

# Category Theory by Example

Ivan Murashko, Alexey Radkov, Marat Minshin

February 3, 2019



# Contents

<b>1</b>	<b>Base definitions</b>	<b>13</b>
1.1	Definitions . . . . .	13
1.1.1	Object . . . . .	13
1.1.2	Morphism . . . . .	14
1.1.3	Category . . . . .	16
1.2	<b>Set</b> category example . . . . .	18
1.3	Programming languages examples . . . . .	23
1.3.1	<b>Hask</b> toy category . . . . .	24
1.3.2	<b>C++</b> toy category . . . . .	25
1.3.3	<b>Scala</b> toy category . . . . .	26
1.4	Quantum mechanics examples . . . . .	27
<b>2</b>	<b>Objects and morphisms</b>	<b>31</b>
2.1	Equality . . . . .	31
2.2	Initial and terminal objects . . . . .	32
2.3	Product and sum . . . . .	34
2.4	Category as a monoid . . . . .	35
2.5	Exponential . . . . .	36
2.6	Programming language examples. Type algebra . . . . .	38
2.6.1	<b>Hask</b> category . . . . .	38
2.6.2	<b>C++</b> category . . . . .	39
2.6.3	<b>Scala</b> category . . . . .	41
2.7	Quantum mechanics . . . . .	41
<b>3</b>	<b>Curry-Howard-Lambek correspondence</b>	<b>43</b>
3.1	<b>Proof</b> category . . . . .	43
3.2	Linear logic and Linear types . . . . .	45
3.3	Quantum logic and quantum computation . . . . .	46

<b>4</b>	<b>Functors</b>	<b>47</b>
4.1	Definitions . . . . .	47
4.2	<b>Cat</b> category . . . . .	49
4.3	Contravariant functor . . . . .	50
4.4	Bifunctors . . . . .	51
<b>5</b>	<b>Natural transformation</b>	<b>53</b>
5.1	Definitions . . . . .	53
5.2	Category of functors . . . . .	55
5.3	Operations with natural transformations . . . . .	57
5.4	Polymorphism and natural transformation . . . . .	58
5.4.1	<b>Hask</b> category . . . . .	59
<b>6</b>	<b>Monads</b>	<b>61</b>
6.1	Monoid in <b>Set</b> category . . . . .	61
6.1.1	Associativity . . . . .	61
6.1.2	Identity presence . . . . .	64
6.1.3	Categorical definition for monoid . . . . .	65
6.1.4	Monoid importance . . . . .	66
6.2	Monoidal category . . . . .	67
6.3	Tensor product in Quantum mechanics . . . . .	68
6.4	Category of endofunctors . . . . .	69
6.5	Monads in programming languages . . . . .	71
6.5.1	Haskell . . . . .	71
6.5.2	C++ . . . . .	72
6.5.3	Scala . . . . .	72
<b>7</b>	<b>Kleisli category</b>	<b>73</b>
7.1	Partiality and Exception . . . . .	74
7.1.1	Haskell example . . . . .	75
7.1.2	C++ example . . . . .	76
7.2	Non-Determinism . . . . .	77
7.2.1	Haskell example . . . . .	77
7.3	Side effects and interactive input/output . . . . .	77
7.4	Continuation . . . . .	78
<b>8</b>	<b>Limits</b>	<b>79</b>
8.1	Definitions . . . . .	79
8.1.1	Limit . . . . .	80
8.1.2	Colimit . . . . .	83
8.2	Cone as natural transformation . . . . .	84

<i>CONTENTS</i>	5
8.3 Categorical constructions as limits . . . . .	85
8.3.1 Initial and terminal objects . . . . .	85
8.3.2 Product and sum . . . . .	86
8.3.3 Equalizer . . . . .	88
<b>9 Yoneda's lemma</b>	<b>91</b>
9.1 Hom functors . . . . .	91
9.2 Yoneda's lemma . . . . .	95
9.3 Examples . . . . .	96
9.3.1 Quantum mechanics . . . . .	96
<b>Index</b>	<b>97</b>



# Notations

$[\mathbf{C}, \mathbf{D}]$  **Fun** category

$\alpha \circ \beta$  Vertical composition of natural transformations (circle dot)

$\alpha \star \beta$  Horizontal composition of natural transformations (star dot)

$\alpha H$  Left whiskering

$\alpha, \beta$  Natural transformation (Greek small letters)

$\alpha : F \rightrightarrows G$  Natural transformation (arrow with dot)

$\text{cocone}(c, f^{(c)})$  Cocone

$\text{cone}(c, f^{(c)})$  Cone

$\text{Hom}_{\mathbf{C}}(-, a)$  Contravariant Hom functor

$\text{Hom}_{\mathbf{C}}(a, -)$  Covariant Hom functor

$\text{hom}_{\mathbf{C}}(a, b)$  Set of morphisms between  $a$  and  $b$  in category  $\mathbf{C}$

$\text{hom}(a, b)$  Set of morphisms between  $a$  and  $b$

$\mathbf{C}++$  **C++** category

$\mathbf{C}_{\mathbf{M}}$  Kleisli category

$\mathbf{Cat}$  **Cat** category

$\mathbf{C}$  Category (bold capital Latin letter)

$\mathbf{C}^{op}$  Opposite category

$\mathbf{FdHilb}$  **FdHilb** category

$\mathbf{Hask}$  **Hask** category

<b>Proof</b>	<b>Proof</b> category
<b>Rel</b>	<b>Rel</b> category
<b>Scala</b>	<b>Scala</b> category
<b>Set</b>	<b>Set</b> category
$\text{cod } f$	Codomain
$\Delta \downarrow F$	Category of cones to $F$
$\Delta_c$	Constant functor
$\text{dom } f$	Domain
$1_{\mathbf{C} \Rightarrow \mathbf{C}}$	Identity functor
$1_{a \rightarrow a}$	Identity morphism
$1_{F \rightarrow F}$	Identity natural transformation
$\text{Im } f$	Image of the function $f$
$\langle M, \mu, \eta \rangle$	Monad
$\mathcal{H}_n$	finite dimensional Hilbert space
$a \cong_f b$	there is an Isomorphism $f$ between $a$ and $b$
$a \oplus b$	Sum
$a \times b$	Product
$a, b$	Objects (Latin small letters)
$a^b$	Exponential
$eq(f, g)$	Equalizer
$F \circ G$	Functor composition (circle dot)
$f \circ g$	Morphism composition (circle dot)
$F \downarrow \Delta$	Category of co-cones from $F$
$F, G$	Functor (capital Latin letter)
$f, g, h$	Morphism (Latin small letter)



$F : \mathbf{C} \Rightarrow \mathbf{D}$  [Functor](#) (double arrow)

$f : a \hookrightarrow b$  [Monomorphism](#) (hook arrow)

$f : a \rightarrow b$  [Morphism](#) (simple arrow)

$H\alpha$  [Right whiskering](#)

$P \implies Q$  [Implication](#)

TBD To Be Defined (later)



# Introduction

You just looked at yet another introduction to Category Theory. The subject mostly consists of a lot of definitions that are related each others and we wrote the book to collect all of them in one place to be easy checked and updated in future when we decide to refresh our knowledge about the field of math. Therefore the book was written mostly for our category theory studying purposes but we will appreciate if somebody else find it useful.

The topics(chapters) cover the base definitions ([Object](#), [Morphism](#) and [Category](#)), [Functor](#), [Natural transformation](#), [Monad](#) and also include important results from the category theory such as Yoneda's lemma (see chapter [9](#)) and Curry-Howard-Lambek correspondence (see chapter [3](#)).

There are a lot of examples in each chapter. The examples cover different category theory application areas. We assume that the reader is familiar with the corresponding area and the example(s) can be passed if not. I.e. anyone can choose the suitable example(s) for (s)he.

The most important examples are related to the set theory. The set theory and category theory are very close related. Each one can be considered as an alternative view to another one.

There are a lot of examples from programming languages which include Haskell, Scala, C++. The source files for programming languages examples (Haskell, C++, Scala) can be found on github repositories:

- Haskell: [\[8\]](#)
- Scala: [\[9\]](#)
- C++: [\[7\]](#)

The examples from physics are related to quantum mechanics that is the most known for me. For the examples We were inspired by the Bob Coecke article [\[1\]](#).

The text is distributed under **Creative Common Public License** (see the text of the license at the end of the book) i.e. any reader has right to copy, store, modify, distribute or build upon the book until the original

authors are pointed in the derivative products. The initial text of the book can be found at [\[15\]](#).

# Chapter 1

## Base definitions

### 1.1 Definitions

#### 1.1.1 Object

**Definition 1.1** (Class). A class is a collection of sets (or sometimes other mathematical objects) that can be unambiguously defined by a property that all its members share.

**Definition 1.2** (Object). In category theory object is considered as something that does not have internal structure (aka point) but has a property that makes different objects belong to the same [Class](#)

**Remark 1.3** (Class of Objects). The [Class](#) of [Objects](#) will be marked as  $\text{ob}(\mathbf{C})$  (see fig. 1.1).



Figure 1.1: Class of objects  $\text{ob}(\mathbf{C}) = \{a, b, c, d\}$

### 1.1.2 Morphism

Morphism is a kind of relation between 2 **Objects**.

**Definition 1.4** (Morphism). A relation between two **Objects**  $a$  and  $b$

$$f_{ab} : a \rightarrow b$$

is called *morphism*. Morphism assumes a direction i.e. one **Object** ( $a$ ) is called *source* and another one ( $b$ ) *target*.

The **Set** of all morphisms between objects  $a$  and  $b$  is denoted as  $\text{hom}(a, b)$ .

The important remark about morphisms is below

**Remark 1.5** (Morphism). The morphism has to be considered as a relation between objects. We will avoid standard (from set theory) notation for morphisms:  $f(a) = b$ . The reason for this is the following. Let  $f_1 : a \rightarrow b$  and  $f_2 : a \rightarrow b$  are 2 different morphisms. The notation  $f_1(a) = b, f_2(a) = b$  leads to incorrect conclusion that  $f_1 = f_2$ .

