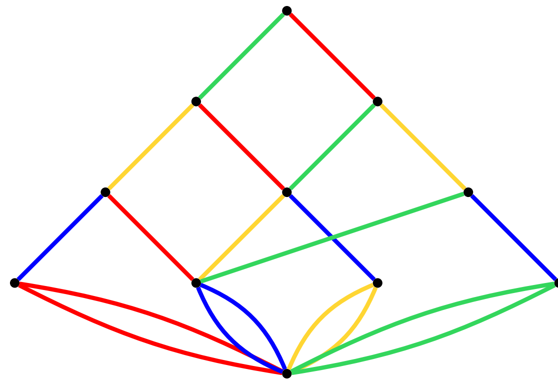
Computing in relatively free bands

An acyclic transducer based approach

Reinis Cirpons



School of Mathematics and Statistics
University of St Andrews

Declaration

I certify that this project report has been written by me, is a record of work carried out by me, and is essentially different from work undertaken for any other purpose or assessment.

Reinis Cirpons

To no one in particular, I just like dedication pages.

Computing in free bands

An acyclic transducer based approach

Reinis Cirpons

Abstract

We investigate computational aspects of free bands. These are the free objects in the variety defined by the identical relation $[x^2 = x]$.

Inspired by the solution to the word problem for the free band presented by Radoszewski and Rytter in [12], we derive a framework for uniquely representing elements of the free band using a certain restricted class of transducers. We apply automata minimization techniques and show that the resulting transducers have size proportional to the smallest possible word representative of the element in the free band.

We then derive a new algorithm for solving the word problem in the free band matching the $\mathcal{O}(|A| \cdot |w|)$ time complexity of [12], where $|A|$ is the size of the alphabet and $|w|$ is the total length of the words to be compared. This is somewhat surprising, given that the obvious method for doing so is exponential in $|A|$. Our algorithm has practical implications for computing in bands as well as being of theoretical interest.

Finally we show how the algorithm of Radoszewski and Rytter fits into our framework.

Our framework is original and shows potential for use in computational problems relating to the free band and bands in general.

Contents

Title page i

Abstract iv

Contents 1

Introduction 2

1 Preliminaries 3
- 1.1 Semigroups 3
  - 1.1.1 Elementary semigroup theory 4
  - 1.1.2 Varieties of semigroups 8
- 1.2 Computation 11
  - 1.2.1 A quick overview of the theory of computation 11
  - 1.2.2 The word problem in semigroups 13
  - 1.2.3 Computational complexity 14
  - 1.2.4 Normal forms and complete invariants 16
- 1.3 Automata and transducers 18
  - 1.3.1 Automata basics 18
  - 1.3.2 Acyclic automata 21
  - 1.3.3 Transducers 23

2 Free band elements via minimal transducers 25
- 2.1 The word problem in the free band 25
- 2.2 From words to trees to transducers and back 28
- 2.3 Minimal transducers are small 31
- 2.4 How to build a minimal transducer 35
  - 2.4.1 The vertical method 36
  - 2.4.2 The horizontal method of [12] 36

Conclusions and further work 38

Bibliography 39

Introduction

Efficient datastructures for representing and manipulating objects are fundamental to the design of algorithms.

When computing with semigroups words are often used to give a combinatorial description of elements. However, the word representation is often computationally lacking in many respects, especially in terms of equality checking.

In this project we set out to explore the computational aspects of a particular semigroup known as the free band. Inspired by various solutions to the word problem in the free band, we propose a method of representing elements of the free band with minimal acyclic transducers and use it to derive a novel algorithm for the word problem in the free band, whose time complexity matches that of the current best known algorithm.

The project is written to be accessible to a final year undergraduate student with an interest in algebra and computation, however we believe that there are interesting insights even for the expert reader, and we also provide an extensive treatment of preliminary material and external resources for anyone that is not well acquainted with the background material. Our intended audience are graduate students specializing in abstract algebra and computation.

Chapter 1 of the project is dedicated to the preliminaries. While a lot of it is standard material, and those familiar with semigroup and computational theory may choose to skip parts of it, we strongly encourage readers to read through Sections 1.1.2, 1.2.4 and 1.3 fully, as they contain non-standard material or provide important context for the rest of the paper.

Chapter 2 is dedicated to showcasing our acyclic transducer representation. At first we review the basic theory of the word problem in the free band, including an important theorem by Green and Rees. We then show how acyclic transducers arise at as representatives of elements in the free band. Afterwards we investigate the size properties of minimal transducers. Afterwards we give a linear method for constructing minimal automata representing a given free band element, and then show how it can be used to solve the word problem. This leads to us deriving an algorithm for solving the word problem in the free band with time complexity $\mathcal{O}(|A| \cdot |w|)$, where $|A|$ is the size of the alphabet and $|w|$ is the total length of the words to be compared. This matches of the best know algorithm for doing so by Radoszewski and Rytter. Finally, we show how the algorithm of Radoszewski and Rytter fits in our framework.

Chapter 1

Preliminaries

The purpose of this chapter is to bring us up to speed on the material that is necessary for understanding the word problem in the free band — the central object of study of this paper, as well as our proposed method for solving it. Furthermore, the ideas laid out should show our motivation for the second half of the project.

Our main inspiration for this chapter comes from the wonderful courses that the author undertook while at the University of St Andrews. Namely, Section 1.1 on semigroups is inspired by the course "MT5863 Semigroups", Section 1.2 on computation is inspired by "MT3852 Automata and Complexity theory", with Subsection 1.2.4 on normal forms and complete invariants discussing a subject that the author was first introduced to in "MT5864 Topics in Groups". Even though these courses serve as our main inspiration, the layout of the content as well as the emphasis is our own, with elements mixed in from other primary sources.

Section 1.3 consists mainly of a literature review in the domain of automata minimization and some original definitions of a class of transducers relevant to our goals.

Since this is meant only as a brief overview, theorems are mainly used to either motivate or make concrete the abstract ideas that we are presenting. Therefore we omit proofs of certain theorems in this chapter, usually because the proof is either too long or not all too relevant to the methods presented in the second chapter. Where possible we try and give a reference to the theorem or definition, so that the curious reader may explore the solution for themselves.

1.1 Semigroups

The following is a very brief overview of the theory of semigroups, with particular focus on the areas most relevant. For a more comprehensive introduction to the theory of semigroups see, for example, [4] or [8].

Before diving into semigroups, we will benefit from a refresher on binary relations.

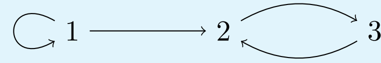
Definition 1.1.1 (Binary relation, special binary relations)**.** A binary relation between sets X and Y is a subset $R \subseteq X \times Y$. If $(x, y) \in R$ then we write $x \sim_R y$. We call X the *domain* and Y the *codomain* of a relation. One can also compose binary relations with compatible domains and codomains — if $R \subset X \times Y$ and $S \subset Y \times Z$ are relations, then their composition is a relation denoted $RS \subset X \times Z$ and it is given by $RS = \{(x, z) : \exists\, y \in Y, \text{ s.t. } (x, y) \in R, (y, z) \in S\}$.

Example 1.1.2 (Special types of relations).

(a) A *graph* is a relation $R \subseteq X \times X$. We call elements $x \in X$ vertices and the pair $(x, y) \in R$ is called an edge from x to y. We can visualize graphs as networks of interconnected nodes by

drawing a node for each vertex and connecting nodes with arrows according to the edges.

For example, the graph $G = \{(1,1),(1,2),(2,3),(3,2)\} \subseteq \{1,2,3\}^2$ can be visualized as



(b) An *equivalence relation* is a relation $R \subseteq X \times X$ such that for all $x, y, z \in X$ the following hold:

- Reflexivity: $x \sim_R x$,
- Symmetry: if $x \sim_R y$ then $y \sim_R x$,
- Transitivity: if $x \sim_R y$ and $y \sim_R z$, then $x \sim_R z$.

Equivalence relations can be thought of as relating elements of X that share a common property. For a given element $x \in X$, its equivalence class is defined to be the set of all elements related to x, formally we write

$$x/R = \{y \in X : x \sim_R y\}$$

We write $X/R = \{x/R : x \in X\}$ for the set of all equivalence classes. One can show that the set of equivalence classes partition X.

(c) A function $f : X \to Y$ defines the relation $R_f \subseteq X \times Y$ such that $x \sim_{R_f} y \Leftrightarrow f(x) = y$. Going the other direction, one can show that any relation that associates to each $x \in X$ a unique $y \in Y$ defines a function. Furthermore, for functions $f : X \to Y$ and $g : Y \to Z$, the function composition gf has associated relation $R_{gf} = R_f R_g$, so, in some sense, composition of function relations is exactly the composition of functions.

(d) A *partial function* $f : X \to Y$ is a function that is allowed to be undefined on some inputs, in such cases we write $f(x) = \bot$. It can be thought of as a relation $R_f \subseteq X \times Y$ that associates to each $x \in X$ at most one $y \in Y$. Partial functions inherit composition from binary relations, that is, if $f : X \to Y, g : Y \to Z$ are partial functions, then $gf : X \to Z$ is a partial function given by

$$gf(x) = \begin{cases} \bot & \text{if } f(x) = \bot \\ g(f(x)) & \text{otherwise} \end{cases}$$

1.1.1 Elementary semigroup theory

This section is essentially a compressed version of the exposition found in [4, Chapters 1, 2]. We will first establish semigroups as certain algebraic objects and then work our way to making them more combinatorial by describing their elements using words and relations.

Definition 1.1.3 (Semigroup, monoid)**.** A *semigroup* is a pair $(S, \cdot)$ consisting of a set S and a binary operation $\cdot : S \times S \to S$ such that

$$\forall\, x, y, z \in S,\; x \cdot (y \cdot z) = (x \cdot y) \cdot z$$

This property of $\cdot$ is also known as *associativity*.

If S in addition has an element $e \in S$ such that $\forall\, x \in S, x \cdot e = x = e \cdot x$, then we call e an identity element and $(S, \cdot)$ a *monoid*.

When the binary operation is unambiguous and clear from context, we sometimes simply refer to the set S as the semigroup.

Example 1.1.4 (Basic examples of semigroups).

(a) Since addition is associative, then the natural numbers $\mathbb{N} = \{1, 2, \ldots\}$ together with addition form a semigroup.

(b) Similarly, the integers, rational, reals or imaginary number together with addition also form a monoid, with 0 as the identity element.

(c) The complex numbers together with multiplication form a monoid.

(d) For a given n, the set of all real $n \times n$ matrices together with matrix multiplication is a semigroup.

(e) Semigroups can sometimes be given via so called *multiplication tables*. In the following table, the entry in the row labeled by x and column labeled by y corresponds to the product $x \cdot y$:

$\cdot$	a	b	c
a	a	b	a
b	a	b	b
c	a	b	c

One can check that the operation is indeed associative and therefore $S = \{a, b, c\}$ together with $\cdot$ form a semigroup. Furthermore, c is an identity and so this semigroup is actually a monoid. It is known as the *flip-flop monoid*.

(f) For any set X, consider the set $\mathcal{T}_X$ of all functions $X \to X$. Then $\mathcal{T}_X$ together with function composition is a semigroup. This is known as *the full transformation monoid* on X.

(g) The set $\mathcal{P}_X$ of all partial functions $f : X \to X$ together with partial function composition is a semigroup, known as *the partial transformation* monoid.

(h) For any set X, the powerset $\mathcal{P}(X) = \{Y : \forall Y \subseteq X\}$ together with the union operation is a monoid.

(i) For a set A, called an *alphabet*, we can define a *word* over A to be any finite sequence of elements of A. So, for example if $A = \{a, b, c\}$, then $(a, b), (a, c, b, c, a)$ and $()$ are all valid words over A, where $()$ is called the empty word. For simplicity we usually omit the parentheses and commas when writing words, so that (a, c, b, c, a) becomes $acbca$, and the empty word is written as ε by convention. Given words $u = u_1u_2 \ldots u_k$ and $v = v_1v_2 \ldots v_l$, define their *concatenation* to be the word $uv = u_1 \ldots u_kv_1 \ldots v_l$. One can easily check that concatenation is associative, and that the empty word is the identity for this operation.

Then A^+, the set of all non-empty words over A, together with concatenation is a semigroup, called the *free semigroup*. And similarly the set A^* of all words (including the empty word) is a monoid, called the *free monoid*.

Note that A does not necessarily have to be finite.

As this example shows, semigroups crop up all over the place in mathematics, and the sets and binary operation they use are often quite different. For example, performing a matrix multiplication is very different to composing two functions or concatenating two words or taking the union of two sets.

Our goal in this project is to derive various methods for computing with semigroups and elements within them. It is therefore useful to come to some sort of standardized description of elements and multiplication, since then we can more accurately reason about, for example, what properties of the elements and multiplication we can use in our methods, and we can better compare our methods to other methods that use the same input description.

Why such a standardized description should exist is not immediately clear, however it turn out that it does and in fact there exist multiple different descriptions that we could adopt. For example, it is possible to represent any semigroup as a collection of functions together with composition [4, Theorem

1.22]. Or, if the semigroup is finite, then we can always come up with a multiplication table for it like in Example 1.1.4(e), so the elements just become formal symbols and the operation consists of looking up a symbol in a table. While these descriptions can be useful, we are more interested in words than in functions.

And as it turns out, any semigroup can be thought of as a set of words that can be multiplied by concatenation with *some additional constraints.*

To make concrete what these additional constraints are, we first introduce the idea of subsemigroups and generation:

Definition 1.1.5 (Subsemigroups, generation)**.** A *subsemigroup* of a semigroup $(S, \cdot)$ is a subset $T \subseteq S$ that forms a semigroup under the same operation, i.e. a subset such that $(T, \cdot)$ is also a semigroup, where we restrict $\cdot$ to the domain $T \times T$. We sometimes write $T \leq S$ to mean that T is a subsemigroup of S.

Given any subset $U \subseteq S$ the smallest subsemigroup $T \leq S$ that contains U is called the subsemigroup *generated by* U. Dually, we call U the *generating set* of T. One can show that such a T always exists and is unique.

It is not hard to see that every semigroup is its own generating set, but often we can find much smaller generating sets, for example we can show that the full transformation monoid $\mathcal{T}_X$ with $|X| \geq 3$ finite is always generated by only 3 elements [4, Exercise 1.11].

Furthermore, if U is a generating set of S, then every element $s \in S$ can be expressed as $s = u_1 u_2 \cdots u_n$ with $u_1, u_2, \ldots u_n, \in U$ and $n \in \mathbb{N}$ (if this were not the case, then U would generate strictly smaller subsemigroup of S).

So, if we fix a generating set, we can express every element of S as a word in the generators of the generating set U, for example we could associate the word $u_1 u_2 \ldots u_n \in U^+$ with $s \in S$ above. However, this expression of an element as a word in generators may not be unique. One can also show that if $x, y \in S$, then xy has a word in the generators equal to the concatenation of a word in generators representing x and a word representing y.

This is all very similar to how the free semigroup U^+ works, however unlike in U^+, in S we can have multiple different sequences of generators representing the same element. To better express this, we need more theory still.

Definition 1.1.6 (Homomorphism, image, kernel)**.** A *homomorphism* between semigroups $(S, \cdot)$, $(T, \star)$ is a function $f : S \to T$ such that $\forall\, x, y \in S,\ f(x \cdot y) = f(x) \star f(y)$.

An *isomorphism* is a bijective homomorphism. We write $S \cong T$ to mean that S and T are isomorphic.

The *image* of a homomorphism is the set $\mathrm{im}(f) = \{f(x) : \forall x \in S\}$, one can show that it is a subsemigroup of T.

The *kernel* of a homomorphism is the relation $\ker(f) = \{(x, y) \in S \times S : f(x) = f(y)\}$. One can show that it is an equivalence relation on S.

Homomorphisms can be thought of as mapping some part of the structure of one semigroup into another. In this sense, we think of $\mathrm{im}(f)$ as a partial representation of S inside of T, with $\ker(f)$ somehow capturing the information about S that was lost or in some sense "forgotten" in the process of embedding S into T.

An isomorphism is then a mapping which preserves all information about S, and furthermore, identifies S fully with T. One can show that S and T have the same semigroup properties, and therefore it is convenient to think that S and T are the same up to a relabeling of elements, in the context of

semigroups. As we will see in this project, however, using different labelings of elements can be very useful in computation and provide much faster algorithms, so the distinction can still be useful if our context is broader than just that of semigroups.

We now formalize this idea of "forgetting" certain information about a semigroup by introducing congruences and quotients.

Definition 1.1.7 (Congruence, quotient)**.** A *congruence* on a semigroup S is an equivalence relation $\rho \subseteq S \times S$ for which the following hold: $\forall\, z \in S$ and $\forall\, (x, y) \in \rho$, we also have $(x \cdot z, y \cdot z) \in \rho$ and $(z \cdot x, z \cdot y) \in \rho$.

