Czech Technical University in Prague
Faculty of Electrical Engineering

BACHELOR THESIS

Coalgebraic approach to automata theory

Prague, 2010

Author: **Matěj Dostál**
Department of Cybernetics
Software Technologies and Management
Intelligent Systems

Advisor: **doc. RNDr. Jiří Velebil, Ph.D.**
Department of Mathematics

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

V Praze dne ______________ ____________________

podpis

Acknowledgements

It is a pleasure to thank all the people who helped me through the time I worked on my bachelor thesis. I want to thank doc. RNDr. Jiří Velebil, Ph.D., my advisor. His everlasting patience has been a great motivation for me. I thank my family. They provided me with so much more than just food and shelter. If it is true that I am still a sane person, it is only because of Janča Lepšová. Thank you!

Abstrakt

Ukážeme, že automaty mohou být jednotně popsány jako koalgebry určitého funktoru. Pro každý takový funktor předvedeme modální jazyk, který nám umožní popsat stavy automatů. U tohoto jazyka prozkoumáme jeho korektnost a úplnost.

Klíčová slova

Koalgebra, modální logika, bisimilarita.

Abstract

We show that automata can be described uniformly as coalgebras for a functor. For every such functor we present a modal language that allows us to speak about states of automata. We discuss soundness and completness of semantics of our language.

Keywords

Coalgebra, modal logic, bisimilarity.

Contents

1 Introduction **1**

1.1 Background and state of art 1

1.2 Informal formulation of the problem and goals of the thesis 2

1.3 Scope of the text 2

1.4 Related work 3

2 Coalgebras and automata **5**

2.1 Basic coalgebraic notions 5

2.2 Automata as coalgebras 13

2.3 Bisimulation 16

3 Modal logic **19**

3.1 Preliminaries 19

3.2 Syntax 20

3.3 Semantics 21

3.4 Bisimilarity and logical formulas 23

4 Conclusions **33**

Bibliography **36**

A On some mathematical constructions **I**

A.1 Constructions on sets I

A.2 Currying III

Chapter 1

Introduction

This thesis is devoted to a modern topic in computer science: the coalgebraic approach to automata and the description of their behaviour by (necessarily) modal logic. In this chapter we will give an overview of the text, outline what has been done in the past in computer science that is relevant to our topic, and give credit to related work which inspired this thesis.

1.1 Background and state of art

Our thesis heavily uses the methods and results of several classical areas of mathematics and computer science. It is mainly automata theory, category theory and coalgebras, and modal logic.

Automata theory is an important part of computer science and its roots can be found in the works of Alan Turing dealing with computation theory through the use of now-called Turing machines (1930s). It has been a subject of thorough research and brought a number of theoretical and practical results. The classical approach to automata theory is well presented in the classical textbook from Hopcroft, Motwani and Ulmann [6], from which we use the definitions of our example automata.

Automata are classically defined as tuples with a set of states, inputs, outputs, and transition functions. A novel approach using coalgebras unifies the presentation of automata to a set of states, a "signature" that describes the interface of the automaton,

and a transition function. Given a standard definition of an automaton, we have a straightforward way to describe it coalgebraically. This approach dates back to 1980s and has been well summarised in Rutten's study [13], along with deep results from universal coalgebra.

A very young and promising branch of research in theoretical computer science combines two disciplines: automata theory and (modal) logic. The main inspiration comes from Kripke semantics of possible worlds. See Chellas [5] for a classical approach, or Blackburn, de Rijke and Venema [3], for a more updated overview of modal logic.

1.2 Informal formulation of the problem and goals of the thesis

Informally, the problem of the study of behavioural equivalence can be described as follows:

> We have two automata A, B of the same kind, considered as "reactive" systems. Two users U_A, U_B execute runs of automata A and B, respectively, and "observe" the pattern of their behaviour. The behaviour may be a series of output symbols or anything else that is intrinsic to the automata.
>
> The question we ask is the following one: when are two states behaviourally indistinguishable?

The above problem is formulated rather vaguely, of course. What one needs is a *formal* approach to the problem since, ideally, we want another machine to test behavioural equivalence.

The goal of the thesis is, therefore, to develop a mathematical formalism for description of automata and a corresponding formal language that is capable of expressing the behaviour of automata.

1.3 Scope of the text

In the text we focus on two topics:

1. In **Chapter 2** we show how category theory offers a unifying framework for describing automata. We present this phenomenon on various examples of automata – deterministic finite automata, non-deterministic finite automata and pushdown automata. To show that two automata have the same behaviour, we introduce the concept of mutual simulation and show how it works on our examples.

2. In **Chapter 3** we introduce a modal language for description of possible behaviours of classes of automata given by coalgebras for a functor. The language results quite necessarily in a many-sorted modal logic. Its semantics is given in Kripke style – states of an automaton play the role of possible worlds. For such languages we prove two results below:

   (a) Each language for a Kripke-polynomial functor has a sound semantics (Proposition 3.4.1). This means that states with the same behaviour cannot be distinguished by a formula.

   (b) Each language for polynomial functors is complete (Proposition 3.4.4). By completeness we mean that any two states satisfying the same formulas have the same behaviour. We also show that completeness does not hold if unbounded nondeterminism is present – this is the essence of Example 3.4.2.

In short, we give a presentation of automata which allows us to easily determine the conditions under which two automata behave the same way, then we show a language that is used to describe the behaviour of these automata, and prove that the language distinguishes automata with different behaviour and cannot distinguish behaviourally equivalent automata.

1.4 Related work

The thesis was much inspired by Martin Rössiger's dissertation [12]. The modal laguage we study in this text differs, however, from his. We also present slightly more economical and compact proofs of the results.

Chapter 2

Coalgebras and automata

The behaviour of various types of automata has attracted large attention in theoretical computer science. Only recently it has become clear that coalgebras provide a uniform framework to deal with various types of automata and that the formal logic for description of behaviour is necessarily some form of modal logic.

In this chapter we will introduce the notion of coalgebra and the constructs needed to define it precisely. We shall see that coalgebras are structures that allow us to study the behaviour of many automata, and that the type of an automaton can be described by a functor of the coalgebra. To see the differences in both classical and coalgebraic approach, we will look at standard definitions of various automata and then define them coalgebraically. The reason we are interested in this approach is that it unifies the view on automata.

2.1 Basic coalgebraic notions

To be able to define automata coalgebraically, we have to introduce some concepts from category theory. Reader who is interested in category theory can find additional details in standard reference [2], from which we take some of the basic notions for our purposes.

In an informal way we can say that a coalgebra is a function $c\colon S \to \langle \cdots S \cdots \rangle$. By such notation we mean that c is a function from a domain S to a more complex codomain, while the domain set is usually part of the codomain [7]. Let us make the

concept clearer by formalising it.

Category

First we need to define a category, the structure on which we will operate the whole time. Roughly speaking, a category consists of objects and arrows between them, where the arrows can be composed according to the usual laws of map composition. We will define categories more precisely in the following definition.

Definition 2.1.1. A *category* $\mathbb{C}$ consists of

- a class of objects $Obj(\mathbb{C})$
- and a class of morphisms $Mor(\mathbb{C})$.

The above data are subject to the following axioms:

Every morphism $m \in Mor(\mathbb{C})$ is tied to a domain object $D \in Obj(\mathbb{C})$ and a codomain object $C \in Obj(\mathbb{C})$. Such morphism is then written as $m\colon D \to C$, or as $D \xrightarrow{m} C$.

For every two morphisms $m : A \to B$ and $n : B \to C$ there exists a composite morphism $n \circ m : A \to C$, and morphism composition is associative: $(p \circ n) \circ m = p \circ (n \circ m)$.

Lastly, every object $A \in Obj(\mathbb{C})$ has an identity morphism $id_A : A \to A$, which acts as an identity element on morphism composition: for a morphism $m : A \to B$ there are morphisms id_A and id_B such that $m \circ id_A = m = id_B \circ m$.

Definition 2.1.2. In a category $\mathbb{C}$, the set[1] of all morphisms between a domain object A and a codomain object B is denoted $\mathbb{C}(A, B)$ and is called a *hom set*.

If $f\colon X \to Y$ is a morphism and A is an object from $\mathbb{C}$, we define $\mathbb{C}(A, f)\colon \mathbb{C}(A, X) \to \mathbb{C}(A, Y)$ as a function that takes a morphism g from $\mathbb{C}(A, X)$ and returns a morphism $(f \circ g)$.

[1] If the morphisms between any two objects of a category form a set, then the category is *locally small*. Note that not every category satisfies this requirement. See [2].

Categories are very general structures. There are many interesting and diverse examples of categories. We will take a look at some of them.