For instance if  $a = b = \mathbb{R}$  then 2 functions  $f_1(x) = x, f_2(x) = -x$  set 2 different ordering on  $\mathbb{R}$  and as result have not to be considered as the same functions.

**Definition 1.6** (Domain). Given a **Morphism**  $f : a \rightarrow b$ , the **Object**  $a$  is called domain and denoted as  $\text{dom } f$ .

**Definition 1.7** (Codomain). Given a **Morphism**  $f : a \rightarrow b$ , the **Object**  $b$  is called codomain and denoted as  $\text{cod } f$ .

**Morphisms** have several properties. <sup>1</sup>

**Axiom 1.8** (Composition). If we have 3 **Objects**  $a, b$  and  $c$  and 2 **Morphisms**

$$f_{ab} : a \rightarrow b$$

and

$$f_{bc} : b \rightarrow c$$

then there exists **Morphism**

$$f_{ac} : a \rightarrow c$$

such that

$$f_{ac} = f_{bc} \circ f_{ab}$$

---

<sup>1</sup>The properties don't have any proof and postulated as axioms

**Remark 1.9** (Composition). The equation

$$f_{ac} = f_{bc} \circ f_{ab}$$

means that we apply  $f_{ab}$  first and then we apply  $f_{bc}$  to the result of the application i.e. if our objects are sets and  $x \in a$  then

$$f_{ac}(x) = f_{bc}(f_{ab}(x)),$$

where  $f_{ab}(x) \in b$ .

**Axiom 1.10** (Associativity). The *Morphisms Composition* (*Axiom 1.8*) should follow associativity property:

$$f_{ce} \circ (f_{bc} \circ f_{ab}) = (f_{ce} \circ f_{bc}) \circ f_{ab} = f_{ce} \circ f_{bc} \circ f_{ab}.$$

**Definition 1.11** (Identity morphism). For every *Object*  $a$  we define a special *Morphism*  $\mathbf{1}_{a \rightarrow a} : a \rightarrow a$  with the following properties:  $\forall f_{ab} : a \rightarrow b$

$$\mathbf{1}_{a \rightarrow a} \circ f_{ab} = f_{ab} \tag{1.1}$$

and  $\forall f_{ba} : b \rightarrow a$

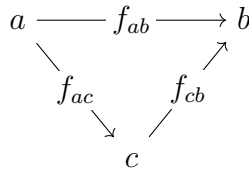
$$f_{ba} \circ \mathbf{1}_{a \rightarrow a} = f_{ba}. \tag{1.2}$$

This morphism is called as *identity morphism*.

Note that *Identity morphism* is unique, see *Identity is unique* (*Theorem 2.3*) below.

**Definition 1.12** (Commutative diagram). A commutative diagram is a diagram of *Objects* (also known as vertices) and *Morphisms* (also known as arrows or edges) such that all directed paths in the diagram with the same start and endpoint lead to the same result by composition

The following diagram commutes if  $f_{ab} = f_{cb} \circ f_{ac}$ .



**Remark 1.13** (Class of Morphisms). The *Class of Morphisms* will be marked as  $\text{hom}(\mathbf{C})$  (see fig. 1.2)



Figure 1.2: Class of morphisms  $\text{hom}(\mathbf{C}) = \{f, g, h\}$ , where  $h = f \circ g$

**Definition 1.14** (Monomorphism). If  $\forall g_1, g_2$  the equation

$$f \circ g_1 = f \circ g_2$$

leads to

$$g_1 = g_2$$

then  $f$  is called *monomorphism*. The monomorphism between  $a$  and  $b$  is denoted as  $f : a \hookrightarrow b$

**Definition 1.15** (Epimorphism). If  $\forall g_1, g_2$  the equation

$$g_1 \circ f = g_2 \circ f$$

leads to

$$g_1 = g_2$$

then  $f$  is called *epimorphism*.

**Definition 1.16** (Isomorphism). A **Morphism**  $f : a \rightarrow b$  is called *isomorphism* if  $\exists g : b \rightarrow a$  such that  $f \circ g = \mathbf{1}_{a \rightarrow a}$  and  $g \circ f = \mathbf{1}_{b \rightarrow b}$ . If there is an isomorphism  $f$  between objects  $a$  and  $b$  then it is denoted as  $a \cong_f b$ .

**Remark 1.17** (Isomorphism). There are can be many different **Isomorphisms** between 2 **Objects**.

If there is an unique isomorphism between 2 objects then the objects can be treated as the same object.

### 1.1.3 Category

**Definition 1.18** (Category). A category **C** consists of





Figure 1.3: Category  $\mathbf{C}$ . It consists of 4 objects  $\text{ob}(\mathbf{C}) = \{a, b, c, d\}$  and 7 morphisms  $\text{ob}(\mathbf{C}) = \{f, g, h = f \circ g, 1_{a \rightarrow a}, 1_{b \rightarrow b}, 1_{c \rightarrow c}, 1_{d \rightarrow d}\}$

- **Class** of **Objects**  $\text{ob}(\mathbf{C})$
- **Class** of **Morphisms**  $\text{hom}(\mathbf{C})$  defined for  $\text{ob}(\mathbf{C})$ , i.e. each morphism  $f_{ab}$  from  $\text{hom}(\mathbf{C})$  has both source  $a$  and target  $b$  from  $\text{ob}(\mathbf{C})$

For any **Object**  $a$  there should be unique **Identity morphism**  $1_{a \rightarrow a}$ . Any morphism should satisfy **Composition** (**Axiom 1.8**) and **Associativity** (**Axiom 1.10**). See fig. 1.3

**Definition 1.19** (Set of morphisms). The set of morphisms between objects  $a$  and  $b$  in the  $\mathbf{C}$  will be denoted as  $\text{hom}_{\mathbf{C}}(a, b)$ . The set will be denoted as  $\text{hom}(a, b)$  if the category does not matter

The **Category** can be considered as a way to represent a structured data. **Morphisms** are the ones which form the structure.

**Definition 1.20** (Opposite category). If  $\mathbf{C}$  is a **Category** then opposite (or dual) category  $\mathbf{C}^{op}$  is constructed in the following way: **Objects** are the same, but the **Morphisms** are inverted i.e. if  $f \in \text{hom}(\mathbf{C})$  and  $\text{dom } f = a, \text{cod } f = b$ , then the corresponding morphism  $f^{op} \in \text{hom}(\mathbf{C}^{op})$  has  $\text{dom } f^{op} = b, \text{cod } f^{op} = a$  (see fig. 1.4)

**Remark 1.21.** Composition on  $\mathbf{C}^{op}$  As you can see from fig. 1.4 the **Composition** (**Axiom 1.8**) is reverted for **Opposite category**. If  $f, g, h = f \circ g \in \text{hom}(\mathbf{C})$  then  $f \circ g$  translated into  $g^{op} \circ f^{op}$  in opposite category.

**Definition 1.22** (Small category). A category  $\mathbf{C}$  is called *small* if both  $\text{ob}(\mathbf{C})$  and  $\text{hom}(\mathbf{C})$  are **Sets**



Figure 1.4: Opposite category  $C^{op}$  to the category from fig. 1.3 . It consists of 4 objects  $\text{ob}(C^{op}) = \text{ob}(C) = \{a, b, c, d\}$  and 7 morphisms  $\text{hom}(C^{op}) = \{f^{op}, g^{op}, h^{op} = g^{op} \circ f^{op}, 1_{a \rightarrow a}, 1_{b \rightarrow b}, 1_{c \rightarrow c}, 1_{d \rightarrow d}\}$

**Definition 1.23** (Locally small category). A category  $C$  is called *locally small* if  $\text{hom}(C)$  is a [Set](#). The set is called *homset*.

**Definition 1.24** (Large category). A category  $C$  is not [Small category](#) then it is called *large*. The example of large category is [Set category](#)

**Definition 1.25** (Empty category). The category that does not contain any [Objects](#) and as result does not contain any [Morphisms](#) is called *Empty category* [21].

**Definition 1.26** (Trivial category). The category that contains only one [Object](#) and only one [Morphism](#) ([Identity morphism](#)) is called *Trivial category*.

## 1.2 Set category example

There are several examples of categories that will also be used later

**Definition 1.27** (Set). Set is a collection of distinct object. The objects are called the elements of the set.

**Definition 1.28** (Binary relation). If  $A$  and  $B$  are 2 [Sets](#) then a subset of  $A \times B$  is called binary relation  $R$  between the 2 sets, i.e.  $R \subset A \times B$ .

**Definition 1.29** (Function). Function  $f$  is a special type of [Binary relation](#). I.e. if  $A$  and  $B$  are 2 [Sets](#) then a subset of  $A \times B$  is called function  $f$  between the 2 sets if  $\forall a \in A \exists! b \in B$  such that  $(a, b) \in f$ . In other words function definition does not allow “multi value”.

**Definition 1.30** (Cartesian product). If  $A$  and  $B$  are two sets then we can define a new set  $A \times B = \{(a, b) | a \in A, b \in B\}$  that is called as the *cartesian product*.

**Definition 1.31** (Set category). In the set category we consider a **Set** of **Sets** where **Objects** are the **Sets** and **Morphisms** are **Functions** between the sets.

The **Identity morphism** is trivial function such that  $\forall x \in X : \mathbf{1}_{X \rightarrow X}(x) = x$ .

In general case when we say **Set** category we assume the set of all sets. But the result is inconsistent because famous Russell's paradox [25] can be applied. To avoid such situations we consider a limitation that is applied on our construction, for instance ZFC [26]. If we apply the limitation we have that set of all sets is not a set itself and as result the **Set** category is a **Large category**

**Remark 1.32** (Set vs Category). There is an interesting relation between sets and categories. In both we consider objects(sets) and relations between them(morphisms/functions).

In the set theory we can get info about functions by looking inside the objects(sets) aka use “microscope” [12]

Contrary in the category theory we initially don't have any info about object internal structure but can get it using the relation between the objects i.e. using **Morphisms**. In other words we can use “telescope” [12] there.

