**Czech Technical University in Prague**
**Faculty of Electrical Engineering**
**Department of Computer Science**

# Induction and coinduction

Bachelor thesis

EMA MORVAYOVÁ

Study Programme: Software engineering and technologies
Thesis Supervisor: Ing. Matěj Dostál, Ph.D.

Prague, 2023

# ZADÁNÍ BAKALÁŘSKÉ PRÁCE

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

## I. OSOBNÍ A STUDIJNÍ ÚDAJE


Příjmení: **Morvayová** Jméno: **Ema** Osobní číslo: **492235**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**


## II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

**Indukce a koindukce**

Název bakalářské práce anglicky:

**Induction and coinduction**

Pokyny pro vypracování:

Induction and recursion are standard mathematical tools with wide applications in theoretical computer science, e.g. in the theory of algebraic data types. Coinduction and recursion are concepts dual to induction and recursion; they allow describing and working with infinite data structures. The goal of this project is to describe the relationship between induction and coinduction.
The student will study the relevant literature on the topic of induction and coinduction, study the necessary concepts from algebra and coalgebra theory, and create a mathematical text explaining the relationship between induction and coinduction in detail. The connection with computer science will be illustrated with appropriate examples from the theory of data types.

Seznam doporučené literatury:


J. Adámek, Introduction to Coalgebra, Theory and Applications of Categories, Vol. 14, No. 8 (2005), 157-199
J. J. M. M. Rutten, Universal coalgebra: a theory of systems, Theoretical Computer Science, Vol. 249, Issue 1 (2000), 3-80
D. Sangiorgi, J. J. M. M. Rutten, Advanced Topics in Bisimulation and Coinduction, Cambridge University Press 2011


Jméno a pracoviště vedoucí(ho) bakalářské práce:

**Ing. Matěj Dostál, Ph.D. katedra matematiky FEL**

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **15.02.2022** Termín odevzdání bakalářské práce: **15.08.2022**

Platnost zadání bakalářské práce: **19.02.2024**

| Ing. Matěj Dostál, Ph.D.<br>podpis vedoucí(ho) práce | podpis vedoucí(ho) ústavu/katedry | prof. Mgr. Petr Páta, Ph.D.<br>podpis děkana(ky) |
|---|---|---|

## III. PŘEVZETÍ ZADÁNÍ

Studentka bere na vědomí, že je povinna vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

| . <br>Datum převzetí zadání | Podpis studentky |
|---|---|



© VUT v Praze, Design: VUT v Praze, VIC

# Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

In Prague, 2023

..............................................
Ema Morvayová

# Acknowledgements

First and foremost, I would like to thank my thesis supervisor, Matěj Dostál. Without his help and valuable guidance throughout the process, this thesis would not have been accomplished.

I am very thankful to my friends for their support and understanding. I would also like to give special thanks to my best friend, Silvia Goldasová, whose words of encouragement “You can do this!” provided me strength and motivation when I needed it the most.

Finally, I am extremely grateful to my parents, whose unceasing love and support are always with me in whatever I pursue.

# Abstract

**[EN]** This thesis focuses on the relationship between induction and coinduction. Firsty, we explain the the necessary basics of category theory, mainly the notion of functor witch will play important role in the explanation of $F$-algebras and $F$-colgebras. The duality of induction and coinduction is then described by showing the duality between initial $F$-algebras and final $F$-colagebras. We support the explanations of these concepts with appropriate examples from data type theory to illustrate the connection with computer science.

***Keywords:*** *algebra, coalgebra, induction, coinduction, functor, initiality, finality, data type theory, data structures*

**[CZ]** Tato práce se zabývá vztahem mezi indukcí a koindukcí. Nejprve si představíme nezbytné základy teorie kategorií, především pojem functor, který bude hrát důležitou roli při výkladu $F$-algeber a $F$-kolgeber. Dualita indukce a koindukce je pak popsána skrze dualitu mezi počátečními $F$-algebrami a konečnými $F$-kolagebrami. Vysvětlení těchto pojmů podporujeme vhodnými příklady z teorie datových typů, abychom ilustrovali spojení s informatikou.

***Klíčova slova:*** *algebra, koalgebra, indukce, koindukce, funktor, počátečnost, konečnost, teorie datových typů, datové struktury*

# Contents


**Introduction** **1**

**1 Algebra** **3**

1.1 Category Theory . . . 3

1.1.1 Category . . . 3

1.1.2 Functor . . . 5

1.2 $F$-algebras . . . 7

1.3 Initial $F$-algebras and induction . . . 9

**2 Coalgebra** **15**

2.1 $F$-coalgebras . . . 15

2.2 Final $F$-coalgebras . . . 17

2.3 Coinduction . . . 20

**3 Relationship between induction and coinduction** **23**

**Conclusion and Future Work** **27**

**Bibliography** **29**

# Introduction

In computer science, standard algebraic techniques are used for representation of various essential data structures. Induction, one of the main algebraic tools, is used to define constructor-generated inductive data types such as finite lists or finite trees. These are modeled by initial algebras and they carry an algebraic structure.

While algebraic representation is sufficient for certain data types, it is difficult to algebraically describe the behavior of (potentially) infinite systems and structures. Examples of these are streams, infinite trees or classes in OOP. In order to deal with infinite data types, the final coalgebras (finality being the dual property of initiality for algebras) were used along with their logical reasoning principle - coinduction. This dual to induction, a less known or understood concept, is used both as a definition principle and as a reasoning principle for coinductive data types.

In this thesis we aim to provide a simplified explanation of the duality between induction and coinduction. The main goal is to explain these notions in understandable way and to show how are they connected to initial algebras and final coalgebras (of a functor). Also, we support our explanations with examples from data type theory in order to make the topic of coalgebra and coinduction better understandable and to show its usability in the world of computers as well.

We also need to mention that none of the notions defined in this thesis is a new finding, all the concepts and examples are well known, only put together in differently structured sections based on the desired outcome of this thesis.

In this thesis we explain the relationship between induction and coinduction and how the duality between these two notions is representable by the dual categorical notion of initiality and finality. The structure of the thesis is as follows:

Chapter 1 begins by introducing the notion of a category, functor and algebras for a functor (or $F$-algebras). We then proceed by explaining the concept of initial $F$-algebras and some of the uses of initiality in computer science. We finish the chapter by describing the connection of initial $F$-algebras and induction through an example from data type theory.

