The Coalgebraic Interpretation of Brzozowski's Automata Minimisation Algorithm

Natalie Ravenhill
ID Number : 1249790

Supervisor: Achim Jung

Dissertation for the degree of BSc Computer Science
Date : 6th April 2015

School of Computer Science
University of Birmingham

Abstract

This report gives an explanation of a proof of correctness of an algorithm by Brzozowski, which gives the minimal automaton of a given automaton that recognises a certain language. The proof in question is by [Bonchi et al., 2014] and gives the algorithm as a corollary of the proof that observability of automata is the dual of reachability of automata. Using this proof and a representation of automata as a combination of an algebra and a coalgebra, we can derive Brzozowski's algorithm. My report explains how this proof works in detail and the category theory, universal algebra and automata theory needed to understand it.

Keywords: *Automata, Algebra, Coalgebra, Automata Minimisation, Category Theory*

Acknowledgements

Firstly, I must thank my supervisor, Professor Achim Jung, for his guidance and the interesting discussions we have had during the course of my project and for aiding me to learn some of the advanced topics in my dissertation over the summer.

I would also like to thank Dr Steve Vickers, for the advice he gave me on my project during the inspection and himself and Dr Martín Escardó for attending my final demonstration and the advice they gave me on my report.

Contents

1 Introduction **5**

2 Background **8**

2.1 Category Theory . . . 8

2.1.1 Category . . . 8

2.1.2 Functors . . . 10

2.1.3 Homomorphism . . . 11

2.1.4 Initial Object . . . 11

2.1.5 Final Object . . . 12

2.1.6 Coproduct . . . 12

2.1.7 Product . . . 12

2.1.8 Product Category . . . 12

2.2 Universal Algebra . . . 13

2.2.1 Types of Universal Algebra . . . 13

2.2.2 Algebras and Categories . . . 16

2.2.3 Representing Data Types as Algebras . . . 17

2.3 Coalgebra . . . 17

2.3.1 Categoric Definition . . . 18

2.3.2 Representing Data Types as Coalgebras . . . 18

2.4 Automata . . . 20

2.4.1 Representation of Automata . . . 20

2.4.2 Algebra and Coalgebra . . . 20

2.4.3 Reachability and Observability . . . 21

2.4.4 Categories of Automata . . . 27

3 Brzozowski's Algorithm **30**

3.1 Powerset Construction . . . 31

3.2 Example of Brzozowski's Algorithm . . . 32

3.3 Proof of Correctness . . . 35

3.3.1 Co-reachable and Co-deterministic Automata . . . 35

4 Brzozowski's Algorithm Coalgebraically **38**

4.1 Contravariant Powerset Functor 38
4.2 Constructing the Reverse Automaton 39
4.2.1 Reverse of Transition Function 39
4.2.2 Switching Initial and Final states 41
4.2.3 Converting Reachability to Observability 42
4.2.4 Converting Observability to Reachability 46
4.3 Getting Brzozowski's Algorithm 49

5 Discussion **53**

6 Conclusion **54**

Chapter 1

Introduction

The aim of my project is to explain a proof of correctness of Brzozowski's automata minimisation algorithm that I found different and interesting, but very difficult to understand. This is because of the presence of coalgebra, abstract algebraic structures in mathematics, that are often used to represent state based systems. This is because they represent structures in which one input is usually mapped to many different outputs. This suits automata, particularly the transition function, as we map transitions from a state to a different state using many different symbols from the input alphabet.

I chose to study this proof as last summer I studied different types of automata minimisation algorithms, including partition based minimisation algorithms (in which we we take a Determinisitic Finite Automaton (DFA) and partition the set of states into classes, where each state has the same behaviour, and these classes form the minimal automaton) such as in [Moore, 1956] and [Hopcroft, 1971].

The last algorithm I studied was *Brzozowski's Algorithm*, defined in [Brzozowski, 1962]. It is based on the reversal of automata and the powerset construction of states of an automaton to get a deterministic one from a non deterministic one. In a powerset construction, we replace the states of the automaton with the powerset of these states, then replace the transitions with those between these sets. The initial state will be the set of all initial states of the automaton and the final states will be all of the state sets including a final state of the original automaton.

We apply the powerset construction to the reverse of our automaton which is obtained by switching the initial and final states and reversing all of the transitions between each state. Then we will have some states that cannot be reached so we remove these. Now we have a minimal automaton recognising the reverse of the language recognised by the original automaton. To get the minimal automaton for the original language we repeat this process, reversing our new automaton, constructing the powerset and taking the reachable part.

Proving the correctness of this algorithm is less straightforward than proving the correctness of partition-based algorithms. I studied two proofs of correctness. The first was given in [Sakarovitch, 2009] and is based on a theorem in which determinising an automaton which is co-deterministic (the opposite of deterministic) and co-reachable (opposite of reachable) gives the minimal automaton, as we can show that this automaton is the same as the one obtained by reversing the first powerset construction. This theorem is not immediately obvious and therefore a better correctness result can be given.

I was then directed to the paper that formed the basis of my dissertation, "Algebra-coalgebra Duality in Brzozowski's Minimization Algorithm" [Bonchi et al., 2014]. I found this paper interesting as I had no knowledge of coalgebra, or indeed any abstract algebra and category theory and was curious to see how this could be applied to the minimisation of automata. Therefore I decided to study this paper in more depth and understand how and why the coalgebraic proof of Brzozowski's automata minimisation algorithm works.

To understand this proof we need to understand some basic category theory (Chapter 2.1), in order to represent the coalgebra categorically to aid us in the proof, and universal algebra and coalgebra to explain the construction of automata used in the proof. (Chapter 2.1.8)

We also need to understand two properties of automata, *reachability*, which means that all states can be reached from the initial state, and *observability*, which means all states recognise a unique language. (Chapter 2.4).

I begin by explaining this background in more detail.

Then we explain how Brzozowski's Algorithm works in detail, following it through with an example. (Chapter 3)

Finally we go through the coalgebraic proof of the correctness of the algorithm. (Chapter 4). There are a few stages to this proof. It is based on a theorem where we prove that observability is the dual (opposite) of reachability.

A minimal automaton is one that is reachable and observable and recognises a given language L. By using this theorem we can obtain that automaton.

This theorem states that by reversing a reachable automaton recognising L we get an observable automaton that recognises $rev\ L$ (The reverse language of L). Eventually we want an automaton that is reachable and observable and recognises L so this will be useful in its construction, as we will see.

To give the reverse construction, we use a functor called the *contravariant powerset functor* (Chapter 4.1). Contravariant means we reverse the directions of all of the functions, and a powerset functor replaces all of the sets with their powersets. This can be applied to our automaton to get the reverse of its components. We apply the contravariant powerset functor to the transition function to get the reverse transition function 2^t and to the initial state function to get the final state function. The new initial state function is just obtained by currying the arguments of the original final state function. (Chapter 4.2)

We can also apply the contravariant powerset functor to the constructions we create to define reachability and observability and get a commuting diagram. (Chapter 4.2.3).

What we have then defined is the powerset construction for the reverse of our original automaton. Taking the reachable part of this automaton gives us a reachable automaton. This automaton accepts the reverse language so we need to apply the reverse construction again to get an automaton that recognises the original language. We have just given a theorem that says reversing a reachable automaton gives an observable automaton, so our new automaton for the original language must be observable. The final step of Brzozowski's algorithm consists of taking the reachable part, so finally we have an automaton that recognises the original language and is both reachable and observable. Therefore it must also be minimal. This is explained in the final section of the report (Chapter 4.3).

Chapter 2

Background

2.1 Category Theory

To begin, we review some basic category theory that we will need to understand the coalgebraic interpretation. Category Theory is a foundation of mathematics in which classes of structures are represented as categories that contain objects and morphisms, which link the objects.

2.1.1 Category

A category (as defined in [Awodey, 2010]) consists of: :

- Objects, denoted by A, B, C, etc...
- Arrows (or morphisms) between objects denoted as f, g, h, etc..
- Operations $dom(f)$ and $cod(f)$ that represent the domain and codomain of a function f
- A composition operation, $\circ$, between morphisms, defined by taking morphisms $f : A \to B$ and $g : B \to C$ and forming $g \circ f : A \to C$. This means that $cod(f) = dom(g)$ must hold.

There are two properties that all categories must satisfy:

1. The composition operation is **associative**, so for all $f : A \to B$, $g : B \to C$ and $h : C \to D$, $h \circ (g \circ f) = (h \circ g) \circ f$.
2. For any object A in a category, there is an **identity** morphism, $1_A : A \to A$, such that for any morphism $f : A \to B$, $f \circ 1_A = f = 1_B \circ f$ (where 1_B is the identiy morphism for the object B)

An Example of a Category

The simplest example of a category is the category of all sets, **Set** (we represent a category by giving its name in bold). In this category the objects are sets and the morphisms are functions between the sets. The composition operation is just the composition of two functions.

One of the benefits of category theory is that we can represent objects and morphisms diagrammatically. Then we can give properties by drawing commutative diagrams, which are diagrams where the result of starting from a certain object and getting to another object by following a series of morphisms will always be the same. The paths we follow are compositions of different morphisms, so these compositions will be equal and we can form equations from them.

For example, if we want to illustrate that the composition of three functions, f, g, and h is associative, then we can give a diagram as follows, where $f : A \to B$, $g : B \to C$ and $h : C \to D$ (as in [Awodey, 2010]) :

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[dr, "g\circ f"'] & B \arrow[d, "g"] \arrow[dr, "h\circ g"] & \\
& C \arrow[r, "h"] & D
\end{tikzcd}$$

To get from A to D, we need morphisms f, g and h. Then we compose these morphisms, which can be done in two ways - we can either compose f and $h \circ g$, or equivalently we can compose $g \circ f$ and h. By the commutative diagram, we get the same result by following any sequence of arrows from A to D. It is the same as giving the equation $(h \circ g) \circ f = h \circ g \circ f = h \circ (g \circ f)$.

We can illustrate the identity functions for **Set** by giving a similar diagram (as in [Awodey, 2010]) :

$$\begin{tikzcd}
A \arrow[r, "1_A"] \arrow[dr, "f\circ 1_A"'] & A \arrow[d, "f"] \arrow[dr, "1_B\circ f"] & \\
& B \arrow[r, "1_B"] & B
\end{tikzcd}$$

Here we use the identity morphisms for A and B, 1_A and 1_B. By the diagram, we get $f \circ 1_A = f = 1_B \circ f$, because we can get from A to B in the diagram by following both of these compositions.

2.1.2 Functors

A *functor* is a morphism between categories. It is defined in [Awodey, 2010] as a structure preserving map between any two categories, **C** and **D**. Formally, a functor $F : \mathbf{C} \to \mathbf{D}$ is a mapping between objects (so that for any object $O \in \mathbf{C}$ there is a corresponding object $F(O) \in \mathbf{D}$) and arrows such that given any objects A,B and any morphism f, then:

- $F(f : A \to B) = F(f) : F(A) \to F(B)$
  (*A functor on a morphism between two objects gives a morphism between the images of the functor applied to those objects*)

- $F(1_A) = 1_{F(A)}$
  (*The functor on the identity morphism of an object A is the identity morphism of the category containing the image F(A))* .

- $F(g \circ f) = F(g) \circ F(f)$
  (*A functor on a composition of morphisms is the composition of the images of the individual morphisms.*)

We can show diagrammatically that a functor F:

$$\begin{tikzcd}\mathbf{C} \arrow[r, "F"] & \mathbf{D}\end{tikzcd}$$

applied to a diagram in **C**:

$$\begin{tikzcd}A \arrow[r, "f"] \arrow[dr, "g\circ f"] & A \arrow[d, "g"] \\ & B\end{tikzcd}$$

gives us a diagram in the category **D**:

$$\begin{tikzcd}F(A) \arrow[r, "F(f)"] \arrow[dr, "F(g\circ f)"] & F(A) \arrow[d, "F(g)"] \\ & F(B)\end{tikzcd}$$

Here f is an **endofunctor** (a functor from a category to itself).