**Definition 1.33** (Categorical approach). The description of a system via its communications we will call as *categorical approach*.

This description is contrary to an ordinary system description via its internal structure.

**Definition 1.34** (Singleton). The *singleton* is a **Set** with only one element.

**Example 1.35** (Domain). Given a function  $f : X \rightarrow Y$ , the set  $X$  is the domain. I.e.  $\text{dom } f = X$

**Example 1.36** (Codomain). Given a function  $f : X \rightarrow Y$ , the set  $Y$  is the codomain. I.e.  $\text{cod } f = Y$

**Definition 1.37** (Image). The *image* of a function  $f : X \rightarrow Y$  is a subset of **Codomain**  $Y$  such that for every element in the subset there is an element in **Domain**  $X$  that maps into the subset:

$$\text{Im } f = \{y \in Y | y = f(x) \text{ for some } x \in X\}$$

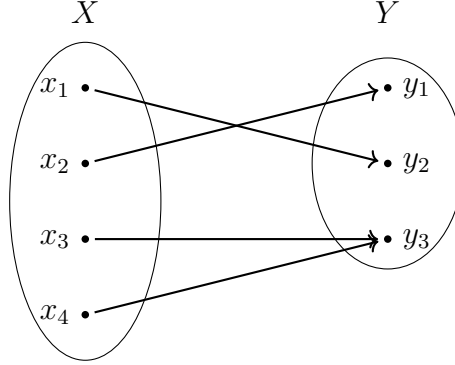


Figure 1.5: A surjective (non-injective) function from domain  $X$  to codomain  $Y$

**Definition 1.38** (Surjection). The function  $f : X \rightarrow Y$  is surjective (or onto) if  $\forall y \in Y, \exists x \in X$  such that  $f(x) = y$  (see figs. 1.5 and 1.9).

**Remark 1.39** (Surjection vs Epimorphism). [Surjection](#) and [Epimorphism](#) are related each other. Consider a non-surjective function  $f : X \rightarrow Y' \subset Y$  (see fig. 1.6). One can conclude that there is not an [Epimorphism](#) because  $\exists g_1 : Y' \rightarrow Y'$  and  $g_2 : Y \rightarrow Y$  such that  $g_1 \neq g_2$  because they operates on different [Domains](#) but from other hand  $g_1(Y') = g_2(Y')$ . For instance we can choose  $g_1 = \mathbf{1}_{Y' \rightarrow Y'}, g_2 = \mathbf{1}_{Y \rightarrow Y}$ . As soon as  $Y'$  is [Codomain](#) of  $f$  we always have  $g_1(f(X)) = g_2(f(X))$ .

As result we can say that an [Surjection](#) is a [Epimorphism](#) in the **Set** category. Moreover there is a proof [19] of that fact.

**Definition 1.40** (Injection). The function  $f : X \rightarrow Y$  is injective (or one-to-one function) if  $\forall x_1, x_2 \in X$ , such that  $x_1 \neq x_2$  then  $f(x_1) \neq f(x_2)$  (see figs. 1.7 and 1.9).

**Remark 1.41** (Injection vs Monomorphism). [Injection](#) and [Monomorphism](#) are related each other. Consider a non-injective function  $f : X \rightarrow Y$  (see fig. 1.8). One can conclude that it is not monomorphism because  $\exists g_1, g_2$  such that  $g_1 \neq g_2$  and  $f(g_1(a_1)) = y_3 = f(g_2(b_1))$ .

As result we can say that an [Injection](#) is a [Monomorphism](#) in **Set** category. Moreover there is a proof [18] of that fact.

**Definition 1.42** (Bijection). The function  $f : X \rightarrow Y$  is bijective (or one-to-one correspondence) if it is an [Injection](#) and a [Surjection](#) (see fig. 1.9).

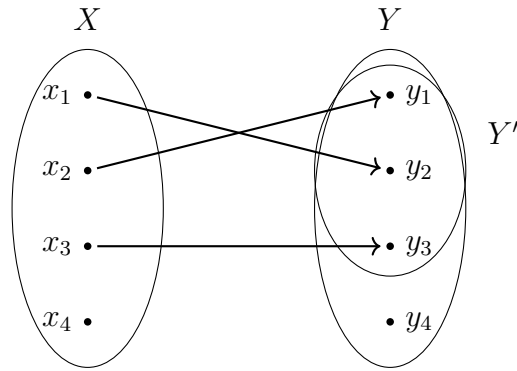


Figure 1.6: A non-surjective function  $f$  from domain  $X$  to codomain  $Y' \subset Y$ .  $\exists g_1 : Y' \rightarrow Y', g_2 : Y \rightarrow Y$  such that  $g_1(Y') = g_2(Y')$ , but as soon as  $Y' \neq Y$  we have  $g_1 \neq g_2$ . Using the fact that  $Y'$  is codomain of  $f$  we got  $g_1 \circ f = g_2 \circ f$ . I.e. the function  $f$  is not epimorphism.

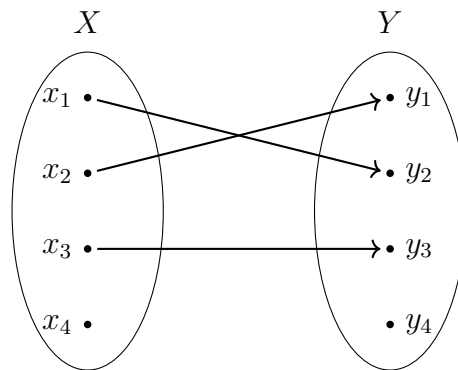


Figure 1.7: A injective (non-surjective) function from domain  $X$  to codomain  $Y$

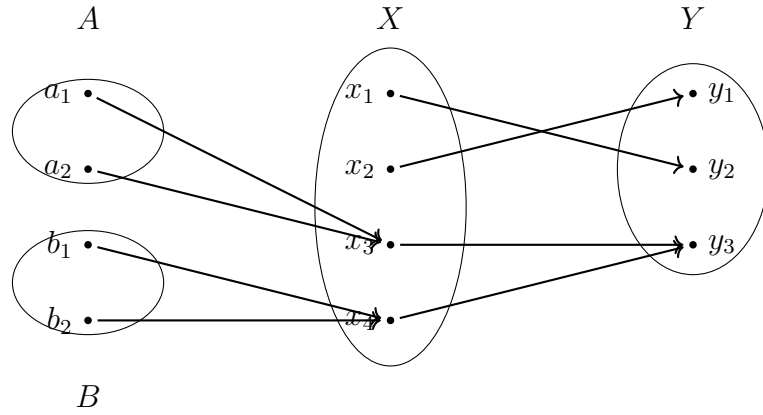


Figure 1.8: A non-injective function  $f$  from domain  $X$  to codomain  $Y$ .  $\exists g_1 : A \rightarrow X, g_2 : B \rightarrow X$  such that  $g_1 \neq g_2$  but  $f \circ g_1 = f \circ g_2$ . I.e. the function  $f$  is not monomorphism.

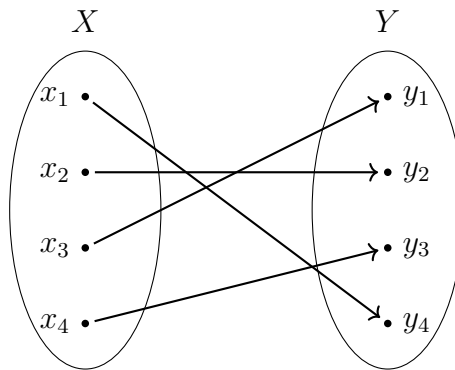


Figure 1.9: An injective and surjective function (bijection)

There is a question what is the categorical analog of a single [Set](#). Main characteristic of a category is a structure but the set by definition does not have a structure. Which category does not have any structure? The answer is [Discrete category](#).

**Definition 1.43** (Discrete category). Discrete category is a [Category](#) where [Morphisms](#) are only [Identity morphisms](#).

## 1.3 Programming languages examples

In the programming languages we consider types as [Objects](#) and functions as [Morphisms](#). The critical requirements for such consideration is that the functions have to be pure functions (without side effects). This requirement mainly is satisfied by functional languages such as Haskell and Scala. From other side the functional languages use lazy evaluation to improve their performance. The laziness can also make category theory axiom invalid (see [Haskell lazy evaluation](#) ([Remark 1.49](#)) ).

Strictly speaking neither Haskell (pure functional language) nor C++ can be considered as a category in general. For the first approximation a functional language (Haskell, Scala) can be considered as a category if we avoid to use functions with side effects (mainly for Scala) and use strict (for both Haskell and Scala) evaluations. Take the fact into consideration and define categories for 3 languages

**Definition 1.44** (**Hask** category). The objects in the **Hask** category are Haskell types and morphisms are functions

**Definition 1.45** (Pure function). The function is pure if it's execution give the same results independently from the environment.

**Definition 1.46** (**Scala** category). The objects in the **Scala** category are Scala types and morphisms are functions. We don't define functions that have a state in the category. I.e. the functions are [Pure functions](#).

**Definition 1.47** (**C++** category). The objects in the **C++** category are Scala types and morphisms are functions. We don't define functions that have a state in the category. I.e. the functions are [Pure functions](#).

In any case we can construct a simple toy category that can be easy implemented in any language. Particularly we will look into category with 3 objects that are types: Int, Bool, String. There are also several functions between them (see [fig. 1.10](#)).



Figure 1.10: Programming language category example. Objects are types: Int, Bool, String. Morphisms are several functions

### 1.3.1 Hask toy category

**Example 1.48** (**Hask** toy category). Types in Haskell are considered as **Objects**. Functions are considered as **Morphisms**. We are going to implement **Category** from fig. 1.10.