Chapter 2 introduces the notion dual to initial algebras (of a functor) - final $F$-coalgebras and its corresponding definition principle – coinduction.

In Chapter 3, we describe the relationship between induction and coinduction based on definitions from previous two chapters. The duality of these two principles is explained and shown through the dual categorical notions of initiality and finality.

# Chapter 1

# Algebra

The goal of this chapter is to reformulate the definition of induction in a more abstract way. Using the initiality of algebras (of functors), we get highly generic description of induction which can be applied to all kinds of algebraic data types and can be easily dualized, thus helping us explain the relationship between induction and coinduction.

In Section 1.1 we will briefly introduce the fundamentals of category theory which will be used throughout the thesis. We will describe the concept of categories and functors, by providing both formal definitions and examples. For more information on concepts from this section we refer the reader to [1] or [2].

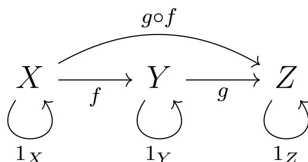
Section 1.2 deals with the concept of $F$-algebras and the last Section 1.3 is dedicated to explanation of initial algebras and induction using examples from data type theory. More thorough treatment of these concepts can be found in [3] and [4].

## 1.1 Category Theory

### 1.1.1 Category

In category theory, a category is a collection of *objects* and arrows between these objects, called *morphisms*. The definition of a category abstract the compositional properties of sets and mappings between sets.

**Example 1.1.1.** To illustrate on an example consider the category *Set* which consists of the collection of all sets together with the collection of all set functions. Objects of this category are sets and its morphisms are functions between them represented by arrows.




We can see that for each object $X$, there is an identity function $1_X : X \to X$ called the *identity morphism* of $X$, which sends all elements of $X$ to themselves. We can also observe that functions $f : X \to Y$ and $g : Y \to Z$ can be composed creating composite function or *composite morphism* $g \circ f : X \to Z$.

By abstracting the above example we get the definition of a categry.

**Definition 1.1.1.** A *category* $\mathcal{C}$ is a collection of *objects* denoted by $X, Y, Z, \ldots$ and *morphisms* $f : X \to Y$, $g : Y \to Z$, ... between them. Morphisms relate two objects with following operations:

- *Domain*, which assigns to each arrow $f$ an object $X =$ *dom(f)*
- *Codomain,* which assigns to each arrow $f$ an object $Y =$ *cod(f)*

These operations on $f$ can be represented by displaying $f$ as an actual arrow starting at its *domain* (or *source*) and ending at its *codomain* (or *target*):

$$f : X \longrightarrow Y \quad \text{or} \quad X \xrightarrow{f} Y$$

Objects and morphisms of a category must satisfy the following laws:

- For each object $X$, there is a morphism $1_X : X \to X$, called the *identity morphism* of $X$.
- For each morphism $f : X \to Y$ and each morphism $g : Y \to Z$, there is a morphism $g \circ f : X \to Z$, called the *composite* of $f$ and $g$. In other words, if codomain of one morphism matches the domain of another morphosm, then there is a *composite morphism.*
- For each morphism $f : X \to Y$ , the following equations hold: $1_Y \circ f = f$ , $f \circ 1_X = f$. This property of identity morphisms can be also observed on the following diagram:

$$\begin{array}{ccc} X & & \\ \downarrow 1_X & \searrow^{f} & \\ X & \xrightarrow{f} & Y \\ & \searrow_{f} & \downarrow 1_Y \\ & & Y \end{array}$$

- Each composition is associative, meaning $(h \circ g) \circ f = h \circ (g \circ f)$ for each

$$X \xrightarrow{f} Y \xrightarrow{g} Y \xrightarrow{h} Z$$

We continue with the definitions of *initial* and *final* objects which will be used later in the thesis when dealing with initial algebras and final coalgebras.

**Definition 1.1.2.** An object $I$ of a category $\mathcal{C}$ is called *initial* if, for each object $A$ in $\mathcal{C}$, there exists a unique morphism $I \to A$.

In other words, there is only one structure-preserving mapping from initial object $I$ to any $A$ in $C$.

**Definition 1.1.3.** An object $T$ of a category $\mathcal{C}$ is called *final* (or *terminal*) if, for each object $A$ in $\mathcal{C}$, there exists a unique morphism $A \to T$.

In *Set*, the category of sets, the *initial* object is the empty set $\emptyset$, as for each set $A$ there is unique function $\emptyset \to A$, also called the empty function. On the other hand, the *final* object in *Set* is any one-element set $S$ called *singleton*, as for each $A$ in *Set* there exists exactly one function $A \to S$, which sends every element of $A$ to the unique element of set $S$.

### 1.1.2 Functor

A *functor* is a mapping between categories that preserves categorical structure. So just like objects in a category are "linked through" arrows or morphisms between them, categories themselves are "linked through" functors, a morphism of categories.

More precisely, a functor $F : \mathcal{C} \to \mathcal{D}$ between two categories $\mathcal{C}$ and $\mathcal{D}$ maps objects from category $\mathcal{C}$ to objects in category $\mathcal{D}$ and morphisms from category $\mathcal{C}$ to morphisms in $\mathcal{D}$.

**Definition 1.1.4.** A functor $F : \mathcal{C} \to \mathcal{D}$ between two categories consists of a mapping of objects $F : \mathcal{C} \to \mathcal{D}$, which sends each object $X$ of $\mathcal{C}$ to an object $F(X)$ of $\mathcal{D}$, and a mapping of morphisms $F : \mathcal{C} \to \mathcal{D}$, which sends each morphism $f : X \to Y$ of $\mathcal{C}$ to a morphism $F(f) : F(X) \to F(Y)$ of $\mathcal{D}$, such that the following conditions are satisfied:

(i) For each object $X$ of $\mathcal{C}$, the morphism $F(1_X) : F(X) \to F(X)$ is equal to the identity morphism $1_{F(X)}$ of $F(X)$.

(ii) For each morphism $f : X \to Y$ and each morphism $g : Y \to Z$ of $\mathcal{C}$, the morphism $F(g \circ f) : F(X) \to F(Z)$ is equal to the composition $F(g) \circ F(f)$.