Example 2.1.3. The category of sets and functions between them. We need to check whether it satisfies all the category properties. Let us define a structure called **Sets** constituted of two classes: the class of all sets and the class of all functions between those sets. Every function has a domain and a codomain, and is therefore tied to two sets from the class of all sets. If there are two functions $f\colon X \to Y$ and $g\colon Y \to Z$, then there is a function $(g \circ f) = h\colon X \to Z$, and function composition is associative. The structure satisfies the morphism composition property from the definition of category. There is an identity function for every set as well, which obeys the rules of the identity morphism. **Sets** is therefore truly a category.

The above example is a very important one. Now we will try to form a category from some algebraic structures together with homomorphisms.

Example 2.1.4. Suppose we have some groups and homomorphisms between them. We know that a composition of homomorphisms is again a homomorphism and that homomorphism compositions are associative. Moreover, every group has an identity homomorphism, which maps every element to itself. This is enough for making a category **Groups** of all groups together with group homomorphisms.

In a similar manner, a category **Monoids**, **Magmas** can be made, with these structures as objects and their corresponding homomorphisms as morphisms.

Viewing morphisms as a sort of "moves", morphism composition is a connection of two moves which can be made one after another. Identity morphism then corresponds to the act of not moving at all. We could apply this view in graph theory:

Example 2.1.5. With a graph (V, E) of vertices V and edges E, we can form a category with vertices as objects. If there is an oriented path from one vertex to another, we join them with a morphism. Then there is a trivial path of length zero, acting as the identity morphism. Morphism composition in this category is simply a connection of two paths.

A category can be thought of as a structure that generalises the concepts of both posets and monoids. Consider a category with one object $\triangle$, and consider a monoid M. The elements of M can be viewed as arrows from $\triangle$ to $\triangle$ in our category, the associativity of the arrows is forced by definition, and so is the existence of the identity arrow. Now consider a poset $(P, \leq)$. It can be viewed as a category with objects from P. The arrows between objects show that the objects are in the relation $\leq$. This kind of generalisation turns out to be fruitful in many more situations than we presented here. More examples of interesting categories include the category of posets and monotone functions, category of metric spaces and metric maps, etc.

Functor

The notion of a functor is important, for it allows us to transform the objects and morphisms of a category to other objects and morphisms (generally in another category) in a well-behaved manner. That is, the transformation preserves domains, codomains, identities and compositions.

Definition 2.1.6. For two categories $\mathbb{C}$ and $\mathbb{D}$, a *functor* F is given by a mapping $F\colon Obj(\mathbb{C}) \to Obj(\mathbb{D})$ and a mapping $F\colon Mor(\mathbb{C}) \to Mor(\mathbb{D})$. We usually shorten this notation as $F\colon \mathbb{C} \to \mathbb{D}$. These mappings have to satisfy the following requirements:

- for a morphism $f \in Mor(\mathbb{C}), f\colon A \to B$ the following holds: $F(f)\colon F(A) \to F(B)$,
- for an identity morphism $id_A \in Mor(\mathbb{C}),\ F(id_A) = id_{F(A)}$, and
- for a morphism composition $(f \circ g) \in Mor(\mathbb{C}),\ F(f \circ g) = F(f) \circ F(g)$.

  In case that the domain and range category is the same, we say that F is an *endofunctor* (of the ambient category).

A composition of two functors is a functor as well.

Proposition 2.1.7. *Let F be a functor $F\colon \mathbb{C}_1 \to \mathbb{C}_2$ and G be a functor $G\colon \mathbb{C}_2 \to \mathbb{C}_3$. Then the composition $G \circ F$ of these functors is a functor.*

Proof. For an arbitrary object X, morphism $f\colon A \to B$, and a pair of composable morphisms g, h, all from the category $\mathbb{C}_1$, we prove that the three functorial requierements hold.

The morphism f is turned into a morphism $(G \circ F)(f) = G(F(f))$, and since for $F(f)$ the domain and codomain is $F(A)$, respectively $F(B)$. After the application of the functor G the domain is $G(F(A)) = (G \circ F)(A)$ and the codomain is $G(F(B)) = (G \circ F)(B)$. So we see that the morphism f is turned into a morphism
$(G \circ F)(f)\colon (G \circ F)(A) \to (G \circ F)(B)$.

The second requirement can be proven easily:

$$(G \circ F)(id_X) = G(F(id_X)) = G(id_{F(X)}) = id_{G(F(X))} = id_{(G \circ F)(X)}.$$

To show that the composition rule is satisfied, it suffices to consider this equation:

$$\begin{aligned} (G \circ F)(g \circ h) &= G(F(g \circ h)) \\ &= G(F(g) \circ F(h)) \\ &= G(F(g)) \circ G(F(h)) \\ &= (G \circ F)(g) \circ (G \circ F)(h). \end{aligned}$$

□

A closer look on functors in **Sets** will be helpful in the construction of coalgebraic automata. We will inspect some basic functors and and their properties. Comments on used notation can be found in Appendix A, further information on functors can be taken from [7].

Firstly, we can make a mapping $Id(X) = X$ for every X from $Obj(\mathbf{Sets})$ and $Id(f) = f$ for every f from $Mor(\mathbf{Sets})$. It is trivially a functor, and we will call it the *identity functor* (on the category **Sets**).

There is also a functor const_S, that maps every set X to a fixed set S, and every function $f\colon X \to Y$ to an identity function $id_S\colon S \to S$. The first functorial requirement holds because for a function $f\colon X \to Y$ the sets become $\mathrm{const}_S(X) = \mathrm{const}_S(Y) = S$

and $\text{const}_S(f) = id_S \colon \text{const}_S(X) \to \text{const}_S(Y)$. In the second requirement we can see that $\text{const}_S(id_A) = id_S = id_{\text{const}_S(A)}$ and the composition property can be proven easily:

$$\text{const}_S(f) \circ \text{const}_S(g) = id_S \circ id_S = id_S = \text{const}_S(f \circ g).$$

Such functor is called a *constant functor* (at a set S). In the following text we are sometimes going to use an abbreviation notation $\text{const}_A = A$, when no confusion can occur.

A *hom functor* on a set A is a functor denoted $\mathbf{Sets}(A, -)$, which turns each set X into a hom set $\mathbf{Sets}(A, X)$ and every function f to $\mathbf{Sets}(A, f)$.

Let us check that this definition indeed gives rise to a functor. A morphism $f \colon X \to Y$ is mapped to a function $\mathbf{Sets}(A, -)(f) = \mathbf{Sets}(A, f) : \mathbf{Sets}(A, -)(X) \to \mathbf{Sets}(A, -)(Y)$ The identity morphism id_X is mapped to an identity function on $\mathbf{Sets}(A, X)$, because $\mathbf{Sets}(A, -)(id_X) = \mathbf{Sets}(A, id_X) = id_{\mathbf{Sets}(A,X)} = id_{\mathbf{Sets}(A,-)(X)}$. Two morphisms $f \colon Y \to Z$ and $g \colon X \to Y$ satisfy the functorial composition property – for any morphism $h \colon A \to X$ we see that

$$\begin{aligned}
(\mathbf{Sets}(A, f) \circ \mathbf{Sets}(A, g))(h) &= \mathbf{Sets}(A, f)(\mathbf{Sets}(A, g)(h)) \\
&= \mathbf{Sets}(A, f)(g \circ h) \\
&= f \circ (g \circ h) \\
&= (f \circ g) \circ h \\
&= \mathbf{Sets}(A, (f \circ g))(h).
\end{aligned}$$

If we are given some set functors F and G, we can combine them in a way that the result is a functor as well. We can take their *product* K: it will be denoted by $K = F \times G$ and we shall define $K(X) = F(X) \times G(X)$, and functions $f \colon X \to Y$ will be transformed to $K(f) = F(f) \times G(f)$.

With this definition the functoriality is guaranteed: for a function $f \colon X \to Y$ we get

$$K(f) = F(f) \times G(f) \colon F(X) \times G(X) \to F(Y) \times G(Y).$$

Since $F(X) \times G(X) = K(X)$ and $F(Y) \times G(Y) = K(Y)$, we see that $K(f)$ is a function $K(X) \to K(Y)$, and the first requirement is met.

The second requirement is easy:

$$K(id_A) = F(id_A) \times G(id_A) = id_{F(A)} \times id_{G(A)} = id_{F(A)\times G(A)} = id_{K(A)}.$$

To prove the composition rule we see that

$$\begin{aligned} K(f) \circ K(g) &= (F(f) \times G(f)) \circ (F(g) \times G(g)) \\ &= (F(f) \circ F(g)) \times (G(f) \circ G(g)) \\ &= F(f \circ g) \times G(f \circ g) \\ &= K(f \circ g). \end{aligned}$$

In a similar manner, we can make a disjoint sum of set functors F and G, called their *coproduct* $C = F + G$. The definition is analogous to product functor – $C(X) = F(X) + G(X)$, function mapping is defined as $C(f) = F(f) + G(f)$.

