\mathcal{T}_\Gamma$ is a function that translates the input word (which is a polynomial set with atoms) into its $\Sigma_{[\mathbb{A}]}$-encoding, and $\mathcal{T}_\Gamma^{-1}$ is a one-way inverse of $\mathcal{T}_\Gamma$.

4.2.6 Compositions of primes ⊆ Regular list transductions

Finally, let us show how to translate compositions of single-use two-way primes into regular list transductions with atoms (the construction is also presented in [BS20, Section E.2]). Since regular list functions are (by definition) closed under compositions, it suffices to show how to translate every prime function:

1. **Letter-to-word homomorphism** In this step, we construct h^*, for any $h : \Sigma \to_{\text{eq}} \Gamma^*$ where Σ and Γ are polynomial orbit-finite. Similar as in Section 4.2.4, we observe that there is a limit k on the length of the outputs of h, which means that we can translate h into an equivalent $h' : \Sigma \to_{\text{eq}} \Gamma^{\leq k}$. As $\Gamma^{\leq k}$ is polynomial orbit-finite, it follows by Lemma 23, that h' is a regular list function with atoms. This means that we can construct h^* in the following way:

$$\Sigma^* \xrightarrow{\texttt{map}\ f'} (\Gamma^{\leq k})^* \xrightarrow{\texttt{map toList}_{\leq k}} (\Gamma^*)^* \xrightarrow{\texttt{concat}} \Gamma^*,$$

where $\texttt{toList}_{\le\texttt{k}}$ is the following tuple-to-list transformation:

$$\texttt{toList}_{\le\texttt{k}} : X^{\le k} \to X^*.$$

We finish the construction by showing that $\texttt{toList}_{\le\texttt{k}}$ is a regular list function. We start with $\texttt{toList}_1$:

$$\texttt{toList}_1 : X \xrightarrow{\texttt{rightI}} X \times I \xrightarrow{\texttt{const}_\epsilon} X \times X^* \xrightarrow{\texttt{cons}} X^*$$

Now, let us continue with $\texttt{toList}_{\le 2}$:

$$\texttt{toList}_2 : X^2 \xrightarrow{\texttt{id}\times\texttt{toList}_1} X \times X^* \xrightarrow{\texttt{cons}} X^*$$

By continuing in this manner, we can construct $\texttt{toList}_\texttt{i}$ for every $i \le k$. Finally, we combine all those functions into $\texttt{toList}_{\le\texttt{k}}$ using the $[f_1, \dots, f_n]$ combinator from Lemma 16.

2. **Single-use propagation** For the sake of clarity, we show how to construct $f_{\text{su-prop}}$ – the construction for $f_{\text{su-prop}} \times \texttt{id}$ is analogous. Consider the following example input:

$$[1,\ \epsilon,\ \epsilon,\ \downarrow,\ \epsilon,\ \epsilon,\ 3,\ \epsilon,\ \downarrow,\ 3,\ 2,\ \epsilon,\ \downarrow,\ \downarrow,\ 3,\ \epsilon]$$

We start by applying h^* for the following h (we have already shown in the previous item that h^* is a regular list function):

$$h(\epsilon) = \epsilon \quad h(\downarrow) =\downarrow \# \quad h(a \in \mathbb{A}) = \# a$$

This transforms the input word into:

$$[\#,\ 1,\ \epsilon,\ \epsilon,\ \downarrow,\ \#,\ \epsilon,\ \epsilon,\ \#,\ 3,\ \epsilon,\ \downarrow,\ \#,\ \#,\ 3,\ \#,\ 2,\ \epsilon,\ \downarrow,\ \#,\ \downarrow,\ \#,\ \#,\ 3,\ \epsilon]$$

Next, we apply the $\texttt{block}$ function:

$$[\,[\,], [1,\ \epsilon,\ \epsilon,\ \downarrow],\ [\epsilon,\ \epsilon],\ [3,\ \epsilon,\ \downarrow],\ [\,],\ [3],\ [2,\ \epsilon,\ \downarrow],\ [\downarrow],\ [\,],\ [3,\ \epsilon]\,]$$