2.1.3 Homomorphism

A functor is a homomorphism between categories. In general, a homomorphism is a structure preserving map between any two structures of the same kind. In category theory we distinguish special kinds of morphisms. A morphism $f : A \to B$ will be called:

- a **monomorphism** if for any $g, h : C \to A$, if $f \circ g = f \circ h$ then $g = h$. In the category of sets, a function is a monomorphism if it is *injective*. We can show this diagrammatically (as in [Awodey, 2010]) as follows:

$$\begin{tikzcd} C \arrow[r, shift left, "g"] \arrow[r, shift right, "h"'] & A \arrow[r, "f"] & B \end{tikzcd}$$

- an **epimorphism** if for any $i, j : B \to D$, if $i \circ f = j \circ f$ then $i = j$. In the category of sets, a function is an epimorphism if it is surjective. We can show this diagrammatically (as in [Awodey, 2010]) as follows:

$$\begin{tikzcd} A \arrow[r, "f"'] & B \arrow[r, shift left, "i"] \arrow[r, shift right, "j"'] & D \end{tikzcd}$$

- an **isomorphism** if there exists an inverse function $g : B \to A$ such that $g \circ f = 1_A$ and $f \circ g = 1_B$. In the category of sets, a function is an isomorphism if it is bijective. As a diagram, we give:

$$\begin{tikzcd} A \arrow[r, "f"'] \arrow[dr, shift left, "1_A"] \arrow[dr, shift right, "g\circ f"'] & B \arrow[d, "g"] \arrow[dr, shift left, "f\circ g"] \arrow[dr, shift right, "1_A"'] & \\ & A \arrow[r, "f"'] & B \end{tikzcd}$$

Another type of morphism is an **endomorphism**, which is a map from an object to itself. Similarly, an endofunctor is a functor from a category to itself.

2.1.4 Initial Object

An initial object, 0, is an object where for any object C there is a unique morphism $0 \to C$. For example, in the category of sets, **Sets**, this is the empty set, $\emptyset$, because there can be only one function from the empty set to any other set.

2.1.5 Final Object

A final object, 1, is an object where for any object C there is a unique morphism $C \to 1$, For example, in the category of sets, **Sets**, this is any singleton set, because there is only one element in this set to make a mapping to.

A final object is the dual (opposite) of an initial object, as a coalgebra is the dual of an algebra, as we will see later.

2.1.6 Coproduct

A coproduct of some objects in a category is formed by taking one of a set of objects, so is a disjoint sum of a set of objects. Given two objects A and B, we form the coproduct $A + B$.

2.1.7 Product

A product of some objects in a category is formed by taking the result of a set of objects, so is the conjunction of a set of objects. Given two objects A and B, we form the product $A \times B$.

2.1.8 Product Category

A product category is a category formed by taking all of a set of categories, so is a conjunction of categories. Given two categories **C** and **D**, we form the product of these two categories $\mathbf{C} \times \mathbf{D}$, where the objects will be pairs of objects (C, D) where $C \in \mathbf{C}$ and $D \in \mathbf{D}$ and the morphisms are of the form (f, g), where $f : C \to C'$ and $g : D \to D'$.

We also have to give composition and identity for this to be a category. They are defined componentwise as in [Awodey, 2010] as

- $(f', g') \circ (f, g) = (f' \circ f, g' \circ g)$
- $I_{(C,D)} = (I_C, I_D)$

2.2 Universal Algebra

Universal Algebra is the study of general algebraic structures. An algebra is represented by a carrier set, with a collection of operations (or signature) on the elements of this set. The operations on elements in an algebra must succeed and return an object from the set of elements contained in the algebra. They must also have a fixed finite arity.

2.2.1 Types of Universal Algebra

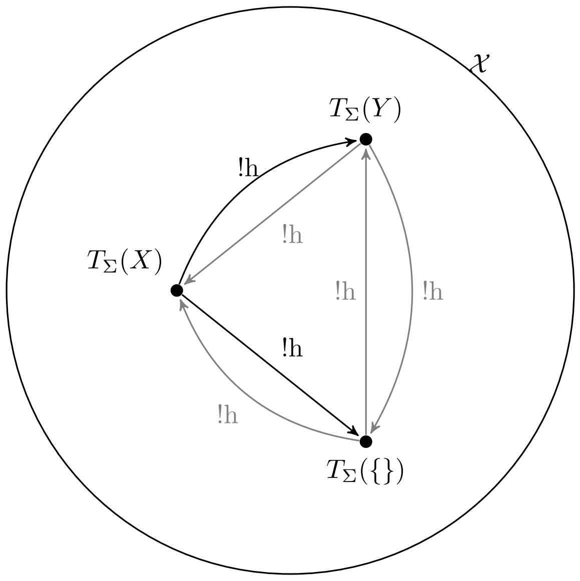
An example of an algebra is $\mathcal{N} = (\mathbb{N}; +, *, succ, 0, 1)$, where $\mathbb{N}$ is the carrier set and is the set of natural numbers, $*$ and $+$ represent conventional addition and multiplication on natural numbers, *succ* is the successor function for natural numbers and 0 and 1 are the identity elements for addition and multiplication.

$\mathcal{N}$ can be generated from the empty set by using the 0 and 1 elements and the $*$ and $+$ operations. We get 2 from 1 + 1, 3 from 2 + 1 and so on until we have all elements of $\mathbb{N}$. An algebra that can be generated from the empty set, $\emptyset$ is an **initial algebra**. If the carrier set was the integers, $\mathbb{Z}$, we could not generate any negative numbers, because we only have addition and multiplication, so this would not generate an initial algebra, as we cannot generate every element of $\mathbb{Z}$.

Although we defined this algebra on addition and multiplication, $+$ and $*$ do not have to be these functions, they can represent any function of the same arity. This means that there are many algebras that can be defined in this way and they all have the same signature (a set of operations and arities of those operations) and may or may not be generated from the empty set.

We can have a class of these algebras, $\mathcal{X}$. Then every algebra $T_\Sigma(X) \in \mathcal{X}$, where X is any set, is a term algebra with a unique homomorphism to any other algebra of the same signature:

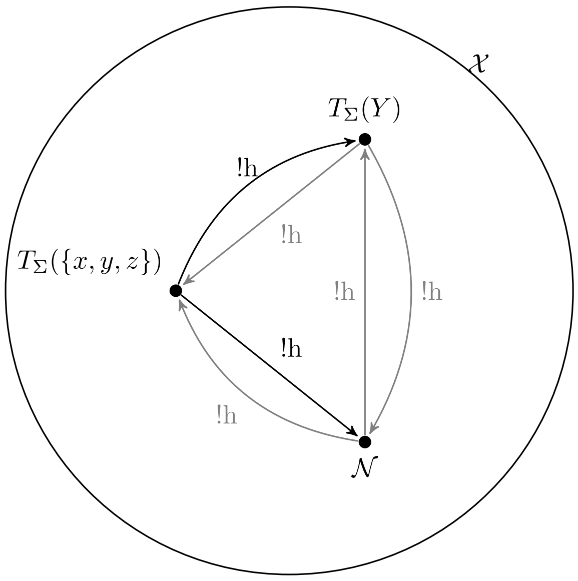
Diagrammatically:



There will be unique homomorphisms between our algebra and the algebras that also have the siganture Σ, even if their carrier sets are empty. This will also be the case for all other algebras of the same signature in $\mathcal{X}$ (shown by the unique homomorphisms in grey).

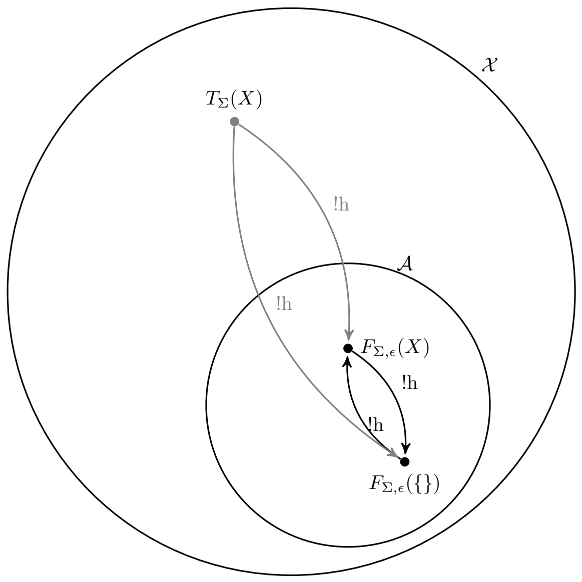
Our algebra $\mathcal{N}$ is the same as the algebra $T_\Sigma(\{\})$ with signature $\Sigma = (\{+, *, succ, 0, 1\}, \{2, 2, 1, 0, 0\})$. This algebra will be a term algebra and an initial algebra because it is generated from the empty set. If we give a non empty carrier set $X = \{x, y, z\}$, containing some variables, we can form another term algebra $T_\Sigma(X)$, then this will not be an initial algebra. However it is still a term algebra of the same signature as $\mathcal{N}$ so there will be a unique homomorphism from it to $\mathcal{N}$ (and vice versa).

We can modify our previous diagram to get the following:



We can restrict the term algebras given in $\mathcal{X}$, by postulating properties (such as associativity and commutativty of $*$) on the term algebras of $\mathcal{X}$. Then we have a subclass of algebras $\mathcal{A} \in \mathcal{X}$ and any algebra in this class is a **free algebra** $F(X)_{(\Sigma,\epsilon)}$, over Σ and ϵ (where X is the carrier set and ϵ is the set of properties we have postulated). Therefore free algebras are a subclass of term algebras and a free algebra may or may not be an initial algebra, depending on the carrier set. If the carrier set is empty, then this means that the algebra can be freely generated from the empty set, so is initial, otherwise it is not initial. The algebras in $\mathcal{A}$ have unique morphisms to each other and a term algebra outside the class has a unique morphism to an algebra inside $\mathcal{A}$, but we can't have a unique morphism from a free algebra in $\mathcal{A}$ to a term algebra outside it.

Diagrammatically, we have:



Therefore we can say that for any signature Σ, any collection of postulates, ϵ and any carrier set X, there exists the free algebra $F(X)_{(\Sigma,\epsilon)}$, which is unique (up to isomorphism).

We can show the diffrences between each type of algebra as a table:

Algebra	Always generated from $\emptyset$?	Always satisifies a set of properties?
Initial Algebra	Yes	No
Term Algebra	No	No
Free Algebra	No	Yes

Non-Example

We can give an example of an algebra that is not a universal algebra. Consider the structure $\mathcal{L} = (\mathit{Lists}; \mathit{head}, \mathit{tail}, [\ ], ::)$, where the carrier set contains all possible lists we can generate and we have the functions *head* and *tail*, an empty list function " [ ] " and an append function " :: ". $\mathcal{L}$ is not a universal algebra because *head* and *tail* of an empty list would not succeed (as the operations are not total). To define this algebra, we would have to allow partial operations, giving us a universal algebra that is generalised to include partial functions.

Example

Stacks can be represented as the algebra $\mathit{Stack} = (S; \mathit{empty}, \mathit{pop}, \mathit{push})$. S is the set of possible stacks we can create, *empty* is a function that gives an empty stack, *pop* is an operation that removes the top element and *push* is a function that adds an element to the top of a stack. The arities of the functions are $(0, 1, 2)$. For this to be an algebra, *pop*(*empty*) must equal *empty*, so that all of the operations are total.

Stack will be an initial algebra because we can form any stack by starting with an empty stack and pushing elements onto that stack with *push*. The elements that are pushed onto the stack are equivalent to the one element stacks in S. Therefore all stacks can be generated from $\emptyset$.

2.2.2 Algebras and Categories

Algebras give rise to categories. To get from algebras to categories, we define a category where the objects are algebras of a certain signature and the morphisms are the homomorphisms between those algebras.

An **initial algebra** is then exactly an initial object in this category, as there will be a unique morphism from this algebra to any other algebra of the same signature.

F-algebras

We can give an endofunctor $F : Set \to Set$ that represents the signature of an algebra. For example, we can have a functor representing the signature $\Sigma = (\{+, *, succ, 0, 1\}, \{2, 2, 1, 0, 0\})$. F will always be an endofunctor because the operations must return values of the carrier set. The functor is then applied to the carrier set of the algebra, so given a set X, we have $F(X) = (X \times X) + (X \times X) + X + 1 + 1$, giving the arities of each operation as a coproduct.

The definition of any F-algebra is an object X in any category C and a morphism $F(X) \to X$, representing how we get the elements of the carrier set by applying the operations.

Using functors to define algebras in this way means that we can show a homomorphism between algebras without giving their signatures (and hide the operations) and we can model an algebra in full by just giving a simple diagram. We can show a homomorphism between two F-algebras diagrammatically as:

$$\begin{tikzcd}
F(A) \arrow[r, "\alpha"] \arrow[d, "F(f)"'] & A \arrow[d, "f"] \\
F(B) \arrow[r, "\beta"'] & B
\end{tikzcd}$$

where (A, α) is an algebra, (B, β) is another algebra of the same signature (represented by the functor F) and f is the homomorphism between them, such that $f \circ \alpha = \beta \circ F(f)$.

Then we can see that there will be a category of F-algebras corresponding to the class of universal algebras for a given signature.