The function **isEven** converts **Int** type into Bool.

```
isEven :: Int -> Bool
isEven x = x `mod` 2 == 0
```

There is also **Identity morphism** that is defined as follows

```
id :: a -> a
id x = x
```

If we have an additional function

```
stringLength :: String -> Int
stringLength x = length x
```

then we can create a **Composition** (**Axiom 1.8**)

```
isStringLengthEven :: String -> Bool
isStringLengthEven = isEven . stringLength
```



**Remark 1.49** (Haskell lazy evaluation). Each Haskell type has a special value  $\perp$ . The fact that the value and lazy evaluations are part of the language, make several category law invalid, for instance [Identity morphism](#) behaviour become invalid in specific cases:

The following code

```
seq undefined True
```

produces *undefined* But the following

```
seq (id.undefined) True
seq (undefined.id) True
```

produces *True* in both cases. As result we have (we cannot compare compare functions in Haskell, but if we could we can get the following)

```
id . undefined /= undefined
undefined . id /= undefined,
```

i.e. [\(1.1\)](#) and [\(1.2\)](#) are not satisfied.

### 1.3.2 C++ toy category

**Example 1.50** (C++ toy category). We will use the same trick as in [Hask toy category](#) ([Example 1.48](#)) and will assume types in C++ as [Objects](#), functions as [Morphisms](#). We also are going to implement [Category](#) from [fig. 1.10](#).

We also define 2 functions:

```
auto isEven = [](int x) {
    return x % 2 == 0;
};

auto stringLength = [](std::string s) {
    return static_cast<int>(s.size());
};
```

Composition can be defined as follows:

```
// h = g . f
template <typename A, typename B>
auto compose(A g, B f) {
    auto h = [f, g](auto a) {
        auto b = f(a);
```

```

    auto c = g(b);
    return c;
};
return h;
};

```

The [Identity](#) morphism:

```

auto id = [](auto x) { return x; };

```

The usage examples are the following:

```

auto isStringLengthEven = compose<>(isEven, stringLength);

auto isStringLengthEvenL = compose<>(id, isStringLengthEven);

auto isStringLengthEvenR = compose<>(isStringLengthEven, id);

```

Such construction will always provides us the category as soon as we use pure function (functions without effects).

### 1.3.3 Scala toy category

**Example 1.51** (Scala toy category). We will use the same trick as in [Hask toy category](#) ([Example 1.48](#)) and will assume types in Scala as [Objects](#), functions as [Morphisms](#). We also are going to implement [Category](#) from [fig. 1.10](#).

```

object Category {
  def id[A]: A => A = a => a
  def compose[A, B, C](g: B => C, f: A => B):
    A => C = g compose f

  val isEven = (i: Int) => i % 2 == 0
  val stringLength = (s: String) => s.length
  val isStringLengthEven = (s: String) =>
    compose(isEven, stringLength)(s)
}

```

The usage example is below

```

class CategorySpec extends Properties("Category") {
  import Category._
}

```

```

import Prop.forAll

property("composition") = forAll { (s: String) =>
  isStringLengthEven(s) == isEven(stringLength(s))
}

property("right id") = forAll { (i: Int) =>
  isEven(i) == compose(isEven, id[Int])(i)
}

property("left id") = forAll { (i: Int) =>
  isEven(i) == compose(id[Boolean], isEven)(i)
}
}

```

## 1.4 Quantum mechanics examples

The most critical property of quantum system is the superposition principle. The **Set category** cannot be used for it because it does not satisfy the principle. but a simple modification of the **Set** category does.

**Definition 1.52** (**Rel** category). We will consider a set of sets (same as **Set category**) i.e. **Sets** as **Objects**. Instead of **Functions** we will use **Binary relations** as **Morphisms**.

The **Rel** category is similar to the finite dimensional Hilber space especially because it assumes some kind of superposition. Really consider **Rel** - the **Rel** category.  $X, Y \in \text{ob}(\mathbf{Rel})$  - 2 sets which consists of different elements. Let  $f : X \rightarrow Y$  - **Morphism**. Each element  $x \in X$  is mapped to a subset  $Y' \subset Y$ . The  $Y'$  can be **Singleton** (in this case no differences with **Set category**) but there can be a situation when  $Y'$  consists of several elements. In the case we will get some kind of superposition that is analogiest to quantum systems.

In the quantum mechanics we say about Hilber spaces.

**Definition 1.53** (Hilbert space). The Hilbert space is a complex vector space with an inner product as a complex number ( $\mathbb{C}$ ).

Later we will consider only finite dimensional Hilber spaces. We will denote a Hilbert space of dimensional  $n$  as  $\mathcal{H}_n$ . Obviously  $\mathcal{H}_1 = \mathbb{C}$ .

**Definition 1.54** (Dual space). Each Hilber space  $\mathcal{H}$  has an associated with it dual space  $\mathcal{H}^*$  that consists of linear functionals

**Example 1.55** (Dirac notation). Consider a ket-vector  $|\psi\rangle \in \mathcal{H}$ . Then the corresponding vector from [Dual space](#) is called bra-vector  $\langle\psi| \in \mathcal{H}^*$ . From the definition of dual space the bra-vector is a linear functional i.e.

$$\langle\psi| : \mathcal{H} \rightarrow \mathbb{C},$$

$\forall |\phi\rangle \in \mathcal{H}$  we have  $\langle\psi|(|\phi\rangle) = (|\psi\rangle, |\phi\rangle)$  - inner product that is often written as  $\langle\psi|\phi\rangle$ .

The transformation between 2 [Hilbert spaces](#) that preserves the structure is called linear map or linear transformations.

**Definition 1.56** (Linear map). The linear map between 2 [Hilbert spaces](#)  $\mathcal{A}$  and  $\mathcal{B}$  is a mapping  $f : \mathcal{A} \rightarrow \mathcal{B}$  that preserves additions

$$f(a_1 + a_2) = f(a_1) + f(a_2),$$

and scalar multiplications:

$$f(c \cdot a) = c \cdot f(a)$$

where  $a, a_{1,2} \in \mathcal{A}$  and  $f(a), f(a_{1,2}) \in \mathcal{B}$ .

**Remark 1.57** (Linear map). Note that [Linear map](#) does not preserve inner product. TBD (verify the statement ???)

If we want to combine 2 Hilbert spaces into one we use a notion of direct sum.

**Definition 1.58** (Direct sum of Hilber spaces). Let  $\mathcal{A}, \mathcal{B}$  are 2 Hilber spaces. The direct sum  $\mathcal{A} \oplus \mathcal{B}$  is defined as follows

$$\mathcal{A} \oplus \mathcal{B} = \{a \oplus b | a \in \mathcal{A}, b \in \mathcal{B}\}.$$

The inner product is defined as follows

$$\langle a_1 \oplus b_1 | a_2 \oplus b_2 \rangle = \langle a_1 | a_2 \rangle + \langle b_1 | b_2 \rangle.$$

**Definition 1.59** (**FdHilb** category). Most common case in quantum mechanics is the case of quantum states in the finite dimensional Hilbert space. We can consider the set of all finite dimensional Hilbert spaces as a category. The [Objects](#) in the category are finite dimensional [Hilbert spaces](#) and [Morphisms](#) are [Linear maps](#). The category is denoted as **FdHilb**. It is very similar to [Rel category](#). The brief relation is described in the table [1.1](#).

	<b>Set</b>	<b>Rel</b>	<b>FdHilb</b>
Object	Set	Set	finite dimensional Hilbert space
Morphism	Function	Binary relation	Linear map
Initial object	empty set	empty set	trivial Hilbert space of dimensional 0
Terminal object	Singleton	Singleton	$\mathbb{C}$
Product	Cartesian product	Cartesian product	Direct sum of Hilber spaces
Sum	Sum (Example 2.15)	Sum (Example 2.15)	Direct sum of Hilber spaces

Table 1.1: Relations between **Set**, **Rel** and **FdHilb** categories

**Example 1.60** (Rabi oscillations). For our example we consider a 2 level atom with states  $|a\rangle$  - excited and  $|b\rangle$  - ground. As soon as we consider a 2-level system we are in the 2 dimensional Hilbert space i.e. have only one Object. Lets call it as  $|\psi\rangle$ . The category in the example will be called as **Rabi**. I.e.  $\text{ob}(\mathbf{Rabi}) = \mathcal{H}_2\{|\psi\rangle\}$ .

The atom interacts with light beam of frequency  $\omega = \omega_{ab}$ . The state of the system is described by the following equation [27]:

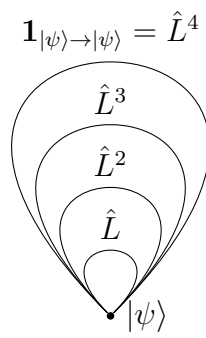
$$|\psi\rangle = \cos \frac{\omega_R t}{2} |a\rangle - i \sin \frac{\omega_R t}{2} |b\rangle ,$$

where  $\omega_R$  - Rabi frequency [27].

The interaction time  $t$  is fixed and corresponds to  $\omega_R t = \pi$  i.e. the interaction can be described a linear operator  $\hat{L}$ .

There are 4 different states and as result 4 Morphisms:

$$\begin{aligned} |\psi\rangle_0 &= |a\rangle , \\ |\psi\rangle_1 &= \hat{L} |\psi\rangle_0 = -i |b\rangle , \\ |\psi\rangle_2 &= \hat{L}^2 |\psi\rangle_0 = -|a\rangle , \\ |\psi\rangle_3 &= \hat{L}^3 |\psi\rangle_0 = i |b\rangle , \end{aligned}$$

Figure 1.11: Rabi oscillations as a category **Rabi**

# Chapter 2

## Objects and morphisms

### 2.1 Equality

The important question is how can we decide whenever an object/morphism is equal to another object/morphism? The trivial answer is possible if an **Object** is a **Set**. In the case we can say that 2 objects are equal if they contain the equivalent collection of elements. Unfortunately we cannot do the same trick for categorical **Objects** as soon as they don't have any internal structure but can use a **Categorical approach** (see **Set vs Category** (Remark 1.32) ): if we cannot use “microscope” lets use “telescope” and define the equality of objects and morphisms of a category **C** in the terms of whole  $\text{hom}(\mathbf{C})$ .