Observe that every $\downarrow$ is the last element of some block, and that the output produced by each $\downarrow$ is equal to the first letter of the $\downarrow$'s block (as long as the block contains at most two letters – the output of a singleton block $[\downarrow]$ is equal to $[\epsilon]$). This means that we can produce the output for each block using the following function:

$$f_{\text{block-out}}(b_1\, b'\, b_n) = \begin{cases} \epsilon\, b'\, b_1 & \text{if } b_n =\downarrow \\ \epsilon\, b'\, \epsilon & \text{otherwise} \end{cases},$$

where b_1 denotes the block's first letter, b_n denotes the last letter, and b' denotes inner letters of the block. Observe, that due to the way the blocks are constructed, we know that:

$$b_1 \in \mathbb{A} + \epsilon, \quad b_n \in \{\epsilon, \downarrow\}, \quad \text{and } b' \in \epsilon^*.$$

This definition of $f_{\text{block-out}}$ only works blocks with at least two letters, but the other cases are very simple: the empty block produces empty output and all one-letter blocks produce $[\epsilon]$. Observe that we can implement the $f_{\text{block-out}}$ function as a regular list function with atoms: First, we use `destr` and `reverse` (together with some structural transformations, such as `distr`) to split b into (b_1, b', b_n). Then, we can use the if-then-else combinator from Example 25 together with `cons`, `reverse` (and some structural functions) to construct the output. This finishes the construction, as we can now use `map` to apply $f_{\text{block-out}}$ to every block, and combine the results using `concat`.

3. **Multiple-use propagation** Again, we only show how to construct f_{prop} – the construction for $f_{\text{prop}} \times \texttt{id}$ is analogous. The idea is very similar to the one for single-use propagation (from the previous item). For this reason, we only show how to implement the key component of the construction, which is the following function:

$$\texttt{replace} : \{\bigcirc, \bullet\} \times \epsilon^* \to \{\bigcirc, \bullet\}^*,$$

which replaces every ϵ in the input word with the letter from the first coordinate. For example:

$$(\bullet, [\epsilon, \epsilon, \epsilon, \epsilon]) \mapsto [\bullet, \bullet, \bullet, \bullet]$$

Since $\{\bigcirc, \bullet\}$ is encoded as $1 + 1$, we can use the if-then-else combinator (from Example 25), and implement `replace` as follows[10]:

$$(\texttt{map}\,\texttt{const}_{\bigcirc}) \; ? \; (\texttt{map}\,\texttt{const}_{\bullet})$$

4. **Map reverse** This function can be implemented in the following way:

$$f_{\text{map-rev}}(\Sigma + \#)^* \xrightarrow{\texttt{block}} (\Sigma^*)^* \xrightarrow{\texttt{map reverse}} (\Sigma^*)^* \xrightarrow{\texttt{concat}} \Sigma^*$$

The only problem with this construction is that it erases the #-separators. However, it is not hard to see that we can reintroduce them before applying the `concat` function.

5. **Map duplicate** We use a similar idea as for $f_{\text{map-reverse}}$. The only difference is that we have to implement the function $\texttt{duplicate} : X^* \to X^*$. We start by showing how to concatenate two lists:

$$\texttt{append} : X^* \times X^* \xrightarrow{\texttt{id}\times\texttt{toList}_1} X^* \times (X^*)^* \xrightarrow{\texttt{append}} (X^*)^* \xrightarrow{\texttt{concat}} X^*$$

Now we can implement `duplicate` in the following way:

$$X^* \xrightarrow{\texttt{copy}_{X^*}} X^* \times X^* \xrightarrow{\texttt{append}} X^*$$

[10] It is worth mentioning that this idea does not work for $\texttt{replace}_{\mathbb{A}} : \mathbb{A} \times \epsilon^* \to \mathbb{A}^*$. In fact, it is not hard to see that $\texttt{replace}_{\mathbb{A}}$ cannot be implemented as a single-use two-way automaton, which by Section 4.2.5, means that it is not a regular list function with atoms.