2.2.3 Representing Data Types as Algebras

We can show the categoric definition of the algebra of stacks as an F-Algebra. To do this, we need to define an endofunctor F on the category **Set**, as the set of all stacks is an object of this category, to enable us to represent our algebra for stacks as $F(\mathit{Stack}) \to \mathit{Stack}$. The elements of this algebra will be all of the possible stacks, and we will represent the operations of the algebra as morphisms to define the endofunctor F:

$$\begin{tikzcd}
1 \arrow[dr, "empty"] & \\
\mathit{Stack} \arrow[r, "pop"] & \mathit{Stack} \\
\mathit{Stack} \times \mathbb{N} \arrow[ur, "push"'] &
\end{tikzcd}$$

We can combine all of these morphisms into a single one from the coproduct (disjoint union) $1 + \mathit{Stack} + \mathit{Stack} \times \mathbb{N}$ to Stack. Therefore we can define a functor $F(X) = 1 + X + X \times \mathbb{N}$ from sets to sets, such that $F(\mathit{Stack}) \to \mathit{Stack}$ will be an instance of it.

For $F(\mathit{Stack}) \to \mathit{Stack}$ to be an initial algebra, we must be able to generate all possible stacks in S from an empty stack (which we have already seen) and be able to have a unique homomorphism from it to any other algebra of the same signature. Instead of Stack we could a have a different data structure where the operations have the same arities, so it is still represented as a functor $1+X+X\times\mathbb{N}$. Therefore the operations don't need to be exactly the same but the stucture has to be preserved.

2.3 Coalgebra

Now we have defined universal algebra, we can introduce a coalgebra as its dual (opposite). For algebras, we usually want to map some inputs to one output, but there may be situations where we need more than one output, so this is where coalgebra are used. This makes them useful for defining the behaviour of systems, particularly state based systems, such as automata, as for example, we may have a transition function where we want to keep track of the next state and the word we are forming.

2.3.1 Categoric Definition

Coalgebra are usually defined categorically, as they are most useful for defining data structures and are rarely used in mathematics, so a formal definition without categories would not be helpful.

Because a coalgebra is the dual of an algebra, this means we can reverse the direction of all of the morphisms in an algebra to get a coalgebra. Therefore we represent a coalgebra as a carrier set X, and a morphism $\alpha : X \to F(X)$ where F is the functor that defines the operations on X.

F-Coalgebra

F will be an endofunctor representing the operations on a coalgebra. Then the objects of the category of F-coalgebras will be the coalgebras with that signature and as with algebras, the morphisms will be homomorphisms between these coalgebra:

$$\begin{tikzcd}
A \arrow[r, "\alpha"] \arrow[d, "f"'] & F(A) \arrow[d, "F(f)"] \\
B \arrow[r, "\beta"'] & F(B)
\end{tikzcd}$$

where (A, α) is an coalgebra, (B, β) is another coalgebra of the same signature (represented by the functor F) and f is the homomorphism between them such that $F(f) \circ \alpha = \beta \circ f$.

Final Coalgebra

A final coalgebra is defined as a final object in the category of F-coalgebras, so they are the dual of initial algebras. This means there is a unique morphism from any other algebra of signature F to the final coalgebra.

2.3.2 Representing Data Types as Coalgebras

Coalgebras are generally used for defining infinite data types or state systems.

An example of an infinite data type is **streams**, lists of elements that can be infinite. A simple example of a stream is given in [Rutten, 2000], where we form words of potentially infinite length. Given an alphabet, A, we can form a set of potentially infinite words A^{∞}. Then we have two operations, one that forms an empty word, ϵ and one that appends a symbol $a \in A$ to another word.

We can give a functor $F(-) = 1 + (- \times -^{*})$, where 1 is a set containing one element, the operation $*$ that forms the empty word and $- \times -^{*}$ takes a pair (a, w) where w is a word and a a symbol that is prepended to it to form the word $a \cdot w$. The $-$ is replaced by the carrier set of the coalgebra.

$$\begin{tikzcd}
 & 1 \\
A^\infty \arrow[ru] \arrow[rd] & \\
 & A \times A^*
\end{tikzcd}$$

Then our coalgebra consists of the set A^∞ of all possible words and the functor $F(A) = 1 + (A \times A^*)$. The structure map in the category of F-coalgebras for our functor F will be $\alpha : A^\infty \to 1 + (A \times A^*)$. This says that given a stream in A^∞, we can see that it is formed from the empty word (represented by 1) and the operation that appends a symbol in A to that word. Therefore, given any word in A^∞, we can get the components that form it.

The typical way to define **automata** is as a coalgebra, where given a set of states X and an alphabet A, we usually define a transition function $\delta : X \times A \to X$, which takes a state and a symbol from A and goes to the next state on that symbol. We can rewrite this as a function $X \to X^A$, which maps a state to all functions $A \to X$. To see this as an example of an F-Coalgebra, we let F be the functor that maps a set X to X^A.

2.4 Automata

2.4.1 Representation of Automata

To represent a deterministic automaton, we are given a set A of input symbols, a set $1 = \{0\}$ and a set $2 = \{0, 1\}$ where 0 corresponds to false and 1 corresponds to true. Then we define an automaton (X, t, i, f) where:

- X is the set of states
- $t : 1 \to X^A$ is a transition function mapping a state $x \in X$ to a function $t(a) : A \to X$ that takes an input symbol and returns the state $t(x)(a)$ (which is the next state under a)
- $i \in X$ is an initial state (or equivalently a function $i : 1 \to X$)
- f is a set of final states (represented as a function $f : X \to 2$ where 1 means the state is a final state and 0 means it is not)

So a deterministic automaton with inputs from A can be represented diagrammatically as:

$$\begin{tikzcd}
1 \arrow[dr, "i"] & & 2 \\
& X \arrow[ur, "f"] \arrow[d, "t"] & \\
& X^A &
\end{tikzcd}$$

2.4.2 Algebra and Coalgebra

In [Bonchi et al., 2014] automata are defined as a combination of algebra and coalgebra, where the algebra defines the initial state:

$$\begin{tikzcd}
1 \arrow[dr, "i"] & & 2 \\
& X \arrow[ur, "f"] \arrow[d, "t"] & \\
& X^A &
\end{tikzcd}$$

The algebra will be the following F-Algebra, where $F(X) = 1 + (A \times X)$

$$\begin{tikzcd}
1 \arrow[dr, "i"] & \\
& X \\
A \times X \arrow[ur, "t"] &
\end{tikzcd}$$

This means that the state set, X, is formed of the initial state, defined by the function i and the states that are the destinations from other states, obtained by the transition function t.

The coalgebra defines the output function and transition structure:

$$\begin{tikzcd}
1 \arrow[dr, "i"] & & 2 \\
& X \arrow[ur, "f"] \arrow[d, "t"] & \\
& X^A &
\end{tikzcd}$$

which will be the F-Coalgebra where $F(X) = 2 \times (X)^A$, each state in the state set is formed as either a final state or not and is the source of a transition to another state.

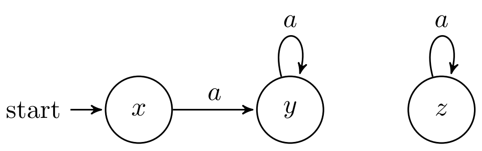
2.4.3 Reachability and Observability

Reachability and Observability are properties from systems theory that were extended to automata, as defined in [Arbib and Zeiger, 1969].

Reachability

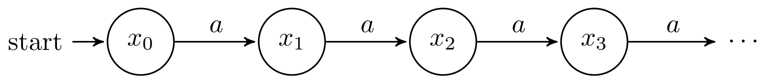
Reachability means for a state $x \in X$ there exists a word $w \in A^*$ such that applying that word to the initial state gives us x. An automaton is *reachable* if every state is reachable, so there is a word for every state where we can follow a series of transitions from the initial state and get to that state.

An example of an automaton that is not reachable is given as follows:



There is no way of getting from the initial state x to state z, so there is no word that we can apply to the initial state to give us z.

An example of an automaton that is reachable is given as the following automaton, $\mathcal{R}$:

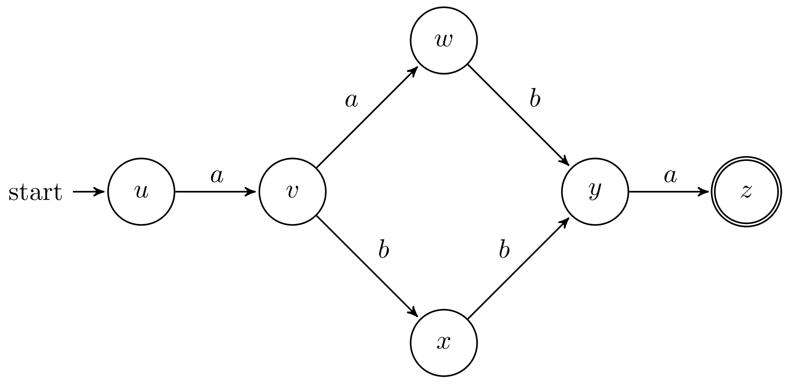


For any $x_i \in X$, there exists a word w where we can get from the initial state x_0 to that state. Our input alphabet is $A = \{a\}$, so a word in A^* will be a repeated i times. Therefore we can repeatedly apply the transition function t. For example, aaa is the word that gets from x_0 to x_3, so $x_3 = t(x_2)(a) = t(t(t(x_0)(a))(a))(a)$. Then this automaton could continue in the same way forever and we just keep appending a for each state we add to get another word in A^*. For the initial state, there is no way of getting from the initial state to itself, so this would just be ϵ, the empty word.

Observability

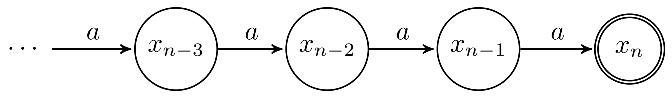
Observability means that a state has unique behaviour, so it recognises a different langauge from the other states in the automaton. The language that a state recognises is the set of words where following a series of transitions on their symbols gives us an accepting state. An automaton is observable if every state recognises a different language.

An example of an automaton that is not observable is given below:



This is not observable because the language accepted by w is the same as that accepted by x, namely {ba}, as this is the only word that can get us to a final state from each of those states..

An example of an observable automaton is given by the following automaton without initial states, $\mathcal{O}$:



This automaton has an input alphabet of $A = \{a\}$ and accepts a string of the symbol a of length n. The final state will accept the language $\{\epsilon\}$. Then the previous state, x_{n-1} accepts the language $\{a\}$ as this is the only symbol on which there is a transition to an accepting state. The state before that, x_{n-2} will accept the language $\{aa\}$ and so on, because each state has a single transition to its successor state on a. Therefore we can see that every state will only accept a string of the symbol a of a length that is 1 more than its successor state, so each state accepts a different language.

Reachability and Observability diagrammatically

We model both of these properties diagrammatically by adding maps from automata that model reachability and observability to our original automaton as follows, using homomorphisms between the automaton modelling each property to our original automaton:

$$\begin{tikzcd}
1 \arrow[d, "\epsilon"] \arrow[dr, "i"] & & 2 \\
A^* \arrow[r, dotted, "r"] \arrow[d, "\alpha"] & X \arrow[r, dotted, "o"] \arrow[ur, "f"] \arrow[d, "t"] & 2^{A^*} \arrow[u, "\epsilon?"'] \arrow[d, "\beta"] \\
(A^*)^A \arrow[r, dotted, "r^A"] & X^A \arrow[r, dotted, "o^A"] & (2^{A^*})^A
\end{tikzcd}$$

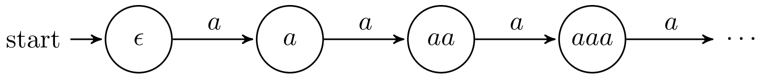
Reachability The left hand side of this diagram is an automaton that helps us to explain **reachability**:

$$\begin{tikzcd}
1 \arrow[d, "\epsilon"] \\
A^* \arrow[d, "\alpha"] \\
(A^*)^A
\end{tikzcd}$$

This is an automaton without final states, as we are only interested in where we can get to from the initial state. A state in the automaton is represented by a word, so instead of having X as the set of states, we now have A^*. If we use our input alphabet from the example given above, then our state set would be $X = A^* = \{\epsilon, a, aa, aaa, aaaa, \dots\}$.

ϵ is our initial state and α is the transition function, which maps a word, $w \in A^*$ to the word obtained by appending a symbol $a \in A$ to it. So α is defined as $\alpha : A^* \to (A^*)^A$ with $a(w)(a) = w \cdot a$.

We can show an example of this by using our example automaton that we used to explain reachability, $\mathcal{R}$ to get an automaton:



We can see that any state in the set A^* is reachable, as we keep increasing the length of a in the word infinitely, so we will be able to get to any word in A from the inital state.