For any relation $\rho \subseteq S \times S$ there exists a least congruence containing ρ. We call this the congruence generated by ρ and denote it $\rho^\sharp$.

The *quotient* of S by a congruence ρ is the semigroup on the set of equivalence classes of ρ, $(S/\rho, \star)$, where multiplication is given by $x/\rho \star y/\rho = (x \cdot y)/\rho$. One can show that this is indeed a valid semigroup. Furthermore, the function $f : S \to S/\rho$ given by $f(x) = x/\rho$ can be shown to be a homomorphism.

Roughly speaking, an equivalence relation captures within an equivalence class all elements that satisfy a certain property. A congruence is an equivalence relation that is compatible with the semigroup structure, therefore its equivalence classes contain all elements that satisfy some semigroup property. In such a manner, when we quotient, we look at how the equivalence classes interact among themselves, forgetting about the differences of elements in the same equivalence class.

If this seems similar to what we said about kernels then it is, because as it turns out, kernels and quotients are equivalent:

Theorem 1.1.8 (Kernels and conguences are equivalent)**.** Given semigroups S, T, and any homomorphism $f : S \to T$, $\ker(f)$ is a congruence on S. And likewise, for every congurence $\rho \subseteq S \times S$, there exists a semigroup T and homomorphism $f : S \to T$ such that $\ker(f) = \rho$.

Therefore congruences are actually equivalent to kernels.

Proof. Let $f : S \to T$ be a homomorphism. We already know that $\ker(f)$ is an equivalence relation on S. To see that it is a congruence, note that if $(x, y) \in \ker(f)$ and $z \in S$, then $f(x) = f(y)$ and so $f(xz) = f(x)f(z) = f(y)f(z) = f(yz)$, therefore $(xz, yz) \in \ker(f)$. Similarly one establishes that $(zx, zy) \in \ker(f)$ too. So $\ker(f)$ is a congruence.

Now let $\rho \subseteq S \times S$ be a congruence. Then let $T = S/\rho$, and let $f : S \to T$ be the homomorphism $f(x) = x/\rho$. Now note that $x/\rho = y/\rho \Leftrightarrow x \sim_\rho y$, so $f(x) = f(y) \Leftrightarrow (x, y) \in \rho$, therefore $\ker(f) = \rho$ as required. △

We now formalize the idea that the image of a homomorphism is a representation of S inside of T with $\ker(f)$ capturing the information that was lost about S in the process.

Theorem 1.1.9 (First isomorphism theorem)**.** Let S, T be semigroups and $f : S \to T$ a homomorphism. Then

$$S/\ker(f) \cong \operatorname{im}(f)$$

Now to bring it all back to our idea of thinking representing an arbitrary semigroup as a collection of words under concatenation with some words representing the same element:

Theorem 1.1.10 (Fundamental property of free semigroups)**.** Let A^+ be the free semigroup.

Then for any semigroup S and function $\varphi : A \to S$ there exists a unique homomorphism $\widehat{\varphi} : A^+ \to S$ such that $\hat{\varphi}(a) = \varphi(a)$ for all $a \in A$.

In particular, if S has a generating set $U \subseteq S$, then taking $\varphi : U \to S$ to be the inclusion map, we conclude that $U^+/\ker\varphi \cong S$.

This means that, given any semigroup S and generating set U, we can think of S as consisting of words over the alphabet U, with multiplication given by concatenation, where there exists a congruence capturing among its equivalence classes all the words that represent the same element.

Now, using these ideas as a basis we are finally ready to define the main way we will represent semigroups and their elements in this paper:

Definition 1.1.11 (Presentation)**.** Let A be a set and $\rho \subseteq A^+ \times A^+$ be any relation. Then we call the pair (A, ρ) a *presentation*, often writing it as $\langle A|\rho\rangle$ instead. If A and ρ are finite, then we call the presentation *finite*.

A semigroup S is said to be *presented by* $\langle A|\rho\rangle$ if $S \cong A^+/\rho^\sharp$.

We can think of the elements $(l, r) \in \rho$ as rules for converting one word into another:

Lemma 1.1.12. [4, Proposition 2.7] Let A be a set and $\rho \subseteq A^+ \times A^+$. If the words $u, v \in A^+$ are in the same equivalence class in $\rho^\sharp$, then there exists a sequence of words $u = w_1, w_2, \ldots, w_n = v$ and relations $(l_1, r_1), \ldots, (r_{n-1}, l_{n-1}) \in \rho$ such that for every $i \in \{1, \ldots, n-1\}$, either there is a subword of w_i equal to l_i such that replacing that subword by r_i we get w_{i+1} or there is a subword of w_i equal to r_i such that replacing it by r_i we get w_{i+1}.

When writing presentations we often omit the curly braces for sets and write relations $(l, r) \in \rho$ as equations $l = r$ instead, which is justified by Lemma 1.1.12. For two words $u, v \in A^+$ we write $u \sim v$ if $(u, v) \in \rho^\sharp$ and the presentation is clear from context.

Example 1.1.13 (Some examples of presentations).

(a) Any semigroup $(S, \cdot)$ is presented by $\langle S \mid xy = x \cdot y, \forall x, y \in S\rangle$. This presentation essentially just encodes the binary operation within the relations.

(b) Consider the presentation $\langle a, b \mid aba = b, bab = b\rangle$. We can show that $aa \sim bb$ within the semigroup presented by this presentation, since

$$aa \sim a(a) \sim a(bab) \sim (aba)b \sim bb$$

1.1.2 Varieties of semigroups

We now consider varieties of semigroups with a particular focus on the variety of bands. The exposition is loosely based upon that of [8, Chapters 4.3-6].

Before we can talk about varieties, we have to introduce the concept of a direct product of semigroups:

Definition 1.1.14 (Direct product)**.** Let $\{S_\alpha\}_{\alpha\in A}$ be a collection of semigroups indexed by the set A (A is allowed to be infinite, even uncountable). Then the *direct product* of $\{S_\alpha\}_{\alpha\in A}$, denoted $\prod_{\alpha\in A} S_\alpha$ is defined to be the semigroup whose elements are functions $f : A \to \bigcup_{\alpha\in A} S_\alpha$ such that $f(\alpha) \in S_\alpha$ for all α, and multiplication is defined componentwise, i.e. $(fg)(\alpha) = f(\alpha)\ g(\alpha)$.

In the case of two semigroups S, T, their direct product can be shown to be equivalent to the semigroup $S \times T$ with product $(s, t)(u, v) = (su, tv)$.

With this in mind we define a variety as follows.

Definition 1.1.15 (Variety of semigroups)**.** A variety $\mathcal{V}$ of semigroups is a class of semigroups that is closed under direct products, homomorphic images and subsemigroups.

This is to say, if $\{S_\alpha\}_{\alpha\in A}$ is a collection of semigroups belonging to the variety $\mathcal{V}$, then $\prod_{\alpha\in A} S_\alpha$ also belongs to $\mathcal{V}$. If S, T belong to $\mathcal{V}$ and $f : S \to T$ is a homomorphism, then $\mathrm{im}(f)$ belongs to $\mathcal{V}$. And finally, if S is a semigroup belonging to the variety $\mathcal{V}$, and $T \leq S$ is a subsemigroup, then T belongs to $\mathcal{V}$ as well.

Remark 1.1.16. We think of a *class* of objects as being any collection that is too large to be a set. For example, there is no set of semigroups, since if such a set did exist, we could encode all sorts of paradoxes akin to Russel's paradox concerning the set containing all sets that do not contain themselves. We therefore talk about the class of semigroups instead. Similarly a variety, as a collection of semigroups, is a class. △

Varieties as defined above are rather difficult objects to understand, especially from a presentation point of view that we set out to take in the previous section. We will now introduce a rather surprising alternative description of varieties that helps us with this.

First we need to introduce a seemingly different way of defining a class of semigroups:

Definition 1.1.17 (Identical relation, equational class)**.** Let X^+ be the free semigroup on some alphabet X. An *identical relation* is simply a pair of words $(u, v) \in X^+ \times X^+$, we often write it as $[u = v]$ instead. A semigroup S is said to *satisfy* the identical relation $[u = v]$ if for every $\varphi : X \to S$ we have that $\widehat{\varphi}(u) = \widehat{\varphi}(v)$ in S, where $\widehat{\varphi}$ is the unique homomorpishm $\widehat{\varphi} : X^+ \to S$ extending φ.

For a given set $\mathcal{R} \subseteq X^+ \times X^+$ of identical relations, its *equational class* is defined to be the collection of all semigroups that satisfy all of the identical relations in $\mathcal{R}$. If $\mathcal{R}$ is finite, then we call the equational class *finitely based.*

To make this more concrete, take for example the set $X = \{x, y, z\}$ and the set of identical equations $R = \{[xyxz = xz], [xzy = zyx]\}$, then their equational class is exactly the collection of all semigroups S such that $\forall\, x, y, z \in S$, $xyxz = xz$ and $xzy = zyx$ holds. That is to say that in general, we can think of the identical relations as specifying an equation that has to be true for all possible assignments of variables.

One thing to note is that multiple different sets of relational equations can define the same equational class.

Example 1.1.18 (Different sets of equations defining the same class)*.* The identical relation $[x = y]$ defines the class of *trivial* semigroups, since any semigroup S in this class satisfies $\forall\, x, y \in S$, $x = y$ implies that S has at most 1 element, so the only semigroups in this class are the trivial semigroup and the empty semigroup.

On the other hand, if we consider the equational class defined by the relations $\{[xy = yx], [xy = x]\}$, then one can see that any semigroup S in this class must satisfy $\forall\, x, y \in S$, $x = xy = yx = y$. So semigroups of this class also satisfy the identical relation $[x = y]$, and we can further show that in fact both classes are equal.

What is surprising is that varieties can be equivalently specified by identical relations:

Theorem 1.1.19 (HSP theorem)**.** [8, Theorem 4.3.1] Every variety $\mathcal{V}$ is an equational class, and every equational class is a variety.

Example 1.1.20 (Examples of varieties).

- The variety defined by the equation $[xy = yx]$ is known as the variety of *commutative semigroups*. So, for example, the set of reals together with multiplication belong to this variety, since real multiplication is commutative. On the other hand, for a fixed $n \geq 2$, the set of all $n \times n$ matrices over the reals with matrix multiplication do not belong to this variety, since matrix multiplication is not commutative.
- The variety defined by the equation $[x^2 = x]$ is known as the *variety of bands*. The flip flop monoid from Example 1.1.4 is a band, since $a^2 = a, b^2 = b, c^2 = c$ from its multiplication table.
- The variety of semilattices is defined by the identical relations $[x^2 = x, xy = yx]$. For any set X, the powerset $\mathcal{P}(X) = \{Y : \forall Y \subseteq X\}$ together with the union operation is an example of a semigroup within this variety.

In the previous section we saw the importance of the free semigroup. We now exhibit an analogue of the free semigroup within a variety:

Definition 1.1.21 (Free-$\mathcal{V}$ semigroup)**.** Let $\mathcal{V}$ be the variety defined by the identical relations $R \subseteq X^+ \times X^+$ over the set X. Then for a set A define the relation ρ by

$$\bigcup_{(u,v)\in R} \{(\widehat{\varphi}(u), \widehat{\varphi}(v)) : \forall\, \varphi : X \to A^+\}$$

Where $\widehat{\varphi} : X^+ \to A^+$ is the unique extension of φ to X^+.

The *free-$\mathcal{V}$ semigroup* over A is then defined to be the semigroup $F_{\mathcal{V}}(A)$ presented by $\langle A \,|\, \rho \rangle$.

As the name suggests, free-$\mathcal{V}$ semigroups satisfy a variation of the fundamental property of free semigroups:

Theorem 1.1.22 (Fundamental property of free-$\mathcal{V}$ semigroups)**.** [8, Section 4.3] Let $\mathcal{V}$ be a variety, A a set and S an arbitrary semigroup belonging to $\mathcal{V}$. Then for any function $\varphi : A \to S$ there exists a unique homomorphism $\widehat{\varphi} : F_{\mathcal{V}}(A) \to S$ such that $\widehat{\varphi}(a) = \varphi(a)\, \forall\, a \in A$.

Therefore in a similar manner, we can talk about presentations for semigroups within a variety. This is why the study of free-$\mathcal{V}$ semigroups is important — it provides us with an important way of representing all semigroups within the variety.

Example 1.1.23 (Examples of free-$\mathcal{V}$ semigroups).

- Consider the variety of commutative semigroups. Then its free-$\mathcal{V}$ semigroups for $A = \{a, b, c\}$ by definition is the semigroups presented by $\langle a, b, c \,|\, xy = yx\, \forall\, x, y \in A^+\rangle$. This presentation has quite a large set of relations, but it can be shown that actually only 3 suffice - the presentation $\langle a, b, c \,|\, ab = ba, bc = cb, ca = ac\rangle$ presents the same semigroup.

  Furthermore, we can rewrite any word $u \in A^+$ using these rules so that all the a's occur first, all the b's occur second and the c's occur last. In this way, every word can be rewritten as $a^i b^j c^k$, $i, j, k \in \{0, 1, \ldots\}$ with $i + j + k \neq 0$.
- Consider the variety of bands. Let $A = \{a, b\}$. Then the free-$\mathcal{V}$ semigroup over A within the variety of bands is presented by $\langle a, b \,|\, x^2 = x\, \forall\, x \in A^+\rangle$.

One could speculate by analogy with the previous example that this presentation is equivalent to $\langle a, b \mid a^2 = a, b^2 = b \rangle$, but this is wrong! Indeed, one can show that $F_{\mathcal{V}}(A)$ has only 6 elements, but $\langle a, b \mid a^2 = a, b^2 = b \rangle$ defines an infinite semigroup!

- For the variety of semilattices one can show that the free-$\mathcal{V}$ semigroup over a finite set A is always isomorphic to the set on non-empty subsets of A under union.

1.2 Computation

We now turn our attention to the questions of computation, with particular emphasis on computing with semigroups.

In this section we will first formally introduce the theory of computation and decidability in order to give a precise formal definition of the word problem in semigroups. Afterwards computational complexity is introduced, and we review two related computational problems and outline how solving them can help us solve the word problem and be otherwise beneficial.

A great book for further reading on theory of computation is [14].

1.2.1 A quick overview of the theory of computation

We base this section on [14, Chapters 3-4].

Very informally, an algorithm could be seen as a set of instructions that can be performed on some input data to produce a result. A Turing machine formalizes this idea by giving a clear definition of input, basic instructions that can be performed, and the way in which a computation stops.

We model it as consisting of a set of states along with state transitions that roughly correspond to different steps of the computation, as well as an infinite tape for memory and a tape head. Initially the tape contains the input to the computation, and the Turing machines tape head is centered on the first input symbol, the rest of the tape is filled with blank symbols, and the Turing machine is in a designated initial state.

A computation then proceeds by reading the symbol on the tape that is under the tape head. Then, depending on the current state and the symbol read, we write a symbol on the tape, move the tape head either left or right and change the current state. This continues until we reach an accepting or rejecting state.

As output, we get whatever is on the tape at the end of the calculation as well as whether we accepted or rejected the input.

Definition 1.2.1 (Turing machine)**.** [15] A *deterministic Turing machine* $\mathcal{M}$ is a 7-tuple

$$(Q, : w\Sigma, \Gamma, q_0, q_a, q_r, \circ)$$

consisting of a finite set of states Q, a finite input alphabet Σ that does not contain the blank symbol "_", a finite tape alphabet $\Sigma \subseteq \Gamma$, states $q_0, q_a, q_r \in Q$ that are the initial, accepting and rejecting states respectively, and finally $\circ : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ the state transition function.

For a given input word $w \in \Sigma^*$, the tape initially consists of only w followed by infinitely many blank symbols. Initially we are in state q_0 and the tape head points to the first letter of w. Afterwards, if our state is q and the symbol that the tape head is pointing to is a, then we use the state transition function to determine $q \circ a = (q', a', D)$, where q' is the new state, a' is the new symbol under the tape head, and $D \in \{L, R\}$ determines whether to move the tape head left or right along the tape respectively (if we are at the leftmost position along the tape the no movement happens).

The computation *terminates* when we reach either q_a or q_r. If we reach q_a, then the input word w

is *accepted* by the Turing machine. Otherwise, if q_r is reached, then the word w is *rejected.* Note that it is possible for a computation to not terminate by never reaching either the accepting or rejecting state.

If a Turing machine accepts the input, then its *output* is the word on the tape at the end of the computation without the infinite tail of empty symbols. In this way, a Turing machine computes a function $f : D \to \Gamma^*$ where $D \subseteq \Sigma^*$ is the set of all words accepted by $\mathcal{M}$.

Now that we have defined a computing device, we can give a formal definition of a computational problem:

Definition 1.2.2 (Decidable language, decision problem)**.** Let Σ be given. We call a subset $L \subseteq \Sigma^*$ a *language.* A language L is *decidable* if there exists a Turing machine $\mathcal{M}$ that accepts all words in L and rejects all words not in L. The *decision problem* for L asks whether L is decidable or not.