**Definition 2.1** (Objects equality). Two **Objects**  $a$  and  $b$  in **Category C** are equal if there exists an unique **Isomorphism**  $a \cong_f b$ . This also means that also exist unique isomorphism  $b \cong_g a$ . These two **Morphisms** ( $f$  and  $g$ ) are related each other via the following equations:  $f \circ g = \mathbf{1}_{a \rightarrow a}$  and  $g \circ f = \mathbf{1}_{b \rightarrow b}$ .

Unlike **Functions** between **Sets** we don't have any additional info <sup>1</sup> about **Morphisms** except category theory axioms which the morphisms satisfy [4]. This leads us to the following definition of morphisms equality:

**Definition 2.2** (Morphisms equality). Two **Morphisms**  $f$  and  $g$  in **Category C** are equal if the equality can be derived from the base axioms:

- **Composition** (Axiom 1.8)
- **Associativity** (Axiom 1.10)
- **Identity morphism**: (1.1), (1.2)

---

<sup>1</sup> for instance info about sets internals. i.e. which elements of the sets are connected by the considered functions

or [Commutative diagrams](#) which postulate the equality.

As an example lets proof the following theorem

**Theorem 2.3** (Identity is unique). *The [Identity morphism](#) is unique.*

*Proof.* Consider an [Object](#)  $a$  and it's [Identity morphism](#)  $\mathbf{1}_{a \rightarrow a}$ . Let  $\exists f : a \rightarrow a$  such that  $f$  is also identity. In the case (1.1) for  $f$  as identity gives

$$f \circ \mathbf{1}_{a \rightarrow a} = \mathbf{1}_{a \rightarrow a}.$$

From other side (1.2) for  $\mathbf{1}_{a \rightarrow a}$  satisfied

$$f \circ \mathbf{1}_{a \rightarrow a} = f$$

i.e.

$$f = f \circ \mathbf{1}_{a \rightarrow a} = \mathbf{1}_{a \rightarrow a}$$

or  $f = \mathbf{1}_{a \rightarrow a}$ . □

## 2.2 Initial and terminal objects

**Definition 2.4** (Initial object). Let  $\mathbf{C}$  is a [Category](#), the [Object](#)  $i \in \text{ob}(\mathbf{C})$  is called *initial object* if  $\forall x \in \text{ob}(\mathbf{C}) \exists ! f_x : i \rightarrow x \in \text{hom}(\mathbf{C})$ .

**Example 2.5** (Initial object). [\[Set\]](#) Note that there is only one function from empty set to any other sets [\[17\]](#) that makes the empty set as the [Initial object](#) in [Set category](#).

**Definition 2.6** (Terminal object). Let  $\mathbf{C}$  is a [Category](#), the [Object](#)  $t \in \text{ob}(\mathbf{C})$  is called *terminal object* if  $\forall x \in \text{ob}(\mathbf{C}) \exists ! g_x : x \rightarrow t \in \text{hom}(\mathbf{C})$ .

**Example 2.7** (Terminal object). [\[Set\]](#) [Terminal object](#) in [Set category](#) is a set with one element i.e [Singleton](#).

As you can see the initial and terminal objects are opposite each other. I.e. if  $i$  is an [Initial object](#) in  $\mathbf{C}$  then it will be [Terminal object](#) in the [Opposite category](#)  $\mathbf{C}^{\text{op}}$ .

**Theorem 2.8** (Initial object is unique). *Let  $\mathbf{C}$  is a category and  $i, i' \in \text{ob}(\mathbf{C})$  two [Initial objects](#) then there exists an unique [Isomorphism](#)  $u : i \rightarrow i'$  (see [Objects equality](#))*





Figure 2.1: Commutative diagram for initial object uniqueness proof



Figure 2.2: Commutative diagram for terminal object uniqueness proof

*Proof.* Consider the following [Commutative diagram](#) (see fig. 2.1). As soon as  $i$  initial object  $\exists! u : i \rightarrow i'$ . From other side  $i'$  is also initial object and therefore  $\exists! u^{-1} : i' \rightarrow i$ . Combining them together via composition we can get  $u^{-1} \circ u : i \rightarrow i$  and  $u \circ u^{-1} : i' \rightarrow i'$ . From the fact that  $i$  is initial object one can get that there exists only one morphism  $1_{i \rightarrow i} : i \rightarrow i$ . The same is the truth for  $i'$ . Therefore  $u^{-1} \circ u = 1_{i \rightarrow i}$  and  $u \circ u^{-1} = 1_{i' \rightarrow i'}$ . These complete the commutative diagram build and finishes the proof.  $\square$

**Theorem 2.9** (Terminal object is unique). *Let  $\mathbf{C}$  is a category and  $t, t' \in \text{ob}(\mathbf{C})$  two [Terminal objects](#) then there exists an unique [Isomorphism](#)  $v : t' \rightarrow t$  (see [Objects equality](#))*

*Proof.* Just got to the [Opposite category](#) and revert arrows in fig. 2.1. The result shown on fig. 2.2 and it proofs the theorem statement.  $\square$

**Example 2.10** (Toy example). In our toy example fig. 1.10 the type `String` is [Initial object](#) and type `Bool` is the [Terminal object](#).



Figure 2.3: Product  $c = c_1 \times c_2$ .  $\forall c, \exists! h \in \text{hom}(\mathbf{C}) : \pi'_1 = \pi_1 \circ h, \pi'_2 = \pi_2 \circ h$ .

## 2.3 Product and sum

The pair of 2 objects is defined via the universal property in the following way:

**Definition 2.11** (Product). Let we have a category  $\mathbf{C}$  and  $c_1, c_2 \in \text{ob}(\mathbf{C})$  -two **Objects** then the product of the objects  $c_1, c_2$  is another object in  $\mathbf{C}$   $c = c_1 \times c_2$  with 2 **Morphisms**  $\pi_1, \pi_2$  such that  $c_1 = \pi_1(c), c_2 = \pi_2(c)$  and the following universal property is satisfied:  $\forall c' \in \text{ob}(\mathbf{C})$  and morphisms  $\pi'_1 : c' \rightarrow c_1, \pi'_2 : c' \rightarrow c_2$ , exists unique morphism  $h$  such that the following diagram (see fig. 2.3) commutes, i.e.  $\pi'_1 = \pi_1 \circ h, \pi'_2 = \pi_2 \circ h$ . In other words  $h$  factorizes  $\pi'_{1,2}$ .

**Example 2.12** (Product). **[Set]** The **Product** of two sets  $A$  and  $B$  in **Set category** is defined as a **Cartesian product**:  $A \times B = \{(a, b) | a \in A, b \in B\}$ .

If we invert arrows in **Product** we will got another object definition that is called sum

**Definition 2.13** (Sum). Let we have a category  $\mathbf{C}$  and  $c_1, c_2 \in \text{ob}(\mathbf{C})$  -two **Objects** then the sum of the objects  $c_1, c_2$  is another object in  $\mathbf{C}$   $c = c_1 \oplus c_2$  with 2 **Morphisms**  $i_1, i_2$  such that  $c = i_1(c_1), c = i_2(c_2)$  and the following universal property is satisfied:  $\forall c' \in \text{ob}(\mathbf{C})$  and morphisms  $i'_1 : c_1 \rightarrow c', i'_2 : c_2 \rightarrow c'$ , exists unique morphism  $h$  such that the following diagram (see fig. 2.4) commutes, i.e.  $i'_1 = h \circ i_1, i'_2 = h \circ i_2$ . In other words  $h$  factorizes  $i'_{1,2}$ .

**Definition 2.14** (Disjoint union). Let  $\{A_i : i \in I\}$  be a family of sets indexed by  $I$ . The *disjoint union* [22] of this family is the set

$$\sqcup_{i \in I} A_i = \cup_{i \in I} \{(x, i) : x \in A_i\}.$$

The elements of the disjoint union are ordered pairs  $(x, i)$ . Here  $i$  serves as an auxiliary index that indicates which  $A_i$  the element  $x$  came from.



Figure 2.4: Sum  $c = c_1 \oplus c_2$ .  $\forall c, \exists! h \in \text{hom}(\mathbf{C}) : i'_1 = h \circ i_1, i'_2 = h \circ i_2$ .



Figure 2.5: Product of morphisms.

**Example 2.15** (Sum). **[Set]** The **Sum** of two sets  $A$  and  $B$  in **Set category** is defined as **Disjoint union**.

The **Product** of objects will provide also a definition for product of morphisms

**Definition 2.16** (Product of morphisms). Let  $\mathbf{C}$  is a category and  $a, a' \in \text{ob}(\mathbf{C})$  and  $b, b' \in \text{ob}(\mathbf{C})$  are 2 pairs of **Objects** that admit definition 2.11. Consider 2 morphisms that connects the objects:  $f : a \rightarrow b, f' : a' \rightarrow b'$  then we can create a new unique morphism that connects the products:  $f \times f' : a \times a' \rightarrow b \times b'$  and makes the diagram commute (see fig. 2.5).

## 2.4 Category as a monoid

Consider the following definition from abstract algebra

**Definition 2.17** (Monoid). The set of elements  $M$  with defined binary operation  $\circ$  we will call as a monoid if the following conditions are satisfied.

1. Closure:  $\forall a, b \in M : a \circ b \in M$
2. Associativity:  $\forall a, b, c \in M : a \circ (b \circ c) = (a \circ b) \circ c$

3. Identity element:  $\exists e \in M$  such that  $\forall a \in M: e \circ a = a \circ e = a$

We can consider 2 **Monoids**. The first one has **Product** as the binary operation and **Terminal object** as the identity element. As result we just got an analog of multiplication in the category theory. This is why the terminal object is often denoted as **1** and the operation is called as the product.

Another one is additional **Monoid** that has **Initial object** as the identity element and the **Sum** as the binary operation. The initial object in that case is often denoted as **0**. I.e. we can see a direct connection with addition in algebra.

If we do such consideration then we can make a step forward and look at the distributive law that sum and multiplication satisfy.