Observability The right hand side of the diagram gives an automaton that helps us to explain **observability**:

$$\begin{tikzcd}
2 \\
2^{A^*} \arrow[u, "\epsilon?"] \arrow[d, "\beta"'] \\
(2^{A^*})^A
\end{tikzcd}$$

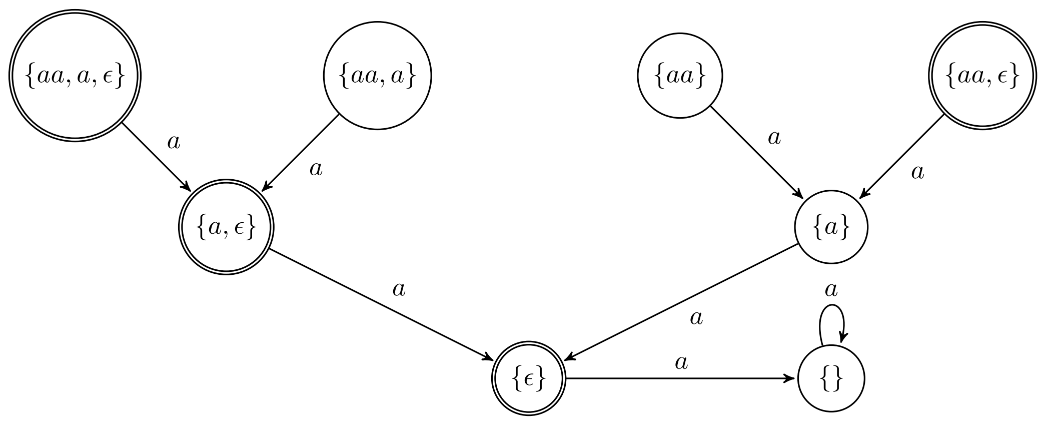
In this automaton, there is no initial state because we are only interested in the language a state accepts (and we obtain this from the final states). The set of states is now the set 2^{A^*}, the set of all languages over the set of words A^*.

The transition function will be defined as $\beta : 2^{A^*} \to (2^{A^*})^A$ with $\beta(L)(a) = \{w \in A^* | a \cdot w \in L\}$ where L is a language and a an input symbol in A.

In words, $\beta(L)(a)$ removes the first symbol on the left hand side of a word if it is not equal to a and keeps it if it is. Then the language we get will be all of the words in A^* that start with a, with a removed. This is determined by the transition function and is the **left a-derivative** of the language L, given a symbol $a \in A$.

The final state function, $\epsilon? : 2^{A^*} \to 2$ will return 1 if the given language contains the empty word, and 0 otherwise. The final states are the languages that contain ϵ, because by removing every symbol from a word, we show that the languages accept a word from the original automaton. They can contain other words too because although a final state in the original automaton accepts a word, it can have transitions to other states too and these words will not be accepted by the original automaton at this point.

We can demonstrate an example of this by creating this automaton for the automaton we defined to explain observability, $\mathcal{O}$. We set $n = 2$ (where n is the length of the longest word accepted by the automaton) to simplify the automaton:



In this automaton we give every unique language possible where $n = 2$. Then these languages are the states of our automaton. The final states are the languages that contain ϵ, as described above. Each state corresponds to a unique language, therefore we can see that the original automaton, $\mathcal{O}$ is observable.

Linking Reachability and Observability Then to link these automata together, we construct two functions, that show how an automaton X can be reachable and observable. These are r and o, respectively. Then we have the entire diagram, as before:

$$\begin{tikzcd}
1 \arrow[d, "\epsilon"] \arrow[dr, "i"] & & 2 \\
A^* \arrow[r, dotted, "r"] \arrow[d, "\alpha"] & X \arrow[ur, "f"] \arrow[r, dotted, "o"] \arrow[d, "t"] & 2^{A^*} \arrow[u, "\epsilon?"'] \arrow[d, "\beta"] \\
(A^*)^A \arrow[r, dotted, "r^A"] & X^A \arrow[r, dotted, "o^A"] & (2^{A^*})^A
\end{tikzcd}$$

r maps A^* to X. Given a word $w \in A^*$, it returns the state i_w, the state that is reached from the initial state $i \in X$ by following the transitions for each character of w in turn. We write this as $r(w) = i_w$. If r is **surjective** (r reaches every element of its codomain), then the automaton is reachable because for every state $x \in X$, there will be a word that takes us from i to that state. There may be more than one path from i to that state, so r may not be bijective (each element of the codomain of r does not have to correspond to a unique element of the domain)).

o maps X to 2^{A^*}. It maps a state x to all of the words accepted by it (the language of x) and is defined as $o = \{w \in A^* | f(x_w) = 1\}$. This will be all of the words where following a series of transitions on the symbols of that word from the given state x gets us to an accepting state.

x_w is the state that is obtained by applying a word $w \in A^*$ to a state $x \in X$. It is defined inductively, so an empty word, ϵ applied to a state, x is just x. Then when we want to add an arbitary symbol a to a non empty word w and apply this to x, it becomes the result of the transition function of the state produced by that word and the symbol a:

- $x_\epsilon = x$
- $x_{w \cdot a} = t(x_w)(a)$

If o is **injective**, then X will be observable because each state can only recognise one language. It may not be bijective because there may be some languages in 2^{A^*} that aren't recognised by any state.

An automaton is **minimal** if it is **both reachable and observable**, because all states must be reachable from the initial state and all states must recognise different languages, so there will be no redundant states.

The functions o and r make the diagram commute, so they are **homomorphisms**.

Reachability r is uniquely determined by i and t. This is because our reachability automaton is an initial algebra of the functor $1 + (A \times -)$, so is the F-algebra $F(A^*) = 1 + (A \times A^*)$ with carrier set A^*.

$$\begin{tikzcd}
1 \arrow[dr, "\epsilon"] & \\
& A \\
A \times A^* \arrow[ur, "\alpha"] &
\end{tikzcd}$$

:

This functor generalises the initial state and transition function of the reachability automaton, so we get an initial algebra on the carrier set A^* of possible words that can be formed by it.

We then give a morphism so that $F(A^*) \to A^*$, gives all the words obtained. The functor will combine the functions i and t, to give the transitions from the empty word (enabling us to generate A^* from an empty set and ensure that our reachability automaton gives an initial algebra). The functor will then be either 1 (the operation ϵ that gives us ϵ, the empty word, or $A \times A^*$, which generalises the transition function to one on the symbols involved, prepending the symbol in A given to the word we have created so far. Therefore we can see that we will have a morphism $X \to 1 + (A \times X)$, for any carrier set X.

This algebra can also be defined as a universal algebra, as $(A^*; \alpha, \epsilon)$ where A^* is the carrier set, and α and ϵ are functions that produce words in the set A^*, ϵ returning the initial state in the set A^* and α returning a word with additional symbol a, which will also be in A^*.

As we previously defined an automaton, partially as an F-algebra with the same functor as this, we can see that r will give a map from the reachability automaton to our original automaton.

Observablity o is uniquely determined by t and f, because 2^{A^*} is a final coalgebra of the functor $2 \times (-)^A$, which is the F-coalgebra $F(2^{A^*}) = 2 + ((2^{A^*})^A \times A^*)$.

This generalises the transition structure and final state function, so we get a final coalgebra on the carrier set 2^{A^*} of languages accepted by the automaton. We give a morphism so that $2^{A^*} \to F(2^{A^*})$ gives the languages obtained by all of the states in this automaton. Then the functor gives us the language accepted by a given state and whether or not it is a final state, so we form this as a product category, to ensure we have both of these things, giving us $F(X) = X^A \times 2$, for any carrier set, X.

As our observability automaton and the coalgebra of our original automaton are both defined by the same F-coalgebra, we can use o to give a unique map from the original automaton to our observability automaton.

2.4.4 Categories of Automata

Using these algebraic structures we can form a category of deterministic automata, **DA**, (as defined in [Bonchi et al., 2014]), where the objects are the automata and the morphisms are automata morphisms. The objects are defined as a combination of a $1 + (A \times -)$-algebra (initial state plus transition structure) and a $2 \times (-)^A$-coalgebra. An automata morphism is a function that respects both of these structures, so a morphism between 2 automata (X, t, i, f) and (Y, s, j, g) is a map $h : X \to Y$ such that the following diagram commutes:

$$\begin{tikzcd}
1 \arrow[d, "i"] \arrow[dr, "j"] & 2 \\
X \arrow[r, "h"] \arrow[ur, "f"] \arrow[d, "t"] & Y \arrow[u, "g"'] \arrow[d, "s"] \\
X^A \arrow[r, "h^A"] & Y^A
\end{tikzcd}$$

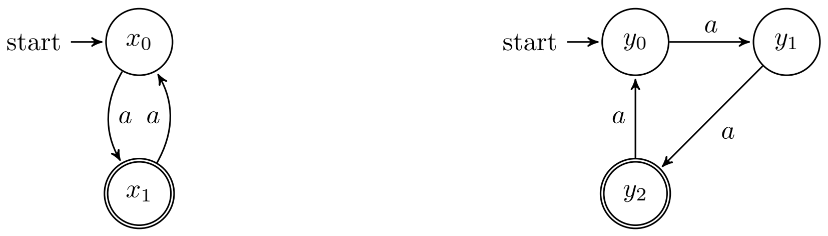
We don't show the maps between the corresponding 1 and 2 of the two automata because they are just identity maps.

Reachability will be a morphism between the algebras of the automata and observability will be a morphism between the coalgebras of the automata. Then h is the combination of these morphisms.

Automata morphisms must preserve the accepted language, so they are homomorphisms. Therefore there must be a unique mapping from a state in X to a state in Y. There are 3 properties that an automata morphism must satisfy. Given our 2 automata and a homomorphism h:

1. If x_0 is the initial state of X, then $h(x_0)$ is the initial state of Y
2. If there is a transition on a symbol a between x_n and x_{n+1} then there is also a transition from $h(x_n)$ to $h(x_{n+1})$ on a
3. If x is an accepting state of X, then $h(x)$ is an accepting state of Y

For example, given the following automata that do not accept the same language:



The first automaton accepts a language $L_1 = \{a, a^3, a^5, \dots\}$ and the second acccepts a language $L_2 = \{a^2, a^5, a^8, \dots\}$.

By the first property of automata homomorphisms, $h(x_0) = y_0$. Then by the second property, there is a transition in X from x_0 to x_1 on a, so $h(x_1)$ will be equal to the state reached by following a transition on a on y_0, which gives us y_1. However in the langauge accepted by X, this should be an accepting state. as $a \in L(X)$. By the third property, x_1 is an accepting state of X, so $h(x_1)$ is an accepting state of Y, but y_1 is not. Therefore we cannot construct a structure preserving map between these two automata and there is no automata morphism.

We can prove that in general automata morphisms preserve the given language:

Theorem 1. *If X and Y are deterministic automata and $h : X \to Y$ is an automata homomorphism, then X and Y accept the same language.*

Proof. Given a word $w \in A^*$, if it is in $L(X)$ (the language of X) it must be in $L(Y)$ and if it is in $L(Y)$ then it must be in $L(X)$. We can represent w using the symbols that compose it, so we have $w = a_1 a_2 \dots a_n$. Let the state set of X be $\{x_0 \dots x_n\}$, where x_0 is the intitial state and x_n is the final state and the state set of Y be $\{y_0 \dots y_n\}$ similarly.

Then to prove it in the first direction, we must prove that if a word is accepted by X, that word will also be accepted by Y. h will be defined from the initial state of X to the initial state of Y. In X we have a transition on a_1 from the initial state of X. By the second property of automata homomorphisms, as there is a transition from $x_0 \to x_1$ there must also be a transition on a_1 from $h(x_0)$ to $h(x_1)$. which will be $y_0 \to y_1$. Then we can do this for every symbol in w. Then we know that there is a homomorphism from x_n to y_n. By the third property of automata homomorphisms, there will be a morphism from the final state of X to the final state of Y and x_n is the final state of X, so y_n will be the final state of Y and there will be no more characters in w to make transitions on. Therefore Y also accepts w and $w \in L(Y)$.