A very useful construction is the *exponent* of a functor. If we have a set functor F and a set A, then the exponent $H = F^A$ is defined as $\mathbf{Sets}(A, -) \circ F$. In coalgebras, we can use the exponent functor to model an input of an automaton.

If we want to generalise the notion of the exponent to non-set functors, we need to introduce the notion of a cotensor. Let $\mathbb{C}$ be a category. Given a set A and an object X in $\mathbb{C}$, we define an *A-fold cotensor of X* to be an object $A \pitchfork X$ in $\mathbb{C}$ together with an isomorphism $\mathbb{C}(Y, A \pitchfork X) \cong \mathbf{Sets}(A, \mathbb{C}(Y, X))$. See that if $\mathbb{C} = \mathbf{Sets}$, then $A \pitchfork X$ is the product of A-many copies of X.

Let $\mathbb{C}$ be a category where all objects have cotensors with a fixed set A. Suppose that $F : \mathbb{C} \to \mathbb{C}$ is a functor. Then $A \pitchfork F$ is a functor from $\mathbb{C}$ to $\mathbb{C}$, that is defined on objects $(A \pitchfork F)(X) = A \pitchfork F(X)$. Due to the definition of cotensors, it is easy to prove that $A \pitchfork F$ is indeed a functor. We will use the notation F^A to denote $A \pitchfork F$.

To describe non-determinism, it is vital to introduce the *powerset functor*. We will denote it by P. The definition is expectable: it transforms sets into their powersets and functions over sets are extended to work over powersets of these sets. For a functor F, we have $P(F)(X) = P(F(X))$ and for functions $P(F)(f) = P(F(f))$. Thus $P(F)$ is defined as the composite $P \circ F$. The functoriality comes straight from the definitions of the powerset operations on sets and functions (see Appendix A).

We now see that functors can be composed and altered in many ways, and the result is still a functor. This allows us to construct complicated functors inductively. We will form the collection of *Kripke-polynomial functors*. They can be described by the following set of rules:

- The identity functor is in the collection.
- For every set S, the constant functor const_S is in the collection.
- The product of two functors in the collection is also in the collection.
- The coproduct of two functors in the collection is also in the collection.
- For every set A and a functor F from the collection, the exponent F^A is also in the collection.
- The powerset of a functor from the collection is also in the collection.

A convenient way to describe such inductive definition is by *Backus-Naur form.*

$$F ::= Id \mid \text{const}_S \mid F \times F \mid F + F \mid F^S \mid P(F)$$

Coalgebra

Now we finally have all the needed constructs to define coalgebra.

Definition 2.1.8. For a category $\mathbb{C}$ and an endofunctor F, an F*-coalgebra* is an object $A \in Obj(\mathbb{C})$ with a morphism $c \in Mor(\mathbb{C})$ in the form $c\colon A \to F(A)$. We write it down as (A, c).

An F-coalgebra on an object A is then determined by the definition of the morphism c. The complexity of the codomain depends on the functor F.

Example 2.1.9. An easy example of a coalgebra is the following. Let A be an alphabet and N set of nodes. Then we can form a coalgebra $t\colon N \to A \times N \times N$ for the functor $F = \text{const}_A \times (Id) \times (Id)$, which for every node gives a symbol from the alphabet and returns two nodes. The functor $\text{const}_A \times (Id) \times (Id)$ describes the behaviour of infinite binary trees which have a label on each node.

As with other structures, we are interested in structure-preserving mappings, coalgebraic homomorphisms.

Definition 2.1.10. Let $c\colon S \to F(S)$, $d\colon T \to F(T)$ be two F-coalgebras. A function $f\colon S \to T$ is a homomorphism from (S, c) to (T, d), if the following diagram

$$\begin{array}{ccc} S & \xrightarrow{f} & T \\ {\scriptstyle c}\downarrow & & \downarrow{\scriptstyle d} \\ F(S) & \xrightarrow[F(f)]{} & F(T) \end{array}$$

commutes, or, in other words, if $F(f) \circ c = d \circ f$.

2.2 Automata as coalgebras

To show that coalgebras can be useful in computer science, we will form more interesting and complex examples. Automata theory gives many examples of systems that are easily transformable into coalgebraic form. We will use the notions defined above to make coalgebraic definitions of deterministic, non-deterministic and pushdown automata. All the standard definitions and results from automata theory presented here are well known and have been summarised in the standard textbook [6].

Deterministic finite automata

We can think of a *deterministic finite automaton* (DFA) as of a simple machine. This machine can be in various states and move from one state to another by receiving some input, namely a symbol from a previously defined alphabet. The machine also has a starting state and an output for every state, telling whether the state is accepting or not. In other words, the output tells us if the sequence of symbols we put into the machine is accepted by the machine.

If we revise what is needed to form a DFA, we see that it is

- a finite set of states (S), also called a state space,
- an alphabet (A) of input symbols,

- a start state ($s_0 \in S$)
- a transition function ($\delta\colon S \times A \to S$),
- and an output function ($\omega\colon S \to O$, where $O = \{0, 1\}$).

From this list we can see that an arbitrary DFA called D can be characterized as a tuple $D = (S, A, s_0, \delta, \omega)$. Alternatively, we could write D as a tuple (S, A, s_0, δ, F), where the output function ω is replaced by a set F of accepting states from S defined by $F = \{s \mid \omega(s) = 1\}$.

We shall take a different view on DFA's. Automata can be studied from a coalgebraic perspective. Borrowing the above notation, we can describe a DFA as a coalgebra $c\colon S \to O \times S^A$. This is possible because the function c can be viewed as a function $c(s) = \langle \omega(s), \sigma(s) \rangle$, with ω being the output function as defined above and the function σ being defined as follows:
For each state $s \in S$, the output of $\sigma(s)$ is a function $i\colon A \to S$, with the definition $i = \{(a, \delta(s, a)) \mid a \in A\}$ and δ being the DFA's transition function. Therefore, we will omit the distinguished start state s_0 from our description. This is nothing grave: any state can be considered as a start state.

Non-deterministic finite automata

Another example of an interesting automaton is a *non-deterministic finite automaton* (NFA). Its main difference from DFA is that its transition function works differently. While with DFA there could be just one possible successor state for any state and input symbol, now it can lead to any subset of the state space. Its expressive power (in terms of describing some language) is, however, equal to DFA.

Definition of an NFA very much resembles that of DFA. We need:

- a finite set of states (S), also called a state space,
- an alphabet (A) of input symbols,
- a start state ($s_0 \in S$)

- an output function ($\omega\colon S \to O$, where $O = \{0,1\}$),

and lastly, making the only difference,

- a transition function ($\delta\colon S \times A \to P(S)$),

P denoting a powerset.

It should not be surprising that NFA can be described coalgebraically as well. Namely we can form a coalgebra $c\colon S \to O \times P(S)^A$, take $c(s) = \langle \omega, \sigma \rangle(s)$ as with deterministic automata. Note that now the definition of σ remains the same as in the DFA case, it just works with the NFA δ transition function.

Pushdown automata

There are also more complicated automata, which can describe formal grammars higher than regular languages. *Pushdown automata* are automata that accept precisely context-free grammars. The difference between our NFA and a pushdown automaton (PDA) is that PDA allows ϵ-transitions, which means that there can be state transitions that do not need any input symbol to be executed. We shall deal with this by extending the domain of the transition function from A to $(A \cup \epsilon)$. More importantly, PDA can work with stack, where it can store additional information, and accordingly to the top symbol on stack and the input, it can decide what state to go to next.

Formally, the pushdown automaton consists of

- a finite set of states S,
- an alphabet A of input symbols,
- an alphabet Γ of *stack symbols*,
- the δ transition function,
- the start state $s_0 \in S$,
- the start symbol $Z_0 \in \Gamma$, which is the only symbol that is on the stack at the beginning, and

- the output function ($\omega\colon S \to O$, where $O = \{0, 1\}$) again.

Let us look closely on the transition function δ. It takes a state, an input symbol or an empty word ϵ and a stack symbol. The result is a finite set of pairs (s, α), where s is the successor state and α a word of Γ symbols. This word replaces the top of the stack of the previous state. The transition function δ is then a function of the form $\delta\colon S \times (A \cup \{\epsilon\}) \times \Gamma \to P(S \times \Gamma^*)$. The notation Γ^* stands for a set of finite words over alphabet Γ. Coalgebra for such automaton is then $c\colon S \to O \times P(S \times \Gamma^*)^{(A+1)\times\Gamma}$.

We have shown that examples of (classical) automata can be described in a uniform way, using set functors. Moreover, the functors have typically a special shape: they are "built" from "simple" ones using products, coproducts, etc. The shape of functors will play a significant role later, see Definition 3.1.1 below.