While the Turing Machine seems like a rather strange and limited machine for performing computations, the *Church-Turing thesis* [5] asserts that the Turing machine is the most general possible device for computation, in the sense that no other reasonable computational device can decide strictly more languages than the Turing machine can. While it was first formulated by Church for the model of computation known as the lambda calculus in [5], in [15], Turing shows the equivalence between lambda calculus and Turing machines. Since its first formulation in 1936, he thesis has withstood the test of time, and it is generally accepted to be the definition of computability.

Remark 1.2.3 (A note on encoding)**.** Although decision problems are defined with respect to words and languages, often we would like to ask similar questions about non-word data. For example we might want to know if a given graph (say a network of cities connected by roads) is connected (if we can get a city to any other city by following the roads), or maybe we want to know if a given presentation corresponds to a finite semigroup.

This is where the notion of encoding comes into play. Essentially every such problem can be encoded into a word in one way or another. For example, we could encode our city network by writing down how many cities there are and then for every road writing which pair of cities it connects. Alternatively, we could write out a matrix row by row, where (i, j)-th entry is 0 or 1 depending on whether the i-th and j-th city have a road between them. Similarly, we can write down a number, for example by using its binary expansion (since our input alphabet has to be finite).

It turns out that the encoding used for the problem does not change whether or not it is decidable, but it could change the running time of the Turing machine deciding it (more on this later). △

We would like to stress that in the rest of this work we will treat the algorithmic aspects less formally. In particular we aim to describe our algorithms at a higher level by either describing the high level idea, or writing out *pseudocode* (a human readable intermediate description) rather than writing out Turing machines for solving the problem explicitly, as that would be extremely impractical and would make the key ideas of the algorithms much harder to understand.

Example 1.2.4 (Some example decision problems)*.*

- For a given triple of integers a, b, c, is $a + b = c$? We could encode this for example by $1^a\#1^b\#1^c$, this would be an encoding of the problem in unary. The problem is decidable, and one way of solving it is by traversing 1^a and 1^b and crossing off a 1 from 1^c for every 1 in 1^a or 1^b. We then accept if we had to cross off exactly as many 1's as there were in 1^c and reject otherwise. So this problem is decidable.
- Another encoding for the $a + b = c$ problem would be to encode a, b and c in binary, and then concatenate them with # as a separating character. This is now a considerably more difficult problem, since we cannot just go along the tape crossing off digits, we need to

actually implement some sort of binary addition scheme.

- For a given number n, is n prime? This too can be decided. We essentially go through all possible numbers less than p and check if each of them divides p or not.
- Given a graph, is it connected? This too can be decided, essentially by starting at a vertex x and marking it as visited, then for every edge (x, y) emanating from x, we visit the vertex y if it isn't marked and repeat the marking and visiting process until we cant go to an unmarked node anymore. If some node is still unmarked after this, clearly the graph is not connected. We then repeat this process with all other vertices in the graph.
- The *halting problem* asks, for a given Turing machine M, and input u, does M halt? Note that as before the specific encoding of M and u is not important.

We finish the section off with a rather surprising result, which showcases that not everything can be computed, even in principle:

Theorem 1.2.5. The halting problem is undecidable.

The proof is beautiful, and can be found in [14, Theorem 4.11]. Many computational problems within the realm of semigroups are undecidable, we will discuss in particular the word problem in the next section.

In fact, the situation is even worse than first appears. For any finite alphabet A, the set of all words A^* can be shown to be countably infinite (i.e. the set of words is in a bijective correspondence with the naturals), therefore the set of all languages $L \subseteq A^*$ is equal to the power set $\mathcal{P}(A^*)$, which can be shown to be uncountable (infinite and strictly larger than the naturals). Finally, the set of Turing machines can be shown to be only countable as well. Therefore, due to the immense difference between the cardinality of all languages, and the cardinality of Turing machines, and therefore decidable problems we can say that, in some sense, almost all languages are undecidable.

1.2.2 The word problem in semigroups

With the above in mind we can define:

Definition 1.2.6 (Word problem)**.** Let A be a and ρ be a relation on A^+. Let S be the semigroup presented by $\langle A \mid \rho \rangle$. Then the *word problem* in S with respect to A, ρ is the decision problem that asks, for any two $u, v \in A^+$, is $u \sim_{\rho^\sharp} v$ or not?

In other words, we are given two products of generators of S and want to find out whether they evaluate to the same element in S or not. Note that we do not require A and ρ to be finite, although very often it is the case that they are. When working with free bands later, we will have A finite and ρ infinite.

A classic result of [11] shows that

Theorem 1.2.7 (The word problem is undecidable)**.** There exists a set A and a finite relation ρ on A^+ such that the associated word problem is undecidable.

Therefore we cannot come up with a general purpose algorithm that would solve the word problem for an arbitrary semigroup!

However, if we fix A and ρ, then within the specific semigroup there might be an algorithm for deciding the word problem. Here is a small selection of word problems:

Example 1.2.8 (Some word problems).

(a) Consider the free commutative semigroup on 3 generators, we saw before that is has the presentation $\langle a, b, c \mid ab = ba, ac = ca, bc = cb\rangle$. Consider the word problem in this semigroup with respect to $A = \{a, b, c\}, \rho = \{(ab, ba), (ac, ca), (bc, cb)\}$.

We showed earlier, that using the relations, any word $w \in A^+$ can be rewritten as $a^i b^j c^k$ with $i + j + k \neq 0$, lets call this the "sorted" form of w, since all the letters of w were sorted in alphabetical order.

Clearly if two words $u, v \in A^+$ have the same sorted form then they are equivalent, since each word is equivalent to its sorted form, and equivalence is transitive. We can further show that the opposite also holds - two words are equivalent if and only if their sorted forms are equal.

But this gives us a simple way of solving the word problem - count the number of times each letter occurs in each word. The compare the counts and accept the pair if the counts are the same and reject otherwise. One can show that this can be done with a Turing machine.

(b) In general, every finitely presented commutative semigroup has a solvable word problem [1].

(c) In [9] it is shown, perhaps surprisingly, that for the set $A = \{a, c, e, f\}$, and relation $\rho = \{(ac, ca), (af^2, f^2a), (fac, cfa), (eca, ce), (ef^3a, f^2e), (a^2c^2e, a^2c^2)\}$, the word problem is undecidable!

(d) Consider $\langle a, b \mid ab^n a = aba \, \forall \, n \in \mathbb{N}\rangle$. This is an example of an infinite presentation with a solvable word problem.

1.2.3 Computational complexity

This section is based on [14, Chapters 7-8].

Knowing whether a problem is decidable or not is important, however, in practice it is equally important to know how long a given computation will take. Indeed, if a problem is decidable, but the best method we have for deciding it takes longer than the lifetime of the universe, then it is of little practical use. Furthermore, as many students can attest, sometimes even if the best possible method is known and fast, it is still possible to manage to implement in a manner which runs for longer than the lifetime of the universe.

Therefore we now embark on defining how to quantify the resources used by a computation. We will primarily be focused on the time that a computation takes and the amount of space it requires.

Definition 1.2.9 (Time and space complexity)**.** Let $\mathcal{M}$ be a deterministic Turing machine that halts on every input. Then the *time-complexity* of $\mathcal{M}$ is the function $t : \mathbb{N} \to \mathbb{N}$ such that $t(n)$ is the maximum number of steps $\mathcal{M}$ takes on any input of length n. Similarly, the *space-complexity* is the function $s : \mathbb{N} \to \mathbb{N}$ such that $s(n)$ is the maximum number of tape cells that $\mathcal{M}$ visits on any input of length n.

If a problem can be solved by a Turing machine with time complexity $t(n)$ and space complexity $s(n)$, then we say that the problem is $t(n)$-time and $s(n)$-space solvable.

As currently stated, the time and space complexity of a Turing machine is a very specific quantity that might be hard to calculate exactly. The following theorem actually shows that, in some sense, instead of calculating the exact time or space complexity, we should really be looking at their growth rate:

Theorem 1.2.10 (Linear speedup theorem, linear compression theorem)**.** In what follows let n be the size of the input.

If a decision problem can be solved in time $t(n)$, then for any $\epsilon > 0$ there exists a Turing machine solving the problem in time at most $\epsilon t(n) + 2n + 3$.

If a decision problem can be solved in space $s(n)$, then for any $\epsilon > 0$ there exists a Turing machine solving the same problem in space at most $\epsilon \cdot s(n) + 2$.

In other words, provided our Turing machine has time complexity at least n (which is a reasonable assumption, since that is how much time it takes to fully read the input), then we can always improve the time complexity by a constant factor. Similarly we can improve the space by a constant factor as well.

So this motivates us to instead consider the asymptotics of the time and space complexity, rather than pinning them down exactly. To this end we introduce $\mathcal{O}$ notation.

Definition 1.2.11 (Big $\mathcal{O}$)**.** In what follows let $\mathbb{R}^+ = \{x \in \mathbb{R} : x \geq 0\}$ to be the set of non-negative reals, as opposed to the set of words over $\mathbb{R}$.

For a given function $f : \mathbb{N} \to \mathbb{R}^+$, we define $\mathcal{O}(f)$ to be the set of all functions $g : \mathbb{N} \to \mathbb{R}^+$ for which there exists constants $C \in \mathbb{R}, N \in \mathbb{N}$ with $C > 0$ such that for all $n > N$, $g(n) \leq Cf(n)$.

In other words, $\mathcal{O}(f)$ captures all the functions that have the same or slower growth than f as $n \to \infty$. We will use $\mathcal{O}$ notation to describe the worst case time or space complexity of algorithms.

Example 1.2.12.

(a) Clearly $f \in \mathcal{O}(f)$ for all $f : \mathbb{N} \to \mathbb{R}^+$.

(b) $0.001n^2 + 2n + 100\ln(n) \in \mathcal{O}(n^2)$, but it is not in $\mathcal{O}(\ln(n))$.

(c) $\mathcal{O}(f + g) = \mathcal{O}(\max(f, g))$, so we only need to consider the largest $\mathcal{O}$-class of any function of a product.

(d) $\mathcal{O}(n) \subsetneq \mathcal{O}(n^2)$ and in general $\mathcal{O}(n^\alpha) \subsetneq \mathcal{O}(n^\beta)$ for $\alpha < \beta$.

From now on we will speak of a problem having time complexity $\mathcal{O}(f)$ and space complexity $\mathcal{O}(g)$ if there exists a Turing machine solving it with time complexity in $\mathcal{O}(f)$ and space complexity in $\mathcal{O}(g)$.

Note that for a given decision problem, the time and space complexity depend on the encoding. So, for example, if we wished to decide if $a + b = c$ with a, b, c given in unary, then this would take at least time $\mathcal{O}(a+b+c)$, since we need to at least read all of the input. However, if we encoded a, b, c in binary, then we can exhibit a $\mathcal{O}\left((\ln(a) + \ln(b) + \ln(c))^2\right)$ time algorithm, since the encoding of a, b, c in binary is exponentially smaller than the actual numbers themselves.

Furthermore, while Turing machines are excellent models for machines in terms of figuring out qualitatively what a machine can or cannot compute, the time and space complexity of a problem can change if we give the Turing machine certain extra capabilities. For example

Example 1.2.13. Consider a new type of Turing machine, called the *two tape* Turing machine, which has an additional tape and tape head that can be manipulated in the same way as the tape for the original Turing machine.

Now consider the decision problem for the language $\{0^k 1^k : k \in \mathbb{N}\}$. We can solve it in $\mathcal{O}(n)$ time on a two tape Turing machine as follows: first copy all the initial zeros onto the second tape, then alternate reading off 1's from the first tape and crossing off 0's from the second tape. Accept if we stopped reading 1's and crossing off 0's at the same time, and there are no more symbols to read on the input tape and no more to cross out, reject otherwise.

On the other hand, one can show [14, Problem 7.47] that the only languages that are decidable

in $\mathcal{O}(n)$ time on a single tape Turing machine are so called *regular languages* (more on this later). But it can be shown that $\{0^k 1^k : k \in \mathbb{N}\}$ is not a regular language.

Similarly, a typical computer has a random access memory that allows it to access any part of the memory in a roughly constant amount of time, in contrast a Turing machine will take at least n steps to reach the n-th tape position.

It is for these reasons that we adopt a primitive based approach to modeling the quantitative aspects of computation. We will assume that there are certain "primitive" operations that take constant time and space to execute and use them to build more complicated algorithms later on. Instead of listing all such primitive operations here, we will mention them as need for them arises.

With this in mind we now analyze a very simple example.

Example 1.2.14. Consider the word problem in the free commutative semigroup on three generators, that is the word problem associated to the presentation $\langle a, b, c \mid ab = ba, ac = ca, bc = ca \rangle$. We showed before that solving this word problem consists of counting the number of letters that occur in each word and checking that the counts for both words are the same. We now outline an algorithm for doing so and analyze its space and time complexity.

For primitives we assume that it is possible to do arithmetic operations in constant time and space, it is possible to assign a value to a variable in constant time and space, and that it is possible to access an arbitrary letter in a word in constant time and space.

The algorithm for solving the word problem `AreEquivalent` then uses a subroutine `LetterCounts` for computing the count of each letter, and then compares the counts.

We have given a pseudocode implementation of `LetterCounts` in Algorithm 1 and of `AreEquivalent` in Algorithm 2, depicted in Figure 1.1. In the notation, $\leftarrow$ denotes variable assignment, lines starting with a pound sign # are comments, **for** is a keyword indicating iteration and **if** is a conditional statement.

Let us now analyze the time and space complexity of this algorithm. The subroutine `LetterCount` initializes 3 variable, each of which takes $\mathcal{O}(1)$ time and space by our assumptions, then it iterates through the letters of w, and performs and arithmetic operation and assignment, each of which take $\mathcal{O}(1)$ time, and this happens once for each letter, so $|w|$ times. Finally we return the values which also takes $\mathcal{O}(1)$ time and space according to our primitives. Therefore we utilize $\mathcal{O}(1 + 1 + 1 + |w| \cdot (1 + 1) + 1) = \mathcal{O}(|w|)$ time, and similarly we can show that we use $\mathcal{O}(|w|)$ space (since we need to store the input).

Finally, `AreEquivalent` calls `LetterCount` once for each of u and v, performs assignments, and then does one comparison for each letter in the alphabet. So this gives a total of $\mathcal{O}(|u| + |v|)$ time and space as well.

We stress that this isn't the time complexity of the problem on a Turing machine. Indeed, arithmetic operations on a Turing machine cannot be implemented in constant time, and similarly we cannot access arbitrary variables in constant time either. Rather, it is the complexity of the algorithm for a machine supporting our primitives, which we selected to roughly model the capabilities of a modern computer.

1.2.4 Normal forms and complete invariants

This section borrows ideas and notation from literature on the graph isomorphism problem, namely the notion of a complete invariant and canonical (normal) form. The latter appears in semigroup theory, however the former is discussed less often. We will also see how these are related to the word problem and how they can aid us in solving it.

Algorithm 1: `LetterCounts`(w)

Data : $w \in \{a, b, c\}^+$
Result : The number of times each letter occurs in w

```
1  # The variables count_x will store the number of occurrences of each letter
2  count_a ← 0
3  count_b ← 0
4  count_c ← 0
5  for letter x in w do
6  |  count_x ← count_x + 1
7  end for
8  return count_a, count_b, count_c
```

Algorithm 2: `AreEquivalent`(u, v)

Data : $u, v \in \{a, b, c\}^+$
Result : True if $u \sim v$ and False otherwise

```
1  count^u_a, count^u_b, count^u_c ← LetterCounts(u)
2  count^v_a, count^v_b, count^v_c ← LetterCounts(v)
3  for x ∈ {a, b, c} do
4  |  if count^u_x ≠ count^v_x then
5  |  |  return False
6  |  end if
7  end for
8  return True
```

Figure 1.1: Algorithms for solving the word problem in the free commutative semigroup on 3 generators.

Definition 1.2.15 (Complete invariant, normal form)**.** Let A be a set and ρ a congruence on $A^+ \times A^+$. Then a *normal form* is any function $\zeta : A^+ \to A^+$ such that $u \sim v$ if and only if $\zeta(u) = \zeta(v)$. Similarly a *complete invariant* is any function $\xi : A^+ \to D$, where D is an arbitrary set, such that $u \sim v$ if and only if $\xi(u) = \xi(v)$.

Clearly, a normal form is a special case of a complete invariant.

Example 1.2.16. Consider the word problem in the free commutative semigroup $\langle a, b, c \,|\, ab = ba, ac = ca, bc = cb \rangle$.

As we showed before, $u \sim v$ if and only if u and v contain the same number of a's, b's and c's.

An example normal form would therefore be $\xi(w) = a^i b^j c^k$, where i is the number of a's, j the number of b's and k the number of c's.