Then to prove it in the other direction $w \in L(X)$, we can also use the homomorphism on the initial state to say that because there is a transition from y_0 to y_1 on a_1, there must have been a transition from x_0 to x_1. Then we can do this for every state of X and Y and each symbol in w until we get to a_n, which will be accepted by y_n, so there is a homomorphism from $x_n \to y_n$. By the third property of automata homomorphisms, $h(x_n) = y_n$ is an accept state so x_n is an accept state. Therefore X accepts the word w so $w \in L(X)$. $\square$

Chapter 3

Brzozowski's Algorithm

Brzozowski's Algorithm, as defined in [Brzozowski, 1962] is an algorithm for automata minimisation based on reversing the automaton and determinising it (usually by powerset construction). In [Bonchi et al., 2014], a simple description of this algorithm for deterministic automata is given:

Definition 1. ***Brzozowski's Algorithm*** *Given an automaton accepting a language L:*

1. *Construct the reverse of the automaton*
2. *Determinise it and take the reachable part*
3. *Construct the reverse of the automaton again*
4. *Determinise it and take the reachable part*

Then the automaton we create is the minimal automaton that accepts L

The reverse of the automaton is the automaton obtained by making the final states initial (and vice versa) and reversing the direction of the transitions. For example, given a transition defined on a state x to state y on the symbol a we have $t(x)(a) = y$, which will become $t'(y)(a) = x$ in our new transition function, which is denoted by t'.

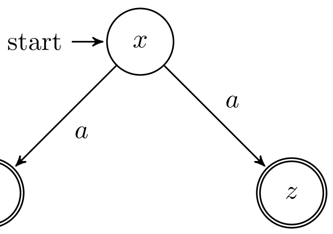
The reachable part is all the states and transitions that can be reached from the initial state. If the automaton obtained by reversing an automaton is non-deterministic, or we want to start with a non deterministic automaton, we must determinise it first. We can do this using the powerset construction.

Reversing an automaton in this way could give us a situation where we have more than one initial state. We can solve this by adding ϵ-moves to the beginning of the automaton or by taking both of the initial states as the initial state of our new automaton when we do the powerset construction.

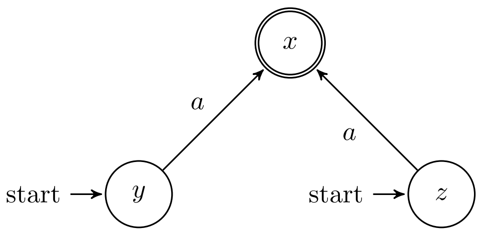
3.1 Powerset Construction

The powerset construction determinises an automaton by creating a new automaton, where the set of states is now the powerset of the state set given. Then the transitions are mapped to every state that they are applicable to. For example, if you had a transition $t(x)(a) = y$, then for every state that contains x there will be a transtition on a to every state that contains y.

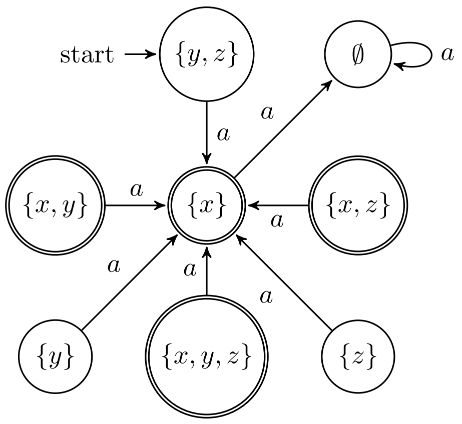
Given the following automaton:



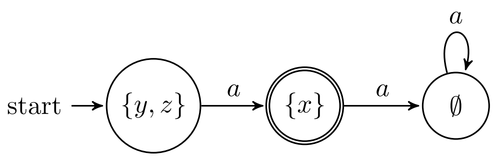
We reverse it to get the following automaton.



The powerset of our state set will be the new set $X = \{\theta, \{x\}, \{y\}, \{z\}, \{x, y\}, \{y, z\}, \{x, z\}, \{x, y, z\}\}$. In our reversed automaton, y and z are the initial states, so the state $\{y, z\}$ is the inital state of our powerset constructed automaton. The final state of our reversed automaton is x , so the final states are all the states that contain x : $\{x\}, \{x, y\}, \{x, z\}$ and $\{x, y, z\}$. We construct the automaton for this set by mapping the original transitions to these states, so we have:

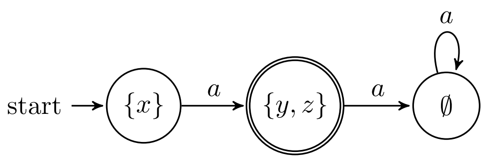


Now we take the reachable part. We remove all of the nodes apart from $\{y, z\}$ and $\{x\}$ because they cannot be reached from the initial state. Now we have the automaton:



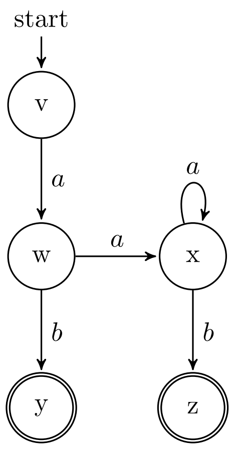
We include the error state, even though it does not lead to an accepting state, to ensure that our automaton is still determinstic, because for an automaton to be deterministic, there must be **exactly one** transition from a state on each symbol.

We have now reversed an automaton, determinised it (by using powerset construction) and taken the reachable part. To complete Brzozowski's algorithm we need to do this process again. However, in this case, we can see that this automaton is already miniminal, so reversing and determinising it would have no effect apart from giving us an automaton for the original language, not the reversed lanaguage. Therefore we get:

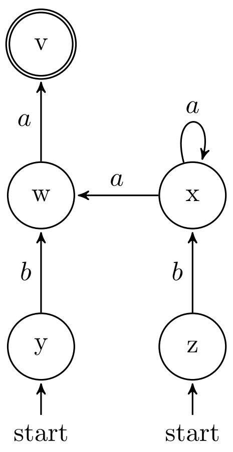


3.2 Example of Brzozowski's Algorithm

To give an example of the whole algorithm, consider this (non-minimal) automaton that accepts the language $a(a^*)b$:



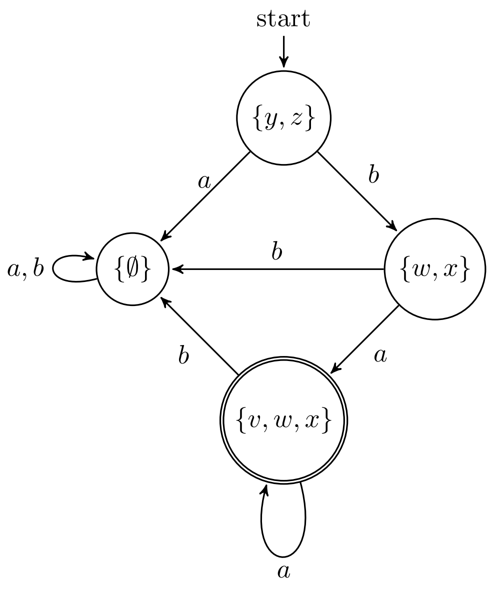
Reversing this automaton we get:



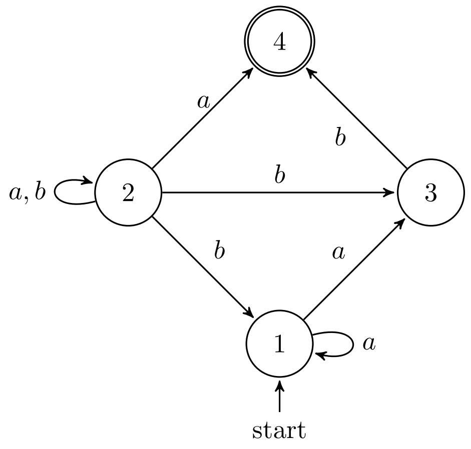
Then we can see that this automaton is non-deterministic, as it has two start states, so we determinise it using the powerset construction. As this automaton has 4 states, the powerset constructed automaton will have 16 states, making it very complex to draw, Therefore, we can give a table of the transitions and use this to construct the reachable part of the powerset automaton:

state	a	b
y	-	{w}
z	-	{x}
w	{v}	-
x	{w,x}	-
v	-	-

Then the start state of our automaton will be the combination of the start states in the reversed algorithm, $\{y, z\}$ and we get the results of all the transitions on both of those states from the table, so this will be $\{\emptyset\}$ for a and $\{w, x\}$ for b. Then we do the same for those states and so on until there are no more transitions from the states we have, This gives us the automaton:



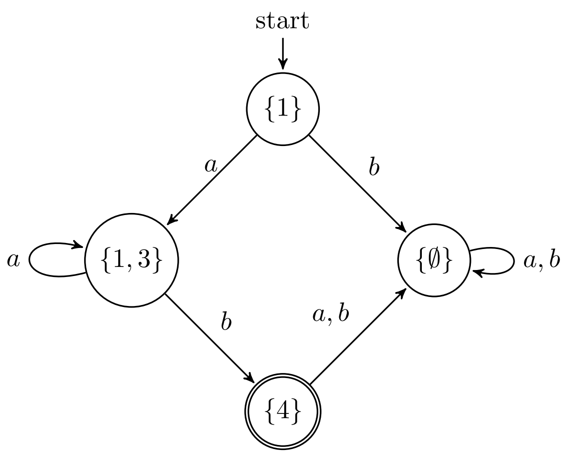
This automaton is minimal but accepts the reverse language, $b(a^*)a$. Therefore we go through the same process again. First we reverse this automaton (renaming the states to simplify the operation):



Then we determinise this by constructing a table as before:

state	a	b
1	{1,3}	-
2	{2,4}	{1,2,3}
3	-	{4}
4	-	-

and then form the automaton, taking {1} as the start state and any states reachable from it that contain 4 as the final states. This gives us:



This is the minimal deterministic automaton, it cannot be reduced any further. and still have exactly one transition from each state on each symbol.

3.3 Proof of Correctness

A proof that this algorithm is correct is found in [Sakarovitch, 2009]. The algorithm is a corollary of the following theorem on automata:

Theorem 2 (Sakarovitch)**.** *The determinisation of a co-deterministic, co-reachable automaton which recognises a language L is the minimal automaton of L.*

This proof is based on some new terminology that we must define:

3.3.1 Co-reachable and Co-deterministic Automata

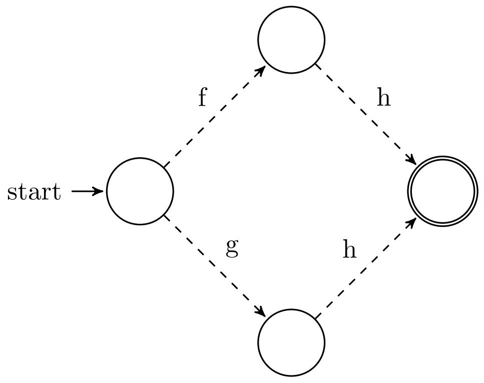
Co-reachable (defined as *co-accessible* in [Sakarovitch, 2009]) is the dual of reachable, so a state q is reachable from a state p if there is a transition from p to it. Then it is reachable if it is reachable from the initial state . Therefore the opposite of this is that a state p is co-reachable from q if there is a transition from p to q. So p will be co-reachable if it is co-reachable to a final state. For an automaton M to be reachable, it must be the case that all of its states are reachable. Then M is co-reachable if all of its states are co-reachable.

Codeterministic An automaton is co-deterministic (as defined in [Sakarovitch, 2009]), if the reversal of M, M^R (which Sakarovitch calls the transpose of M), is deterministic. So for all $q \in X$ and for all $a \in A$, there will be at most one transition to q on a and there is a unique final state.

Now we can consider the proof. This proof is given in [Sakarovitch, 2009], but here we use our notation and explain how the proof is constructed in more detail and diagrammatically:

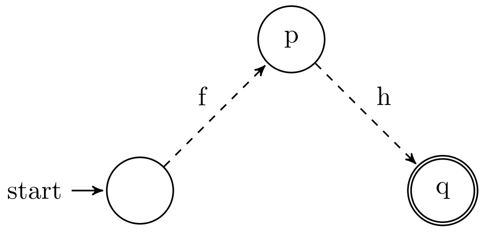
Proof. Let M be any co-deterministic, co-reachable automaton which accepts a language L and M_{det} be its determinisation by the powerset construction.