2.3 Bisimulation

In the definition 2.1.10 we wrote about the homomorphism between coalgebras, which is a structure-preserving morphism and therefore shows that the two coalgebras are very similar. It is, however, more strict than we need, so we are going to introduce a concept which generalises that of a homomorphism.

We want to be able to say that some automata (or parts of them) are indistinguishable for the observer, even if their inner structure is different. This can be reformulated as a demand that the automata have to be able to simulate their behaviour mutually. The mutual relation is called a bisimulation and we are going to define it categorically with a number of examples to enlighten the definition.

Definition 2.3.1 (Rutten [13]). For a functor F and two F-coalgebras (S, f) and (T, g), an F-*bisimulation* is a relation $R \subseteq S \times T$, for which a morphism $r\colon R \to F(R)$ exists to make the following diagram commute:

$$\begin{array}{ccccc} S & \xleftarrow{\pi_S} & R & \xrightarrow{\pi_T} & T \\ f\downarrow & & \downarrow r & & \downarrow g \\ F(S) & \xleftarrow[F(\pi_S)]{} & F(R) & \xrightarrow[F(\pi_T)]{} & F(T) \end{array}$$

The morphisms π_S and π_T are the corresponding projections of the product $S \times T$. We say that two states s and t are *bisimilar* if there exists a bisimulation R with $(s,t) \in R$.

We shall take some concrete examples of coalgebras, see what the requirements for a bisimilarity in these specific cases are, and how they arise from the general definition of bisimulation written above.

Example 2.3.2. Let us inspect the DFA first. It is a coalgebra for a functor $F = O \times (Id)^A$. So if we take two different coalgebraic DFA's (S,f) and (T,g), when are some two states $s \in S$, $t \in T$ bisimilar?

Firstly, they must yield the same output $o \in O$, because the function $r\colon R \to F(R)$ makes only one output and it remains the same after making either one of the projections $F(\pi_S)$ or $F(\pi_T)$, and this is true for every r.

Secondly, we see that if $s \mapsto (o,\alpha)$ and $t \mapsto (o,\beta)$, where α and β are functions generated by the coalgebras, then it obviously has to hold $(s,t) \mapsto (o,(\alpha \times \beta))$, and since for every input $a \in A$ the output $(\alpha \times \beta)(a)$ must belong to R, we can state the following: For every $a \in A$, the ordered pair $(\alpha(a),\beta(a))$ must be in R.

Example 2.3.3. Nondeterministic finite automata are modelled by a coalgebra belonging to a functor $G = O \times P(Id)^A$. We are given two G-coalgebras (S,f), (T,g) and two states $s \in S$, $t \in T$. The application of the function f on the state s gives us $f(s) = (o,\alpha)$, where $o \in O$ is the output of the automaton and $\alpha\colon A \to P(S)$ a function that takes some input from A and returns a set of states $S' \subseteq S$, which can be thought of as a next-states set.

As with the DFA, it is quite clear that if the states are to be bisimilar, they should have the same output. The function α in DFA returns just one state, though. Now we have to deal with the non-deterministic aspect brought by the powerset functor.

Having two bisimilar states, we can simulate the behaviour of one automaton by the other one and vice versa. This approach can help us imagine what the requirements for

bisimilarity should be. For every input $a \in A$ we get a set of states $\alpha(a) = (S' \subseteq S)$, and for every state $s' \in S'$ we should be able to find a state $t' \in T' = \beta(a)$, which is bisimilar to s'. This way we can simulate the behaviour of (S, f) by (T, g), if we start both automata at the states s, g. Of course, the same simulation has to be possible in the other way, from (T, g) to (S, f).

Example 2.3.4. Taking the same notation as in the previous two examples, we change the functor to $H = O \times P(S \times \Gamma^*)^{(A+1)\times\Gamma}$. For s, t to be bisimilar, they have to satisfy (remember that ω denotes the output function):

1. The output is the same for both automata, $\omega(s) = \omega(t)$.
2. For all a in $\{A \cup \epsilon\}$ and γ in Γ, for all (s', w) in $\alpha(a, \gamma)$ there exists (t', w) in $\beta(a, \gamma)$ such that s' and t' are bisimilar.
3. For all a in $\{A \cup \epsilon\}$ and γ in Γ, for all (t', w) in $\beta(a, \gamma)$ there exists (s', w) in $\alpha(a, \gamma)$ such that t' and s' are bisimilar.

Chapter 3

Modal logic

Given a specific automaton and its state, is it possible to describe its behaviour by some formal language? In this chapter we shall see that such a language exists, that it is structurally a many-sorted modal logic, and its specific structure is given by the structure of the automaton. This language will moreover enable us to express bisimilarity of some two states by stating that the states satisfy the same formulas in the logic, which we are going to prove in this chapter. The construction presented here can be seen in the dissertation [12].

3.1 Preliminaries

The behaviour of an automaton is determined by its coalgebra and the underlying functor of the coalgebra. If we want to create a language that would be able to express the behaviour of automata of a given functor, then the language should probably be constructed accordingly to the structure of the functor. Because we take only Kripke-polynomial functors into account, the language is defined inductively with respect to the inductive structure of the functor.

Definition 3.1.1. If a functor F is constructed from two functors F_1 and F_2 such that $F \in \{F_1 \times F_2, F_1 + F_2\}$, F_1 and F_2 are called *direct ingredients* of F. If $F = P(G)$, then G is a direct ingredient of F. Also const_S and H are direct ingredients of F, if $F = H^S$. Transitive closure of the direct ingredient relation with forced reflexivity forms an *ingredient* relation denoted by $\leq$.

If we have a Kripke-polynomial functor F, we see that it has a tree structure (if we loosen up the view on what a tree is a little bit[1]), with the nodes being ingredients of F, and from every functor to the direct ingredients used for its construction there are (possibly many) edges. The tree we describe will bear more structural information than just the one about direct ingredients. We will create more edges and name them to be able to use them in further work. If $G = G_1 \times G_2$, then the edge from G to G_1 or G_2 is named $\langle \pi_1 \rangle$ or $\langle \pi_2 \rangle$ respectively. If $G = G_1 + G_2$, then the edge from G to G_1 or G_2 is named $\langle \kappa_1 \rangle$ or $\langle \kappa_2 \rangle$ respectively. If $G = H^S$, then for every s in S there is an edge from G to H named $\langle ev_s \rangle$. If $G = P(H)$, then there are two edges from G to H named $\langle P \rangle$ and $[P]$.

The sorts of our language shall relate to the ingredients of F. Let us form a category $Ing(F)$ of the ingredients of the functor F as its structural tree described above, with an added vertex, named $\langle next \rangle$, from every Id ingredient directly to F. This change will enable us to decribe the coalgebra transition, which takes a state from S and returns an element from $F(S)$.

It will be technically more pleasant for us to work with the opposite category of $Ing(F)$.

Definition 3.1.2. For a category $\mathbb{C}$ with objects $Obj(\mathbb{C})$ and morphisms $Mor(\mathbb{C})$, an *opposite category* $\mathbb{C}^{op}$ is a category with objects $Obj(\mathbb{C})$, and its morphisms are morphisms of $\mathbb{C}$ with opposite orientation, that is, for a morphism $m\colon A \to B$ in $Mor(\mathbb{C})$ of $\mathbb{C}$, there is a morphism $m\colon B \to A$ in $Mor(\mathbb{C})$ of $\mathbb{C}^{op}$. If $h = g \circ f$ in $\mathbb{C}$, then $h = f \circ g$ in $\mathbb{C}^{op}$.

3.2 Syntax

We shall first introduce the syntax of the language informally. The structural tree of F has some leaves, namely Id functors and constant functors. When we form a category $Ing(F)$ from the structural tree, it is no longer a tree, but we can still think of the constant functors as of "quasi-leaves". Starting in such a leaf const_A in the category

[1] It is rather a *multitree* in the sense of the difference between graphs and multigraphs.

$Ing(F)^{op}$, we can take an element a from A and it will be a formula of sort const_A. If there is a morphism named $\langle\varpi\rangle$ from sort G_1 to G_2, we can turn a formula φ of sort G_1 (denoted $\varphi : G_1$) into a formula $\langle\varpi\rangle\varphi : G_2$. The set of formulas of a sort can be also closed under boolean connectives. The following definition of the language syntax offers a different view than the one presented above, but the reader should see that the construction of the formulas follows the same principle in both cases.