**Definition 2.18** (Distributive category). A category **C** is *distributive* if [24] it has finite **Products** and **Sums** such that  $\forall a, b, c \in \text{ob}(\mathbf{C})$ :

$$a \times b \oplus a \times c \cong a \times (b \oplus c)$$

and

$$a \times 0 \cong 0$$

where **0** is the **Initial object**.

**Example 2.19** (Distributive category). **Set category** is an example [24] of **Distributive category**

From other hand not all categories which have both product and sum are distributive. One of such example is a category of all groups [24] **Grp** where groups are considered as objects and group homomorphisms as morphisms.

## 2.5 Exponential

We are going to talk about functions (aka morphisms) as **Objects**.

**Example 2.20** (Hom set). Consider 2 sets  $A$  and  $B$  the set of functions between the 2 sets form a new set that is called as *Hom-set* and denoted as  $A \rightarrow B$ . Thus if  $A, B \in \text{ob}(\mathbf{Set})$  then the Hom-set will also  $A \rightarrow B \in \text{ob}(\mathbf{Set})$ .

The construction of **Hom set** (Example 2.20) is applied to **Set category** but not to an arbitrary category because the Hom-set is a set and therefore the object in the **Set** category. I.e. if **C** is a category and  $a, b \in \text{ob}(\mathbf{C})$  then the Hom-set  $a \rightarrow b \in \text{ob}(\mathbf{Set})$  but we now want to construct something like to the Home-set but that is an object in **C**. This will be called as the function object. We will use the universal construction for the object definition.



Figure 2.6: Exponential object

**Definition 2.21** (Exponential). Let  $\mathbf{C}$  is a category and  $z, y \in \text{ob}(\mathbf{C})$ . We also assume that  $\mathbf{C}$  allows all **Products** with  $y$ , i.e.  $\forall z' \in \text{ob}(\mathbf{C}), \exists z' \times y$ . An object  $z^y$  together with a **Morphism**  $e : z^y \times y \rightarrow z$  is an *exponential object* if  $\forall e' \in \text{hom}(\mathbf{C})$  and  $\forall z' \in \text{ob}(\mathbf{C})$  exists an unique morphism  $h : z' \rightarrow z^y$  such that the **Commutative diagram** shown in fig. 2.6 commutes:

$$e' = e \circ (h \times \mathbf{1}_{y \rightarrow y})$$

**Example 2.22** (Exponential). [**Set**] Lets look at the **Exponential** in **Set**. We want to show that the object corresponds to the function. Really if we want to define a function  $f : X \rightarrow Y$  then we should look at the **Hom set** (**Example 2.20**)  $F = X \rightarrow Y$ .  $f \in F$  - is an element of the Hom-set. For the function application we have to take the argument  $x \in X$  and the function we want to apply  $f \in F$ . Then we construct the pair  $(f, x) \in F \times X$ . For the function application we have to call a **Morphism**  $e : F \times X \rightarrow Z$ .<sup>2</sup> I.e. the application  $e(f, x)$  gives us  $e(f, x) = y \in Y$  - the function value.

**Definition 2.23** (Cartesian closed category). If a category  $\mathbf{C}$  satisfies the following conditions then it is called *Cartesian closed category*

1. It has **Terminal object**
2.  $\forall a, b \in \text{ob}(\mathbf{C})$  exists **Product**  $a \times b \in \text{ob}(\mathbf{C})$ .
3.  $\forall a, b \in \text{ob}(\mathbf{C})$  exists **Exponential**  $a^b \in \text{ob}(\mathbf{C})$

**Theorem 2.24** (Cartesian closed category). *If  $\mathbf{C}$  is a **Cartesian closed category** with finite **Sum** then it is a **Distributive category**.*

*Proof.* TBD

□

---

<sup>2</sup> $e$  from the word “eval”

## 2.6 Programming language examples. Type algebra

### 2.6.1 Hask category

**Example 2.25** (Initial object). [Hask] If we avoid lazy evaluations in Haskell (see [Haskell lazy evaluation](#) (Remark 1.49) ) then we can find several types as candidates for initial and terminal object in Haskell. [Initial object](#) in [Hask category](#) is a type without values

```
data Void
```

i.e. you cannot construct an object of the type.

There is only one function from the initial object:

```
absurd :: Void -> a
```

The function is called `absurd` because it does an absurd action. Nobody can prove that it does not exist. For the existence proof the following absurd argument can be used: “Just provide me an object type **Void** and I will provide you the result of evaluation”.

There is no function in the opposite direction because it would have been used for the **Void** object creation.

**Example 2.26** (Terminal object). [Hask] Terminal object (unit) in [Hask category](#) keeps only one element

```
data () = ()
```

i.e. you can create only one element of the type. You can use the following function for the creation:

```
unit :: a -> ()
unit _ = ()
```

**Example 2.27** (Product). [Hask] The [Product](#) in [Hask category](#) keeps a pair and the constructor is defined as follows

```
(,) :: a -> b -> (a, b)
(,) x y = (x, y)
```

There are 2 projectors:

```
fst :: (a, b) -> a
fst (x, _) = x
snd :: (a, b) -> b
snd (_, y) = y
```

**Example 2.28** (Sum). [Hask] The **Sum** in **Hask** category defined as follows

```
data Either a b = Left a | Right b
```

The typical usage is via pattern matching for instance

```
factor :: (a -> c) -> (b -> c) -> Either a b -> c
factor f _ (Left x) = f x
factor _ g (Right y) = g y
```

**Example 2.29** (Distributive category). [Hask] As soon as **Hask** is a **Cartesian closed category** then by theorem 2.24 it is a **Distributive category** i.e. one can conclude that

```
(a, Either b c)
```

is the same to

```
Either (a, b) (a, c)
```

**Example 2.30** (Exponential). [Hask] It's not surprisingly that the **Exponential** in **Hask** is a function object i.e.  $b^a$  can be written as **a -> b**.

**Example 2.31** (Type algebra). example 2.30 gives interesting results with types manipulations. For instance the type  $a^{b+c}$  can be written as

```
Either b c -> a
```

for the function we should have both functions **b -> a** and **b -> c**. I.e. the code is equivalent to the following one

```
(b -> a, c -> a)
```

These transformations correspond to the following simple algebraic equation

$$a^{b+c} = a^b a^c.$$

This is also called as *type algebra*.

### 2.6.2 C++ category

**Example 2.32** (Initial object). [C++] In C++ exists a special type that does not hold any values and as result cannot be created: **void**. You cannot create an object of that type i.e. you will get a compiler error if you try.

**Example 2.33** (Terminal object). [C++] C++ 17 introduced a special type that keeps only one value - `std::monostate`:

```
namespace std {
    struct monostate {};
}
```

**Example 2.34** (Product). [C++] The `Product` in `C++ category` keeps a pair and the constructor defined as follows

```
namespace std {
    template< class A, class B > struct pair {
        A first;
        B second;
    };
}
```

There is a simple usage example

```
std::pair<int, bool> p(0, false);

std::cout << "First projector: " << p.first << std::endl;
std::cout << "Second projector: " << p.second << std::endl;
```

Really any `struct` or `class` can be considered as a product.

**Example 2.35** (Sum). [C++] If we consider `Objects` as types then `Sum` is an object that can be either one or another type. The corresponding C/C++ construction that provides an ability to keep one of two types is `union`.

C++17 suggests `std::variant` as a safe replacement for `union`. The example of the `factor` function is below

```
template <typename A, typename B, typename C, typename D>
auto factor(A f, B g, const std::variant<C, D>& either) {
    try {
        return f(std::get<C>(either));
    }
    catch(...) {
        return g(std::get<D>(either));
    }
};
```

The simple usage as follows:



```
std::variant<std::string, int> var = std::string("abc");
std::cout << "String length:" <<
factor<>(stringLength, id, var) << std::endl;
var = 4;
std::cout << "id(int):" <<
factor<>(stringLength, id, var) << std::endl;
```

TBD

### 2.6.3 Scala category

**Example 2.36** (Initial object). [Scala] We used a same trick as for [Initial object](#) (Example 2.25) in [Hask category](#) and define [Initial object](#) in [Scala category](#) as a type without values

```
sealed trait Void
```

i.e. you cannot construct a object of the type.

**Example 2.37** (Terminal object). [Scala] We used a same trick as for [Terminal object](#) (Example 2.26) in [Hask category](#) and define [Terminal object](#) in [Scala category](#) as a type with only one value

```
abstract final class Unit extends AnyVal
```

TBD i.e. you can create only one element of the type.

TBD

## 2.7 Quantum mechanics

**Example 2.38** (Initial object). [FdHilb] We will use a Hilber space of dimensional 0 as the [Initial object](#). I.e. the set that does not have any states in it.

**Example 2.39** (Terminal object). [FdHilb] We will use a Hilber space of dimensional 1 as the [Terminal object](#). I.e. the set of complex numbers  $\mathbb{C}$ .

**Example 2.40** (Product). [FdHilb] The [Product](#) in [FdHilb category](#) is a [Direct sum of Hilber spaces](#).

**Example 2.41** (Sum). [FdHilb] The [Sum](#) in [FdHilb category](#) is a [Direct sum of Hilber spaces](#).

TBD



# Chapter 3

## Curry-Howard-Lambek correspondence

There is an interesting correspondence between computer programs and mathematical proofs. Different types of logic correspond to different computational models. This allows to build a theory of computation on the base of math logic. First of all consider a category of proofs

### 3.1 Proof category

**Definition 3.1** (Proposition). *Proposition* is a statement that either true or false.

There are 2 main propositions

**Definition 3.2** (True). A true statement is one that is correct, either in all cases or at least in the sample case [20].

and

**Definition 3.3** (False). A false statement is one that is not correct [20].

**Example 3.4** (Proposition). There is an example of correct (true) proposition

$$\forall n \in \mathbb{R} : n^2 \geq 0$$

There is an example of incorrect (false) proposition

$$\forall n \in \mathbb{C} : n^2 \geq 0,$$

for instance  $i \in \mathbb{C}$  gives  $i^2 = -1$ .