Further work

Throughout the thesis, I have pointed out several open problems. This section gathers them all in one place. For more information, about each of those problems follow the references to the sections where they were originally discussed, or contact me directly.

1. **Semantic definition of single-use functions.** The definition of single-use functions is quite syntactic in its nature, which limits their domain to polynomial orbit-finite sets. This open question asks for a semantic definition of single-use functions. (Compare with syntactic and semantic definitions of equivariance in Section 1.1.) A possible approach might be to consider a version of sets with atoms equipped with an action of all functions $\mathbb{A} \to \mathbb{A}$ or even all relations $\mathbb{A} \times \mathbb{A}$ (rather than only atom bijections). See the introduction of Section 2.2 for context.

2. **Nondeterministic single-use automata.** A straightforward way of introducing nondeterminism to single-use automata results in a model that is too strong: It is not hard to construct a *nondeterministic single-use automaton* that recognizes the language "The first letter appears again" (which cannot be recognized by a deterministic single-use automaton). This does not fit well in the picture of definitional robustness presented in this thesis. This open question asks if there is a notion of nondeterminism compatible with the single-use restriction. This seems to be connected with developing a good notion of *single-use relations.* For context, see Section 2.4.

3. **Unambiguous single-use automata.** It is worth noting that both examples from Section 2.4, which demonstrate that nondeterministic single-use automata are stronger than deterministic ones, use automata that are ambiguous (which means that some accepted words will always have more than one accepting run). It follows that those examples cannot be used to show that unambiguous automata are stronger than deterministic ones. In fact, the question of whether unambiguous nondeterministic automata are equivalent to deterministic single-use automata remains open. If the two models turned out to be equivalent, it would open a path to a machine-based definition of single-use rational transductions. For context, see Footnote 17 on Page 152.

4. **Local semigroup transductions over arbitrary orbit-finite sets.** Note that local semigroup transductions (Definition 19) are defined for all orbit-finite alphabets, but their Krohn-Rhodes theorem only works for polynomial orbit-finite alphabets. A counterexample is the single-use propagation of $\binom{\mathbb{A}}{2}$, which can be constructed as a local semigroup transduction but not as a composition of single-use primes. One way to address this issue would be to extend the set of single-use primes with the generalized single-use propagation for every orbit-finite X (i.e. an extended version of the function from Claim 26). However, the current proof of the theorem only works for polynomial orbit-finite alphabets. This leaves the question of whether compositions of these generalized primes are equivalent to local rational semigroup transductions over orbit-finite alphabets open. An analogous open question can be asked about local rational semigroup transductions. For context, see the footnotes in Theorems 9 and 14.

5. **Local semigroup transductions as two Mealy machines.** Elgot-Mezei Theorem ([EM63, Theorem 7.8]) shows that every rational function can be expressed as a composition of one left-left-to-right and one right-to-left Mealy machine. In contrast, in the proof of Theorem 14, we have used multiple left-to-right and multiple right-to-left single-use Mealy machines. It remains an open problem whether the Elgot-Mezei Theorem can be generalized for single-use Mealy machines. For context see Footnote 22 on Page 156.

6. **Single-use restriction for total-order atoms.** In this thesis, we have considered a set of atoms, whose elements can only be compared with respect to equality. However, there are also other types of atoms that are studied in the literature (see [Boj19, Chapter 3]). One example is the total-order atoms, i.e. the set $\mathbb{Q}$ equipped with the relation $\leq$. Interestingly, Claim 23 fails for some of those other atoms (including the total-order atoms) which breaks most of the proofs presented in this thesis. For this reason, developing a single-use theory for other kinds of atoms remains an open problem. We are currently working on it together with Nathan Lhote. For context, see Footnote 11 on Page 114.

7. **Polyregular functions over infinite alphabets.** *Polyregular functions* is a class over finite alphabets that extends *regular functions* while keeping many of their desirable properties (see [Boj22]). Our final open question concerns finding a well-behaved class of *polyregular functions over infinite alphabets*. It is harder than one might expect, as polyregular functions seem to be very good at bypassing the single-use restriction. For context, see introduction to Chapter 3.