Then let $f, g \in A^*$ be two words in L such that the left f derivative of L is equal to the left g derivative of L. The left f derivative is $\{h \in A^* | f \cdot h \in L\}$, so it is all of the words in L that start with the word f. We can represent this as follows, the dotted lines representing potentially more than one transition:

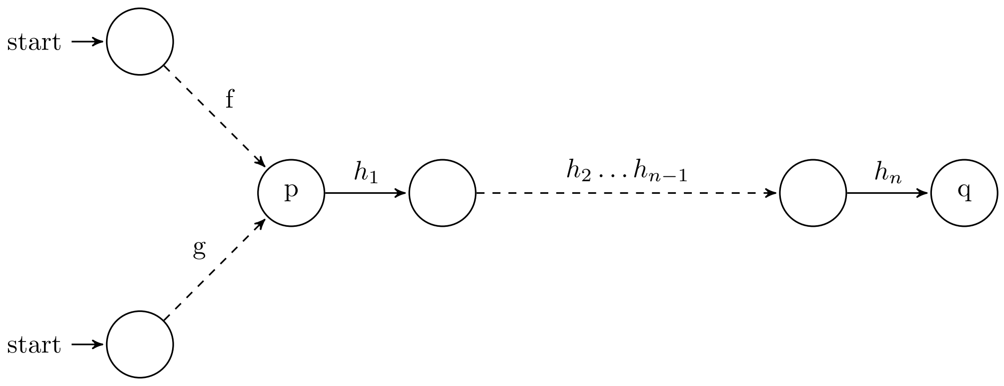


If we have this then we can show that $t(i)(f) = t(i)(g)$, so the state reached by following transitions on each symbol of f will be equivalent to the state reached by following the transitions on the symbols of g. Therefore when we determinise M to get M_{det} any states where we have the above situation for any words f and g in A^* will become a single state, and the states of M_{det} will be in bijection with those of the minimal automaton (there is a one to one correspondence between the states of M_{det} and the minimal automaton), satisfying our theorem.

To show this, let p be a member of $t(i)(f)$ (M is not assumed to be deterministic, so we can have multiple transitions on a given state and symbol, so the result of t is now a set). Then because M is co-reachable, we can get to an accepting state from p, which we will call q. There will exist a word h such that q (the final state) is in $t(p)(h)$ and $fh \in L$:



Then by our hypothesis, it is also the case that $fh \in L = gh \in L$ and $q \in t(i)(gh)$. Because M is co-deterministic, by definition, there is only one transition on h from p to q, because there is only one transition from the final symbol of h to q, Then for the state that accepts the final symbol of h, there will be only one transition on the penultimate symbol of h and so on until all the symbols of h have been used:



Now we have $p \in t(i)(g)$, but p is also in $t(i)(f)$, therefore, the states in $t(i)(g)$ must be a subset or equal to $(t)(i)(f)$. Symmetrically, it is the case that $t(i)(g) \subseteq t(i)(f)$. Therefore is must be the case that $t(i)(f) = t(i)(g)$. There will now only be one state in each of these sets so we have determinised them and got the statement we want to prove, so the states of M_{det} will be in bijection with the states of the mininal automaton.

□

Brzozowski's Algorithm is then a corollary of this proof, as in [Sakarovitch, 2009]:

Corollary 1 (Sakarovitch)**.** *Let M be an automaton and $L = L(M)$ be the language that it recognises. Then $A_L = (((A^R)_{det})^R)_{det}$*

Proof. We begin with the automaton M that recognises L. Then, we can compute the determinisation of the reversal of M, $(M^R)_{det}$.

By definition, $(M^R)_{det}$ is reachable, so its reversal, $((M^R)_{det})^R)$ will be co-reachable, because the final state and start states have swapped, so now we want to know that all states have a transition to the final state.

Then since $(M^R)_{det}$ is equivalent to (accepts the same language) as (M^R) and the reversal of $(M^R)_{det}$ will be co-deterministic, we get $M_{cod} = ((M^R)_{det})^R)$ which is equivalent to M (because $((M^R)_{det})^R)$ accepts the same language as $((M^R)^R$, which accepts the same language as M).

Determinising this gives us

$$((((M^R)_{det})^R)_{det}) = (M_{cod})_{det}$$

which is the minimal automaton. We can see this is true using the theorem with M_{cod} to obtain its determinisation, as it is a co-reachable, co-deteministic automaton. □

This proof gives the correctness of the algorithm, but is restricted to deterministic automata. Therefore the coalgebraic proof gives us the benefit of being able to prove the correctness of Brzozowski's algorithm for other types of automata too. Now we will be able to see the difference between the above proof and the coalgebraic proof.

Chapter 4

Brzozowski's Algorithm Coalgebraically

4.1 Contravariant Powerset Functor

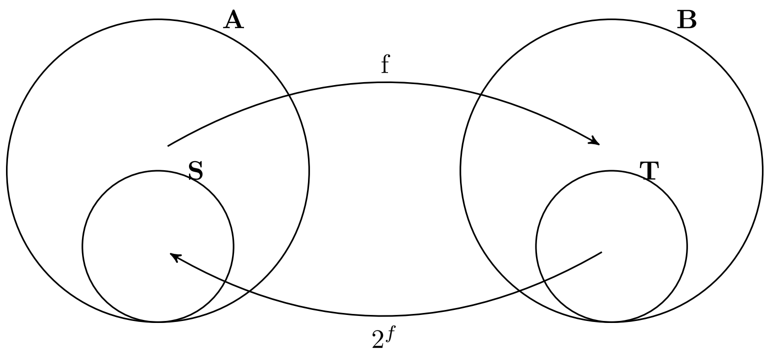
To express Brzozowski's Algorithm, we need to explain another functor that we use in the proof. This is the *contravariant powerset functor*. A contravariant functor reverses the direction of the morphisms. Given a morphism $f : A \to B$, the functor gives us $F(f) : F(B) \to F(A)$.

A powerset functor maps sets and the functions between them to the powersets of the sets and the morphisms between them. Therefore the functor will map the sets in a powerset of one set to the sets in a powerset of another set. Given a morphism $f : A \to B$ and a subset S of A, then the functor gives $f(S) \subseteq B$, a subset of B obtained as the image of f on A.

Combining both of these functors gives us the contravariant powerset functor, $2^{(-)} : Set \to Set$ (where the $^-$ can be substituted for any set). It is defined in [Bonchi et al., 2014] as :

- Given any set A, then $2^A = \{S \mid S \subseteq A\}$ which is the combination of all of the subsets of V (so S is any subset of V)
- Given a morphism $f : A \to B$, for every $T \subseteq B$ (any subset of B), $2^f : 2^B \to 2^A$ is defined as $2^f(T) = \{a \in A | f(a) \in T\}$. This says for any subset T of B we get a a set of elements in A for which the images will be in the subset T of B after applying the original morphism f. Equivalently we can say that 2^f gives the inverse image of $T \subseteq B$.

  Diagrammatically we have:



Diagrammatically, we can illustrate that given $f : A \to B$:

$$\begin{tikzcd} A \arrow[r, "f"] & B \end{tikzcd}$$

Applying the contravariant powerset functor to the function f and the sets V and W gives us:

$$\begin{tikzcd} 2^B \arrow[r, "2^{(-)}"] & 2^A \end{tikzcd}$$

4.2 Constructing the Reverse Automaton

We can now use the functor we just described to demonstrate the construction of the reverse automaton of an automaton (X, i, t, f) that is applied in [Bonchi et al., 2014]:

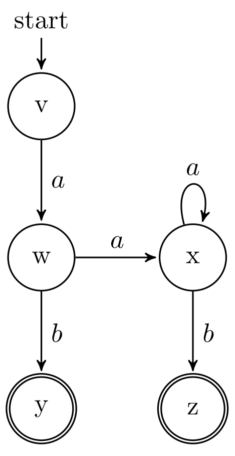
4.2.1 Reverse of Transition Function

We start with the transition function $t : X \to X^A$ and apply the contravariant powerset functor in three steps:

1. Before applying $2^{(-)}$, we uncurry the function, so we can get the appropriate sets for the powerset functor. Given $t(x)(a)$ we now have $t(x, a)$, giving the parameters as a pair. Therefore we get $X \times A \to X$ from $X \to X^A$
2. Now we apply $2^{(-)}$ to this new function to get $2^{(X \times A)} \to 2^X$
3. Then we curry this again to get the new transition function, $2^t : (2^X)^A \to 2^X$ from $2^{(X\times A)} \to 2^X$. Then we have $2^t(S)(a) = \{x \in X | t(x)(a) \in S\}$

Our new transition function takes S, a subset of the state set X and an input symbol a, and returns the set of states where appending a gives a state in S, as oppososed to our previous transition function, where we give a state and a symbol a and return the state obtained by following a transition on a.

We will show this on the following automaton, which is our example from before:



The original transition function for this automaton is:

$$\begin{aligned}
t(v)(a) &= w \\
t(w)(a) &= x \\
t(x)(a) &= x \\
t(w)(b) &= y \\
t(x)(b) &= z \\
t(\_)(\_) &= \emptyset
\end{aligned}$$

any other states and symbols will just be $\emptyset$ as we will have an error state.

Then 2^t will be the will be defined on each state set and each symbol of a. Because there are 5 states in our automaton, there will be $2^5 = 32$ different transitions, so we will just give a sample of them below:

$$\begin{array}{ll}
2^t(\emptyset)(a) = \emptyset & 2^t(\emptyset)(b) = \emptyset \\
2^t(\{v\})(a) = \{w\} & 2^t(\{x\})(b) = \emptyset \\
2^t(\{w\})(a) = \{x\} & 2^t(\{w\})(b) = y \\
2^t(\{x\})(a) = \{x\} & 2^t(\{x\})(b) = \{z\} \\
2^t(\{y\})(a) = \emptyset & 2^t(\{z\})(b) = \emptyset \\
2^t(\{y\})(a) = \emptyset & 2^t(\{z\})(b) = \emptyset \\
2^t(\{v,w\})(a) = \{w,x\} & 2^t(\{v,w\})(b) = \emptyset \\
2^t(\{v,x\})(a) = \{w,x\} & 2^t(\{v,x\})(b) = \{z\} \\
2^t(\{v,y\})(a) = \{w\} & 2^t(\{v,y\})(b) = \emptyset \\
2^t(\{v,z\})(a) = \{x,y\} & 2^t(\{v,z\})(b) = \emptyset \\
2^t(\{w,x\})(a) = \{x\} & 2^t(\{w,x\})(b) = \{y\} \\
2^t(\{w,y\})(a) = \{x\} & 2^t(\{w,y\})(b) = \{y\} \\
2^t(\{w,z\})(a) = \{x\} & 2^t(\{w,z\})(b) = \{y\} \\
2^t(\{x,y\})(a) = \{x\} & 2^t(\{x,y\})(b) = \{z\} \\
2^t(\{y,z\})(a) = \emptyset & 2^t(\{y,z\})(b) = \emptyset \\
2^t(\{v,w,x\})(a) = \{w,x\} & 2^t(\{v,w,x\})(b) = \{y,z\} \\
 & \vdots
\end{array}$$

4.2.2 Switching Initial and Final states

We can apply $2^{(-)}$ to the initial state function, $i : 1 \to X$ of our automaton, giving us the function $2^i : 2^X \to 2^1$. Applying the contravariant powerset functor to 1 gives us a set with two elements, $\{\emptyset, 1\}$, which can be rewritten as 2. This means that overall we have a function from our new state set to the set of final states, so this will be the final state function of our new automaton.

In our example, the initial state is $i(*) = x$, The new final state function will return 1 for any set of states that include y or z. These states will include:

$$\begin{aligned}
&2^i(\{y\}) = 1\\
&2^i(\{z\}) = 1\\
&2^i(\{v, y\}) = 1\\
&2^i(\{v, z\}) = 1\\
&2^i(\{w, y\}) = 1\\
&2^i(\{w, z\}) = 1\\
&2^i(\{x, y\}) = 1\\
&2^i(\{x, z\}) = 1\\
&2^i(\{y, z\}) = 1\\
&2^i(\{v, w, y\}) = 1\\
&2^i(\{v, w, z\}) = 1\\
&\vdots
\end{aligned}$$

Then any other state will just be $2^i(\_) = 0$.

We can apply $2^{(-)}$ to the final state function, $f : 2 \to X$, giving us $2^f : 2^2 \to 2^X$, but this would not give us the initial state function, as we need a function from 1 and this is not the powerset of 2. Instead we obtain the initial state function by currying, getting $1 \to 2^X$, defined as $S \in 2^X$, where S is a subset of states. In the original f we give a state and return a different value depending on if it is final or not, but now we return the set of states that are final.

In our example, the final state function is:

$$\begin{aligned}
&f(v) = 0\\
&f(w) = 0\\
&f(x) = 0\\
&f(y) = 1\\
&f(z) = 1
\end{aligned}$$

So the new initial state function will be the set of the final states of the reversed automaton, so $f = \{y, z\}$.

4.2.3 Converting Reachability to Observability

For our original automaton $(X, t, i.f)$, we modelled reachability and observability by giving an automaton for each of these properties. Reachability was modelled using an automaton based on the initial state of the function, but now we have converted it to a final state. Observability was modelled using an automaton based on the final state. Therefore we will have to convert our automaton for reachability into one for observability.

To begin to change reachability to observability, we apply $2^{(-)}$ to the entire reachability automaton and to the function that relates reachability to the rest of the automaton.