Definition 3.2.1 (Rössiger [12]). For a functor F we define a family $(\mathcal{L}_G)_{G\leq F}$ of languages, where G is an ingredient of F by simultaneous induction using the Backus-Naur form:

$$
\begin{array}{llll}
G = \mathrm{const}_A : & \varphi ::= & \bot \mid \varphi \to \varphi \mid a, & \text{where } a \in A,\\
G = Id : & \varphi ::= & \bot \mid \varphi \to \varphi \mid \langle next\rangle\psi, & \text{where } \psi \in \mathcal{L}_F,\\
G = F_1 \times F_2 : & \varphi ::= & \bot \mid \varphi \to \varphi \mid \langle\pi_i\rangle\psi, & \text{where } \psi \in \mathcal{L}_{F_i},\\
G = F_1 + F_2 : & \varphi ::= & \bot \mid \varphi \to \varphi \mid \langle\kappa_i\rangle\psi, & \text{where } \psi \in \mathcal{L}_{F_i},\\
G = H^A : & \varphi ::= & \bot \mid \varphi \to \varphi \mid \langle ev_a\rangle\psi, & \text{where } \psi \in \mathcal{L}_H,\\
G = P(H) : & \varphi ::= & \bot \mid \varphi \to \varphi \mid \langle P\rangle\psi \mid [P]\psi, & \text{where } \psi \in \mathcal{L}_H.
\end{array}
$$

We have defined a language for each ingredient of the functor F. See that by constructing the formula of sort G, we follow the walk on the graph induced by the category $Ing(F)^{op}$, starting in the node of the constant functor and ending in G, so our informal view really catches the way the formulas are constructed. Indeed, we can form a functor $\mathcal{L}$ between the category $Ing(F)^{op}$ and the category **Sets** – it sends the sort G to the set of all formulas of the sort G, and the morphisms in $Ing(F)^{op}$ are turned into functions that transform the formulas from one sort to another accordingly to the definition of the syntax above.

3.3 Semantics

In order to give the formulas some meaning, we have to introduce semantics for our new language. Let us have a coalgebra $c\colon S \to F(S)$. Defining the meaning of the formulas created by the inductive structure mentioned above, we will have to proceed inductively as well. For a sort G, the meaning of a formula $\varphi : G$ is a certain subset $||\varphi : G||_c$ of

$G(S)$. We shall be especially interested in the formulas of sort Id, because these will be giving us the information about the behaviour of the states of our coalgebra.

The semantics of boolean connectives for the formulas of any sort are defined in a standard way, disjunction as union of the meanings of the formulas, negation as complement etc.

Definition 3.3.1. Given an F-coalgebra $c : S \to F(S)$, we define for every ingredient G of the functor F the semantics of the family of languages $(\mathcal{L}_G)_{G \leq F}$ in the following way (see the appendix for the definition of the inverse image of a function):

- If $G = \text{const}_A$, then $||a : \text{const}_A||_c = \{a\}$.
- If $G = F_1 \times F_2$, then $||\langle \pi_i \rangle \varphi : F_1 \times F_2||_c = \pi_i^{-1}(||\varphi : F_i||_c)$.
- If $G = F_1 + F_2$, then $||\langle \kappa_i \rangle \varphi : F_1 + F_2||_c = \kappa_i(||\varphi : F_i||_c)$.
- If $G = H^A$, then $||\langle ev_a \rangle \varphi : H^A||_c = ev_a^{-1}(||\varphi : H||_c)$. (Function ev_a takes a function f and returns $f(a)$.)
- If $G = P(H)$, then $||[P]\varphi : P(H)||_c = P(||\varphi : H||_c)$. Also $||\langle P \rangle \varphi : P(H)||_c = \{k \in P(H(S)) \mid \exists l \in k \text{ such that } l \in ||\varphi : H||_c\}$.
- If $G = Id$, then $||\langle next \rangle \varphi : Id||_c = c^{-1}(||\varphi : F||_c)$.

For a state s in S, we denote by $s \Vdash_c \varphi$ the fact that s is an element of $||\varphi : Id||_c$, and say that s satisfies the formula φ.

We saw that the meaning of a G-sorted formula is a subset of $G(S)$. This may lead us to a thought that the powerset of $G(S)$ is the set of all the possible meanings a G-sorted formula can have. We will denote this set $\mathcal{M}(G)$. Since this can be done with every sort, we can form a functor $\mathcal{M}$ from the category $Ing(F)^{op}$ to the category **Sets**, which takes the sorts and transforms them into the sets of all possible meanings of the sorted formulas. What will the functions between the sorts then be? If we want $\mathcal{M}$ to be a functor, we need to define them exactly as the functions that define the semantics of our language above. For example, if we take one possible meaning m of sort F_1 and want to turn it into a meaning of sort $G = F_1 \times F_2$, we get $\pi_1^{-1}(m)$.

Because of the way we defined the functor $\mathcal{M}$, we see that the semantics of our language, as introduced in our definition, acts like a natural transformation between the functors $\mathcal{L}$ and $\mathcal{M}$.

Definition 3.3.2. For functors F and G between the categories $\mathbb{C}$ and $\mathbb{D}$, we call η a *natural transformation* from F to G if it gives for every object X from $\mathbb{C}$ a morphism $\eta_X : F(X) \to G(X)$ in $\mathbb{D}$, such that the following diagram

$$\begin{array}{ccc} F(X) & \xrightarrow{F(f)} & F(Y) \\ \downarrow{\scriptstyle \eta_X} & & \downarrow{\scriptstyle \eta_Y} \\ G(X) & \xrightarrow[G(f)]{} & G(Y) \end{array}$$

commutes for every morphism $f : X \to Y$.

3.4 Bisimilarity and logical formulas

In this section we will show that the language $\mathcal{L}_{Id}$ is sound, that is, two bisimilar states s and t satisfy the same formulas in the language. This is certainly a useful result, allowing us to distinguish non-bisimilar states just by finding a formula that is satisfied for one state and not satisfied for the other one. Moreover, we will specify the conditions under which the language $\mathcal{L}_{Id}$ is complete – which means that two states satisfying the same formulas are bisimilar. It will be shown that for Kripke-polynomial functors without the powerset functor (called just *polynomial functors*) $\mathcal{L}_{Id}$ truly is complete, which makes it a perfect language to describe the behaviour of an automaton up to bisimilarity.

Proposition 3.4.1 (Soundness of $\mathcal{L}_{Id}$). *Two bisimilar states satisfy the same formulas.*

We are going to prove this proposition by structural induction on the complexity of the underlying functor of the coalgebra.

Proof. Let us take two coalgebras $c : S \to F(S)$ and $d : T \to F(T)$, a bisimulation R on these two coalgebras, and a pair of bisimilar states $s \in S$ and $t \in T$, where

sRt. We are going to prove that if $s \Vdash_c \varphi$, then $t \Vdash_d \varphi$, and conversely. See that $\varphi : Id = \langle next \rangle \psi : F$. We will deal with the base cases first.

If $F = \text{const}_A$, the formula $\psi : \text{const}_A$ is either atomic or open. Suppose it is atomic: then $\psi = a$, where a is an element of A. Then it follows that $c(s) \in ||\psi||_c$, which means that $c(s) = a$. And because the following diagram

$$\begin{array}{ccccc} S & \xleftarrow{\pi_S} & R & \xrightarrow{\pi_T} & T \\ {\scriptstyle c}\downarrow & & \downarrow{\scriptstyle r} & & \downarrow{\scriptstyle d} \\ A & \underset{id_A}{=\!=} & A & \underset{id_A}{=\!=} & A \end{array}$$

commutes, it is also true that $d(t) = a$, and therefore $t \in d^{-1}(\{a\})$, from which we can directly see that $t \Vdash_d \varphi$. It is easy to see that starting with $t \Vdash_d \varphi$, we could prove $s \Vdash_c \varphi$ in the same way.

Now we know that s and t satisfy the same atomic formulas. Showing that they satisfy the same formulas built inductively from the atomic formulas using the standard logical connectives is then a matter of an easy induction on the complexity of the syntax tree of the formula. Because this will be true for all the cases we are going to cover in the proof, we are going to restrict ourselves to atomic formulas and keep in mind that the proof for non-atomic formulas follows immediately.

If $F = Id$, we cover three cases.

- Suppose $\psi = \bot$: Then φ does not hold for s and does not hold for t, because $c(s) = s'$ and s' is not an element of $\emptyset$, similarly for $d(t) = t'$.
- Taking $\psi = (\bot \to \bot) = \top$, we see that $c(s) \in S = ||\top : Id||_c$ and $d(t) \in T = ||\top : Id||_d$, so $s \Vdash_c \varphi$ and $t \Vdash_d \varphi$.
- Let us form an induction hypothesis: If s' and t' are R-bisimilar, then $s' \Vdash_c \psi$ if and only if $t' \Vdash_d \psi$.

Now we know that there are states $s' = c(s)$ and $t' = d(t)$, and that s' and t' are bisimilar as well. Because $s \Vdash_c \varphi$, it follows that $s' \Vdash_c \psi$, and using the induction hypothesis, $t' \Vdash_d \psi$. And since $t \in d^{-1}(t')$, we finally get $t \Vdash_d \varphi$.

It can be shown very similarly that $t \Vdash_d \varphi$ implies $s \Vdash_c \varphi$.