An example of a complete invariant would be $\xi : A^+ \to \mathbb{N}^3$ given by $\xi(w) = (i, j, k)$, i.e. we map a word to a vector containing the number of letters in each of its coordinates. Note that Algorithm 1 `LetterCounts` computes exactly this invariant.

From a purely theoretical point of view, normal forms are not particularly interesting, since if we can get a normal form for any set and congruence instantly by invoking the axiom of choice. However, if the normal form can be efficiently computed, then we can use it to solve the word problem.

Indeed, if ζ is a normal form, and we can compute $\zeta(w)$ in time $\mathcal{O}(f(n))$ where $n = |w|$, and the size of the output $|\zeta(w)|$ is in $\mathcal{O}(s(n))$, then to check if $u \sim v$, we simply compute the normal form of u and v and then compare if the normal forms are identical. We can check if two words are identical in $\mathcal{O}(n)$, so the total runtime is $\mathcal{O}(f(n) + s(n))$ where $n = |u| + |v|$ is the total size of the input. (Note that we assume that f and s are superadditive for this time analysis, that is $f(a + b) \geq f(a) + f(b)$ for all $a, b \in \mathbb{N}$, this is a reasonable assumption).

Similarly, if we have a complete invariant $\xi : A^+ \to D$ and there is an algorithm for comparing elements of D, then we can solve the word problem by computing the complete invariant of each element and comparing the invariants. The total time complexity is then $\mathcal{O}(f(n) + g(s(n)))$ where $\mathcal{O}(f)$ is the time complexity of computing the invariant, $\mathcal{O}(s)$ is the size of the complete invariant, and $\mathcal{O}(g)$ is the time complexity of comparing two invariants.

Using normal forms or complete invariants has additional advantages. For example, if we had a collection of words and we wanted to see which words a given word is equivalent to, we could compute

the complete invariant of each element in the collection in $\mathcal{O}(f(n))$ time once, and then store the output of these computations. Now every subsequent equality test only costs us $\mathcal{O}(g(s(n)))$ time, which can be considerably less than $\mathcal{O}(f(n))$.

If $s(n)$ is less than n, then storing the complete invariants takes less space than storing the original words, so this is also an advantage.

Finally, if we can endow the set of complete invariants D with a computable binary operation, turning it into a semigroup in a such a way that $D \cong A^+/\rho$, then we gain a computational representation of the semigroup A^+/ρ, which is very desirable in computational algebra.

Example 1.2.17. Taking the free commutative semigroup as before, we can see that while storing a word takes $\mathcal{O}(i+j+k)$ letters, where i, j, k are the count of each letter in the alphabet respectively, storing the complete invariant (i, j, k) only takes $\mathcal{O}(\ln(i)+\ln(j)+\ln(k))$ if we represent each number in binary, an exponential improvement!

Furthermore, if $\xi(u) = (i_1, j_1, k_1)$ and $\xi(v) = (i_2, j_2, k_2)$, then we can show that $\xi(uv) = (i_1 + i_2, j_1 + j_2, k_1 + k_2)$, and this addition can be efficiently implemented.

So by using a complete invariant we have represented the free commutative semigroup inside of $\mathbb{N}^3$ with componentwise addition, and shown that the complete invariants can be efficiently stored and multiplied.

Our motivation is exactly as laid out above — in this masters project we derive a complete invariant for elements of the free band that have desirable size and comparison properties, and then endow them with an efficient multiplication.

1.3 Automata and transducers

Automata are a very simple type of computing machine, with much lesser capabilities than a Turing machine. They are still useful, however, since they are great for matching patterns in words, and their simplicity allows for very efficient storage and computation with them.

Our main use for these structures will be in deriving a specific type of complete invariant based on minimal acyclic transducers. Therefore a large part of this section is dedicated to the theory and algorithms for acyclic automata, and then showing that these port over nicely to the type of transducer that we will be using in this project.

1.3.1 Automata basics

From a purely structural point of view, an automaton is just a directed edge colored graph with certain distinguished vertices. What makes them interesting objects of study, however, is the interpretation of automata as machines for detecting certain patterns in words.

To do this, when given a word, an automaton reads it letter by letter, and modifies its state depending on its current state and the current letter. Once there are no more letters to read, the automaton then checks whether its current state is accepting, in which case the word does indeed contain the pattern that the automaton was looking for.

We formalize these ideas as follows:

Definition 1.3.1 (Automaton, regular language)**.** A *deterministic finite state automaton* $\mathcal{A}$ is a 5-tuple $(Q, A, q_0, T, \circ)$ consisting of a finite set of states Q, a finite alphabet A, an initial state $q_0 \in Q$, a set of terminal states $T \subseteq Q$ and a **partial** function $\circ : Q \times A \to Q$ called a state transition function.

It is often convenient to extend the function $\circ$ to act on words $w \in A^*$. This is done by processing

w letter by letter, i.e., if $w = w_1 w_2 \dots w_n$ with $w_1, w_2, \dots, w_n \in A$, then $q \circ w = (\dots((q \circ w_1) \circ w_2) \circ \dots) \circ w_n$. Also, $q \circ \varepsilon = q$.

A word $w \in A^*$ is *recognized* by $\mathcal{A}$ if $q_0 \circ w \in T$. The set of all words recognized by $\mathcal{A}$ is known as the language of $\mathcal{A}$. We call a subset $L \subseteq A^+$ a *regular language* if L is the language of some automaton.

Recall from Definition 1.1.1 that a partial function $f : X \to Y$ is a function that is allowed to be undefined on some inputs. If f is undefined on x, then we write $f(x) = \bot$, where $\bot$ is taken to be a formal symbol disjoint from Y. Composition is taken in the usual manner with $f(\bot) = \bot$.

Remark 1.3.2. Note that in some textbooks deterministic automata are required to have a total transition function, for example [14, Definition 1.5], i.e. we need to have $q \circ a \in Q$ for all $q \in Q, a \in A$, and cannot leave it undefined at some inputs as we did above. However, in literature relating to automata minimization, partial transition functions are used, for example they are explicitly used in [6] and implicitly in [13].

While this difference in definition is slightly confusing, it turns out that the set of languages recognized by automata of either definition coincide. This is because, if we are given a partial transition automaton $\mathcal{A} = (Q, A, q_0, T, \circ)$, we may create a new automaton $\mathcal{A}' = (Q \cup \{r\}, A, q_0, T, \circ')$ with new a “garbage” state r, and make all the undefined transitions lead to r, that is

$$\forall\, a \in A,\, \forall\, q \in Q,\, q \circ' a = \begin{cases} r & \text{if } q \circ a = \bot \\ q \circ a & \text{otherwise} \end{cases} \text{ and } r \circ' a = r$$

Clearly $\mathcal{A}'$ has a complete state transition function, and one can show that $\mathcal{A}$ and $\mathcal{A}'$ recognize the same language.

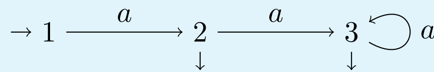
Since the partial function definition is prevalent in literature on automata minimization, it is the one we take in this work. △

Example 1.3.3 (Basic examples of automata and regular languages).

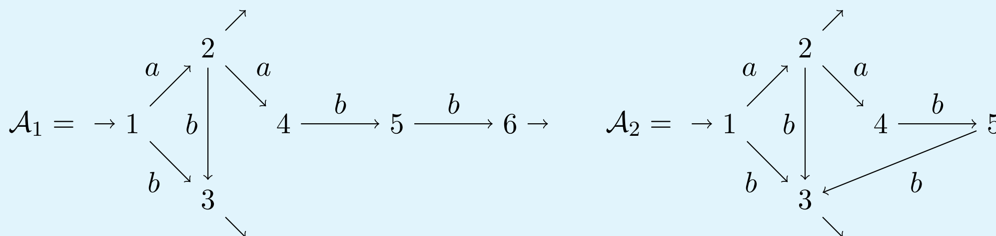
(a) The language $\{a^n : n \geq 1\}$ is regular. Indeed, an automaton recognizing it is given by $\mathcal{A} = (Q, A, q_0, T, \circ)$ with $A = \{a\}$, $Q = \{1, 2, 3\}, q_0 = 1$, $T = \{2, 3\}$ and transition function $\circ$ given by $1 \circ a = 2, 2 \circ a = 3, 3 \circ a = 3$.

Automata can be drawn in a graphical manner similar to graphs — we draw nodes labeled by the states Q, and connect two states q_1, q_2 by an arrow labeled x is there is a transition $q_1 \circ x = q_2$. The initial state is denoted by an unlabeled arrow which points into the node, and terminal states are denoted by unlabeled arrows pointing outside the node.

In this way, the above automaton can also be drawn as:



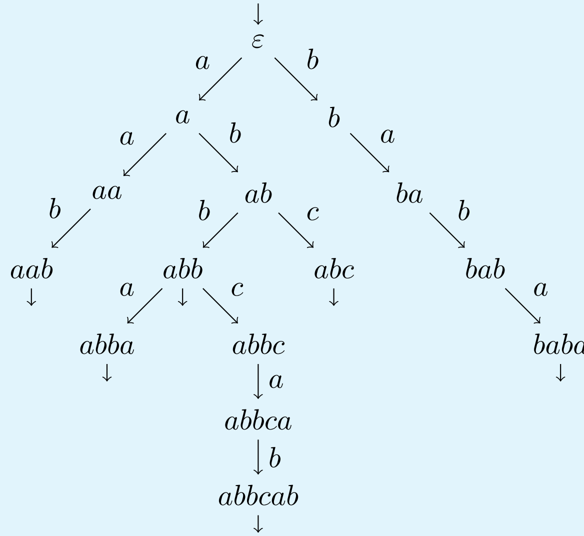
(b) The language $L = \{a, b, ab, aabb\}$ is regular. Note that a language can be recognized by multiple automata, here we exhibit two automata recognizing L:



(c) In a similar vein, any finite language $W = \{w_1, w_2, \dots w_n\} \subseteq A^*$ can be shown to be regular.

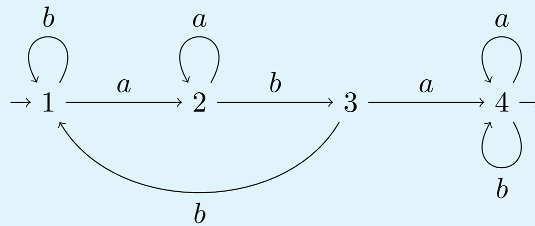
Indeed, let $\mathcal{A} = \{Q, A, q_0, T, \circ\}$ be an automaton with one state for every possible prefix, $Q = \{u \in A^+ : u \text{ is a prefix of some } w \in W\}$, the initial state being the empty word $q_0 = \varepsilon$, and every word in the language being a terminal state, i.e. $T = W$. Finally, for all $x \in A$, we define $\circ$ ny $q \circ x = qx$ if qx is a prefix, and define $q \circ x = \perp$ to be undefined otherwise. We can show that this automaton accepts exactly W. It is known as the *trie* for the language W.

To demonstrate, for $W = \{aab, abb, abba, baba, abbcab, abc\}$, the trie automaton is given by



(d) The language $\{w \in \{a, b\}^+ : w \text{ contains } aba \text{ as a subword}\}$ is regular.

It is recognized by



Now we give a notion of isomorphism for automata:

Definition 1.3.4 (Automata isomorphism)**.** Two automata $\mathcal{A}_1 = \{Q_1, A_1, q_{0,1}, T_1, \circ_1\}$ and $\mathcal{A}_2 = \{Q_2, A_2, q_{0,2}, T_2, \circ_2\}$ are called *isomorphic* if they have the same alphabet $A_1 = A_2$, and there exists a bijection $f : Q_1 \to Q_2$ such that $f(q_{0,1}) = q_{0,2}$, $f(T_1) = \{f(t) : t \in T_1\} = T_2$ and $f(q \circ_1 x) = f(q) \circ_2 x$ for all $q \in Q_1$, $x \in A_1$.

Put simple, two automata are isomorphic if they differ by a relabeling of their states. One can show that automata isomorphism can be decided in time and space $\mathcal{O}(n)$ where n is the number of transitions of the automaton. The algorithm proceeds by trying to construct a bijection, starting by identifying the initial states of both automata, and then building up the isomorphism by following the transitions.

The same language can be recognized by multiple different automata. It turns out that there is actually a unique smallest automaton recognizing any given regular language!

Theorem 1.3.5 (Existance of a minimal automaton)**.** [3] Let L be a regular language. Let $\mathcal{A}$ be an automaton recognizing L such that $\mathcal{A}$ has the least amount of states among all automata recognizing L (clearly such an automaton exists, as the number of states is a natural number, and

at least one automaton recognizes a regular language by definition). The any other automaton recognizing L with the same number of states as $\mathcal{A}$ is isomorphic to $\mathcal{A}$.

We call this automaton the *minimal* automaton recognizing L.

The process of taking an automaton $\mathcal{A}$ and finding the minimal automaton recognizing the same language as $\mathcal{A}$ is called *minimization*. One can show that minimization can be achieved by removing and merging together certain states of $\mathcal{A}$.

This is made more formal in [3]:

Definition 1.3.6. Let $\mathcal{A} = (Q, A, q_0, T, \circ)$ be an automaton.

A state $q \in Q$ is called *accessible* if we can reach it by following state transitions from the initial state, i.e. if there exists $w \in A^*$ such that $q_0 \circ w = q$. A state $q \in Q$ is *co-accessible* if we can reach a terminal state from q, i.e. if there exists $w \in A^*$ such that $q \circ w \in T$. $\mathcal{A}$ is called *connected* if every state is both accessible and co-accessible.

The *right-language* of a state $q \in Q$ is the set of all word that are accepted by the automaton if the initial state were q_0, i.e. $\{w \in A^+ : q \circ q \in T\}$. Two states are *equivalent* if they have the same right-languages and *inequivalent* otherwise.

Theorem 1.3.7 ([3], Definition 1)**.** An automaton is minimal if and only if it is connected and every pair of states is inequivalent.

Given any automaton, one can construct a connected automaton accepting the same language in $\mathcal{O}(n)$ time and space, where n is the total number of transitions of the automaton. To do this, we do two traversals of the automaton. We start the first traversal at the initial state, and mark every state we visit red, these will be the accessible states. Then we proceed by doing a traversal from the terminal nodes and following the transitions in reverse, marking states blue as we go, these are the co-accessible states. Finally we create a new automaton keeping only those nodes that have been marked both red and blue.

Removing equivalent states is much more difficult. Hopcroft's algorithm for general automata minimization can do this in time $\mathcal{O}(n \ln(n))$ and is the best known algorithm for doing so. See [2] for a more in depth look at automata minimization algorithms.

Since the minimal automaton is unique, minimization produces a normal form for automata, where we call two automata *equivalent* if they recognize the same language.

1.3.2 Acyclic automata

We now turn our attention to a much simpler class of automata, known as acyclic automata.

Definition 1.3.8 (Acyclic automaton)**.** An automaton $\mathcal{A} = \{Q, A, q_0, T, \circ\}$ is called *acyclic* if $\forall q \in Q, w \in A^*$, $q \circ w \neq q$.

In other words, there is no state such that following some sequence of transitions leads us back to the same state.

We can characterize the languages accepted by acyclic automata quite easily

Theorem 1.3.9. Every finite language is accepted by an acyclic automaton, and every acyclic automaton accepts a finite language.

Proof. As we saw in Example 1.3.3 (c), every finite language is accepted by a trie automaton, and it can be easily verified that it is acyclic, since every transition increases the length of a states label by one.

On the other hand, if an automaton is acyclic and has n states, then it cannot accept a word of length $n+1$, since otherwise, by the pigeonhole principle, some state must be repeated as we traverse the word to an accepting state. But this repetition implies a cycle, a contradiction. △

So acyclic automata recognize a very limited set of languages. At the same time, they can be minimized with better time and space complexity than a general automaton.

Theorem 1.3.10 (Revuz minimization [13])**.** An acyclic automaton can be minimized in $\mathcal{O}(n)$ time and space, where n is the number of transitions of the automaton.

Recall from Theorem 1.3.7 that an automaton is minimal if and only if it is connected and no two states are equivalent. We already saw that creating an equivalent connected automaton can be done in time and space $\mathcal{O}(n)$.

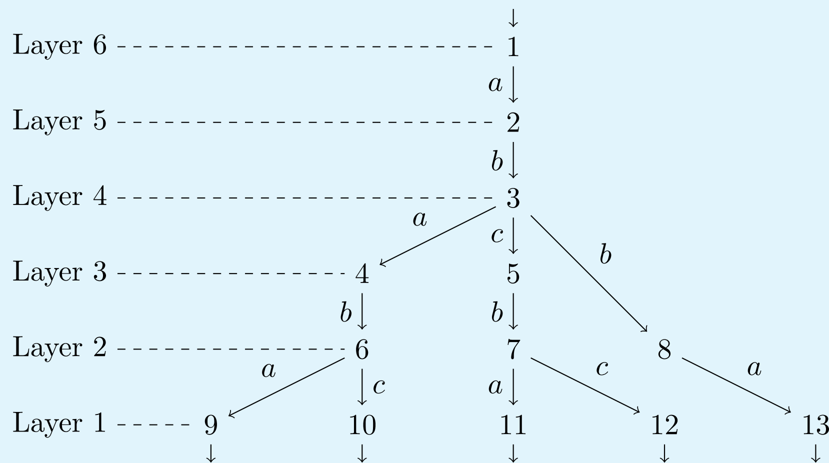
The key insight of Revuz is that the states of a connected acyclic automaton can be partitioned into layers. At the bottom layer are all the states that have no outgoing transition. At the next layer above are all the states that have only transitions to states in the bottom layer. In general, the k-th layer contains all the states that have transitions only to states in a lower layer, and at least one transition to a state in the $k-1$-st layer.