The following diagram represents these functor laws:

$$
\begin{array}{ccc}
X \circlearrowleft 1_X & \Longrightarrow & F(X) \circlearrowleft 1_{F(X)} \\
f \downarrow \searrow g \circ f & & F(f) \downarrow \searrow F(g \circ f) \\
Y \xrightarrow{g} Z & & F(Y) \xrightarrow{F(g)} F(Z)
\end{array}
$$

When $\mathcal{C} = \mathcal{D}$, we call functor $F$ an *endofunctor* on $\mathcal{C}$. That is, the endofunctor is a functor that maps a category to the same category. For example, for any category $\mathcal{C}$ the identity functor $1_{\mathcal{X}}$ is an endofunctor because it maps each object of $\mathcal{X}$ to itself and each morphism of $\mathcal{X}$ to itself.

**Example 1.1.2.** Consider the functor

$$\begin{aligned} Set &\xrightarrow{List} Set \\ X &\longmapsto List(X) \end{aligned}$$

$List(X)$ is the set of all finite sequences of elements of the set $X$, for a function

$$\begin{aligned} X &\xrightarrow{f} Y \\ x &\mapsto f(x) \end{aligned}$$

we can define

$$\begin{aligned} List(X) &\xrightarrow{List(f)} List(Y) \\ (x_1, ..., x_n) &\longmapsto (f(x_1), ..., f(x_n)) \end{aligned}$$

which sends a finite list $(x_1, ..., x_n)$ of elements of $X$ to the list $(f(x_1), ..., f(x_n))$ of elements of $Y$ by applying $f$ element wise.

This obviously constitutes a functor.

**Example 1.1.3.** Consider the functor

$$Set \xrightarrow{Stream} Set$$

On objects, *Stream* maps a set $X$ to the set of all infinite sequences (streams) of elements of $X$ (sequences indexed by natural numbers). On morphisms, *Stream* acts 'point-wise' as in the *List* example: given a function

$$X \xrightarrow{f} X \ ,$$

the sequence (stream) $(x_i)_{i=1}^{\infty}$ is mapped by

$$Stream(X) \xrightarrow{Stream(f)} Stream(Y)$$

to the stream $(f(x_i))_{i=1}^{\infty}$. The requirements for *Stream* to be a functor are again obviously satisfied.

The above example is dealing only with polynomial functor of the sort $Set \rightarrow Set$, acting on sets and functions between them. And just like in this example, throughout the thesis we will be dealing with functors acting on the category of sets and functions. These functors will be built up with identity functors, constants, products, coproducts and/or powersets.

## 1.2 $F$-algebras

The notion of $F$-algebras for an endofunctor $F$ allows us to study algebras more abstractly with categorical methods - it generalizes the usual notion of an algebra as it is defined in general algebra. This generality of definition of $F$-algebras is especially useful when dealing with representation of various finite data structures used in programming, such as lists or trees.

Using the terms and definitions from previous sections, we can now define algebra of functor or $F$-algebra, and show how functors can be used to describe signatures of operations.

**Definition 1.2.1.** Let $F$ be a functor. An *algebra of functor* $F$ (or, a $F$-algebra) is a pair consisting of a set $A$ and a function $a : F(A) \to A$. The set $A$ is called the carrier of the algebra, and the function $a$ the algebra structure, or the operation of the algebra.

The following are examples of $F$-algebras:

**Example 1.2.1.** Let us denote by $\mathbb{N}$ the set of natural numbers. We can endow this set with two operations: a nullary operation (choosing the number 0)

$$0 : 1 \longrightarrow \mathbb{N}$$

and a unary operation (the successor function)

$$\begin{array}{rcl} S : \mathbb{N} & \longrightarrow & \mathbb{N} \\ n & \longmapsto & n+1 \end{array}$$

These operations can be 'coupled together' into a single mapping

$$[0, S] : 1 + \mathbb{N} \longrightarrow \mathbb{N}$$

This shows that $\mathbb{N}$ endowed with 0 and $S$ can be modeled as an $F$-algebra for the functor $F : Set \to Set$ defined on objects by the assignment $x \mapsto 1 + x$.

(Here, by + we mean the disjoint sum operation on sets.)

**Example 1.2.2.** Let us denote by $P$ the *parity* set consisting of two elements $\{0, 1\}$. We can endow this set with two operations: a nullary operation (choosing the number 0)

$$0 : 1 \longrightarrow P$$

and a swaping operation (the successor function)

$$swap : P \longrightarrow P$$

These operations can be 'coupled together' into a single mapping

$$[0, swap] : 1 + P \longrightarrow P$$

.

This shows that $P$ endowed with 0 and $swap$ can be modeled as an $F$-algebra for the functor $F : Set \to Set$ defined on objects by the assignment $x \mapsto 1 + x$.

(Here, by + we mean the disjoint sum operation on sets.)

**Example 1.2.3.** Consider the data type $List(A)$ of lists whose elements are elements of set $A$, list-forming operations are:

$$\begin{array}{lrcl} \textbf{nil}: & 1 & \longrightarrow & List(A) \\ \textbf{cons}: & A \times List(A) & \longrightarrow & List(A) \end{array}$$

here **nil** is given by the empty list and **cons** which maps an element $a \in A$ and a list $\alpha = (a_1, ..., a_n) \in List(A)$ to the list $cons(a, \alpha) = (a, a_1, ..., a_n) \in List(A)$, obtained by prefixing $a$ to $\alpha$. Combining operations nil and cons we get an algebra structure:

$$[\textbf{nil, cons}]\colon (1 + A \times List(A)) \longrightarrow List(A),$$

which represents an algebra of a functor $F$ sending $X$ to $1 + (A \times X)$. (By $\times$ we mean the Cartesian product of sets.)

This representation of list type as one of algebraic data types is also used in functional programming languages such as Haskell and ML.

Here is a more concrete example of how a list, specifically singly linked list, would be declared in Haskell:

```
data List a = Nil | Cons a (List a)
```

Here, `Nil` represents an empty list and operation `Cons x xs` represents combination of a new element $x$ with a list $xs$ creating a new list.

This definition of a list differs for many languages. For instance, in Haskell we can also use [] for `Nil`, and : or :: for `Cons`. So `Cons 1 (Cons 2 (Cons 3 Nil))` would be written as `1:2:3:[]` or `[1,2,3]`.

**Example 1.2.4.** Consider the functor

$$\begin{array}{lcc} T: Set & \longrightarrow & Set \\ X & \longmapsto & 1 + (X \times A \times X) \end{array}$$

We can form an algebra $1 + (Tree(A) \times A \times Tree(A)) \to Tree(A)$, where $Tree(A)$ is set of $A$-labeled finite binary trees, by considering two operations:

$$nil: 1 \quad \longrightarrow \quad Tree(A)$$

choosing the empty tree, and

$$Tree(A) \times A \times Tree(A) \quad \longrightarrow \quad Tree(A)$$

constructing a tree out of two (sub)trees and a labelled node as the new root.