**Definition 3.5** (Implication). An *implication* is a [Proposition](#) of the form  $P \implies Q$  i.e. if  $P$  then  $Q$  [11].

The main logical deduction rule is the following

**Definition 3.6** (Modus ponens). If  $P$  is true and  $P \implies Q$  is true then  $Q$  is also true. The rule is often written as [11]

$$\frac{\begin{array}{c} P \\ P \implies Q \end{array}}{Q}$$

where if statements above the line are true then the statement below the line is also true.

**Definition 3.7** (Proof). *Proof* is a verification [11] of a [Proposition](#) by a chain of logical deduction from a base set of axioms.

Propositions can be combined into new propositions via the following logical operations

**Definition 3.8** (Conjunction). Conjunction or logical AND is the operation with following rules

$a$	$b$	$a \wedge b$
True	True	True
True	False	False
False	True	False
False	False	False

Table 3.1: Conjunction

**Definition 3.9** (Disjunction). Disjunction or logical OR is the operation with following rules

$a$	$b$	$a \vee b$
True	True	True
True	False	True
False	True	True
False	False	False

Table 3.2: Disjunction

Operations in Boolean logic follow the distributive law:

$$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c) \quad (3.1)$$

i.e. the operation  $\wedge$  corresponds to multiplication and  $\vee$  to sum. Therefore the **Proof** can be considered as a [Distributive category](#).

**Definition 3.10 (Proof category).** The **Proof** category is a category where [Propositions](#) are [Objects](#) and [Proofs](#) are [Morphisms](#). I.e. proofs are used as connectors between different propositions.

Consider different objects and constructions of the proof (logic) theory from the categorical point of view

**Example 3.11** (Initial object). [**Proof**] The *false* statement can be considered as the initial object because for any other statement exists only one proof from the false statement to that one.

**Example 3.12** (Terminal object). [**Proof**] The *true* statement can be considered as the terminal object

**Example 3.13** (Product). [**Proof**] [Conjunction](#) can be considered as [Product](#) in [Proof category](#).

**Example 3.14** (Sum). [**Proof**] [Disjunction](#) can be considered as [Sum](#) in [Proof category](#).

Thus we can declare the following correspondence (see table 3.3) between logic proofs and [Cartesian closed category](#) and therefore also between programming languages.

<a href="#">Proof category</a>	Programming language	<a href="#">Cartesian closed category</a>
<a href="#">Proposition/Implication</a>	Type	<a href="#">Object</a>
<a href="#">Proof</a>	Function type	<a href="#">Exponential</a>
<a href="#">Conjunction</a>	Product type	<a href="#">Product</a>
<a href="#">Disjunction</a>	Sum type	<a href="#">Sum</a>
<a href="#">True</a>	unit type	<a href="#">Terminal object</a>
<a href="#">False</a>	bottom type	<a href="#">Initial object</a>

Table 3.3: Relation between logic proofs and programming languages

## 3.2 Linear logic and Linear types

Linear logic [2] is one of refinements of classical logic in the logic the [Implication](#) has been modified. In the classical logic the both statements  $P$  and

$Q$  are valid after implication  $P \implies Q$ . But in linear logic we have another situation when the statement  $P$  can be used only once and become invalid after the usage. The situation then a resource can be used only once is useful in different types of computations especially in concurrency. TBD

### 3.3 Quantum logic and quantum computation

Different modifications of logic rules give us new computational models. One of example is the quantum computations. The quantum logic differs from Boolean one in the missing distributive law (3.1). TBD

# Chapter 4

## Functors

### 4.1 Definitions

**Definition 4.1** (Functor). Let  $\mathbf{C}$  and  $\mathbf{D}$  are 2 categories. A mapping  $F : \mathbf{C} \Rightarrow \mathbf{D}$  between the categories is called *functor* if it preserves the internal structure (see fig. 4.1):

- $\forall a_C \in \text{ob}(\mathbf{C}), \exists a_D \in \text{ob}(\mathbf{D})$  such that  $a_D = F(a_C)$
- $\forall f_C \in \text{hom}(\mathbf{C}), \exists f_D \in \text{hom}(\mathbf{D})$  such that  $\text{dom } f_D = F(\text{dom } f_C), \text{cod } f_D = F(\text{cod } f_C)$ . We will use the following notation later:  $f_D = F(f_C)$ .
- $\forall f_C, g_C$  the following equation holds:

$$F(f_C \circ g_C) = F(f_C) \circ F(g_C) = f_D \circ g_D.$$

- $\forall x \in \text{ob}(\mathbf{C}) : F(1_{x \rightarrow x}) = 1_{F(x) \rightarrow F(x)}$ .



Figure 4.1: Functor  $F : \mathbf{C} \Rightarrow \mathbf{D}$  definition

**Remark 4.2** (Functor). When we say that functor preserve internal structure we assume that the functor is not just mapping between [Objects](#) but also between [Morphisms](#).

Thus functor is something that allows map one category into another. The initial category can be considered as a pattern thus the mapping is some kind of searching of the pattern inside another category.

Programming languages can be considered as a good platform for the functor examples. The functor can be defined in Haskell as follows <sup>1</sup>

**Example 4.3** (Functor). [Hask]

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

In Scala it can be defined in the same way

**Example 4.4** (Functor). [Scala]

```
trait Functor[F[_]] {
  def fmap[A, B](f: A => B): F[A] => F[B]
```

In C++ the definition differs

**Example 4.5** (Functor). [C++] In C++ templates can be considered as type constructors in Haskell and therefore can convert one type for another. For instance the list of strings can be got with the following construction:

```
using StringList = std::list<std::string>;
StringList a = {"1", "2", "3"};
```

i.e. we have [Objects](#) mapping out of the box. Therefore we need to define fmap operation for [Morphisms](#) mapping to complete the [Functor](#) definition. It can be declared as follows

```
template < template< class ...> class F, class A, class B>
F<B> fmap(std::function<B(A)>, F<A>);
```

The template specialization for the `std::list` can be written as follows

```
// file: functor.h
template <class A, class B>
std::list<B> fmap(std::function<B(A)> f, std::list<A> a) {
  std::list<B> res;
  std::transform(a.begin(), a.end(), back_inserter(res), f);
  return res;
}
```

---

<sup>1</sup>the real definition is quite different from the current one



The simple usage example is the following

```
StringList a = {"1", "2", "3"};
std::function<int(std::string)> f = [](std::string s) {
    return 2 * atoi(s.c_str());
};
auto res = fmap<>(f, a);
```

**Definition 4.6** (Endofunctor). Let  $\mathbf{C}$  is a [Category](#). The [Functor](#)  $E : \mathbf{C} \Rightarrow \mathbf{C}$  i.e. the functor from a category to the same category is called *endofunctor*.

**Definition 4.7** (Identity functor). Let  $\mathbf{C}$  is a [Category](#). The [Functor](#)  $1_{\mathbf{C} \Rightarrow \mathbf{C}} : \mathbf{C} \Rightarrow \mathbf{C}$  is called *identity functor* if for every object  $a \in \text{ob}(\mathbf{C})$

$$1_{\mathbf{C} \Rightarrow \mathbf{C}}(a) = a$$

and for every [Morphism](#)  $f \in \text{hom}(\mathbf{C})$

$$1_{\mathbf{C} \Rightarrow \mathbf{C}}(f) = f$$

**Remark 4.8** (Identity functor). First of all notice that [Identity functor](#) is an [Endofunctor](#).

There is difference between identity functor and [Identity morphism](#) because the first one has deal with both [Objects](#) and [Morphisms](#) while the second one with the objects only.

**Definition 4.9** (Functor composition). If we have 3 categories  $\mathbf{C}, \mathbf{D}, \mathbf{E}$  and 2 functors between them:  $F : \mathbf{C} \Rightarrow \mathbf{D}$  and  $G : \mathbf{D} \Rightarrow \mathbf{E}$  then we can construct a new functor  $H : \mathbf{C} \Rightarrow \mathbf{E}$  that is called *functor composition* and denoted as  $H = G \circ F$ . TBD

## 4.2 Cat category

The [Functor composition](#) is associative by definition. Therefore [Identity functor](#) with the associative composition allow us to define a category where other categories are considered as objects and functors as morphisms:

**Definition 4.10** ([Cat](#) category). The category of small categories (see [Small category](#)) denoted as  $\mathbf{Cat}$  is the [Category](#) where objects are small categories and morphisms are [Functors](#) between them.

We can construct an extension of Cartesian product as follows

**Definition 4.11** (Category Product). If we have 2 categories  $\mathbf{C}$  and  $\mathbf{D}$  then we can construct a new category  $\mathbf{C} \times \mathbf{D}$  with the following components:

- **Objects** are the pairs  $(c, d)$  where  $c \in \text{ob}(\mathbf{C})$  and  $d \in \text{ob}(\mathbf{D})$
- **Morphisms** are the pair  $(f, g)$  where  $f \in \text{hom}(\mathbf{C})$  and  $g \in \text{hom}(\mathbf{D})$
- **Composition** (Axiom 1.8) is defined as follows  $(f_1, g_1) \circ (f_2, g_2) = (f_1 \circ f_2, g_1 \circ g_2)$
- Identity is defined as follows:  $\mathbf{1}_{\mathbf{C} \times \mathbf{D} \rightarrow \mathbf{C} \times \mathbf{D}} = (\mathbf{1}_{\mathbf{C} \rightarrow \mathbf{C}}, \mathbf{1}_{\mathbf{D} \rightarrow \mathbf{D}})$

**Definition 4.12** (Constant functor). Let consider a trivial functor  $\Delta_c$  from Category  $\mathbf{A}$  to category  $\mathbf{C}$  such that  $\forall a \in \text{ob}(\mathbf{A}) : \Delta_c a = c$  -fixed object in  $\mathbf{C}$  and  $\forall f \in \text{hom}(\mathbf{A}) : \Delta_c f = \mathbf{1}_{c \rightarrow c}$ . The trivial functor is called *constant functor*.

**Example 4.13** (Initial object).  $[\mathbf{Cat}]$  Empty category is the Initial object in  $\mathbf{Cat}$  category [21].