Minimization the proceeds from bottom to top. It is quite easy to see which states are equivalent in the bottom layer (indeed, all states in the very bottom layer must be equivalent), so we can just merge them. In general, once we have merged all the equivalent states in the first $k-1$ layers, then two states in the k-th layer are equivalent if and only if all the transitions from the two states are the same (have the same labels and lead to the same state), since no pair of states in a lower layer are equivalent.

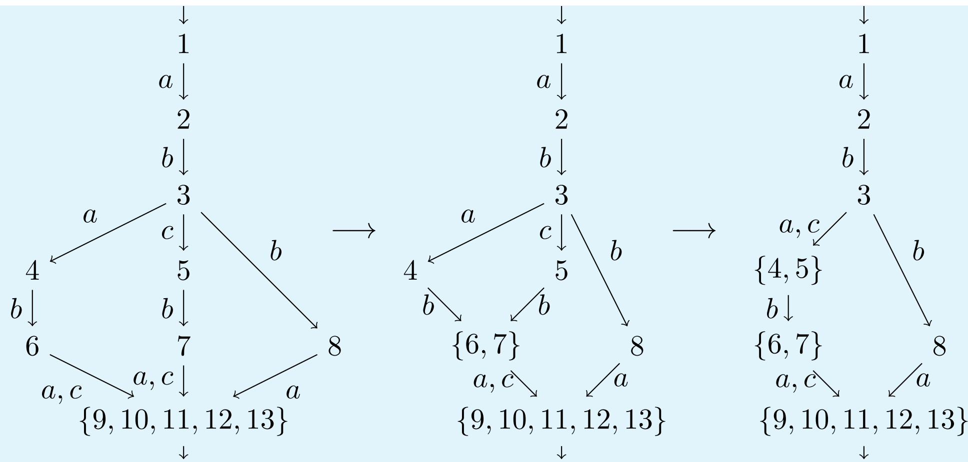
To actually merge a layer we perform a *radix sort*, which takes linear time and space to sort a collection of tuples.

In practice, instead of merging states straight away we usually pick a single state from each equivalence class and make all relevant transitions point to this state. Then at the end we remove all non-accessible states again.

Example 1.3.11. Consider the trie for $\{ababa, abcba, abba, ababc, abcbc\}$, we have drawn it below, drawing all the nodes in the same layer at the same height:



Note that all the nodes in the first layer are equivalent, since their right languages are $\{\varepsilon\}$, so after merging all of the we would be left with the transducer drawn on the left below:



Note that now, states 6 and 7 both have exactly two transitions, one labeled with a and one labeled with c to $\{9, 10, 11, 12, 13\}$, therefore they are equivalent so we perform another merge (center). Finally we merge $4, 5$ in Layer 3 (right). One can show that this automaton is indeed equivalent to the original and is furthermore minimal.

1.3.3 Transducers

Whereas an automaton can be thought of as a machine for detecting patterns in words, a transducer is a machine computing a relation between two sets of words. The theory of transducers in general is quite rich, however we will only be interested in them in a very narrow part of that.

To this end, we define the transducers used in this work:

Definition 1.3.12 (Transducer)**.** A *deterministic synchronous finite state* transducer $\mathcal{T}$ is a 7-tuple $(Q, \Sigma, \Gamma, q_0, T, \circ, *)$ consisting of a finite set of states Q, an input alphabet Σ, an output alphabet Γ, an initial state $q_0 \in Q$, a set of terminal states $T \subseteq Q$, a **partial** state transition function $\circ : Q \times \Sigma \to Q$ and a **partial** output function $* : Q \times \Sigma \to \Gamma$. We require that the domains of $\circ$ and $*$ coincide.

When processing a word $w \in \Sigma^*$, if we are in the state q and read the letter a, then we think of $q \circ a$ as being the next state we transition into, and $q * a$ as the letter we write to the output.

We extend $\circ$ to words $w \in \Sigma^*$ in a similar manner to how we did with automata.

The *function realized* by $\mathcal{T}$ is the partial function $f_{\mathcal{T}} : \Sigma^* \to \Gamma^*$ given by

$$f_{\mathcal{T}}(a_1 a_2 \dots a_n) = q_0 * a_1 \cdot (q_0 \circ a_1) * a_2 \cdot (q_0 \circ a_1 a_2) * a_3 \cdots (q_0 \circ a_1 a_2 \dots a_{n-1}) * a_n$$

if $q_0 \circ a_1 \dots a_n \in T$ and $f_{\mathcal{T}}(a_1 a_2 \dots a_n) = \perp$ otherwise.

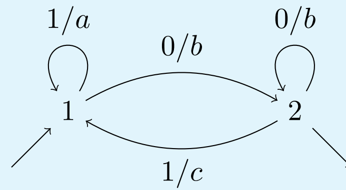
Here are some examples to demonstrate what we mean:

Example 1.3.13 (Simple examles of transducers).

(a) Consider the transducer $\mathcal{T} = (Q, \Sigma, \Gamma, q_0, T, \circ, *)$ with $Q = \{1, 2\}$, $\Sigma = \{0, 1\}$, $\Gamma = \{a, b, c\}$, $q_0 = 1$, $T = \{2\}$, transition function given by $1 \circ 0 = 2, 1 \circ 1 = 1, 2 \circ 0 = 2, 2 \circ 1 = 1$ and output function given by $1 * 0 = b, 1 * 1 = a, 2 * 0 = b, 2 * 1 = c$.

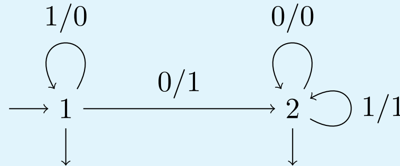
Similar to automata, there is a visual way of drawing transducers, where we denote the transition $q_1 \circ a = q_2$ with output $q_1 * a = b$ by an arrow from q_1 into q_2 labeled a/b.

So we can draw this same transducer as:



The function realized by this transducer $f_{\mathcal{T}}$ is defined for all words ending in 0. Some example evaluation of it are $f_{\mathcal{T}}(010001) = \perp$, $f_{\mathcal{T}}(10010) = abbcb$ and $f_{\mathcal{T}}(1110001100010) = aaabbbcabbbcb$

(b) Consider the transducer given in the picture below



The function that it realizes $f_{\mathcal{T}}(x)$ can be seen to compute the first n digits of $x+1$, where n is the number of digits of x and we interpret the word x as a number given in binary with little endian convention (smallest bit is leftmost). So for example 23 in this convention becomes 11101, and $f_{\mathcal{T}}(11101) = 00011$ which is the little endian representation of 24.

Transducers might seem quite similar to automata, and that is because they are! In fact we can establish an equivalence between transducers and automata:

Theorem 1.3.14. Every deterministic synchronous finite state transducer with input alphabet Σ and output alphabet Γ is equivalent to a finite state automaton with alphabet $\Sigma \times \Gamma$.

Proof. In essence, a transducer $\mathcal{T} = (Q, \Sigma, \Gamma, q_0, T, \circ, *)$ becomes the automaton $\mathcal{A} = (Q, \Sigma\times\Gamma, q_0, T, \circ')$, where $q \circ' (a, b) = q \circ a$ if $q * a = b$, and $q \circ' (a, b) = \perp$ otherwise, for all $q \in Q$, $(a, b) \in \Sigma \times \Gamma$. One can show that the language accepted by $\mathcal{A}$ is actually the relation of the function realized by $\mathcal{T}$.

This map from transducers into automata can be seen to be injective, hence we think of the image of a transducer under this map as its automaton representation. △

Therefore the notions of isomorphism, equivalence, acyclic properties and minimization theory and algorithms can be applied to transducers if in a slightly modified manner from their automata counterparts.

To state some of these explicitly: two transducers are isomorphic if they are identical up to a relabeling of the states. They are equivalent if they realize the same function. A transducer is acyclic if no state has a sequence of transition that leads back to it. Every transducer has a unique minimal transducer realizing the same function. The algorithms for testing equivalence, making a transducer connected and minimizing acyclic transducers carry over from the automata case with the same complexity bounds.

Chapter 2

Free band elements via minimal transducers

2.1 The word problem in the free band

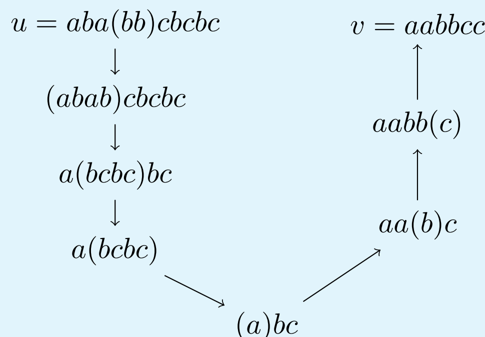
Recall from Example 1.1.20, that the variety of bands is the collection of all semigroups S satisfying $s^2 = s \, \forall s \in S$. The *free band* generated by the set A, denoted $\mathrm{FB}(A)$, is in some sense the largest possible band generated by A. It has the property that every other band that is generated by $|A|$ elements is isomorphic to a quotient of $\mathrm{FB}(A)$. Therefore studying the word problem in the free band can yield important insights into solving the word problem for bands in general.

From Definition 1.1.21 it follows that $\mathrm{FB}(A)$ has the infinite presentation $\langle A \,|\, s^2 = s, \, \forall \, s \in A^+ \rangle$. We let $\beta = \{(s^2, s) : s \in A^+\}^\sharp$ be the least congruence determined by the relations of the free band presentation.

In particular, two words $u, v \in A^+$ are equivalent in the free band $u \sim_\beta v$ if and only if there is a sequence of steps transforming u into v, where each step consists of either replacing a subword s with s^2 (inserting a repetition) or doing the opposite, taking a subword s^2 and replacing it with s (removing a repetition).

In accordance with Definition 1.2.6, the word problem in the free band asks for an algorithm deciding whether $u \sim_\beta v$ or not.

Example 2.1.1. Consider $\mathrm{FB}(\{a, b, c\})$ and $u = ababbcbcbc, v = aabbcc$. Then we can show $u \sim_\beta v$ by rewriting u into v using the repetition insertion and removal steps as described earlier



Here we have enclosed the subword that we modify at each step with parentheses. And so $u \sim_\beta v$

The method we used in Example 2.1.1 is quite simple - we first removed all possible repetitions from the word u until we got the word abc with no repetitions and then started repeating subwords of abc until we got v. One might try to generalize and ask, does a similar method always work? If it did

work then it would suggest a particularly simple normal form for elements of the free band — one with all the possible repetitions removed.

Sadly, this naive method doesn't work, as we can show that $u = abcdabdcab, v = abcdcab$ are equivalent, however neither of them has a repetition. So in particular, any sequence of transformations taking u into v must start with the introduction of a repetition, which lengthens u.

To help us tackle this problem we introduce some notation:

Definition 2.1.2. Let $w = w_1 w_2 \dots w_n \in A^+$ with $w_i \in A$ for all i, then define the *content* of w $\texttt{cont}(w) = \{w_i : i \in \{1, \dots n\}\}$ to be the set of letters occurring in w.

Define $\texttt{pref}(w) = w_1 w_2 \dots w_j$ to be the longest prefix of w that has one less unique letter than w, i.e. such that $|\texttt{cont}(w_1 w_2 \dots w_j)| = |\texttt{cont}(w)| - 1$. With this in mind let $\texttt{ltof}(w) = w_{j+1}$ to be the letter immediately after $\texttt{pref}(w)$, where j is as in the previous sentance. We call $\texttt{pref}(w)$ *the prefix* of w and $\texttt{ltof}(w)$ the *last letter to occur first*, since when reading w left to right, every other letter of $\texttt{cont}(w)$ occurs strictly before $\texttt{ltof}(w)$.

Dually, we define $\texttt{suff}(w) = w_j \dots w_n$ to be the longest suffix of w such that $|\texttt{cont}(w_j \dots w_n)| = |\texttt{cont}(w)| - 1$. And similarly $\texttt{ftol}(w) = w_{j-1}$ to be the letter immediately preceding $\texttt{suff}(w)$. Call $\texttt{suff}(w)$ *the suffix* of w and $\texttt{ftol}(w)$ the *first letter to occur last*.

Note that for the single letter word $a \in A$, $\texttt{pref}(a) = \texttt{suff}(a) = \varepsilon$ is empty.

Example 2.1.3. Lets calculate these values for $w = ababbbcbcbc$,

- $\texttt{cont}(w) = \{a, b, c\}$.
- $\texttt{pref}(w) = ababbb$, since $\texttt{cont}(ababbb) = \{a, b\}$ has one less letter than $\texttt{cont}(w)$, and adding in the next letter gives the prefix $aababbbc$, which now has 3 distinct letters.
- $\texttt{ltof}(w) = c$, since its the next letter after the prefix $ababbb$ in $ababbbcbcbc$.
- $\texttt{suff}(w) = bbbcbcbc$.
- $\texttt{ftol}(w) = a$.

Note that if $u \sim_\beta v$, then $\texttt{cont}(u) = \texttt{cont}(v)$, since repeating a subword cant introduce a letter that wasn't already there, and every letter removed by removing a repetition persists in the part of the repetition that wasn't removed. Similarly we can show that $\texttt{cont}(\texttt{pref}(u))$ and $\texttt{cont}(\texttt{suff}(u))$ are invariants, and therefore $\texttt{ltof}(u), \texttt{ftol}(u)$ are also invariants. It turns out that we can go further with these quantities.

Green and Rees [7] give a very elegant solution to the word problem in the free band in terms of them with the following theorem, here we present it as it appears in [8]:

Theorem 2.1.4 ([8] Lemma 4.5.1, the Green-Rees theorem)**.** For all $v, w \in A^+$, $v \sim_\beta w$ if and only if all of the following hold:

- $\texttt{cont}(v) = \texttt{cont}(w)$
- $\texttt{pref}(v) \sim_\beta \texttt{pref}(w)$
- $\texttt{ltof}(v) = \texttt{ltof}(w)$
- $\texttt{ftol}(v) = \texttt{ftol}(w)$
- $\texttt{suff}(v) \sim_\beta \texttt{suff}(w)$

It may seem that this theorem only makes the word problem worse, since suddenly to figure out if $u \sim_\beta v$, we now need to figure out two more equivalences for $\texttt{pref}(u) \sim_\beta \texttt{pref}(v)$ and $\texttt{suff}(u) \sim_\beta \texttt{suff}(v)$.

However the key insight is that the content of the suffix and prefix are strictly smaller, so eventually the suffix and prefix will be empty and then out calculation "bottoms-out" so to speak.

We use this idea to derive Algorithm 3.

Algorithm 3: `FBEquivalent`

Data : $u, v \in A^+$
Result : True if $u \sim_\beta v$, False otherwise

```
1  if u = v = ε then
2  |   return True
3  else if cont(u) ≠ cont(v) then
4  |   return False
5  else if ftol(u) ≠ ftol(v) then
6  |   return False
7  else if ltof(u) ≠ ltof(v) then
8  |   return False
9  else if not FBEquivalent(pref(u), pref(v)) then
10 |   return False
11 else if not FBEquivalent(suff(u), suff(v)) then
12 |   return False
13 else
14 |   return True
```

Theorem 2.1.5. Algorithm 3 is correct and runs in time $\mathcal{O}\left(2^{|A|} \cdot (|u| + |v|)\right)$.

Proof. Correctness is clear from the Green-Rees theorem.

For the runtime estimate, note that $\texttt{cont}(w), \texttt{pref}(w), \texttt{ltof}(w), \texttt{ftol}(w), \texttt{suff}(w)$ can all be calculated in time $\mathcal{O}(n)$, where $n = |w|$, by a linear scan through the word w. These then contribute the $\mathcal{O}(|u| + |v|)$ factor to the time bound.

Furthermore, at every invocation of `FBEquivalent`, at worst we need to recursively calculate it two more times for inputs whose content is one less than the original, up until the point where the input has empty content. This creates a full binary tree of height at most $|A|$, therefore giving the exponential part of the bound. △

The Green-Rees theorem can be slightly simplified:

Remark 2.1.6. In fact the condition $\texttt{cont}(v) = \texttt{cont}(w)$ may be dropped from Theorem 2.1.4. △

Proof. To show this all we have to do is show that $\texttt{pref}(u) \sim_\beta \texttt{pref}(v)$, $\texttt{ltof}(u) = \texttt{ltof}(v)$, $\texttt{ftol}(u) = \texttt{ftol}(v)$ and $\texttt{suff}(u) \sim_\beta \texttt{suff}(v)$ together imply that $\texttt{cont}(u) = \texttt{cont}(v)$.