Bibliography

[AČ11] Rajeev Alur and Pavol Černý. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 599–610, 2011.

[AČe10] Rajeev Alur and Pavol Černý. Expressiveness of streaming string transducers. In *Foundations of Software Technology and Theoretical Computer Science*, 2010.

[AFR14] Rajeev Alur, Adam Freilich, and Mukund Raghothaman. Regular combinators for string transformations. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, New York, NY, USA, 2014. Association for Computing Machinery.

[Asp98] Andrea Asperti. Light affine logic. In *Proceedings. Thirteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No. 98CB36226)*, pages 300–308. IEEE, 1998.

[BC18] Mikołaj Bojańczyk and Wojciech Czerwiński. Automata toolbox. *URL: https://www. mimuw. edu. pl/˜ bojan/upload/reduced-may-25. pdf*, 2018.

[BDK18] Mikołaj Bojańczyk, Laure Daviaud, and Shankara Narayanan Krishna. Regular and first-order list functions. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 125–134, 2018.

[BDM$^+$11] Mikołaj Bojańczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data words. *ACM Trans. Comput. Logic*, 12(4), jul 2011.

[BKM21] Mikołaj Bojańczyk, Bartek Klin, and Joshua Moerman. Orbit-finite-dimensional vector spaces and weighted register automata. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13. IEEE, 2021.

[BNê23] Mikolaj Bojańczyk and Lê Thành Dũng Nguyễn. Algebraic Recognition of Regular Functions. working paper or preprint, February 2023.

[Boj13] Mikołaj Bojańczyk. Nominal monoids. *Theory of Computing Systems*, 53(2):194–222, 2013.

[Boj18] Mikołaj Bojańczyk. Polyregular functions. *arXiv preprint arXiv:1810.08760*, 2018.

[Boj19] Mikołaj Bojańczyk. Slightly infinite sets, 2019.

[Boj20] Mikołaj Bojańczyk. Languages recognised by finite semigroups, and their generalisations to objects such as trees and graphs, with an emphasis on definability in monadic second-order logic. *arXiv e-prints*, pages arXiv–2008, 2020.

[Boj22] Mikolaj Bojanczyk. Transducers of polynomial growth. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 1–27, 2022.

[Boj23] Mikołaj Bojańczyk. Folding interpretations. *arXiv preprint arXiv:2301.05101*, 2023.

[BS20] Mikołaj Bojańczyk and Rafał Stefański. Single use register automata for data words, 2020.

[CE12] Bruno Courcelle and Joost Engelfriet. *Graph structure and monadic second-order logic: a language-theoretic approach*, volume 138. Cambridge University Press, 2012.

[CHL$^+$19] Taolue Chen, Matthew Hague, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. Decision procedures for path feasibility of string-manipulating programs with complex operations. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019.

[CLP15] Thomas Colcombet, Clemens Ley, and Gabriele Puppis. Logics with rigidly guarded data tests. *Log. Methods Comput. Sci.*, 11(3), 2015.

[Col07] Thomas Colcombet. A combinatorial theorem for trees. In *International Colloquium on Automata, Languages, and Programming*, pages 901–912. Springer, 2007.

[Cou94] Bruno Courcelle. Monadic second-order definable graph transductions: a survey. *Theoretical Computer Science*, 126(1):53–75, 1994.

[DFJL17] Luc Dartois, Paulin Fournier, Ismaël Jecker, and Nathan Lhote. On reversible transducers. *arXiv preprint arXiv:1702.07157*, 2017.

[DH19] Hossep Dolatian and Jeffrey Heinz. Learning reduplication with 2-way finite-state transducers. In *International Conference on Grammatical Inference*, pages 67–80. PMLR, 2019.

[DL09] Stéphane Demri and Ranko Lazić. Ltl with the freeze quantifier and register automata. *ACM Transactions on Computational Logic (TOCL)*, 10(3):1–30, 2009.

[Dub41] Paul Dubreil. Contribution à la théorie des demi-groupes. i. *Mem. Acad. Sci. Paris*, 63:1–52, 1941.