Binary trees with elements at the leaves is another example of an algebraic data type. Implementation of this type of algebraic structure would in Haskell look like this:

```
data Tree = Empty | Leaf Int | Node Tree Tree
```

Here, Empty represents an empty tree, data constructor `Leaf` is a function `Int -> Tree`, an argument of type integer produces a value of the type `Tree`. `Node` organizes the data into branches by taking two arguments of the type `Tree` itself - it is a recursive data type.

On the examples above we can see that there are many different $F$-algebras for the same functor $F$ and that in fact, an $F$-algebra does not have to be unique, unlike initial $F$-algebras.

Having an idea of what algebras of a functor are and how they look like, we now define the notion of a homomorphisms of algebras. It is a structure preserving functions between algebras, that is between the carrier sets of the algebras which commutes with the operations.

**Definition 1.2.2.** Let $F$ be a functor with algebras $a : F(A) \to A$ and $b : F(B) \to B$ . A *homomorphism of algebras*, or an algebra map from $(A, a)$ to $(B, b)$ is a function $f : A \to B$ between the carrier sets which commutes with the operations: $f \circ a = b \circ F(f)$ in

$$\begin{array}{ccc} F(A) & \xrightarrow{F(f)} & F(B) \\ {\scriptstyle a}\downarrow & & \downarrow{\scriptstyle b} \\ A & \xrightarrow[f]{} & B \end{array}$$

Using the notion of homomorphism of algebras we can now formulate the concept of "initiality" for algebras (of a functor) and give the formal definition of initial $F$-algebras.

## 1.3 Initial $F$-algebras and induction

The notion of initial algebras captures algebraic structures which are generated by constructor operations, and gives rise to the familiar principles of definition by induction and proof by induction. Using the notion of homomorphism of algebras explained in previous section, we can now formulate the concept of "initiality" for algebras (of a functor) and give the formal definition of initial $F$-algebras.

An algebra of a functor $F$ is *initial* if for an arbitrary algebra of the same functor there is a unique homomorphism of algebras.

$$\begin{pmatrix} \text{initial} \\ \text{algebra} \end{pmatrix} \xrightarrow[\text{homomorphism}]{\text{unique}} \begin{pmatrix} \text{arbitrary} \\ \text{algebra} \end{pmatrix}$$

[3]

Initiality requires *unique existence* which has two aspects:

- **existence** of an algebra map out of the initial algebra to another algebra, and
- **uniqueness** in the form of equality of any two algebra maps going out of the initial algebra to some other algebra.

Existence corresponds to ordinary definition by induction, meanwhile uniqueness will be used as an inductive proof principle. Uniqueness proofs are done by showing that two functions acting on an initial algebra are the same by showing that they are both homomorphisms (to the same algebra). [5]

**Definition 1.3.1.** An algebra $a : F(A) \to A$ of a functor $F$ is **initial** if for each algebra $b : F(B) \to B$ there is a unique homomorphism of algebras $f : A \to B$ from $(A, a)$ to $(B, b)$, such that $f \circ a = b \circ F(f)$. In other words, an initial $F$-algebra is an $F$-algebra $(A, a)$ such that $A$ is the initial object in the category of $F$-algebras. Following is the diagram of initial algebra, its uniqueness is expressed by a dashed arrow:

$$\begin{array}{ccc} F(A) & \overset{F(f)}{\dashrightarrow} & F(B) \\ \downarrow a & & \downarrow b \\ A & \underset{f}{\dashrightarrow} & B \end{array}$$

The above definition of initial algebras also captures algebraic structures which are generated by constructor operations. Constructors contain instructions on how to generate (algebraic) data elements. An example being empty list constructor *nil* and the prefix operation *cons*. Data types obtained through these constructor operations, that is they are defined by initial $F$-algebras, are known as algebraic data types.

Recall examples 1.2.1 and 1.2.2 from the previous section. The algebra from example 1.2.1 is an initial F-algebra for the functor $F(X) = 1 + X$. We will now illustrate this on the following example of homomorphisms of algebras.

**Example 1.3.1.** The mapping $[0, S] : 1 + \mathbb{N} \to \mathbb{N}$ is an initial algebra of this functor $F$. We will show how initiality of $[0, S]$ corresponds to the principle of definition by induction using Example 1.2.2. Recall that in this example we defined the $F$-algebra $[0, swap] : 1 + P \to P$. Specifying such algebra corresponds precisely to a definition of a function $p : \mathbb{N} \to P$,

$$p(0) = 0 \tag{1.1}$$

and

$$p(n+1) = \begin{cases} 0, & p(n) = 1 \\ 1, & p(n) = 0 \end{cases} \tag{1.2}$$

by induction. In fact, recall that initiality of $[0, S] : a + \mathbb{N} \to \mathbb{N}$ implies that there is a unique map $p : \mathbb{N} \to P$ such that the diagram

$$\begin{array}{ccc} 1 + \mathbb{N} & \xrightarrow{1+p} & 1 + P \\ \downarrow{\scriptstyle [0,S]} & & \downarrow{\scriptstyle [0,swap]} \\ \mathbb{N} & \xrightarrow[p]{} & P \end{array}$$

commutes. In elementary terms, $p \circ [0, S] = [0, swap] \circ (1 + p)$ must hold for every element $x \in 1 + \mathbb{N}$. We will show this requirement entails for the two possible cases: $x \in 1$ and $x \in \mathbb{N}$.

Starting in the upper-left corner of the commuting diagram with the first case $x = \{*\} \cup \mathbb{N}$ and applying $p \circ [0, S]$ we get

$$p([0, S](\{*\} \cup \mathbb{N})) = p(0) = 0,$$

and for $[0, swap] \circ (1 + p)$ we get

$$[0, swap]((1 + p)(\{*\} \cup \mathbb{N})) = [0, swap](\{*\} \cup 0, 1) = p(0) = 0.$$

For the second case $x = n$ we do the same:

$$p(S(n)) = p(n + 1) = swap(p(n)) = swap((1 + p)(n)).$$

The **existence** of a homomorphism $p : (\mathbb{N}, [0, S]) \to (P, [0, swap])$ thus states that there exists a function $p : \mathbb{N} \to P$ satisfying 1.1 and 1.2.