In the case the functor F is more complex, we are going to introduce the induction hypothesis – that for every direct ingredient of F, bisimilar states of any two coalgebras of that ingredient satisfy the same formulas.

If $F = F_1 \times F_2$, then $\psi = \langle \pi_i \rangle \tilde{\psi}$, with $i \in \{1, 2\}$. Because the following diagram

$$\begin{array}{ccccc} S & \xleftarrow{\pi_S} & R & \xrightarrow{\pi_T} & T \\ \downarrow c & & \downarrow r & & \downarrow d \\ F(S) & \xleftarrow{F(\pi_S)} & F(R) & \xrightarrow{F(\pi_T)} & F(T) \\ \downarrow \pi_i & & \downarrow \pi_i & & \downarrow \pi_i \\ F_i(S) & \xleftarrow{F_i(\pi_S)} & F_i(R) & \xrightarrow{F_i(\pi_T)} & F_i(T) \end{array}$$

commutes, we see that R is a bisimulation for coalgebras (S, c') and (T, d'), where $c' = \pi_i \circ c$ and $d' = \pi_i \circ d$. Since[2] $\varphi = \langle next \rangle \langle \pi_i \rangle \tilde{\psi}$ and $||\langle next \rangle \langle \pi_i \rangle \tilde{\psi}||_c = c^{-1}(\pi_i^{-1}(||\tilde{\psi}||_{c'}))$, it follows that $s \Vdash_{c'} \langle next \rangle \tilde{\psi}$, and from induction hypothesis $t \Vdash_{d'} \langle next \rangle \tilde{\psi}$. And because $d' = \pi_i \circ d$, it means that $t \in d^{-1}(\pi_i^{-1}(||\tilde{\psi}||_d))$, from which it directly follows that $t \Vdash_d \varphi$. The proof in the opposite direction is analogous.

If $F = F_1 + F_2$, then $\psi = \langle \kappa_i \rangle \tilde{\psi}$, with $i \in \{1, 2\}$. This shows us that $c(s) \in \kappa_i(||\tilde{\psi}||)$. We need to construct a function that acts as an inverse of κ_i, but has $F(S)$ as a domain. Since $F_1(S) + F_2(S)$ is necessarily nonempty, we can suppose, without loss of generality, that $F_1(S)$ is nonempty. Let us take a dummy element $\triangle \in F_1(S)$. Then we can make a function k_i that takes $w = (v, i)$ to $\kappa_i^{-1}(w) = v$, and takes $x = (z, j)$ to $\triangle$, where $j \neq i$. The function k_i defined this way is a total function from $F(S)$ to $F_i(S)$ and, moreover, makes the following diagram

$$\begin{array}{ccccc} S & \xleftarrow{\pi_S} & R & \xrightarrow{\pi_T} & T \\ \downarrow c & & \downarrow r & & \downarrow d \\ F(S) & \xleftarrow{F(\pi_S)} & F(R) & \xrightarrow{F(\pi_T)} & F(T) \\ \downarrow k_i & & \downarrow k_i & & \downarrow k_i \\ F_i(S) & \xleftarrow{F_i(\pi_S)} & F_i(R) & \xrightarrow{F_i(\pi_T)} & F_i(T) \end{array}$$

[2] We use the fact that $||\tilde{\psi}||_c = ||\tilde{\psi}||_{c'}$, which comes immediately from the inductive definition of the syntax.

commute. If we take $c' = k_i \circ c$ and $d' = k_i \circ d$, it comes from the diagram that R is a bisimulation for (S, c') and (T, d'). Because $s \in c'^{-1}(||\tilde{\psi}||_{c'})$ means that $s \Vdash_{c'} \langle next \rangle \tilde{\psi}$ and this implies $t \Vdash_{d'} \langle next \rangle \tilde{\psi}$ from induction hypothesis, we see that $t \in d'^{-1}(||\tilde{\psi}||_d)$, which turns into $t \in d^{-1}(k_i^{-1}(||\tilde{\psi}||_d))$, and we can conlude that $t \Vdash_d \langle next \rangle \langle \kappa_i \rangle \tilde{\psi}$, and because $\langle next \rangle \langle \kappa_i \rangle \tilde{\psi} = \varphi$, the proof is complete.

If $F = G^A$, then $\psi = \langle ev_a \rangle \tilde{\psi}$, where a is an element of A. The following diagram

$$\begin{array}{ccccc}
S & \xleftarrow{\pi_S} & R & \xrightarrow{\pi_T} & T \\
\downarrow c & & \downarrow r & & \downarrow d \\
F(S) & \xleftarrow{F(\pi_S)} & F(R) & \xrightarrow{F(\pi_T)} & F(T) \\
\downarrow ev_a & & \downarrow ev_a & & \downarrow ev_a \\
G(S) & \xleftarrow{G(\pi_S)} & G(R) & \xrightarrow{G(\pi_T)} & G(T)
\end{array}$$

commutes, and therefore, for $c' = ev_a \circ c$ and $d' = ev_a \circ d$ it is clear that R is a bisimulation for coalgebras (S, c') and (T, d'). Now we see from $s \in ||\varphi||_c$ that $s \in c^{-1}(ev_a^{-1}(||\tilde{\psi}||_c))$, use the definition of c' to write $s \in c'^{-1}(||\tilde{\psi}||_{c'})$, from induction principle get $t \in d'^{-1}(||\tilde{\psi}||_{d'})$, by definition of d' see that $t \in d^{-1}(ev_a^{-1}(||\tilde{\psi}||_d))$, and conlude that $t \in ||\varphi||_d$, which is what we wanted to prove. Again, the proof in the opposite direction is done similarly.

If $F = P(G)$, then suppose that $\psi = [P]\tilde{\psi}$. (For the case where $\psi = \langle P \rangle \tilde{\psi}$, we can take $\langle P \rangle = \neg[P]\neg$.)

We are going to use slightly different induction hypothesis for the powerset case. Because G is an ingredient of F, we are going to assume that bisimilar states of $G(S)$ and $G(T)$ satisfy the same formulas.

Let us denote $c(s) = s'$ and $d(t) = t'$. Since s' is an element of $P(G(S))$, it follows that $s' \subseteq G(S)$, and similarly $t' \subseteq G(T)$. From bisimilarity we get that for every $x \in s'$ there exists some $y \in t'$ that is bisimilar to x, and conversely, for every $y \in t'$ there is an element $x \in s'$ such that x and y are bisimilar as well. Now taking any $x \in s'$, we see that it satisfies $\tilde{\psi}$. We are now going to show that t' is a subset of $||\tilde{\psi}||_d$. That would be true if every $y \in t'$ was an element of $||\tilde{\psi}||_d$. We can find an element $x \in s'$ such that x and y are bisimilar, and since $x \in ||\tilde{\psi}||_c$, we see from induction hypothesis that y is

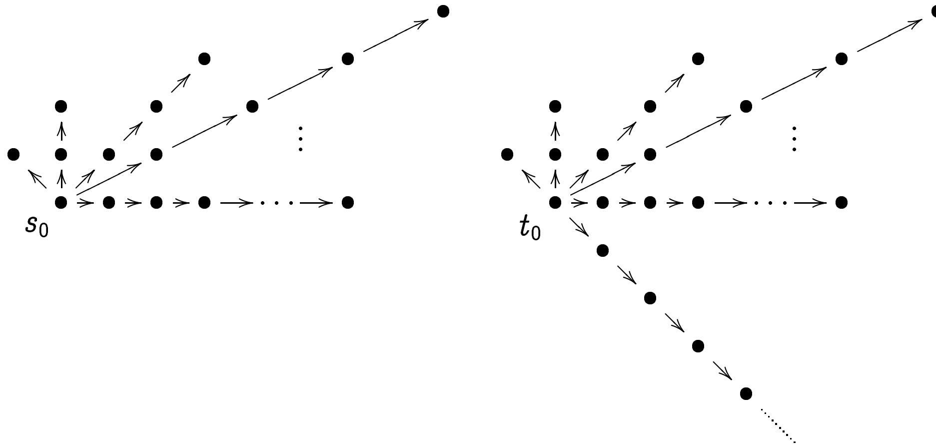
an element of $||\tilde{\psi}||_d$. Therefore, it is truly satisfied that $t' \subseteq ||\tilde{\psi}||_d$, and from this we can easily conclude that $t' \Vdash_d [P]\tilde{\psi}$, and $t \Vdash_d \langle next\rangle[P]\tilde{\psi}$. The proof is now complete in one direction, and to prove the opposite direction, we would follow the same steps analogously.

□

We have proven that $\mathcal{L}_{Id}$ is sound, and now we are going to show that under certain conditions it is complete as well.

First we are going to show that the fact that two states satisfy the same formulas does not always imply that they are bisimilar.