Well, since $\texttt{pref}(u) \sim_\beta \texttt{pref}(v)$, then $\texttt{cont}(\texttt{pref}(u)) = \texttt{cont}(\texttt{pref}(v))$. But $\texttt{cont}(u) = \texttt{cont}(\texttt{pref}(u)) \cup \{\texttt{ltof}(u)\} = \texttt{cont}(\texttt{pref}(v)) \cup \{\texttt{ltof}(v)\} = \texttt{cont}(v)$. △

We end the section by introducing some abuse of notation: for $s \in \mathrm{FB}(A)$ we sometimes refer to $\texttt{cont}(s), \texttt{pref}(s), \texttt{ltof}(s), \texttt{ftol}(s), \texttt{suff}(s)$. Due to the Green-Rees Theorem, if we take any word $w \in A^+$ representing s, then if we define $\texttt{cont}(s) = \texttt{cont}(w), \texttt{ltof}(s) = \texttt{ltof}(w), \texttt{ftol}(s) = \texttt{ftol}(w)$ and $\texttt{pref}(s) = \texttt{pref}(w)/\beta, \texttt{suff}(s) = \texttt{suff}(w)/\beta$, then these are indeed well defined.

2.2 From words to trees to transducers and back

We will now adapt the result of the Green-Rees theorem to actually generate a complete invariant for elements of the free band.

Before we do so, we first define "direction-agnostic" versions of the first to occur last, last to occur first and prefix, suffix functions.

Definition 2.2.1. Define $\texttt{atob}_0 = \texttt{ftol}$ and $\texttt{atob}_1 = \texttt{ltof}$. Similarly, let $\texttt{affx}_0 = \texttt{pref}$ and $\texttt{affx}_1 = \texttt{suff}$.

We can extend the definition of affx to allow subscripts from $\{0,1\}^*$ as follows: let $\texttt{affx}_\varepsilon(w) = w$ for all $w \in A^+$ and for every $x \in \{0,1\}^*$, $x = x_1x_2\dots x_n$ with $x_k \in \{0,1\}$, define $\texttt{affx}_x$ recursively as

$$\texttt{affx}_x(w) = \texttt{affx}_{x_n}(\texttt{affx}_{x_1\dots x_{n-1}}(w))$$

Similarly, we also extend the definition of atob by

$$\texttt{atob}_x(w) = \texttt{atob}_{x_n}(\texttt{affx}_{x_1x_2\dots x_{n-1}}(w))$$

We also define $\texttt{atob}_\varepsilon(w) = \perp$ to be undefined. One can note that $\texttt{atob}_x(w) = \texttt{atob}_x(\texttt{affx}_\varepsilon(w))$ for $x \in \{0,1\}$, so this extension is reasonable.

The extensions of affx and atob essentially allow us to traverse the branches of the recursion in Algorithm 3 from earlier, and the binary string x enumerates the branch we take.

Once we "bottom out" in the recursive algorithm when applying the Green-Rees theorem recursively, we are left with a collection of equalities that were tested, and these equalities are necessarily a complete invariant. The extended versions of affx and atob will help us formalize this invariant as follows:

Definition 2.2.2. Let $w \in A^+$, $k = |\texttt{cont}(w)|$. We define $g_w : \{0,1\}^k \to A^k$ by

$$g_w(x_1x_2\dots x_k) = \texttt{atob}_{x_1}(w)\texttt{atob}_{x_1x_2}(w)\texttt{atob}_{x_1x_2x_3}(w)\cdots\texttt{atob}_{x_1x_2\dots x_k}(w)$$

Theorem 2.2.3. Let $u, v \in A^+$, then $u \sim_\beta v$ if and only if $g_u = g_v$.

Proof. First we show some small technical properties of atob, affx and g. Note that for $n \geq 1$, $x = x_1x_2\dots x_n \in \{0,1\}^+$ and $w \in A^+$,

$$\begin{aligned}\texttt{affx}_x(w) = \texttt{affx}_{x_n}(\texttt{affx}_{x_1x_2\dots x_{n-1}}(w)) = \\ =\texttt{affx}_{x_n}(\texttt{affx}_{x_{n-1}}(\dots(\texttt{affx}_{x_1}(w))\dots)) = \texttt{affx}_{x_2x_3\dots x_n}(\texttt{affx}_{x_1}(w))\end{aligned}$$

Therefore if $n \geq 2$,

$$\begin{aligned}\texttt{atob}_x(w) = \texttt{atob}_{x_n}(\texttt{affx}_{x_1\dots x_{n-1}}(w)) = \\ =\texttt{atob}_{x_n}(\texttt{affx}_{x_2x_3\dots x_{n-1}}(\texttt{affx}_{x_1}(w))) = \texttt{atob}_{x_2\dots x_n}(\texttt{affx}_{x_1}(w))\end{aligned}$$

And so we have that

$$\begin{aligned}g_w(x_1\dots x_k) = \texttt{atob}_{x_1}(w)\texttt{atob}_{x_1x_2}(w)\dots\texttt{atob}_{x_1x_2\dots x_k}(w) = \\ =\texttt{atob}_{x_1}(w)\texttt{atob}_{x_2}(\texttt{affx}_{x_1}(w))\dots\texttt{atob}_{x_2\dots x_k}(\texttt{affx}_{x_1}(w)) = \texttt{atob}_{x_1}(w)g_{\texttt{affx}_{x_1}(w)}(x_2\dots x_k)\end{aligned}$$

We now prove the statement by induction on $|\texttt{cont}(u)|$. Clearly if $|\texttt{cont}(u)| \neq |\texttt{cont}(v)|$, then $u \not\sim_\beta v$ by the Green-Rees theorem, and $g_u \neq g_v$ since they have different domains. So we assume that $|\texttt{cont}(u)| = |\texttt{cont}(v)|$ for the rest of the proof.

As an inductive base, if $|\mathtt{cont}(u)| = |\mathtt{cont}(v)| = 1$, then $\mathtt{affx}_x(u) = \mathtt{affx}_x(v) = \varepsilon$ for all $x \in \{0,1\}$, $\mathtt{atob}_0(u) = \mathtt{atob}_1(u)$ and $\mathtt{atob}_0(v) = \mathtt{atob}_1(v)$ so, from the Green-Rees theorem, $u \sim_\beta v$ if and only if $\mathtt{atob}_0(u) = \mathtt{atob}_0(v)$.

Furthermore, if $|\mathtt{cont}(w)| = 1$, then g_w is defined by $g_w(0) = \mathtt{atob}_0(w), g_w(1) = \mathtt{atob}_1(w)$, and $\mathtt{atob}_0(w) = \mathtt{atob}_1(w)$ if $|\mathtt{cont}(w)| = 1$. So $g_u = g_v$ if and only if $\mathtt{atob}_0(u) = \mathtt{atob}_0(v)$. This establishes the base case.

Now assume that we have shown that the statement holds for all u, v with $|\mathtt{cont}(u)|, |\mathtt{cont}(v)| < k$. Then for u, v with $|\mathtt{cont}(u)| = |\mathtt{cont}(v)| = k$, by Green-Rees theorem we have that $u \sim_\beta v$ if and only if $\mathtt{atob}_x(u) = \mathtt{atob}_x(v)$ and $\mathtt{affx}_x(u) \sim_\beta \mathtt{affx}_x(v)$ for all $x \in \{0,1\}$ (note that we dropped the condition $\mathtt{cont}(u) = \mathtt{cont}(v)$ as per Remark 2.1.6).

On the other hand, we know that $g_w(x_1 \dots x_k) = \mathtt{atob}_{x_1}(w) g_{\mathtt{affx}_{x_1}(w)}(x_2 \dots x_k)$. Therefore $g_u = g_v$ if and only if $\mathtt{atob}_{x_1}(u) = \mathtt{atob}_{x_1}(v)$ and $g_{\mathtt{affx}_{x_1}(u)}(x_2 \dots x_k) = g_{\mathtt{affx}_{x_1}(v)}(x_2 \dots x_k)$ for all $x_1, \dots x_k \in \{0,1\}$. But this condition is equivalent to $\mathtt{atob}_x(u) = \mathtt{atob}_x(v)$ and $g_{\mathtt{affx}_x(u)} = g_{\mathtt{affx}_x(v)}$ for all $x \in \{0,1\}$. But now, from the inductive hypothesis, since $\mathtt{affx}_x(u), \mathtt{affx}_x(v)$ have strictly smaller content than u and v, then it follows that $g_{\mathtt{affx}_x(u)} = g_{\mathtt{affx}_x(v)}$ if and only if $\mathtt{affx}_x(u) \sim_\beta \mathtt{affx}_x(v)$. But then putting it all back together we get that $g_u = g_v$ if and only if $\mathtt{atob}_x(u) = \mathtt{atob}_x(v)$ and $\mathtt{affx}_x(u) \sim_\beta \mathtt{affx}_x(v)$ for all $x \in \{0,1\}$.

This completes the induction and therefore the proof. △

As it turns out, g_w can be realized by a deterministic acyclic synchronous transducer.

For this we introduce the following notation: for a set X and integer k, let $X^{\leq k} = \{x \in X^* : |x| \leq k\}$ denote the set of all word over X of length no more than k.

With this in mind, we construct gives a transducer computing g_w which is in some sense "tree-like":

Definition 2.2.4. Given $w \in A^+$, let $k = |\mathtt{cont}(w)|$ and define $G_w = (Q, \Sigma, \Gamma, q_0, T, \circ, *)$ be a transducer with $Q = \{0,1\}^{\leq k}$, $\Sigma = \{0,1\}$, $\Gamma = A$, $q_0 = \varepsilon$, $T = \{0,1\}^k$ and state and letter transition functions given respectively by:

$$q \circ x = qx \qquad\qquad q * x = \mathtt{atob}_x(\mathtt{affx}_q(w))$$

Theorem 2.2.5. For all $w \in A^+$, the function realized by G_w is exactly g_w. Furthermore, $w \sim_\beta v$ if and only if G_w, G_v are isomorphic as transducers.

Proof. Note that the only path from the initial state $q_0 = \varepsilon$ to a terminal state $x = x_1 x_2 \dots x_k \in \{0,1\}^k$ is by following the transition labeled by x_1 from ε to x_1, then the one labeled x_2 from x_1 to $x_1 x_2$, and so on until we follow the transition labeled x_k to $x_1 x_2 \dots x_k$. The word emitted by these transitions is

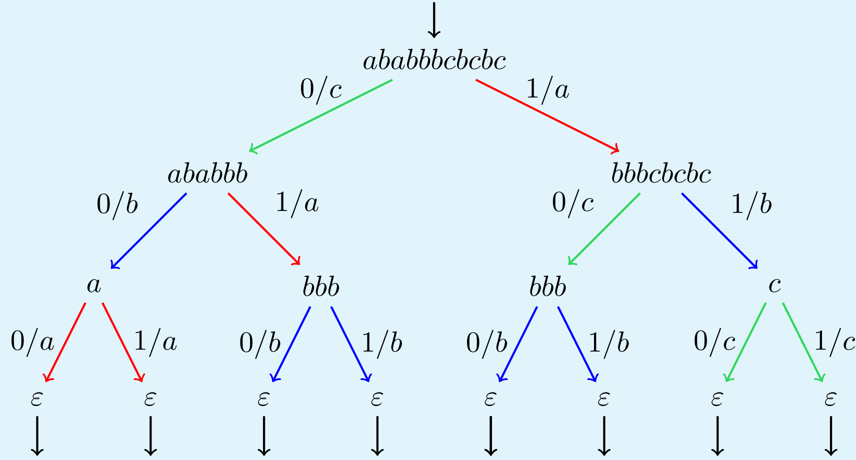
$$\begin{aligned}
&\varepsilon * x_1 \cdot x_1 * x_2 \cdot x_1 x_2 * x_3 \dots x_1 x_2 \dots x_{k-1} * x_k = \\
=&\mathtt{atob}_{x_1}(\mathtt{affx}_\varepsilon(w))\mathtt{atob}_{x_2}(\mathtt{affx}_{x_1}(w)) \dots \mathtt{atob}_{x_k}(\mathtt{affx}_{x_1 \dots x_{k-1}}) = \\
=&\mathtt{atob}_{x_1}(w)\mathtt{atob}_{x_1 x_2}(w) \dots \mathtt{atob}_{x_1 \dots x_k}(w) = g_w(x_1 \dots x_k)
\end{aligned}$$

This completes the proof. △

Therefore G_w is also a complete invariant for the word problem in the free band. By abuse of notation we will also sometimes talk about g_s, G_s for $s \in \mathrm{FB}(A)$, where in this case $g_s = g_w, G_s = G_w$ for some word representative $w \in A^+$ of s.

Example 2.2.6. Consider the word $w = ababbbcbcbc$ as before. We will not write out g_w in full here, but as an example we can see that $g_w(000) = cba$, $g_w(010) = cab$, $g_w(111) = abc$.

The transducer we get is isomorphic to the following, where we choose the state labels to be the corresponding affixes of the word, and the edges are colored depending on the output letter:



We now seek a reverse process — turning a transducer representing a free band element into a word representative:

Definition 2.2.7. Given a transducer $\mathcal{T} = (Q, \Sigma, \Gamma, q_0, T, \circ, *)$ and a state $q \in Q$, define the q-flatted transducer word $\texttt{word}_{\mathcal{T}}(q)$ to be

$$\texttt{word}_{\mathcal{T}}(q) = \begin{cases} \varepsilon & \text{if } q \in T \\ \texttt{word}_{\mathcal{T}}(q \circ 0) \cdot (q * 0) \cdot (q * 1) \cdot \texttt{word}_{\mathcal{T}}(q \circ 1) \end{cases}$$

Theorem 2.2.8. For all $s \in \mathrm{FB}(A)$, if $\mathcal{T}$ is a transducer with start state q_0 realizing g_s, then $\texttt{word}_{\mathcal{T}}(q_0)$ is a word representative of s.

Proof. We proceed by induction on $|\texttt{cont}(s)|$.

If $|\texttt{cont}(s)| = 1$, then $q_0 \circ 0, q_0 \circ 1 \in T$, since it realizes $g_s : \{0, 1\} \to A$. Therefore $\texttt{word}_{\mathcal{T}}(q_0) = q_0 * 0 \cdot q_0 * 1$. But we know that $q_0 \circ 0 = g_s(0) = \texttt{atob}_0(s) = \texttt{atob}_1(s) = g_s(1) = q_1 \circ 1$, so indeed $\texttt{word}_{\mathcal{T}}(q_0) = \texttt{atob}_0(s)^2$ is a word representative for s.

Now assume the statement hold for all $s \in \mathrm{FB}(A)$ with content strictly less than k. Then for $s \in \mathrm{FB}(A)$ with $|\texttt{cont}(s)| = k$ we have that $g_s(x_1 \ldots x_k) = \texttt{atob}_{x_1}(s) g_{\texttt{affx}_{x_1}(s)}(x_2 \ldots x_k)$, therefore $q_0 * x_1 = \texttt{atob}_{x_1}(s)$, since this is the first letter for all inputs starting with x_1.

Furthermore, by the same reasoning we can show that the function realized by $\mathcal{T}$ with initial vertex at $q_0 \circ x_1$ must be $g_{\texttt{affx}_{x_1}}$. But then by the inductive hypothesis, since $\texttt{affx}_{x_1}(s)$ has strictly smaller content, we get that $\texttt{word}_{\mathcal{T}}(q_0 \circ x_1)$ is a representative of $\texttt{affx}_{x_1}(s)$.

Putting this all together, we get that $\texttt{atob}_x(\texttt{word}_{\mathcal{T}}(q_0)) = q_0 * x = \texttt{atob}_x(s)$, and $\texttt{affx}_x(\texttt{word}_{\mathcal{T}}) = \texttt{word}_{\mathcal{T}}(q_0 \circ x) = \texttt{affx}_x(s)$. So by the Green-Rees theorem, $\texttt{word}_{\mathcal{T}}(q_0)$ is indeed a word representative for s. △

Example 2.2.9. Consider the transducer $\mathcal{T}$ from Example 2.2.6, then the flattening of this trans-

ducer we get

$$\texttt{word}_{\mathcal{T}}(q_0) = \texttt{word}_{\mathcal{T}}(q_0 \circ 0) \cdot ca \cdot \texttt{word}_{\mathcal{T}}(q_0 \circ 0) =$$
$$= \texttt{word}_{\mathcal{T}}(q_0 \circ 01) \cdot ba \cdot \texttt{word}_{\mathcal{T}}(q_0 \circ 01) \cdot ca \cdot \texttt{word}_{\mathcal{T}}(q_0 \circ 10) \cdot cb \cdot \texttt{word}_{\mathcal{T}}(q_0 \circ 11) = aababbcabbcbcc$$

2.3 Minimal transducers are small

Note that in the previous section we constructed for each element of the free band a complete invariant in the form of a tree-like transducer. This transducer is rather large, however, — it has $2^{|A|+1} - 1$ states regardless of the element $s \in \mathrm{FB}(A)$ that its representing. In comparison to representing elements of $\mathrm{FB}(A)$ by words in A^+, this can sometimes be an exponential blowup, for example if $A = \{a_1, a_2, \ldots, a_n\}$, then the word $a_1 a_2 \ldots a_n$ is a size $|A|$ word representative of its equivalence class, yet its transducer has $2^{|A|+1} - 1$ states.