In our example automaton, our original reachability function, r, maps the words in A^* to the state that is reached by applying that word to the initial state, so we have:

$$\begin{aligned}
&r(\epsilon) = v\\
&r(a) = w\\
&r(ab) = y\\
&r(aa) = w\\
&r(\text{a string of } a \text{ of length} \geq 3) = y\\
&r(\text{a string of } a \text{ of length} \geq 3 \text{ followed by a } b) = y
\end{aligned}$$

This is slightly incorrect as we should give each word individually, but here we give all of the possible results of r to simplify the function. Then $2^r : 2^X \to 2^{A^*}$ returns all the words where applying a word to the initial state gets us to the given state:

$$\begin{aligned}
&2^r(\{w\}) = \{\epsilon\}\\
&2^r(\{v\}) = \{a\}\\
&2^r(\{x\}) = \{a^n | n \geq 1\}\\
&2^r(\{y\}) = \{ab\}\\
&2^r(\{v\}) = \{a^n b | n \geq 1\}\\
&2^r(\{v, w\}) = \{\epsilon, a\}\\
&2^r(\{v, x\}) = \{\epsilon\} \cup \{a^n | n \geq 1\}\\
&2^r(\{v, y\}) = \{\epsilon, ab\}\\
&2^r(\{v, z\}) = \{\epsilon\} \cup \{a^n b | n \geq 1\}\\
&\vdots
\end{aligned}$$

and $2^r(\emptyset)$ will give any other words that can be formed from the input set that are not accepted by the automaton.

(For observability, we want the reverse words of these languages, as the words in the original language where following transitions from the initial state to get to a certain state will be the same as the reverse words where following the transitions from that state on that word will get to the final state.)

We apply $2^{(-)}$ to the entire left hand side of the diagram we drew previously to show reachability and observability on our automaton:

$$\begin{tikzcd}
 & 2 \\
2^X \arrow[ur, "2^i"] \arrow[r, "2^r"] \arrow[d, "2^t"] & 2^{A^*} \arrow[u, "2^\epsilon"'] \arrow[d, "2^\alpha"] \\
(2^X)^A \arrow[r, "2^{r^A}"] & (2^{A^*})^A
\end{tikzcd}$$

The initial state function is replaced with the final state function, as we previously defined, and the functor $2^{(-)}$ is applied to the functions ϵ and α of the reachability automaton and we give our new transition function, 2^t.

To apply the functor to α in the same way that we did for t, first uncurrying the function to give us $A^* \times A \to A^*$. Then we apply $2^{(-)}$ to this, giving us $2^\alpha : 2^{A^*} \to 2^{A^* \times A}$. Finally we curry this to get $(2^{A^*}) \to (2^{A^*})^A$, defined as $2^\alpha(L)(a) = \{w \in A^* | \cdot a \in L\}$.

In our example, our input alphabet is $\{a, b\}$ so α is:

$$\begin{array}{ll}
\alpha(\epsilon)(a) = a & \alpha(\epsilon)(b) = b \\
\alpha(a)(a) = aa & \alpha(b)(a) = ba \\
\vdots & \vdots
\end{array}$$

and so on, for every symbol and every possible combination of a and b, (so there are an infinite number of values of α as it is not restricted by a language). Then 2^α will be defined for our input alphabet, but is defined for any language in 2^{A^*}, so the states of our automaton do not affect it, and like α, there are an infinite number of values for it.

Applying the functor to ϵ gives us $2^\epsilon : 2^{A^*} \to 2$, which maps a language to 1 if it contains ϵ, otherwise it maps it to 0. This is equivalent to ϵ in the definiton of the original automaton, as they both do the same thing.

It will be the case that $2^\epsilon(L) = \epsilon?(L)$. The function 2^α is almost equivalent to β, but uses $w \cdot a$ instead of $a \cdot w$, so $2^\alpha(L)(a) = \{w \in A^* | w \cdot a \in L\}$.

Connecting the new automaton with observability

Our function $2^{\alpha}(L)(a)$, accepts the right (a) derivatives of a language L. To get a function that is the same as β, we want the left derivatives. $2^{\alpha}(L)(a)$ is equivalently accepts the left (a) derivative of the reverse language $rev(L)$, so by composing the function rev (which reverses a given language) and 2^{α}, we will be able to get the left (a) derivative.

Theorem 3. *For any language L and any $w \in A^*$, $rev^A \circ 2^{\alpha} = \beta \circ rev$*

Proof. Let a and b be symbols in A and w be a word in A^* such that bwa is a word in L. Then if we apply rev to L, $rev(L)$ will contain the word awb. Applying $\beta(L)(a)$ to this gives us the left derivative, which is wb. Then $2^{\alpha}(L)$ will give the right derivative of bwa, which is bw. Then rev^A reverses this derivative, to give us wb. This is equal to the result of $\beta(L)(a)bwa$ so the functions give the same result. □

Using this, we get the following diagram:

$$\begin{tikzcd}
 & 2 \\
2^{A^*} \arrow[ur, "2^{\epsilon}"] \arrow[r, "rev"] \arrow[d, "2^{\alpha}"] & 2^{A^*} \arrow[u, "2^{\epsilon ?}"] \arrow[d, "2^{\beta}"] \\
(2^{A*})^A \arrow[r, "rev^A"] & (2^{A^*})^A
\end{tikzcd}$$

If we compose this diagram with the reachabilty diagram we obtained by using the contravariant powerset functor, we will get a commuting diagram:

$$\begin{tikzcd}
 & & 2 \\
2^X \arrow[rru, "2^i"] \arrow[r, "2^r"] \arrow[d, "2^t"] & 2^{A^*} \arrow[ru, "2^\epsilon"] \arrow[r, "rev"] \arrow[d, "2^\alpha"] & 2^{A^*} \arrow[u, "2^{A^*}"] \arrow[d, "\beta"] \\
(2^X)^A \arrow[r, "2^{r^A}"] & 2^{A^*A} \arrow[r, "rev^A"] & (2^{A^*})^A
\end{tikzcd}$$

We can simplify this to show that observability , $O = rev \circ 2^r$, is the composition of the reverse language and 2^r, the reversed reachability function (which now maps a state in the reverse automaton to the language it accepts). Diagrammatically we have:

$$\begin{tikzcd}
 & 2 \\
2^X \arrow[ru, "2^i"] \arrow[r, "O"] \arrow[d, "2^t"] & 2^{A^*} \arrow[u, "2^{\epsilon?}"] \arrow[d, "2^\beta"] \\
(2^X)^A \arrow[r, "2^{O^A}"] & (2^{A^*})^A
\end{tikzcd}$$

Then for any $S \subseteq X$, (so any subset of states from the original state set), $O(S) = \{w^R \in A^* | i_w \in S\}$. This gives the set of all reversed words in A^* where the original word is accepted by a state in S. If O is injective, then our reversed automaton is observable. Each set of states will have a unique set of reversed words where the original automaton accepted the original word, which means that in the reversed automaton each state will accept a different reverse language.

We can show that the diagram above should commute:

Theorem 4. *For any language L and any symbol $a \in A$, $\beta \circ O = O^A \circ 2^t$*

Proof. $O(S)$ gives us all of the reversed words of the words that when applied to the initial state give states in the subset of states S. Then applying $\beta(L)(a)$ gives us the left (a) derivative of this set. Given a word $w \cdot a \in L$, then $a \cdot w$ will be in $O(S)$ and w will be in $\beta(L)(a) \circ O(S)$.

$2^t(S)(a)$ gives us the states where following a transition on a gives a state in S. If a state $x \in S$ accepts w, then there will be a state $y \in 2^t(S)(a)$ that accepts the word $w \cdot a$. Then O^A is the observability function on a word $w \cdot a$ without the final symbol a. Applying the word w to the initial state gives us the state x, so O^A returns the reversed word w. This is the same as the result of $\beta(L)(a) \circ O(S)$, so the result of both of these operations is equal. $\square$

Applying O to an example automaton will give:

$$\begin{aligned}
&2^r(\{w\}) = \{\epsilon\}\\
&2^r(\{v\}) = \{a\}\\
&2^r(\{x\}) = \{a^n | n \geq 1\}\\
&2^r(\{y\}) = \{ba\}\\
&2^r(\{v\}) = \{ba^n | n \geq 1\}\\
&2^r(\{v, w\}) = \{\epsilon, a\}\\
&2^r(\{v, x\}) = \{\epsilon\} \cup \{a^n | n \geq 1\}\\
&2^r(\{v, y\}) = \{\epsilon, ba\}\\
&2^r(\{v, z\}) = \{\epsilon\} \cup \{ba^n | n \geq 1\}\\
&\vdots
\end{aligned}$$

and $O(\emptyset)$ is all of the words formed by the input alphabet that are not part of the reverse lanaguage. Now we can easily see, by looking at our ouput for 2^r, that the output of O is just the reverse language it.

4.2.4 Converting Observability to Reachability

We converted the final state function into an initial state function and the initial state function is used in the definition of the automaton modelling reachability for our original automaton. This is equivalently the initial algebra $(A^*; \epsilon, \alpha)$, so our new final state function can be used to form the initial algebra $(2^X; f, 2^t)$. This algebra will have the same signature as the initial algebra representing reachability as our new version of f is the same as i, and α and 2^t are the same functions on different carrier sets, so there will exist a unique morphism between them both of these algebras. This morphism will be $R : A^* \to 2^X$ and will demonstrate reachability, as it maps a word in A^* to the set of states that accepts that word, as reachability in the original automaton mapped a word in A^* to the state in X accepting that word.

For our example automaton, to construct R, we give the reachability function for the new automaton as follows:

$$\begin{aligned}
&R(\epsilon) = \{y, z\}\\
&R(b) = \{w, x\}\\
&R(ba \text{ followed by any number of } a) = \{v, w, x\}\\
&R(\_) = \emptyset
\end{aligned}$$

As with r before, this is slightly incorrect as we should give each word individually, but here we give all of the possible results of r to simplify the function.

This will give all of the states in the reachable part that accept words formed by the input alphabet of our original automaton. It is not defined for every state in the powerset constructed automaton, so is not surjective, so we can see that the whole automaton is not reachable.

Then we can see that we can get from the initial state set to any other state set, so our new automaton will be reachable, as the morphism is surjective. We show this diagrammatically as:

$$\begin{tikzcd}
1 \arrow[d, "\epsilon"] \arrow[dr, "f"] & \\
A^* \arrow[r, "R"] \arrow[d, "\alpha"] & 2^X \arrow[d, "2^t"] \\
(A^*)^A \arrow[r, "R^A"] & (2^X)^A
\end{tikzcd}$$

We can prove that this diagram commutes:

Theorem 5. *Let w be a word in A^* and a be a symbol in a such that $2^t \circ R = R^A \circ \alpha$*

Proof. Applying α to w and a gives us the word $w \cdot a$. Then applying R^A to this is the same as applying R to the original word w, so gives us the set of states reached by following transitions on w from the initial state, S_w.

Applying R to the word $w \cdot a$ gives us the set of states $S_{w \cdot a}$. Then $2^t(S_{w \cdot a})$ gives us the set of states obtained by following a transition on any state in the set on a gives us $S_{w \cdot a}$. This set will be S_w, which is the same as what we get from $R^A \circ \alpha$, so the operations are equal. □

Now we have everything we need for our new automaton:

$$\begin{tikzcd}
1 \arrow[d, "\epsilon"] \arrow[dr, "f"] & & 2 \\
A^* \arrow[r, dotted, "R"] \arrow[d, "\alpha"] & 2^X \arrow[ur, "2^i"] \arrow[r, dotted, "O"] \arrow[d, "2^t"] & 2^{A^*} \arrow[u, "\epsilon?"'] \arrow[d, "\beta"] \\
(A^*)^A \arrow[r, dotted, "R^A"] & (2^X)^A \arrow[r, dotted, "O^A"] & (2^{A^*})^A
\end{tikzcd}$$

and the new automaton is defined as a 4-tuple $(2^X, 2^t, f, 2^i)$ and an input alphabet A, where:

- 2^X is the set of states
- 2^t is the transition function, $2^t : 2^X \to 2^{X^A}$, which maps a subset of states $S \in 2^X$ to a function $2^t(S)(a) : 2^X \to 2^{X^A}$. Given a set of states, S and a symbol $a \in A$, this function gives us the set of all the states that accept the same words but with a removed from the end
- f is an initial state, so is a subset of states state $S \in 2^X$ (or equivalently a function $i : 1 \to 2^X$, where $1 = \{0\}$ is a set with one element). As there is only one initial state, this function will return the initial state, given the element 0
- 2^i is a set of final states (represented as a function $2^i : 2^X \to 2$ where $2 = \{0, 1\}$. Then $2^i(S) = 1$ means that a subset of states, S, is a final state and $f(S) = 0$ means it is not final).