The **uniqueness** of this homomorphism corresponds to the fact that the requirements 1.1 and 1.2 on $p$ specify the function $p$ uniquely.

We have seen an example of an initial algebra from general algebra. Next, we will show an example of how an algebraic data type can be represented as an initial algebra, and we will show how initiality can be used to define functions by induction. It requires that one puts an appropriate algebra structure on the codomain of the intended function, corresponding to the induction clauses that determine the function. As an example we will take the *length* function which is taking as an argument a list and returns its length.

**Example 1.3.2.** Consider a fixed set $A$ and the earlier-mentioned list-functor $F(X) = 1 + (A \times X)$. The initial algebra of this functor $F$ is the set $List(A) = \cup_{n=N} A^n$ of finite sequences of elements of $A$, together with the following functions:

- $1 \to List(A)$ given by the empty list $nil = ()$

- $A \times List(A) \to List(A)$ which maps an element $a \in A$ and a list $\alpha = (a_1, ..., a_n) \in List(A)$ to the list $cons(a, \alpha) = (a, a_1, ..., a_n) \in List(A)$, obtained by prefixing $a$ to $\alpha$.

These two functions can be combined into a single function:

$$[nil, cons] : 1 + (A \times List(A)) \to List(A) \ .$$

This is an initial algebra as there is a unique homomorphism $f : List(A) \to X$ of algebras making the following diagram commute:

$$\begin{array}{ccc} 1 + (A \times List(A)) & \xrightarrow{id+(id\times f)} & 1 + (A \times X) \\ {\scriptstyle [nil,cons]}\downarrow & & \downarrow{\scriptstyle [x,y]} \\ List(A) & \xrightarrow[f]{} & X \end{array}$$

for an arbitrary algebra $[x, y] : 1 + (A \times X) \to X$ of the list-functor $F$. In the above diagram, where $f$ can be specified as follows:

$$f(\alpha) = \begin{cases} x & \text{if } \alpha = nil \\ y(a, f(\beta)) & \text{if } \alpha = cons(a, \beta) \end{cases} \tag{1.3}$$

**Example 1.3.3.** Now that we seen how initiality can be used to define general function by induction, we will show on the specific example of the *length* function $len : List(A) \to \mathbb{N}$, which can be inductively defined as

$$len(\alpha) = \begin{cases} 0 & \text{if } \alpha = nil \\ len(\alpha') + 1 & \text{if } \alpha = cons(a, \alpha') \end{cases} \tag{1.4}$$

for all $a \in A$ and $\alpha \in A$, which returns length of finite list $\alpha$. In other words, for every empty list (also called *nil*) the function returns value 0, and for every list of form $cons(a, \alpha')$, that is a list created from list $\alpha'$ by prefixing $a$, it returns $len(\alpha')+1$.

For this function we consider two operations: nullary operation

$$\begin{array}{lcc} x : 1 & \longrightarrow & \mathbb{N} \\ \emptyset & \longmapsto & n + 1 \end{array}$$

and the successor function

$$\begin{array}{lcc} y : A \times \mathbb{N} & \longrightarrow & \mathbb{N} \\ (a, n) & \longmapsto & n + 1 \end{array}$$

These operations can be 'coupled together' into a single mapping

$$[x, y] : 1 + (A \times \mathbb{N}) \ \longrightarrow \ \mathbb{N}$$

This is an initial algebra as there is a unique homomorphism $len : List(A) \to \mathbb{N}$ making the following diagram commute:

$$\begin{array}{ccc} 1 + (A \times List(A)) & \xrightarrow{id+(id \times len)} & 1 + (A \times \mathbb{N}) \\ \Big\downarrow{\scriptstyle [nil,cons]} & & \Big\downarrow{\scriptstyle [x,y]} \\ List(A) & \xrightarrow[len]{} & \mathbb{N} \end{array}$$

Specification of operations align with the definition for $List(A) \xrightarrow{len} \mathbb{N}$ by induction.

# Chapter 2

# Coalgebra

In this chapter we introduces the notion of (final) $F$-coalgebra, which is categorical dual to the initial algebra (of a functor). Within the theory of coalgebras, the corresponding proof and definition principle coinduction will also be explained providing the necessary definitions and appropriate examples. For more detailed treatment, see source [3], [4] and [6], listed in Reference section. For the related topic to coalgebra theory we refer the reader to see also [7].

## 2.1 $F$-coalgebras

To better understand the duality between algebras and coalgebras, we can think of the difference between an inductively defined data type in a functional programming language (an algebra) and a class in an object-oriented programming language (a coalgebra).

The algebraic data type is determined by its *constructors*: algebraic operations of the form $F(X) \to X$ going into the data type. The class on the other hand contains an internal state, given by the values of all the public and private fields of the class. Using pubic fields and methods, one can observe and modify this state. Operations of a class act on a state (or object) and are naturally described as *destructors* pointing out of the class: they are of the coalgebraic form $X \to F(X)$.

**Definition 2.1.1.** A *coalgebra* of a functor $F$ (or a *$F$-coalgebra*) is a pair $(A, a)$ consisting of a set $A$ and a function $a : A \to F(A)$. Similarly as for algebras, the set $A$ is called the *carrier* or the *state space* and the function $a$ the *structure* or *operation* of the coalgebra $(A, a)$.

Essentially, the difference betweenan algebra $F(X) \to X$ and a coalgebra $X \to F(X)$ is the difference between construction and observation:

- An algebra tells us how to construct elements in $X$ with a function $F(X) \to X$ going into this carrier set $X$.

- A coalgebra does not tell us how to form elements of $X$, it only gives us some information about $X$ with a function $X \to F(X)$ going out of carier set $X$.

Both coalgebras and algebras can be seen as models of a signature of operations, but while algebras model signatures of constructor operations, coalgebras of destructor/observer operations.

In other words, while an algebra have constructor functions that build members of the carrier set, a coalgebra has destructor functions that split members of the carrier set into the components they are built from.
So the carrier of a coalgebra is the *domain of its destructors* (or *observers*) which tell us what we can observe about its data elements. An example being, the head and tail operations which tell us all about infinite lists: head gives a direct observation, and tail returns a next state.

In computer science, the first systematic approach to data types relied on initiality of algebras. These algebraic structures generate finite objects, while many data types of interest consist of infinite objects, such as streams or infinite trees. The need for such infinite structures brought to use the (final) coalgebras, which can be mainly represented in functional programming languages (such as Haskell) or in logical programming languages. [2]

For a better initial understanding of coalgebras, we start with some examples of a coalgebra.