However, recall that transducers can be minimized, and for every transducer there exists a unique minimal transducer computing the same function. Since every $s \in \mathrm{FB}(A)$ is uniquely determined by g_s, and there exists a unique minimal transducer M_s computing g_s, then the minimal transducer is also easily seen to be a complete invariant for s.

The main motivation for studying the transducer representation comes from the following theorem:

Theorem 2.3.1 (Main theorem)**.** Let $s \in \mathrm{FB}(A)$ and w be the shortest word representing s. Then M_s, the minimal transducer computing g_s, has no more than $|w| \cdot (|A| + 1)$ states.

Theorem 2.3.1 is great news since, as far as we know, there is no efficient way of computing a shortest word representative of an element of the free band, but as we know, the minimal transducer can be computed in linear time proportional to the number of transitions (since the input alphabet is fixed to $\{0, 1\}$ in our case, then this means we can minimize in time proportional to the number of states).

Before we prove Theorem 2.3.1 we will establish a method of extending a transducer representing $s \in \mathrm{FB}(A)$ into a transducer representing sa for some $a \in A$. This will then give us a method of building up a transducer "letter by letter", which is crucial in establishing the state bound.

The following Lemma gives us an idea of how the important parameters of s change when we right multiply it by a generator:

Lemma 2.3.2. Let $s \in \mathrm{FB}(A)$ and $a \in A$, then

$$\texttt{pref}(sa) = \begin{cases} s & \text{if } a \notin \texttt{cont}(s) \\ \texttt{pref}(s) & \text{otherwise} \end{cases}$$

$$\texttt{ltof}(sa) = \begin{cases} a & \text{if } a \notin \texttt{cont}(s) \\ \texttt{ltof}(s) & \text{otherwise} \end{cases}$$

$$\texttt{suff}(sa) = \begin{cases} \texttt{suff}^2(s)a & \text{if } a = \texttt{ftol}(s) \\ \texttt{suff}(s)a & \text{otherwise} \end{cases}$$

$$\texttt{ftol}(sa) = \begin{cases} \texttt{ftol}(\texttt{suff}(s)) & \text{if } a = \texttt{ftol}(s) \\ \texttt{ftol}(s) & \text{otherwise} \end{cases}$$

Proof. Note that it suffice to prove the theorem for a single word representative of s. Let $w \in A^+$ be a word representative for s.

We proceed by casework. First we consider `pref` and `ltof`:

- If $a \notin \texttt{cont}(w)$, then $|\texttt{cont}(w)| = |\texttt{cont}(wa)| - 1$, furthermore clearly w is the longest such prefix of wa. Therefore $\texttt{pref}(wa) = w$, and consequently $\texttt{ltof}(wa) = a$
- If $a \in \texttt{cont}(w)$, then $|\texttt{cont}(wa)| = |\texttt{cont}(w)| = |\texttt{cont}(\texttt{pref}(w))| + 1$, and $\texttt{pref}(w)$ is also a prefix of wa. Furthermore, $\texttt{pref}(w)$ must be the longest such prefix in wa, since otherwise we can find a longer prefix in w, a contradiction. So $\texttt{pref}(wa) = \texttt{pref}(w)$ and therefore $\texttt{ltof}(wa) = \texttt{ltof}(w)$.

Now consider `suff` and `ftol`:

- If $a = \texttt{ftol}(w)$, then $\texttt{cont}(\texttt{suff}(w)a) = \texttt{cont}(\texttt{suff}(w)) \cup \{a\} = \texttt{cont}(\texttt{suff}(w)) \cup \{\texttt{ftol}(w)\} = \texttt{cont}(w) = \texttt{cont}(wa)$. But then, note that $|\texttt{cont}(\texttt{suff}^2(w)a)| = |\texttt{cont}(w)| - 1 = |\texttt{cont}(wa) - 1|$, and $\texttt{suff}^2(w)a$ is a suffix of wa. Furthermore, it must be the longest such suffix. Otherwise, we could find a suffix longer than $\texttt{suff}^2(w)$ with size of content equal to $|\texttt{cont}(w)| - 2 = |\texttt{cont}(\texttt{suff}(w))| - 1$, but this contradicts the maximality of $\texttt{suff}^2(w)$ as the suffix of $\texttt{suff}(w)$. Therefore $\texttt{suff}(wa) = \texttt{suff}^2(s)a$, and therefore it follows that $\texttt{ftol}(wa) = \texttt{ftol}(\texttt{suff}(w))$.
- If $a \neq \texttt{ftol}(w)$, then $\texttt{ftol}(w) \notin \texttt{cont}(\texttt{suff}(w)a)$, as $\texttt{ftol}(w) \notin \texttt{cont}(\texttt{suff}(w))$ by definition. Therefore $|\texttt{cont}(\texttt{suff}(w)a)| = |\texttt{cont}(wa)| - 1$, and $\texttt{suff}(w)a$ is a suffix of wa. Furthermore it is the largest such suffix as otherwise we could find a larger suffix of w by removing the final letter a. So $\texttt{suff}(wa) = \texttt{suff}(w)a$ which means that $\texttt{ftol}(wa) = \texttt{ftol}(w)$.

△

We now use the above lemma in a constructive manner, to modify a transducer realizing g_w into a transducer realizing g_{wa}. This happens by adding a vertical strip of states and linking them to previous states in a specific manner that mirrors the properties of wa given in Lemma 2.3.2.

Definition 2.3.3. Let $s \in \mathrm{FB}(A)$ with $|\texttt{cont}(s)| = k$, $\mathcal{T} = (Q, \{0,1\}, A, q_0, T, \circ_1, *_1)$ be a transducer realizing g_s and $a \in A$ be any generator.

Then the *right-action* of a on $\mathcal{T}$ is defined as the transducer $\mathcal{T}a = (Q \cup R, \{0,1\}, A, r_0, T, \circ_2, *_2)$, where $R = \{r_0, r_1, \ldots, r_k\}$ if $a \notin \texttt{cont}(s)$ and $R = \{r_0, r_1, \ldots, r_{k-1}\}$ otherwise. For convenience, let $q_i = q_0 \circ_1 1^i$ for all $i \in \{1, \ldots, k\}$ (note that q_k is then a terminal state).

We define $\circ_2, *_2$ to act on Q in the same way that $\circ_1, *_1$ do, i.e. $\forall\ q \in Q, x \in \{0,1\}$, we let $q \circ_2 x = q \circ_1 x, q *_1 x = q *_2 x$.

Now we consider two cases:

- If $a \notin \texttt{cont}(s)$, then let $r_k \circ_2 x = q_k$ lead to a terminal state, and $r_k *_2 x = a$ for all $x \in \{0,1\}$. For $i \in \{0, \ldots, k-1\}$ define

$$r_i \circ_2 0 = q_i \qquad r_i \circ_2 1 = r_{i+1}$$
$$r_i *_2 0 = a \qquad r_i *_2 1 = q_i *_1 1$$

- If $a \in \texttt{cont}(s)$, then let $r_{k-1} \circ_2 x = q_{k-1}$ lead to a terminal state, and $r_{k-1} *_2 x = a$ for all $x \in \{0,1\}$.

  Let j be such that $q_j *_1 1 = a$ (as $a \in \texttt{cont}(s)$, and $\mathcal{T}$ realizes g_s, such a j must exist and furthermore is unique). For $i \in \{0, \ldots, j-1\}$ define

$$r_i \circ_2 0 = q_i \circ_1 0 \qquad r_i \circ_2 1 = r_{i+1}$$
$$r_i *_2 0 = q_i *_1 0 \qquad r_i *_2 1 = q_i *_1 1$$

For $i = j$ define

$$r_j \circ_2 0 = q_j \circ_1 0 \qquad r_j \circ_2 1 = r_{j+1}$$
$$r_j *_2 0 = q_j *_1 0 \qquad r_j *_2 1 = q_{j+1} *_1 1$$

For $i \in \{j+1, \dots, k-2\}$ define

$$r_i \circ_2 0 = q_{i+1} \qquad r_i \circ_2 1 = r_{i+1}$$
$$r_i *_2 0 = a \qquad r_i *_2 1 = q_{i+1} *_1 1$$

Theorem 2.3.4. Let $s \in \mathrm{FB}(A)$, $\mathcal{T}$ and $a \in A$ be as in Definition 2.3.3. Then $\mathcal{T}a$ realizes g_{sa}.

Proof. The proof proceeds by establishing that the induced subtransducer of $\mathcal{T}a$ rooted at r_i (that is, the transducer $\mathcal{T}a$ with initial state changed to be equal to r_i instead) realizes the function $g_{\mathtt{suff}^i(wa)}$, where $\mathtt{suff}^0(wa) = wa$.

Before we do so, note that the induced subtransducer of $\mathcal{T}$ rooted at q_i realizes the function $g_{\mathtt{suff}^i(w)}$, this is immediate since we showed that $g_w(x_1 \dots x_k) = \mathtt{atob}_{x_1} g_{\mathtt{affx}_{x_1}(w)}(x_2 \dots x_k)$ in the proof of Theorem 2.2.3.

We now proceed by induction.

Consider two cases:

- $a \notin \mathtt{cont}(w)$. Then also $a \notin \mathtt{cont}(\mathtt{suff}^i(w))$ for all $i \in \{0, \dots, k\}$, and $a \neq \mathtt{ftol}(\mathtt{suff}^i(w))$ for all $i \in \{0, \dots, k-1\}$. Therefore from Lemma 2.3.2 it follows that $\mathtt{suff}^i(wa) = \mathtt{suff}^{i-1}(\mathtt{suff}(wa)) = \mathtt{suff}^{i-1}(\mathtt{suff}(w)a) = \dots = \mathtt{suff}^i(w)a$ for all $i \in \{0, \dots, k\}$.

  For our inductive base we consider $i = k$. In this case, we can see that $\mathtt{suff}^k(w)a = \varepsilon a = a$. And we have that $r_k \circ_2 x = q_k$ which is a terminal state, and $r_k *_2 x = a$ for all $x \in \{0, 1\}$, so the function realized by the induced subtransducer is indeed g_a.

  Now assume that the statement holds for all values strictly larger than i, i.e. that the subtransducer at r_l realizes $g_{\mathtt{suff}^l(wa)}$ for all $l > i$.

  Then by applying the Lemma again we have that

  – $\mathtt{pref}(\mathtt{suff}^i(w)a) = \mathtt{suff}^i(w)$

  – $\mathtt{ltof}(\mathtt{suff}^i(w)a) = a$

  – $\mathtt{suff}(\mathtt{suff}^i(w)a) = \mathtt{suff}^{i+1}(w)a$

  – $\mathtt{ftol}(\mathtt{suff}^i(w)a) = \mathtt{ftol}(\mathtt{suff}^i(w))$

  but now note that $r_i *_2 0 = a = \mathtt{ltof}(\mathtt{suff}^i(w)a)$, $r_i *_2 1 = q_i *_1 1 = \mathtt{ftol}(\mathtt{suff}^i(w))$ since q_i realizes $\mathtt{suff}^i(w)$. Furthermore $r_i \circ_2 0 = q_i$, and $r_i \circ_2 1 = r_{i+1}$, which according to the inductive hypothesis induces $g_{\mathtt{suff}^{i+1}(wa)}$. But then if f is the function realized by the subtransducer tooted at r_i, by putting this all together we get that

  $$f(0x_{i+1} \dots x_k) = a g_{\mathtt{suff}^i(w)}(x_{i+1} \dots x_k) = g_{\mathtt{suff}^i(w)a}(0x_{i+1} \dots x_k) = g_{\mathtt{suff}^i(wa)}(0x_{i+1} \dots x_k)$$
  $$f(1x_{i+1} \dots x_k) = \mathtt{ftol}(\mathtt{suff}^i(w)) g_{\mathtt{suff}^{i+1}(wa)}(x_{i+1} \dots x_k) = g_{\mathtt{suff}^i(wa)}(1x_{i+1} \dots x_n)$$

  For all $x_{i+1}, \dots, x_n \in \{0, 1\}$. But then $f = g_{\mathtt{suff}^i(wa)}$ as required.

  This establishes the induction and so the induced subtransducer rooted at r_0, which is simply the whole of $\mathcal{T}a$, realizes the function $g_{\mathtt{suff}^0(wa)} = g_{wa}$ as required.

- $a \in \texttt{cont}(w)$. By using the lemma we can show that $\texttt{suff}^i(wa) = \texttt{suff}^i(w)a$ for $i \in \{0, \ldots j\}$ and $\texttt{suff}^i(wa) = \texttt{suff}^{i+1}(w)a$ for $i \in \{j+1, \ldots, k-1\}$.

  We note that for $i \in \{j+1, k-1\}$, we have that $a \notin \texttt{suff}^i(w)$, since $r_j *_2 1 = a$ implies that $\texttt{ftol}(\texttt{suff}^j(w)) = a$, so clearly $a \notin \texttt{suff}^{j+1}(w)$.

  But this means that we can use the proof of the previous case $a \notin \texttt{cont}(w)$ above to show that the subtransducer rooted at r_i realizes the function $g_{\texttt{suff}^i(wa)}$ for all $i \in \{j+1, \ldots, k-1\}$.

  Now consider r_j. Then since $a = \texttt{ftol}(\texttt{suff}^j(w))$, the Lemma tells us that

  – $\texttt{pref}(\texttt{suff}^j(w)a) = \texttt{pref}(\texttt{suff}^j(w))$

  – $\texttt{ltof}(\texttt{suff}^j(w)a) = \texttt{ltof}(\texttt{suff}^j(w))$

  – $\texttt{suff}(\texttt{suff}^j(w)a) = \texttt{suff}^2(\texttt{suff}^j(w))a = \texttt{suff}^{j+2}(w)a = \texttt{suff}^{j+1}(wa)$

  – $\texttt{ftol}(\texttt{suff}^j(w)a) = \texttt{ftol}(\texttt{suff}(\texttt{suff}^j(w))) = \texttt{ftol}(\texttt{suff}^{j+1}(wa))$

  But now note that $r_j *_2 0 = q_j *_1 0 = \texttt{ltof}(\texttt{suff}^j(w))$, $r_j *_2 1 = q_{j+1} *_1 1 = \texttt{ftol}(\texttt{suff}^{j+1}(wa))$. Further, $r_j \circ_2 0 = q_j \circ_1 0$ and $q_j \circ_1 0$ realizes $g_{\texttt{pref}(\texttt{suff}^j(w))}$ by assumption. Finally, $r_j \circ_2 1 = r_{j+1}$ and as we just showed, r_{j+1} realizes $g_{\texttt{suff}^{j+1}(wa)}$. And so in the same manner as before we establish that the subtransducer rooted at r_j realizes $g_{\texttt{suff}^j(wa)}$.

  Finally, we establish $i \in \{0, \ldots, j-1\}$ by reverse induction. r_j serves as the inductive base. We assume that the statement holds for all values larger than i.

  Note now that by the Lemma

  – $\texttt{pref}(\texttt{suff}^i(w)a) = \texttt{pref}(\texttt{suff}^i(w))$

  – $\texttt{ltof}(\texttt{suff}^i(w)a) = \texttt{ltof}(\texttt{suff}^i(w))$

  – $\texttt{suff}(\texttt{suff}^i(w)a) = \texttt{suff}(\texttt{suff}^i(w))a = \texttt{suff}^{i+1}(w)a = \texttt{suff}^{i+1}(wa)$

  – $\texttt{ftol}(\texttt{suff}^i(w)a) = \texttt{ftol}(\texttt{suff}^i(w))$

  But then, since $r_i *_2 0 = q_i *_1 0 = \texttt{ltof}(\texttt{suff}^i(w))$ and $r_i *_2 1 = q_i *_1 1 = \texttt{ftol}(\texttt{suff}^i(w))$. Similarly, $r_i \circ_2 0 = q_i \circ_1 0$, and by assumption $q_i \circ_1 0$ realizes $g_{\texttt{pref}(\texttt{suff}^i(w))}$. Finally $r_i \circ_2 1 = r_{i+1}$, and by the inductive assumption r_{i+1} realizes $g_{\texttt{suff}^{i+1}(wa)}$. Therefore we establish that the subtransducer rooted at r_i realizes $g_{\texttt{suff}^i(wa)}$ as required.