Example 3.4.2. Let us think of coalgebras for a functor $F = P(Id)$. On the figure below we see a standard counterexample showing that there are two coalgebras for F which have two states satisfying the same formulas without being bisimilar.



The figure on the left shows a coalgebra with a state from which there goes a branch of length n for each natural number $n \geq 1$. The figure on the right shows the same coalgebra with the addition of exactly one infinite branch. These two coalgebras satisfy the same formulas is states s_0 and t_0, respectively, even though the coalgebra on the left hand side cannot simulate the infinite behaviour of the coalgebra on the right hand side.

If we limit ourselves to polynomial functors and their coalgebras, the completness of our language holds. First we give a definition of polynomial functors.

Definition 3.4.3. A functor is *polynomial* if it is constructed with respect to the following Backus-Naur form:

$$F ::= Id \mid \text{const}_S \mid F \times F \mid F + F \mid F^S.$$

For this type of functors we can present the completeness theorem and its proof.

Proposition 3.4.4 (Completeness of $\mathcal{L}_{Id}$). *Suppose two coalgebras of a polynomial functor. If their two states satisfy the same formulas, they are bisimilar.*

Proof. Let us take a coalgebra $c : S \to F(S)$, coalgebra $d : T \to F(T)$ and a pair of states $s \in S$, $t \in T$ that satisfy the same formulas. We want to show that s and t are bisimilar. Let us form a relation R that contains all tuples (s', t'), where s' satisfies the same formulas as t'. Evidently (s, t) is in R as well. All that is needed to do is to find a function r such that the following diagram

$$\begin{array}{ccccc} S & \xleftarrow{\pi_S} & R & \xrightarrow{\pi_T} & T \\ \downarrow c & & \downarrow r & & \downarrow d \\ F(S) & \xleftarrow[F(\pi_S)]{} & F(R) & \xrightarrow[F(\pi_T)]{} & F(T) \end{array}$$

commutes.

Suppose $F = \text{const}_A$. Then the function r must make this diagram

$$\begin{array}{ccccc} S & \xleftarrow{\pi_S} & R & \xrightarrow{\pi_T} & T \\ \downarrow c & & \downarrow r & & \downarrow d \\ A & \underset{id_A}{=\!=} & A & \underset{id_A}{=\!=} & A \end{array}$$

commute. Since for any element (s, t) from R we know that for some $a \in A$ it holds that $s \Vdash_c \langle next \rangle a$, it is also true that $t \Vdash_d \langle next \rangle a$. Looking at the meaning of $\langle next \rangle a$, we see that $c(s) = a$ and $d(t) = a$. But then we can define $r((s, t)) = a$ and the diagram indeed commutes.

Suppose $F = Id$. To make this diagram

$$\begin{array}{ccccc} S & \xleftarrow{\pi_S} & R & \xrightarrow{\pi_T} & T \\ \downarrow c & & \downarrow r & & \downarrow d \\ S & \xleftarrow{\pi_S} & R & \xrightarrow{\pi_T} & T \end{array}$$

commute, we denote $c(s) = s'$ and $d(t) = t'$ for every (s,t) in R, define $r((s,t)) = (s',t')$ and show that $s'Rt'$. This requirement is equivalent to the requirement that s' and t' have to satisfy the same formulas. Because s and t satisfy the same formulas, it comes immediately that s' and t' satisfy the same formulas too. Suppose $s' \Vdash_c \psi$ and $t' \nVdash_c \psi$. That would yield a contradiction, since it would imply that $s \Vdash_c \langle next\rangle\psi$ and $t \nVdash_c \langle next\rangle\psi$.

Suppose $F = F_1 \times F_2$. The diagram

$$\begin{array}{ccccc} S & \xleftarrow{\pi_S} & R & \xrightarrow{\pi_T} & T \\ \downarrow c & & \downarrow r & & \downarrow d \\ F(S) & \xleftarrow[F(\pi_S)]{} & F(R) & \xrightarrow[F(\pi_T)]{} & F(T) \end{array}$$

commutes exactly when we set $r = (r_1 \times r_2)$ and the diagram

$$\begin{array}{ccccc} S & \xleftarrow{\pi_S} & R & \xrightarrow{\pi_T} & T \\ \downarrow c_i & & \downarrow r_i & & \downarrow d_i \\ F_i(S) & \xleftarrow[F_i(\pi_S)]{} & F_i(R) & \xrightarrow[F_i(\pi_T)]{} & F_i(T) \end{array}$$

commutes for $i \in \{1,2\}$, where $c_i = \pi_i \circ c$ and $d_i = \pi_i \circ d$. To ensure that r_i can really be constructed in a way that makes the preceding diagram commute, we prove that for every (s,t) in R, $s \Vdash_{c_i} \varphi$ if and only if $t \Vdash_{d_i} \varphi$ for every φ. Since we know that $||\langle next\rangle\langle\pi_i\rangle\psi||_c = ||\langle next\rangle\psi||_{c_i}$, it is sufficient to show that $s \Vdash_c \langle next\rangle\langle\pi_i\rangle\psi$ if and only if $t \Vdash_d \langle next\rangle\langle\pi_i\rangle\psi$. We know that from the fact that s and t satisfy the same formulas for c and d.

Suppose $F = G^A$. If we define $c_a = ev_a \circ c$, $d_a = ev_a \circ d$ and $r_a = ev_a \circ r$, then the diagram

$$\begin{array}{ccccc} S & \xleftarrow{\pi_S} & R & \xrightarrow{\pi_T} & T \\ \downarrow c & & \downarrow r & & \downarrow d \\ F(S) & \xleftarrow[F(\pi_S)]{} & F(R) & \xrightarrow[F(\pi_T)]{} & F(T) \end{array}$$

commutes if for every $a \in A$ the following diagram

$$\begin{array}{ccccc} S & \xleftarrow{\pi_S} & R & \xrightarrow{\pi_T} & T \\ \downarrow{c_a} & & \downarrow{r_a} & & \downarrow{d_a} \\ G(S) & \xleftarrow[G(\pi_S)]{} & G(R) & \xrightarrow[G(\pi_T)]{} & G(T) \end{array}$$

commutes. This comes from a rather technical derivation, but the idea should be clear. We are going to show a part of the derivation. To make this square

$$\begin{array}{ccc} S & \xleftarrow{\pi_S} & R \\ \downarrow{c} & & \downarrow{r} \\ F(S) & \xleftarrow[F(\pi_S)]{} & F(R) \end{array}$$

commute, it must hold that $c \circ \pi_S = F(\pi_S) \circ r$. Let $z = (s,t)$, $c(s) = f$ and $r(z) = g$. Then the following equations are equivalent:

$$\begin{array}{llrcl}
 & & c \circ \pi_S & = & F(\pi_S) \circ r \\
(\forall z \in R) & & c \circ \pi_S(z) & = & F(\pi_S) \circ r(z) \\
(\forall z \in R) & & c(s) & = & F(\pi_S) \circ r(z) \\
(\forall z \in R) & & f & = & F(\pi_S)(g) \\
(\forall z \in R) & & f & = & G(\pi_S) \circ g \\
 & & & & \\
(\forall z \in R) & (\forall a \in A) & f(a) & = & G(\pi_S)(g(a)) \\
(\forall z \in R) & (\forall a \in A) & ev_a \circ f & = & G(\pi_S) \circ ev_a \circ g \\
(\forall z \in R) & (\forall a \in A) & ev_a \circ c(s) & = & G(\pi_S) \circ ev_a \circ r(z) \\
(\forall z \in R) & (\forall a \in A) & ev_a \circ c \circ \pi_S(z) & = & G(\pi_S) \circ ev_a \circ r(z) \\
 & & & & \\
 & (\forall a \in A) & ev_a \circ c \circ \pi_S & = & G(\pi_S) \circ ev_a \circ r \\
 & (\forall a \in A) & c_a \circ \pi_S & = & G(\pi_S) \circ r_a.
\end{array}$$

The last equation means that the following diagram

$$\begin{array}{ccc} S & \xleftarrow{\pi_S} & R \\ \downarrow{c_a} & & \downarrow{r_a} \\ G(S) & \xleftarrow[G(\pi_S)]{} & G(R) \end{array}$$

must commute for every $a \in A$. Doing the same procedure for the square with R and T, we show that finding the function r is equivalent to finding a function r_a for every $a \in A$. We then just define $r(z) = h$ and $h(a) = r_a(z)$ for every z and a. To show that s and t satisfy the same formulas under c_a and d_a, we use the same argument as in the previous case. If there was a formula that would distinguish s and t under c_a and d_a, then it would distinguish them even under coalgebras c and d, which would lead to a contradiction.