**Example 2.1.1.** Consider a machine with two buttons: *value* and *next*. Pressing the value-button gives an observation related to the current state of the machine, an element of a fixed set $A$. Meanwhile by pressing the next-button the machine transitions into a new internal state. This machine can be represented as an $F$-coalgebra:

$$\langle val, next \rangle : X \to A \times X$$

We may imagine the carrier set $X$ of this coalgebra as a *black box*. By doing so, the observable behaviour of the machine consists only of the sequence of values that can be seen after consecutively presses of the 'next' button.

Let $F$ be the functor from *Set* to *Set* defined by $F(X) = A \times X$. For each state $x \in X$ we may associate its observable behaviour:
the infinite sequence

$$(a_0, a_1, a_2, ...) \in A^{\mathbb{N}}$$

where each $a_i$ corresponds to the *value* that can be seen after pressing the *next* button $i$ times.

Let us take the set $A = \{a, b\}$, meaning that the possible value of $a_i$ is either $a$ or $b$; and set $X = \{0, 1\}$. From this we can form an coalgebra

$$X \xrightarrow{\langle val, next \rangle} A \times X.$$

Let us now consider a concrete example of such a machine, which consist of two functions **val:** $X \to A$ and **next:** $X \to X$. With these two operations, given an element $x \in X$ we can produce an element in $A$ with $val(x)$, and we can produce next element in $U$ with $next(x)$. The functions are defined as follows:

$$\begin{array}{ll} val(0) = a & next(0) = 1 \\ val(1) = b & next(1) = 0 \end{array}$$

**Definition 2.1.2.** A *homomorphism of coalgebras* (or coalgebra map) from a $F$-coalgebra $A \to F(A)$ to another $F$-coalgebra $B \to F(B)$ consists of a function $f : A \to B$ between the carrier sets which commutes with the operations: $b \circ f = F(f) \circ a$ as expressed by the following diagram.

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ \downarrow a & & \downarrow b \\ F(A) & \xrightarrow[F(f)]{} & F(B) \end{array}$$

Similarly as we did in previous chapter, using the notion of homomorphism of coalgebras we can now formulate the concept of "finality" for coalgebras (of a functor) and give the formal definition of final (or terminal) $F$-coalgebras.

## 2.2 Final $F$-coalgebras

The notion of a final coalgebra can be used to characterize infinite data types in a way that is similar to how algebraic data types are characterized.

A coalgebra (of a functor) is *final* if for an arbitrary coalgebra (of the same functor), there is a *unique homomorphism* of coalgebras.

$$\begin{pmatrix} \text{arbitrary} \\ \text{coalgebra} \end{pmatrix} \xrightarrow[\text{homomorphism}]{\text{unique}} \begin{pmatrix} \text{final} \\ \text{coalgebra} \end{pmatrix}$$

[3]

**Definition 2.2.1.** A coalgebra $a : A \to F(A)$ of a functor $F$ is **final** if for each coalgebra $b : B \to F(B)$ there is a unique homomorphism of coalgebras $f : B \to A$ from $(B, b)$ to $(A, a)$, such that $a \circ f = F(f) \circ b$. In other words, a final $F$-coalgebra is an $F$-coalgebra $(A, a)$ such that $A$ is the final object in the category of $F$-coalgebras. Following is the diagram of final coalgebra with uniqueness expressed by a dashed arrow:

$$\begin{array}{ccc} B & \xrightarrow{f} & A \\ \downarrow b & & \downarrow a \\ F(B) & \xrightarrow[F(f)]{} & F(A) \end{array}$$

If we compare the above definition to the definition of initial algebras we see that while the initiality defines function out of an initial algerba, the property of finality for coalgebras allow us to define function into a final coalgebra

Now let's see how final $F$-coalgebras can be used to represent data types. Let us consider the data type of natural numbers. We can represent this data type as a final $F$-coalgebra as follows:

**Example 2.2.1.** Consider the set $A^N$ of infinite sequences over $A$, or *streams*, which can be represented as a coalgebra structure by declaring

$$head(\alpha) = a_0$$
$$\text{and}$$
$$tail(\alpha) = (a_1, a_2, a_3, ...)$$

for $\alpha = (a_0, a_1, a_2, ...) \in A^N$, so we get

$$\langle \text{head, tail} \rangle : A^N \to A \times A^N$$

forming an final $F$-coalgebra of the functor $F(X) = A \times X$. We will show that this is in fact final coalgenra through homomorphisms of coalgebras by using an example 2.1.1 of an $F$-coalgebra $\langle val, next \rangle : X \to A \times X$ . The coalgebra map from $\langle val, next \rangle$ to $\langle head, tail \rangle$ is a function $b : X \to A^{\mathbb{N}}$ defined by following operations:

$$b(0) = (a, b, a, b, a, ...)$$
$$b(1) = (b, a, b, a, b, ...)$$

The commuting diagram of the function $b$ is following:

$$\begin{array}{ccc} X & \xrightarrow{b} & A^{\mathbb{N}} \\ \downarrow{\scriptstyle \langle val,next \rangle} & & \downarrow{\scriptstyle \langle head,tail \rangle} \\ A \times X & \xrightarrow[A \times b]{} & A \times A^{\mathbb{N}} \end{array}$$

Now we have to show that $\langle head, tail \rangle \circ b = (A \times b) \circ \langle val, next \rangle$ must hold for every element $x \in X$. Because $X = \{0, 1\}$ we will show this for two possible cases of such $x$ which are $x = 0$ and $x = 1$.

Starting in the upper-left corner of the commuting diagram with the first case $x = 0$ and applying $(A \times b) \circ \langle val, next \rangle$ we get

$$(A \times b)(\langle val, next \rangle(0)) = (A \times b)(a, 1) = (a, babab...),$$

and for $\langle head, tail \rangle \circ b$ we get

$$\langle head, tail \rangle(b(0)) = \langle head, tail \rangle(ababab...) = (a, babab...).$$

For the second case $x = 1$ we do the same:

$$(A \times b)(\langle val, next \rangle(1)) = (A \times b)(b, 0) = (b, ababa...)$$
$$\langle head, tail \rangle(b(1)) = \langle head, tail \rangle(bababa...) = (b, ababa...)$$

Hence we can see that $(A \times b) \circ \langle val, next \rangle$, for all $x \in X$, which proves one aspect of final coalgebras - an existence of an coalgebra map out of the arbitrary coalgebra to a final coalgebra.