  This finishes the induction.

$\triangle$

Proof of Theorem 2.3.1. For $a \in A$ define the transducer $\mathcal{S}_a$ to be as below

$$\to 1 \xrightarrow[0,1/a]{} 2 \to$$

It is easy to see that $\mathcal{S}_a$ realizes g_a.

Now for $w = w_1 \ldots w_n \in A^+$, consider the transducer gained by repeatedly applying the right action of A to $\mathcal{S}_{w_1}$, i.e. the transducer $\mathcal{U} = (\ldots((\mathcal{S}_{w_1} w_2) w_3) \ldots) w_n$. It realizes $g_{w_1 w_2 \ldots w_n} = g_w$ by repeated application of Theorem 2.3.4.

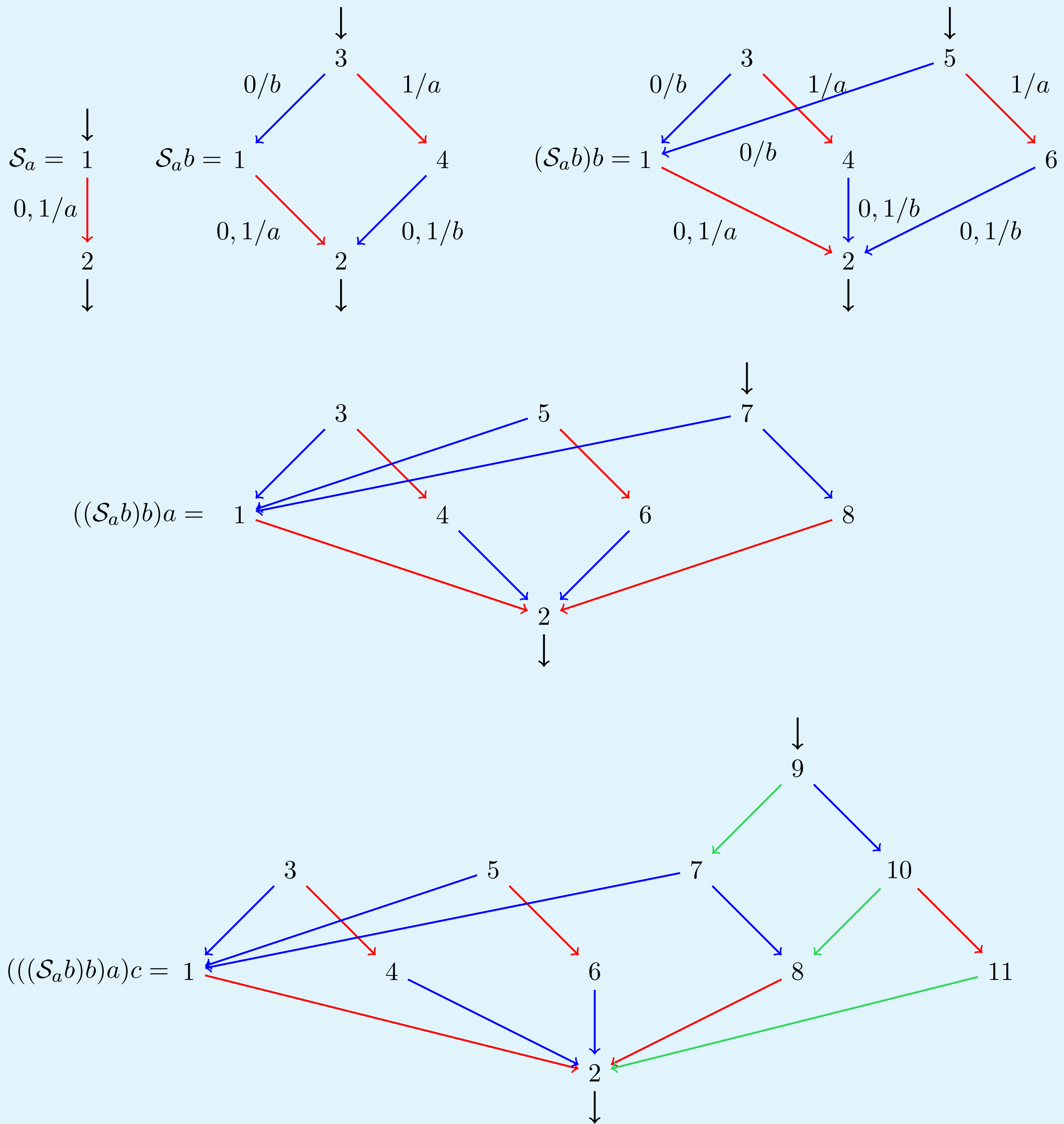
But note that for an arbitrary transducer $\mathcal{T}$ realizing g_u for some $u \in A^+$, the number of states of $\mathcal{T}a$ is related to the number of states of $\mathcal{T}$ by the inequality $|\mathcal{T}a| \leq |\mathcal{T}| + |A| + 1$ by the construction given in Definition 2.3.3, since we introduce at most $|\texttt{cont}(u)| + 1$ new states.

Therefore $\mathcal{U}$ has at most $|w| \cdot (|A| + 1)$ states. But now note that for any $x \in \mathrm{FB}(A)$, $g_x = g_w$ for any word representative w. Therefore we may pick w to be the shortest possible word representative of

x. Then the bound implies that there is a transducer realizing g_x with no more than $|w| \cdot (|A|+1)$ states, and so clearly the minimal transducer realizing g_x will have no more than $|w| \cdot (|A|+1)$ states as required. △

Example 2.3.5. This example applies the construction in the proof of Theorem 2.3.1 to construct a small transducer for the free band element given by the word $w = abbac$. We color the transitions depending on the output symbols for, and omit the edge labels in later drawings for visual clarity.

The various stages of the construction, starting with $\mathcal{S}_a$, then $\mathcal{S}_a b$, $(\mathcal{S}_a b)b$, $((\mathcal{S}_a b)b)a$ and finally $(((\mathcal{S}_a b)b)a)c$ are drawn below:



2.4 How to build a minimal transducer

In the previous sections, for each word $w \in A^+$ we defined a complete invariant function g_w and then gave an exponential size transducer G_w realizing g_w. We then argued how the minimal transducer M_w realizing g_w has a size bounded by $N(|A|+1)$ where N is the length of the shortest word in that is equivalent to w. Now we wish to actually construct such a minimal transducer.

One approach we could envision would be to construct G_w and then apply a transducer minimization

algorithm, but this effort take at least $\mathcal{O}\left(2^{|A|}|w|\right)$ time to even construct G_w due to the exponential nature of the transducer and the cost of computing `affx` and `atob`.

As it turns out we can do better.

2.4.1 The vertical method

Recall that the proof of Theorem 2.3.1 was constructive, and gave us a way of constructing a transducer realizing g_w with at most $|w|(|A|+1)$ states. It is not hard to actually turn the proof into an algorithm that works in time and space $\mathcal{O}(|w| \cdot |A|)$ to compute a transducer realizing g_w.

But now, since our input alphabet is fixed to be $\{0,1\}$, then the number of transitions is proportional to the number of states, therefore, if we apply the Revuz minimization algorithm to this transducer, we get a minimal transducer in time and space $\mathcal{O}(|w| \cdot (|A|+1)) = \mathcal{O}(|w| \cdot |A|)$.

This algorithm proceeds by adding each letter one by one using the right action of A on transducers defined in Definition 2.3.3. Upon doing so, we add a "vertical slice" from an initial state to a terminal state. Hence we call this the vertical method.

Once we have a minimal transducer solving the word problem is easy, since the minimal transducer is a complete invariant, it suffice to just test transducer isomorphism which can be done in time and space linear in the number of states.

So in total for alphabet A and words $u, v \in A^+$, to decide $u \sim_\beta v$ we need $\mathcal{O}(|A| \cdot (|u|+|v|))$ time and space to compute transducers realizing g_u, g_v via the vertical method, then we need $\mathcal{O}(|A| \cdot (|u|+|v|))$ time and space to minimize them and finally $\mathcal{O}(n_1 + n_2)$ time and space to test isomorphism of transducers, where n_1 and n_2 are the number of states of the minimal transducers.

2.4.2 The horizontal method of [12]

In [12] the authors derive an algorithm which uses $\mathcal{O}(|A| \cdot (|u|+|v|))$ time and $\mathcal{O}(|u|+|v|)$ space to decide if $u \sim_\beta v$. We will now show how their algorithm fits into our framework and gives us a different method of generating minimal transducers.

First we introduce some notation and describe the general idea of the algorithm.

Definition 2.4.1. For $x \in \mathrm{FB}(A)$, a transducer $\mathcal{T} = (Q, \{0,1\}, A, q_0, T, \circ, *)$ realizing g_x and a state $q \in Q$, define the *content of* q as the set

$$\texttt{cont}(q) = \left\{r * t : t \in \{0,1\}, r \in Q \text{ s.t. } r \text{ is reachable from } q\right\}$$

In other words it is the set of all output symbols along every path from q to an accepting state.

Then the *k-th layer of* $\mathcal{T}$ is defined to be $\mathcal{I}_k = \{q \in Q : |\texttt{cont}(q)| = k, q \text{ is reachable from } q_0\}$.

Remark 2.4.2. It is not hard to see that the set of all states reachable from q_0 is $\bigcup_{k=0}^{|A|} \mathcal{I}_k$. Furthermore if $q \in \mathcal{I}_k$, then $q \circ t \in \mathcal{I}_{k-1}$ for all $t \in \{0,1\}$. △

So each layer consists of states with a fixed size of content, and the k-th layer only has state transitions into the $k-1$-st layer, therefore justifying the idea of these being layers of the automaton.

The basic idea of the algorithm of Radoszewski and Rytter can then be expressed as follows:

- To test if $u \sim_\beta v$ construct the minimal transducer of each and compare them as transducers.
- To construct the minimal transducer of a word w, we will build it up layer by layer starting from $\mathcal{I}_0$.
- At the k-th step, assume we have access to a collection of states consisting of $\mathcal{I}_k$ and some unreachable states.

- Then to build $\mathcal{I}_{k+1}$, we perform a linear scan across the word and add in one state for every subword of w containing $k+1$ distinct letters, that is maximal in this respect in a concrete way defined in the paper.
- We can then add transitions from every subword in $\mathcal{I}_{k+1}$ to its prefix and suffix in $\mathcal{I}_k$.
- Then we union together equivalent states of this layer. The result is a set of states containing $\mathcal{I}_{k+1}$ and some unreachable states. Since the previous layer was minimized in the previous step, this can be done very efficiently.
- After the last layer is made, identify the initial state q_0 and perform a traversal from it to identify all reachable states.
- Now create a new transducer consisting only of the reachable states. This is the minimal transducer for u.

Note that in [12] the minimal transducer construction is not the main goal, and it is only constructed implicitly. In particular unreachable states do not get removed, and at each point only the current and previous layers are kept. This is done to reduce space complexity.

In essence, Radoszewski and Rytter come up with a method of constructing a transducer realizing g_w with $\mathcal{O}(|A| \cdot |w|)$ states. They do this in a *horizontal* manner by efficiently building up a layer of the transducer at every step. These layers are exactly the same layers as the ones in Revuz minimization. They then interleave Revuz minimization steps with layer construction steps, allowing them to avoid storing the full transducers and rather only store one layer of the transducer at a time. This allows them to keep the space complexity to only $\mathcal{O}(|u| + |v|)$.

Conclusions and further work

In this work we have exhibited a novel way of representing elements in the free band by minimal acyclic transducers. We show that the minimal transducer for an element $x \in \mathrm{FB}(A)$ have size proportional to $N \cdot |A|$ where N is the length of the smallest word representative of x. Furthermore, since the best know algorithm for solving the word problem fits neatly as an instance of our transducer representation, we believe that there is potential in exploring the computational aspects of our representation further.

In terms of future work, we believe that we are able to use our minimal acyclic transducer framework introduced in this paper to show the following conjectures. To paraphrase Fermat, “we have discovered a truly remarkable proof of these conjectures which this masters project is too small to contain.”

Conjecture 2.4.3. Let A be an alphabet and $s, t \in \mathrm{FB}(A)$ and M_s, M_t be the minimal transducers for s and t respectively.

Then it is possible to compute M_{st} the minimal transducer for the product st in time and space $\mathcal{O}\left(|M_s| + |M_t| + |A|^2\right)$, where $|M_s|, |M_t|$ is the number of states of each minimal transducer.

In particular, by Theorem 2.3.1, it follows that the time and space is $\mathcal{O}\left(|A| \cdot (N_t + N_s + |A|)\right)$, where N_t, N_s are the lengths of the minimal word representatives for t and s respectively.

This would equip the set of minimal transducers with a multiplication that makes them isomorphic to the free band. This has interesting theoretical implications too, and as a further goal we could study extensions of the transducer multiplication to the set of all transducers (as the current minimal transducers are very constrained). This could be beneficial in solving the word problem in arbitrary finitely presented bands.

Furthermore, a subvariety $\mathcal{W}$ of a variety $\mathcal{V}$ is a proper subclass of $\mathcal{V}$ that is a variety in its own right. The lattice of subvarieties of bands is well studied and characterized, for example see [10]. A relatively free band is the free object in a subvariety of bands. We believe that using our framework, the horizontal method of Radoszewski and Rytter can be unified with the solution to the word problem in relatively free bands by Petrich [10] to yield the following conjecture:

Conjecture 2.4.4. Let A be and alphabet, $\mathcal{W}$ be a subvariety of the variety of bands, $F_{\mathcal{W}}(A)$ be the free object generated by A in this variety.

Then the word problem in $F_{\mathcal{W}}(A)$ for word $u, v \in A^+$ can be solved in $\mathcal{O}\left(|A|^2 \cdot (|u| + |v|)\right)$ time and space.

Neither of the two problems are currently known to be solvable in subexponential time. We intend to prove these theorems in a subsequent publication.

An even further topic of interest would be that of finitely presented bands. That is, it would be interesting to apply our framework to the word problem in finite quotients of the free band, and see what sort of computational complexity we can achieve.

Bibliography

[1] A. M. Ballantyne and D. S. Lankford. "New decision algorithms for finitely presented commutative semigroups". In: *Computers & Mathematics with Applications* 7.2 (1981), pp. 159–165. ISSN: 0898-1221. DOI: 10.1016/0898-1221(81)90115-2.

[2] Jean Berstel et al. *Minimization of Automata*. 2010. arXiv: 1010.5318 [cs.FL].

[3] J. Bubenzer. "Cycle-aware minimization of acyclic deterministic finite-state automata". In: *Discrete Applied Mathematics* 163 (2014). Stringology Algorithms, pp. 238–246. ISSN: 0166-218X. DOI: 10.1016/j.dam.2013.08.003.

[4] A. J. Cain. *Nine Chapters on the Semigroup Art*. 2020. URL: http://www-groups.mcs.st-andrews.ac.uk/~alanc/pub/c_semigroups/index.html.

[5] A. Church. "An Unsolvable Problem of Elementary Number Theory". In: *American Journal of Mathematics* 58.2 (1936), pp. 345–363.

[6] J. Daciuk et al. "Incremental construction of minimal acyclic finite-state automata". In: *Computational linguistics* 26.1 (2000), pp. 3–16.

[7] J. A. Green and D. Rees. "On semi-groups in which $x^r = x$". In: *Mathematical Proceedings of the Cambridge Philosophical Society* 48.1 (1952), pp. 35–40. DOI: 10.1017/S0305004100027341.

[8] J.M. Howie. *Fundamentals of Semigroup Theory*. LMS monographs. Clarendon Press, 1995. ISBN: 9780198511946.

[9] G. S. Makanin. "On the identity problem in finitely defined semigroups". In: *Dokl. Akad. Nauk SSSR* 171 (1966), pp. 285–287. ISSN: 0002-3264.

[10] M. Petrich and P. V. Silva. "Structure of Relatively Free Bands". In: *Communications in Algebra* 30.9 (2002), pp. 4165–4187. DOI: 10.1081/AGB-120013311.

[11] E. L. Post. "Recursive Unsolvability of a Problem of Thue". In: *The Journal of Symbolic Logic* 12.1 (1947), pp. 1–11. ISSN: 00224812.

[12] J. Radoszewski and W. Rytter. "Efficient Testing of Equivalence of Words in a Free Idempotent Semigroup". In: *SOFSEM 2010: Theory and Practice of Computer Science*. Jan. 2010, pp. 663–671. DOI: 10.1007/978-3-642-11266-9_55.

[13] D. Revuz. "Minimisation of acyclic deterministic automata in linear time". In: *Theoretical Computer Science* 92.1 (1992), pp. 181–189. ISSN: 0304-3975. DOI: 10.1016/0304-3975(92)90142-3.

[14] M. Sipser. *Introduction to the Theory of Computation*. Second. Course Technology, 2006.

[15] A. M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* s2-42.1 (1937), pp. 230–265. DOI: https://doi.org/10.1112/plms/s2-42.1.230.