[EH01] Joost Engelfriet and Hendrik Jan Hoogeboom. Mso definable string transductions and two-way finite-state transducers. *ACM Transactions on Computational Logic (TOCL)*, 2(2):216–254, 2001.

[Eil74] Samuel Eilenberg. *Automata, languages, and machines*, volume B. Academic press, 1974.

[EM63] Calvin C Elgot and Jorge E Mezei. Two-sided finite-state transductions. In *Proceedings of the Fourth Annual Symposium on Switching Circuit Theory and Logical Design (swct 1963)*, pages 17–22. IEEE, 1963.

[FR17] Emmanuel Filiot and Pierre-Alain Reynier. Copyful streaming string transducers. In *Reachability Problems: 11th International Workshop, RP 2017, London, UK, September 7-9, 2017, Proceedings 11*, pages 75–86. Springer, 2017.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical computer science*, 50(1):1–101, 1987.

[GP02] Murdoch J Gabbay and Andrew M Pitts. A new approach to abstract syntax with variable binding. *Formal aspects of computing*, 13(3):341–363, 2002.

[Gre51] James A Green. On the structure of semigroups. *Annals of Mathematics*, pages 163–172, 1951.

[Har72] Juris Hartmanis. On non-determinancy in simple computing devices. *Acta Informatica*, 1(4):336–344, 1972.

[Iba71] Oscar H Ibarra. Characterizations of some tape and time complexity classes of turing machines in terms of multihead and auxiliary stack automata. *Journal of Computer and System Sciences*, 5(2):88–117, 1971.

[JL11] Marcin Jurdziński and Ranko Lazić. Alternating automata on data trees and xpath satisfiability. *ACM Transactions on Computational Logic (TOCL)*, 12(3):1–21, 2011.

[KF94] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.

[KR65] Kenneth Krohn and John Rhodes. Algebraic theory of machines. i. prime decomposition theorem for finite semigroups and machines. *Transactions of the American Mathematical Society*, 116:450–464, 1965.

[LKB14] Sławomir Lasota, Bartek Klin, and Mikołaj Bojańczyk. Automata theory in nominal sets. *Logical Methods in Computer Science*, 10, 2014.

[Mea55] George H Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, 1955.

[MSS^{+}17] Joshua Moerman, Matteo Sammartino, Alexandra Silva, Bartek Klin, and Michał Szynwelski. Learning nominal automata. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 613–625, 2017.

[NDRP20] Max Nelson, Hossep Dolatian, Jonathan Rawski, and Brandon Prickett. Probing rnn encoder-decoder generalization of subregular functions using reduplication. In *Proceedings of the Society for Computation in Linguistics 2020*, pages 167–178, 2020.

[Nê21] Lê Thành Dũng Nguyễn. *Automates implicites en logique linéaire et théorie catégorique des transducteurs*. Theses, Université Paris-Nord - Paris XIII, December 2021.

[NSV04] Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Transactions on Computational Logic (TOCL)*, 5(3):403–435, 2004.

[Pin10] Jean-Éric Pin. Mathematical foundations of automata theory. *Lecture notes LIAFA, Université Paris*, 7, 2010.

[Pit13] Andrew M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2013.

[Sch55] Marcel-Paul Schützenberger. Une théorie algébrique du codage. *Séminaire Dubreil. Algebre et théorie des nombres*, 9:1–24, 1955.

[She59] John C Shepherdson. The reduction of two-way automata to one-way automata. *IBM Journal of Research and Development*, 3(2):198–200, 1959.

[Sim90] Imre Simon. Factorization forests of finite height. *Theoretical Computer Science*, 72(1):65–94, 1990.

[Ste18] Rafał Stefański. An automaton model for orbit-finite monoids. Master's thesis, University of Warsaw - Faculty of Mathematics, Informatics and Mechanics, 2018.

[UMB23] Henning Urbat, Stefan Milius, and Fabian Birkmann. Nominal topology for data languages. *arXiv preprint arXiv:2304.13337*, 2023.

[Wad89] Philip Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359, 1989.

[Wad90] Philip Wadler. Linear types can change the world! In *Programming concepts and methods*, volume 3, page 5. Citeseer, 1990.