The uniqueness of $b$ means that $b$ is the only map making the above diagram commute. This can be shown by proving that any other coalgebra map $f : (X, \langle val, next \rangle) \to (A^{\mathbb{N}}, \langle head, tail \rangle)$ is equal to $b$. Explicitly, we need to show that $f(0) = (a, b, a, b, a, ...)$ and $f(1) = (b, a, b, a, b, ...)$. This can be proved by a straightforward induction.

We can also represent other infinite data types using final $F$-coalgebras.

**Example 2.2.2.** Consider the $Set$-endofunctor

$$F(X) = X \times A \times X.$$

An $F$-algebra for this functor is a mapping

$$f : X \longrightarrow X \times A \times X$$

which can be thought of as a 'branching machine' with an output of type $A$. Any such mapping $f$ can be described as a triple $\langle left, val, right \rangle$ of mappings, where $val : X \to A$ represents the output of the machine at a given interval state, whereas the maps $left : X \to X$ and $right : X \to X$ give the 'left and right' successor states, respectively.

**Example 2.2.3.** Consider a $F$-coalgebra from the previous example. The behaviour of every such $F$-coalgebra at a given state can be modeled by an infinite binary tree with nodes labelled by elements of set $A$. Considering all such trees, we obtain a set $InfTree(A)$ which itself can be endowed with coalgebraic structure. We have a mapping

$$\langle l, o, r \rangle : InfTree(A) \longrightarrow InfTree(A) \times A \times InfTree(A)$$

which, given an infinite binary tree $t$, outputs its left subtree $l(t)$, right subtree $r(t)$, and the label $o(t)$ of its root, respectively.

It is possible (but out of the scope of this text) to show that the coalgebra $\langle l, o, r \rangle$ is final for $F$.

Final $F$-coalgebras have a number of useful properties. For example, they are unique up to isomorphism. This means that if $(X, f)$ and $(Y, g)$ are both final $F$-coalgebras on the same set $A$, then there is an isomorphism of $F$-coalgebras $(X, f) \to (Y, g)$.

## 2.3 Coinduction

Coinduction is the dual principle to induction and is used to prove properties of coinductively-defined data types such as infinite streams, infinite trees, and coterms. These data types are typically final coalgebras of a functor.

In this section we introduce the concept of coinduction. We will give the definition of coinduction. In the previous section, we have shown how finality can be used to define elements of infinite data types. In this section we show what role final coalgebras play in the notion of coinduction. Coinduction is a way of defining functions on infinite objects, such as infinite lists. The key idea is to define a function on an infinite object by recursion on a final coalgebra.

**Definition 2.3.1.** Let $A$ be a set and $(B, b)$ a final $F$-coalgebra. To give a *coinductive definition* of a function $f : A \longrightarrow B$ is to specify an $F$-coalgebra $(A, a)$ (with the carrier set $A$).

**Example 2.3.1.** In this example we will show how the finality of the coalgebra

$$\langle head, tail \rangle : A^{\mathbb{N}} \to A \times A^{\mathbb{N}}$$

can be used to give a coinductive definition of a function

$$even : A^{\mathbb{N}} \to A^{\mathbb{N}}$$

which, for every stream $a = (a_0, a_1, a_2, ...)$, return the stream $even(a) = (a_0, a_2, a_4, ...)$ of all the elements of $a$ with even indices. (In other words, $even(a_i)_{i=0}^{\infty} = (a_{2i})_{i=0}^{\infty}$ .)

First, let us write down a specification of the *even* function. The heads of the stream $a$ and of the stream $even(a)$ should be the same:

$$head(even(a)) = head(a).$$

Second, to compute the rest of the stream (i.e., the stream $tail(even(a))$ ), is the same thing as removing the first two elements of $a$ and applying to the resulting stream:

$$tail(even(a)) = even(tail(tail(a))).$$

To show that these two requrements indeed define uniquely a function $even : A^{\mathbb{N}} \to A^{\mathbb{N}}$, we will show that there is a suitable coalgebra

$$\langle x, y \rangle : A^{\mathbb{N}} \to A \times A^{\mathbb{N}}$$

such that the unique homomorphism from $\langle x, y \rangle$ to $\langle head, tail \rangle$ satisfies the requirements for **even**.

Indeed, choose the mappings

$$x : A^{\mathbb{N}} \to A \text{ and } y : A^{\mathbb{N}} \to A^{\mathbb{N}}$$

as follows:

$$x = head \text{ and } y = tail \circ tail.$$

If we call the resulting homomorphism *even*, we see that it makes the diagram

$$\begin{array}{ccc} A^{\mathbb{N}} & \xrightarrow{even} & A^{\mathbb{N}} \\ \downarrow{\scriptstyle \langle head, tail \circ tail \rangle} & & \downarrow{\scriptstyle \langle head, tail \rangle} \\ A \times A^{\mathbb{N}} & \xrightarrow[id \times even]{} & A \times A^{\mathbb{N}} \end{array}$$

commute. Now taking an arbitrary stream $a$ and following it through the commutative diagram yields

$$\langle head, tail \rangle (even(a)) = (id \times even) \circ (\langle head, tail \circ tail \rangle)(a)$$

This equality can be rewritten as two equalities:

$$head(even(a)) = id(head(a)) = head(a) \text{ and } tail(even(a)) = even(tail(tail((a))).$$

However, these are precisely the equations given by the requirements for the function *even*.

We have thus used finality of $\langle head, tail \rangle$ to coinductively define *even*.

## Chapter 3

# Relationship between induction and coinduction

In this thesis, we have explored the relationship between induction and coinduction, and how they are dual due to them being representable by dual categorical notions of initiality and finality.

Informally, the key difference between induction and coinduction is well expressed by the following remark: "A property holds by induction if there is good reason for it to hold; whereas a property holds by coinduction if there is no good reason for it not to hold" [5]. This can be also seen as a consequence of the duality between initial algebras and final coalgebras. Because if a property $p$ holds for the constructors of an initial algebra, then it also holds for all data elements generated by these constructors. Meanwhile, if a property $p$ holds for the destructors of a final coalgebra, then it also holds for all data elements that can be observed by these destructors.

The duality between induction and coinduction can be also described/understood as the duality between least and greatest fixed points (of a monotone function). These notions generalise to least and greatest fixed points of a functor, which are suitably described as initial algebras and final coalgebras, see [7] for more details in this direction.