Suppose $F = F_1 + F_2$. Then it must hold for every $s' \in S$ that either $s' \Vdash_c \langle next \rangle \langle \kappa_1 \rangle \top$ or $s' \Vdash_c \langle next \rangle \langle \kappa_2 \rangle \top$, but not both of them, since in that case we would get $c(s') = (a, 1)$ and $c(s') = (b, 2)$ for some $a \in F_1(S)$ and $b \in F_2(S)$, which yields a contradiction, because it would imply that $1 = \pi_2(c(s')) = 2$. From the induction hypothesis we get that R can be divided into two partitions, R_1 and R_2, where for $(s', t') \in R_i$ it holds that both s' and t' satisfy $\langle next \rangle \langle \kappa_i \rangle \top$.

Consider a demarking function $k_i : F(S) \to F_i(S)$, which is defined the same way as the function k_i in the proof of the Proposition 3.4.1, coproduct part. We can then define $c_i = k_i \circ c$, $d_i = k_i \circ d$ for $i = \{1, 2\}$. If we ask the diagram

$$\begin{array}{ccccc} S & \xleftarrow{\pi_S} & R & \xrightarrow{\pi_T} & T \\ \downarrow c & & \downarrow r & & \downarrow d \\ F(S) & \xleftarrow[F(\pi_S)]{} & F(R) & \xrightarrow[F(\pi_T)]{} & F(T) \end{array}$$

to commute, it is equivalent to ask the diagram

$$\begin{array}{ccccc} S & \xleftarrow{\pi_S} & R_i & \xrightarrow{\pi_T} & T \\ \downarrow c_i & & \downarrow r_i & & \downarrow d_i \\ F_i(S) & \xleftarrow[F_i(\pi_S)]{} & F_i(R) & \xrightarrow[F_i(\pi_T)]{} & F_i(T) \end{array}$$

to commute for $i \in \{1, 2\}$, if we set $r(z_i) = (r_i(z_i), i)$, where $z_i \in R_i$. To show that every s and t satisfy the same formulas under c_i and d_i, we use the same logic as in the preceding cases. The proof is therefore complete.

□

We have proven that $\mathcal{L}_{Id}$ is complete for polynomial functors. In fact, Propositions 3.4.1 and 3.4.4 give the best result one can expect: modal logic for polynomial

functors captures exactly the behaviour of the relevant automata. The above completeness result therefore meets the goals of the thesis.

Chapter 4

Conclusions

We presented a logical calculus for description of behaviour of automata. The approach we took was that of a modal (many-sorted) language for coalgebras. We showed that the given semantics of the language is sound with regard to bisimulation. We also proved a partial converse: the semantics is complete provided the construction of the relevant automata does not use use unbounded non-determinism.

Thus, the proposed goals of the thesis have been fulfilled: each type of automata given by a polynomial functor admits a formal language that is capable to describe the behaviour of the relevant class of automata.

Due to the properties of our language, its applications could arise in the fields of formal specification and verification of simple systems.

The problems we have studied admit a natural generalisation. For example, one might be interested in modal languages describing automata on a category, different from the category of sets [11]. Another interesting area of research is to understand better how the modal language and its semantics arise naturally from logical connections, making a bridge between the category of coalgebras and the category of the underlying propositional logic [4], [8].

Bibliography

[1] Henk Barendregt, Erik Barendsen, 2000. *Introduction to Lambda Calculus*, Lecture Notes available online at http://www.cs.ru.nl/ henk/courses.html

[2] Michael Barr, Charles Wells, 1990. *Category Theory for Computing Science*, Prentice Hall.

[3] Patrick Blackburn, Maarten de Rijke, Yde Venema, 2002. *Modal Logic*, Cambridge University Press.

[4] Marcello M. Bonsangue, Alexander Kurz, 2005. Duality for Logics of Transition Systems, *Lecture Notes in Comput. Sci.*, volume 3441, pages 455-469, Springer.

[5] Brian F. Chellas, 1980. *Modal Logic: An Introduction*, Cambridge University Press.

[6] John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman, 2000. *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*, Addison Wesley.

[7] Bart Jacobs, 2005. *Introduction to coalgebra. Towards Mathematics of States and Observations*, available online at

http://www.cs.ru.nl/B.Jacobs/PAPERS/index.html

[8] Bartek Klin, 2007. Coalgebraic modal logic beyond sets, *Electron. Notes Theoret. Comput. Sci.*, volume 173, pages 177-201, Elsevier B. V., Amsterdam.

[9] Alexander Kurz, 2001. *Coalgebras and Modal Logic*, Course Notes for ESSLI 2001, available electronically at http://www.cs.le.ac.uk/people/akurz/cml.html

[10] Saunders MacLane, 1994. *Categories for the Working Mathematician*, Springer, New York.

[11] Larry Moss, Ignacio Viglizzo, 2005. Harsanyi Type Spaces and Final Coalgebras Constructed from Satisfied Theories, *Electron. Notes Theoret. Comput. Sci.*, volume 106, pages 279-295, Elsevier B. V., Amsterdam.

[12] Martin Rössiger, 2000. *Coalgebras, Clone Theory, and Modal Logic*, Ph.D. dissertation, Dresden University of Technology.

[13] J.J.M.M. Rutten, 2000. Universal coalgebra: a theory of systems, *Theoret. Comput. Sci.*, volume 249, pages 3-80, Elsevier B. V., Amsterdam.

Appendix A

On some mathematical constructions

A.1 Constructions on sets

Reader can consult [7] for in-depth treatment of the following constructions.

Image of function

Let us have a function $f\colon A \to B$ and a set X, which is a subset of A. The notation $f[X]$ then stands for the set $f[X] = \{b \mid x \in X, f(x) = b\}$.

Inverse image

For a function f, we denote $f^{-1}(z) = \{x \mid f(x) = z\}$.

Powerset function

The powerset of a set A is defined as the set $P(A) = \{X \mid X \subseteq A\}$.

Let Q be a subset of A. If we have a function $f\colon A \to B$, we can define the powerset function $P(f)\colon P(A) \to P(B)$, defined as $P(f)(Q) = f[Q]$.

Product

The product of two sets A and B, denoted by $A \times B$, is defined as the set of ordered pairs $\{(a, b) \mid a \in A, b \in B\}$.

The product of the functions $f\colon A \to B$ and $g\colon C \to D$ is the function

$$(f \times g)\colon (A \times C) \to (B \times D).$$

If $f(a) = b$ and $g(c) = d$, then $(f \times g)(a, c) = (b, d)$.

The product can be defined categorically by its universal property. Let A, B and C be objects from a category $\mathbb{C}$. The object $A \times B$ and two projection morphisms π_A, π_B form together a product of A and B, if for every object C and morphisms $C \xrightarrow{f} A$ and $C \xrightarrow{g} B$ there exists a unique morphism $C \xrightarrow{p} A \times B$. That means, if the following diagram

$$\begin{array}{ccccc} A & \xleftarrow{\pi_A} & A \times B & \xrightarrow{\pi_B} & B \\ & \nwarrow^{f} & \uparrow{\scriptstyle p} & \nearrow_{g} & \\ & & C & & \end{array}$$

commutes. The dashed arrow indicates that the morphism p is unique.

Coproduct

The coproduct of two sets A and B, marked $A + B$, is defined as the set of ordered pairs $\{(a, 1) \mid a \in A\} \cup \{(b, 2) \mid b \in B\}$.

The coproduct of the functions $f\colon A \to C$ and $g\colon B \to D$ is the function

$$(f + g)\colon (A + B) \to (C + D).$$

If $x = (a, 1)$, then $(f + g)(x) = (f(a), 1)$. If $x = (b, 2)$, then $(f + g)(x) = (g(b), 2)$.

Coproduct can be defined by an universal property as well. Moreover, its categorical definition is dual to the definition of a product. This means that to define the coproduct, we can take the commutative diagram defining the universal property of a product and reverse the orientation of the morphisms. In other words, the object $A + B$ and two injection morphisms κ_A, κ_B are together a coproduct of A and B, if the following diagram

$$\begin{array}{ccccc} A & \xrightarrow{\kappa_A} & A+B & \xleftarrow{\kappa_B} & B \\ & {\scriptstyle f}\searrow & \downarrow{\scriptstyle c} & \swarrow{\scriptstyle g} & \\ & & C & & \end{array}$$

commutes for every C.

A.2 Currying

Currying is often used when we try to transform automata into coalgebraic form. Basically it turns functions of many variables into a function of one variable [1]. Suppose we have a function

$$f\colon (A \times B) \to C,$$

which gives us $f(a,b) = c$. If the arguments are a and b, the result is c. There is another way to look at the function, though. We can view it as a function that takes only one argument and returns another function. In our example, we could create a function

$$\lambda.f\colon A \to C^B,$$

for which the following holds: $\lambda.f(a) = g$, where g is a function and $g(b) = c$. It is not hard to prove that there is a bijection between f and $\lambda.f$, and the bijection is, moreover, natural.