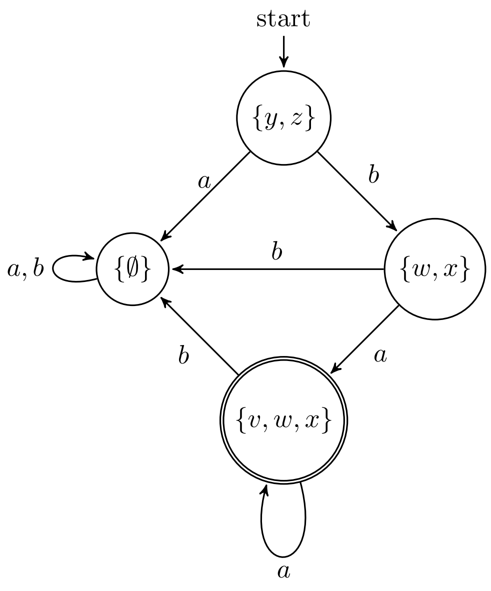
This is represented diagrammatically as:

$$\begin{tikzcd}
1 \arrow[dr, "f"] & & 2 \\
& 2^X \arrow[ur, "2^i"] \arrow[d, "2^t"] & \\
& 2^{X^A} &
\end{tikzcd}$$

Then for our example we have:

- The set of states $2^X = \{\emptyset, \{x\}, \{y\}, \{z\}, \{x, y\}, \{x, z\}, \{y, z\}, \{x, y, z\}\}$
- 2^t is the transition function, as previously defined.
- f is the initial state as previously defined.
- 2^i is the set of final states as previously defined.

The automata diagram of this from our example will contain 32 states and be very complicated to represent, so taking the reachable part of this gives us the following:



Which will be the minimal deterministic automaton that recognises the reverse language of our automaton, because we have taken only the states that can be reached from the inital state, so R is now surjective and our new automaton is both reachable and observable and therefore reachable.

4.3 Getting Brzozowski's Algorithm

Brzozowski's Algorithm is the corollary of the proof of the combination of two theorems given in [Bonchi et al., 2014]:

Theorem 6 (Bonchi). *Let $\mathcal{X} = (X, t, i, f)$ be a determinsitic automaton and $\mathcal{R} = (2^X, 2^t, f, 2^i)$ be the reverse deterministic automaton of that automaton. If $\mathcal{X}$ is reachable, then $\mathcal{R}$ is observable and if $\mathcal{X}$ accepts a language L, then $\mathcal{R}$ accepts the reverse langauge $rev(L)$.*

We need to prove two lemmas to complete the proof:

Lemma 1. *If a function $f : A \to B$ is surjective, then applying the contarvariant powerset functor to it will give the injective function $2^f : 2^B \to 2^A$.*

Proof. f will be defined for every $b \in B$. Therefore the inverse of f, $f^{-1} : B \to A$ is defined for all $b \in B$. Applying $2^{(-)}$ to f gives us $2^f : 2^B \to 2^A$, so this will be defined for all $S \in 2^B$, where S is a subset of B. For $2^f : 2^B \to 2^A$ to be injective, every S must map to a *unique* subset of A. We can prove this by contradiction: assume there are two subsets of B that map to the same subset of A.

If the subset of A is $\emptyset$ then the only subset of B that can be mapped to this is $\emptyset$, by the definition of surjectivity (because the original function f would have to be defined for every element of B, so every other subset of B will be mapped to some non-empty subset of A).

If the subset of A is non-empty, and more than one subset of B maps to it, then the original function f on each element of the subset of A will map to every value in the subsets of B so will also not be a function as these maps will be many to one relations. $\square$

Lemma 2. *rev is a bijective function*

Proof. rev must be surjective, so it must be the case that for a set of languages 2^{A^*}, every reversed language L^R in that set is the result of $rev(L)$, where L is a language in 2^{A^*}. We can prove this by induction on the set of languages.

The empty language $\{\}$ will be the result of $rev(\{\})$ and the language that just contains ϵ will be the result of $rev(\{\epsilon\})$. Then for any non empty language, we have a set of words with at least one symbol, and/or ϵ. A reversed word with one symbol is exactly the same as the original word, as there is nothing to reverse except itself.

To show that any reversed word accepted by L^R with more than one symbol is the result of reversing the word accepted by L, we can form the word by giving the right and left derivatives, respectively. Given a language containing the one symbol word w, we get the language that contains the word obtained by prepending a symbol a to w by $\beta(L)(a) = \{w' \in A^* | a \cdot w' \in L\}$. Then the reverse of this word will be $w \cdot a$. We can get this by getting the right derivate of L, using $2^{\alpha}(L)(a) = \{w^{R'} \in A^* | w^{R'} \cdot a \in L\}$. Therefore wa is a word in the reverse language and we obtain it by reversing the language containing aw. We can then repeatedly do the same thing again with any symbol s (including the one we just used) to get was and saw, then $wass'$ and $s'saw$ and so on to get a word of potentially infinite length, and see that it is the reverse of a word in L.

rev must also be injective, so every $L \in 2^{A^*}$ must have a unique reversed language. Our original automaton is observable, so every language L is unique. Therefore every reversed language will also be unique. $\square$

We use these lemmas to show that reachability of $\mathcal{X}$ implies observability of $\mathcal{R}$, as follows:

Proof. For part (1), we know that $\mathcal{X}$ is reachable, so r must be surjective. We use Lemma 1 to show that applying $2^{(-)}$ to r, gives us the injective function 2^r. Then O is the composition of 2^r and rev. We proved in Lemma 2 that rev is bijective, so overall O is injective and therefore the reverse automaton $\mathcal{R}$ is observable.

For part (2) of the theorem, we prove that we get the reverse language by using our observability function for reversed automata, O on the initial states of our new automaton, $\mathcal{R}$, but giving the function, for f, as that is now the initial state. This gives us $\{w \in A^* | 2^i(f_w) = 1\}$. This is equivalent to the original definition, because the original definition was on an arbitrary set of states, S, but here we define it on the set of ïnitial" states. As this is now f, we already have the reversed words, so we give w instead of w^R and as we wanted to know if we get to a state from the inital state on w, we now want to know if given w from a state gets us to a final state, so we replace $i_w \in S$ with the new final state function 2^i on f_w (as f is the new i). Therefore we can say that $O(f) = \{w \in A^* | 2^i(f_w) = 1\} = \{w^R \in A^* | i_1 \in f\}$

Now we have all the reversed words where the original word applied to the initial state of $\mathcal{X}$ is accepted by the automaton. This is exactly the same as the reverse language of $\mathcal{X}$. We defined the function rev to give the reverse language previously, so we can see that $\{w^R \in A^* | i_1 \in f\} = rev(\{w \in A^* | i_1 \in f\})$

Then we can also see that $\{w \in A^* | i_1 \in f\}$ is the same as $\{w \in A^* | f(i_1) = 1\}$, which is the observability function applied to the initial state, i,of $\mathcal{X}$, $o(i)$. Therefore $rev(\{w \in A^* | i_1 \in f\}) = rev(o(i))$.

Now we have proved that $O(f) = rev(o(i))$. The observability function, o of the initial automaton, maps a state to the language accepted by it, so it maps the initial state to all languages accepted by words from it, so this will be the accepted language of $\mathcal{X}$. Then the observability function, O, of the reverse automaton sends the new initial state, f, to the language it accepts, which will be the accepted language of the entire reversed language of the automaton. Therefore it is true that if $\mathcal{X}$ accepts L, then $\mathcal{R}$ accepts $rev(L)$

□

Now we can give Brzozowski's Algorithm for deterministic automata as a corollary of this proof, as in [Bonchi et al., 2014]

Corollary 2 (Bonchi)**.** *Given an deterministic automaton accepting a language L:*

1. *Construct the reverse of the automaton*
2. *Take the reachable part*
3. *Construct the reverse of the automaton again*
4. *Take the reachable part*

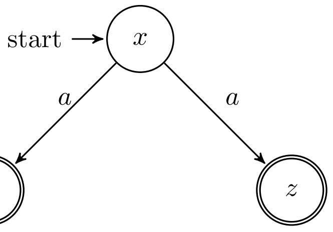
Then the automaton we create is the minimal automaton that accepts L

Proof. Applying our reverse construction and taking the reachable part gives us the powerset construction of the reversed automaton. This is reachable (as we have taken the reachable part) and accepts $rev(L)$.

Applying the reverse construction again will give us an observable automaton (by Theorem 5) that accepts $(rev\ (rev\ L))$. This language is the same as L.

Then taking the reachable part again will give us a reachable, observable automaton that accepts L. We previously defined a minimal automaton to be one that is both reachable and observable, so this automaton is minimal (as all states can be reached from the initial state and all accept a different language, so there are no redundant nodes). □

If the original automaton is already reachable, then the automaton obtained after one reversal is the minimal automaton that accepts L. If we go back to our example we used to explain powerset construction before:



We can see that every state is reachable from the start state and that is why we only needed to minimise this automaton once.

Chapter 5

Discussion

Now that we have proved the theorem that Brzozowski's Algorithm is a corollary of, we have everything we need to give the correctness of the algorithm coalgebraically, having understood basic category theory including the definition of functors and homomorphisms, universal algebra and coalgebra and the properties of automata - reachability and observability, of which the proof of duality enables us to prove the correctness of Brzozowski's Algorithm.

We have understood how the proof works, so we could now go on to see how this proof can be generalised to other types of automata. In [Bonchi et al., 2014], the proof is generalised to other types of automata including non-deterministic automata, Kleene Algebra with Tests (algebra that can be used to represent regular languages with an additional boolean subalgebra, as defined in [Kozen, 1997]) and weighted automata (automata where the transitions are given weights), so it would be interesting to study this in more detail.

The authors of [Bonchi et al., 2014] propose other generalisations and extensions to the proof. The proof can be generalised to other types of automata such as probabilistic automata (automata where there are additional probablities of whether or not a transition can be made, as defined in [Rabin, 1963]). There is also scope to find connections to other methods of minimising automata such as to well-pointed coalgebras (coalgebras with a designated initial state, as defined in [Adámek et al., 2013]).

In my proposal, I wanted to consider some of these options; in hindsight, that was very ambitious for an undergraduate student to take on. It would be a possibility to explore this further in the future, but not at an undergraduate level.

Therefore I am very happy with what I achieved. I have learned a lot about very abstract areas of Computer Science that I had not previously studied including Category Theory and Universal Algebra and coalgebra, that I can apply to my future studies. I also managed to understand a complicated proof in these areas and elaborate on the details including why the homomorphisms being formed in the proof commute and the statements in the proof of the duality between reachability and accessibility that were not immediately obvious to an undergraduate student.

Chapter 6

Conclusion

In my report I have discussed the stages of Brzozowski's algorithm and how this can be applied to an example, reachability and observabilty of automata and the duality between these properties. I have also discussed how to represent an automaton as both an algebra and coalgebra and how reversing this representation of an automaton using the contravariant powerset functor can be used to prove the correctness of Brzozowski's Algorithm.

Bibliography

Jirí Adámek, Stefan Milius, Lawrence S. Moss, and Lurdes Sousa. Well-pointed coalgebras. *Logical Methods in Computer Science*, 9(3), 2013.

M. A. Arbib and H. P. Zeiger. On the relevance of abstract algebra to control theory. *Automatica*, 5(5): 589–606, September 1969.

Steve Awodey. *Category Theory.* Oxford University Press, USA, 2010. ISBN 0199237182.

Filippo Bonchi, Marcello M. Bonsangue, Helle H. Hansen, Prakash Panangaden, Jan J. M. M. Rutten, and Alexandra Silva. Algebra-coalgebra duality in brzozowski's minimization algorithm. *ACM Trans. Comput. Logic*, 15(1):3:1–3:29, March 2014. ISSN 1529-3785.

Janusz A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. In *Mathematical theory of Automata*, Volume 12 of MRI Symposia Series, pages 529–561. Polytechnic Press, Polytechnic Institute of Brooklyn, N.Y., 1962.

John Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, Stanford University, 1971.

Dexter Kozen. Kleene algebra with tests. *Transactions on Programming Languages and Systems*, 19(3): 427–443, May 1997.

Edward F. Moore. *Gedanken-experiments on sequential machines*, chapter Automata studies, pages 129 – 153. Number 34 in Annals of mathematics studies. Princeton University Press, 1956.

M. Rabin. Probabilistic automata. *Information and Control*, 6(3):230 – 245, 1963.

J. J. M. M. Rutten. Universal coalgebra: A theory of systems. *Theoretical Computer Science*, 249(1):3–80, October 2000. ISSN 0304-3975.

Jacques Sakarovitch. *Elements of Automata Theory.* Cambridge University Press, New York, NY, USA, 2009. ISBN 0521844258, 9780521844253.