In this thesis we firstly introduced the fundamentals of category theory by providing definitions of category and functor. Examples of functors were provided for both finite and infinite data types:

$$Set \xrightarrow{List} Set \qquad \text{and} \qquad Set \xrightarrow{Stream} Set$$

The notion of functor plays important role throughout this thesis as it describes signatures of operations of algebras and colagebras of a functor. $F$-algebras and $F$-coalgebras generalize the notion of general algebras and coalgebras, which we used for representation of finite and infinite data structures.

Using definitions and examples of different data structures we explained the main difference between $F$-algebras and $F$-coalgebras through the difference between construction and observation.

An algebra $F(X) \to X$ can be seen as model of a signature of constructor operations and it tells us how to construct elements in $X$ with a function $F(X) \to X$ going into this carrier set $X$. In other words, constructors of $F$-algebras contain instructions on how to generate (algebraic) data elements. As an example of such constructor operations we introduced empty list constructor *nil* and the prefix operation *cons* (Example 1.2.3).

A coalgebra $X \to F(X)$ can be seen as model of signature of destructor (or observer) operations while it gives us information about $X$ with a function $X \to F(X)$ going out of carrier set $X$, but it does not give us instructions on how to form elements of $X$. That is, the carrier of a coalgebra is the domain of its destructors which tell us what we can observe about its data elements. An example of such destructor operations of infinite lists were *head* and *tail*, head gives a direct observation and tail returns a next state (Example 2.3.1).

After understanding the duality of $F$-algebras and $F$-coalgebras, we then defined the notion of *homomorphism* between such algebras (and coalgebras) of a functor which allowed us to define initial algebras (and final coalgebras).

The *initiality* of $F$-algebras provided us more abstract definition of induction which can be applied to all kinds of algebraic data types. We showed this on an example (1.3.2) of list-functor $F(X) = 1+(A\times X)$ by showing that there is a unique homomorphism $f : List(A) \to X$ from initial algebra $[nil, cons] : 1+(A\times List(A)) \to List(A)$ to an arbitrary algebra $[x, y] : 1 + (A \times X) \to X$ making the following diagram commute:

$$\begin{array}{ccc} 1+(A\times List(A)) & \xrightarrow{id+(id\times f)} & 1+(A\times X) \\ \downarrow{\scriptstyle [nil,cons]} & & \downarrow{\scriptstyle [x,y]} \\ List(A) & \xrightarrow[f]{} & X \end{array}$$

On the other hand, the *finality* of $F$-coalgebras, which are categorical duals to the initial algebras (of a functor), can be used to characterize infinite data types in a way that is similar to how algebraic data types are characterized. As we mentioned before, coalgebras have no construction operations for constructing elements in state space of a coalgebra. However, we can form elements in state space of a certain coalgebra if we know that this coalgebra is final. As we explained in the previous chapter, coidunction is the dual principle to induction and is used to define functions on infinite objects by recursion on a final coalgebra. We showed this in the example (2.3.1) of final coalgebra $\langle head, tail\rangle : A^{\mathbb{N}} \to A \times A^{\mathbb{N}}$. By using the finality of this coalgebra we gave a coinductive definition of a function $even : A^{\mathbb{N}} \to A^{\mathbb{N}}$ which makes the following diagram commute:

$$\begin{array}{ccc} A^{\mathbb{N}} & \xrightarrow{even} & A^{\mathbb{N}} \\ \downarrow{\scriptstyle \langle head, tail\circ tail\rangle} & & \downarrow{\scriptstyle \langle head, tail\rangle} \\ A\times A^{\mathbb{N}} & \xrightarrow[id\times even]{} & A\times A^{\mathbb{N}} \end{array}$$

This duality between constructing and observing contributes to understanding the difference between inductive and coinductive definition of a function. An inductive definition of a function $f$ defines its value of $f$ on all constructors, while the coinductive definition defines its value of $f$ of all destructors on each outcome $f(x)$. In other words, it determines the observable behavior of each $f(x)$. Therefore, by understanding the duality of notions of initiality and finality, and how they can be used to define functions, we can understand the relationship between induction and coinduction.

# Conclusion and Future Work

In this thesis we dealt with the notions of induction and coinduction with the aim to provide the explanation of the relationship between these two concepts. The main observation was that duality of these two concepts can be shown through the dual categorical notion of initiality and finality. More specifically, we explored how are they connected to initial algebras and final coalgebras (of a functor). We supported our explanations of these concepts by illustrating on examples from computer science - we shown how different finite and infinite data types and functions can be represented as initial $F$-algebras and final $F$-coalgebras, respectively.

We also mentioned that the notion of coinduction can be used not only only as a definition principle (that is to define possibly infinite data types) but also as a proof principle. However, such theoretical concepts would require more detailed explanations than those presented in this thesis. Moreover, an interesting point of view on relationship between induction and coinduction worth exploring is through least and greatest fixed point of a monotone function, which is well explained in some of the sources cited in this work. We therefore defer these topics to future work.

# Bibliography


1. MACLANE, Saunders. *Categories for the Working Mathematician*. Springer-Verlag, 1971. Graduate Texts in Mathematics, Vol. 5.
2. JACOBS, Bart. *Introduction to coalgebra : towards mathematics of states and observation*. 2017. Cambridge tracts in theoretical computer science. Available from DOI: 10.1017/cbo9781316823187.
3. JACOBS, Bart; RUTTEN, Jan. An introduction to (co)algebra and (co)induction. In: SANGIORGI, Davide; RUTTEN, Jan J. M. M. (ed.). *Advanced Topics in Bisimulation and Coinduction*. Cambridge University Press, 2012, sv. 52, pp. 38–99. Cambridge tracts in theoretical computer science.
4. ADÁMEK, Jiří. Introduction to coalgebra. *Theory and Applications of Categories [electronic only]*. 2005, roč. 14, pp. 157–199. Available also from: http://eudml.org/doc/125538.
5. KOZEN, DEXTER; SILVA, ALEXANDRA. Practical coinduction. [B.r.]. Available from DOI: 10.1017/s0960129515000493.
6. RUTTEN, J.J.M.M. Universal coalgebra: a theory of systems. [B.r.]. Available from DOI: 10.1016/s0304-3975(00)00056-6.
7. SANGIORGI, Davide. Origins of bisimulation and coinduction. In: *Advanced Topics in Bisimulation and Coinduction*. Cambridge University Press, 2012, sv. 52, pp. 1–37. Cambridge tracts in theoretical computer science.