**Example 4.14** (Terminal object).  $[\mathbf{Cat}]$  Trivial category is the Terminal object in  $\mathbf{Cat}$  category.

The good example can be found in  $\mathbf{Hask}$  category.

**Example 4.15** (Constant functor).  $[\mathbf{Hask}]$

```
data Const c a = Const c
fmap :: (a -> b) -> Const c a -> Const c b
fmap f (Const c a) = Const c
```

### 4.3 Contravariant functor

Ordinary functor preserves the direction of morphisms and often called as **Covariant functor**. The functor that reverses the direction of morphisms is called as **Contravariant functor**.

**Definition 4.16** (Covariant functor). If we have categories  $\mathbf{C}$  and  $\mathbf{D}$  then the ordinary **Functor**  $\mathbf{C} \Rightarrow \mathbf{D}$  is called *covariant functor*.

**Definition 4.17** (Contravariant functor). If we have categories  $\mathbf{C}$  and  $\mathbf{D}$  then the **Functor**  $\mathbf{C}^{\text{op}} \Rightarrow \mathbf{D}$  is called *contravariant functor*.

**Example 4.18** (Contravariant functor).  $[\mathbf{Hask}]$  Function mapping inside a functor is made via **fmap** (see example 4.3) but sometimes the function that has to be mapped is  $\mathbf{a} \rightarrow \mathbf{b}$  but the result mapping has an inverse order:  $\mathbf{f} \mathbf{b} \rightarrow \mathbf{f} \mathbf{a}$ . In the case the contravariant functor can help

```
class Contravariant f where
  contramap :: (a -> b) -> f b -> f a
```

The contravariant functor should follow the following laws

```
contramap id = id
contramap f . contramap g = contramap (g . f)
```

Consider the following task. We have a predicate for **Int** type that returns **True** if the number is greater than **10** otherwise it returns **False**:

```
newtype Predicate a = Predicate { runPredicate :: a -> Bool}

intgt10 :: Predicate Int
intgt10 = Predicate ( \i -> i > 10 )
```

Now we want to create a predicate that accepts a string and verify if length greater than 10 or not. I.e. we want to have something of the following type:

```
strgt10 :: Predicate String
```

In the case the **Contravariant functor** helps.

```
instance Contravariant Predicate where
  contramap f (Predicate p) = Predicate ( p . f )

strgt10 :: Predicate [Char]
strgt10 = contramap length intgt10
```

## 4.4 Bifunctors

**Definition 4.19** (Bifunctor). Bifunctor is a **Functor** whose **Domain** is a **Category Product**. I.e. if  $\mathbf{C}_1, \mathbf{C}_2, \mathbf{D}$  are 3 categories then the **Functor**  $F : \mathbf{C}_1 \times \mathbf{C}_2 \Rightarrow \mathbf{D}$  is called *bifunctor*.

**Example 4.20** (Bifunctor). **[Set]** Lets  $A, B, C$  and  $D$  are sets and  $f : A \rightarrow C, g : B \rightarrow D$  are two **Functions**. Then the **Cartesian product** with **Product of morphisms** form a **Bifunctor**  $\times$ .

**Example 4.21** (Maybe as a bifunctor). **[Hask]** Lets show how the **Maybe** a type can be constructed from different **Functors** and as result show that the **Maybe a** is also a **Functor**.

```

data Maybe a = Nothing | Just a
-- This is equivalent to
data Maybe a = Either () (Identity a)
-- Either is a bifunctor and () == Const () a
-- Thus Maybe is a composition of 2 functors

```

**Definition 4.22** (Profunctor). If we have a category  $\mathbf{C}$  then the [Bifunctor](#)  $\mathbf{C}^{\text{op}} \times \mathbf{C} \Rightarrow \mathbf{C}$  is called *profunctor*.

**Example 4.23** (Profunctor). [\[Hask\]](#) TBD

```

class Profunctor p where
  dimap :: (a' -> a) -> ( b -> b' ) -> p a b -> p a' b'
  -- p a b == a -> b
  dimap f g h = g . h . f

```

# Chapter 5

## Natural transformation

Natural transformation is the most important part of the category theory. It provides a possibility to compare **Functors** via a standard tool.

### 5.1 Definitions

The natural transformation is not an easy concept compare other ones and requires some additional preparations before we can give the formal definition.

Consider 2 categories  $\mathbf{C}, \mathbf{D}$  and 2 **Functors**  $F : \mathbf{C} \Rightarrow \mathbf{D}$  and  $G : \mathbf{C} \Rightarrow \mathbf{D}$ . If we have an **Object**  $a \in \text{ob}(\mathbf{C})$  then it will be translated by different functors into different objects of category  $\mathbf{D}$ :  $a_F = F(a), a_G = G(a) \in \text{ob}(\mathbf{D})$  (see fig. 5.1). There are 2 options possible

1. There is not any **Morphism** that connects  $a_F$  and  $a_G$ .
2.  $\exists \alpha_a \in \text{hom}(a_F, a_G) \subset \text{hom}(\mathbf{D})$ .



Figure 5.1: Natural transformation: object mapping



Figure 5.2: Natural transformation: morphisms mapping



Figure 5.3: Natural transformation: commutative diagram

We can of course to create an artificial morphism that connects the objects but if we use *natural* morphisms <sup>1</sup> then we can get a special characteristic of the considered functors and categories. For instance if we have such morphisms then we can say that the considered functors are related each other. Opposite example if there are no such morphisms then the functors can be considered as unrelated each other.

The functor is not just the object mapping but also the morphisms mapping. If we have 2 objects  $a$  and  $b$  in the category  $\mathbf{C}$  then we potentially can have a morphism  $f \in \text{hom}_{\mathbf{C}}(a, b)$ . In this case the morphism is mapped by the functors  $F$  and  $G$  into 2 morphisms  $f_F$  and  $f_G$  in the category  $\mathbf{D}$ . As result we have 4 morphisms:  $\alpha_a, \alpha_b, f_F, f_G \in \text{hom}(\mathbf{D})$ . It is natural to impose additional conditions on the morphisms especially that they form a [Commutative diagram](#) (see fig. 5.3):

$$f_f \circ \alpha_b = \alpha_a \circ f_G.$$

---

<sup>1</sup>the word natural means that already existent morphisms from category  $\mathbf{D}$  are used

**Definition 5.1** (Natural transformation). Let  $F$  and  $G$  are 2 **Functors** from category  $\mathbf{C}$  to the category  $\mathbf{D}$ . The *natural transformation* is a set of **Morphisms**  $\alpha \subset \text{hom}(\mathbf{D})$  which satisfy the following conditions:

- For every **Object**  $a \in \text{ob}(\mathbf{C})$   $\exists \alpha_a \in \text{hom}(a_F, a_G)$ <sup>2</sup> - **Morphism** in category  $\mathbf{D}$ . The morphism  $\alpha_a$  is called the component of the natural transformation.
- For every morphism  $f \in \text{hom}(\mathbf{C})$  that connects 2 objects  $a$  and  $b$ , i.e.  $f \in \text{hom}_{\mathbf{C}}(a, b)$  the corresponding components of the natural transformation  $\alpha_a, \alpha_b \in \alpha$  should satisfy the following conditions

$$f_G \circ \alpha_a = \alpha_b \circ f_F, \quad (5.1)$$

where  $f_F = F(f), f_G = G(f)$ . In other words the morphisms form a **Commutative diagram** shown on the fig. 5.3.

We use the following notation (arrow with a dot) for the natural transformation between functors  $F$  and  $G$ :  $\alpha : F \rightarrowtail G$ .

## 5.2 Category of functors

The functors can be considered as objects in a special category **Fun**. The morphisms in the category are **Natural transformations**.

To define a category we need to define composition operation that satisfied **Composition** (**Axiom 1.8**), identity morphism and verify **Associativity** (**Axiom 1.10**).

For the composition consider 2 **Natural transformations**  $\alpha, \beta$  and consider how they act on an object  $a \in \text{ob}(\mathbf{C})$  (see fig. 5.4). We always can construct the composition  $\beta_a \circ \alpha_a$  i.e. we can define the composition of natural transformations  $\alpha, \beta$  as  $\beta \circ \alpha = \{\beta_a \circ \alpha_a | a \in \text{ob}(\mathbf{C})\}$ .

The natural transformation is not just object mapping but also morphism mapping. We will require that all morphisms shown on fig. 5.5 commute. The composition defined in such way is called **Vertical composition**.

**Definition 5.2** (Vertical composition). Let  $F, G, H$  are functors between categories  $\mathbf{C}$  and  $\mathbf{D}$ . Also we have  $\alpha : F \rightarrowtail G, \beta : G \rightarrowtail H$  - natural transformations. We can compose the  $\alpha$  and  $\beta$  as follows

$$\alpha \circ \beta : F \rightarrowtail H.$$

This composition is called *vertical composition*.

---

<sup>2</sup>  $a_F = F(a), a_G = G(a)$



Figure 5.4: Natural transformation vertical composition: object mapping

Figure 5.5: Natural transformation vertical composition: morphism mapping  
- commutative diagram



**Definition 5.3** (**Fun category**). Let  $\mathbf{C}$  and  $\mathbf{D}$  are 2 categories. The category that contains functors  $F : \mathbf{C} \Rightarrow \mathbf{D}$  as objects and **Natural transformation** as morphisms is called as *functor category*. The morphism composition is the **Vertical composition** in the category. The *functor category* between categories  $\mathbf{C}$  and  $\mathbf{D}$  is denoted as  $[\mathbf{C}, \mathbf{D}]$ .

Uniqueness of **Natural transformation** is the same to uniqueness of morphisms in the target category as soon as the natural transformation is a set of **Morphisms** in it. This fact leads to the following examples for initial and terminal objects in **Fun category**.

**Example 5.4** (Terminal object). [**Fun**] Let  $[\mathbf{C}, \mathbf{D}]$  is the functor category between  $\mathbf{C}$  and  $\mathbf{D}$ . If  $t \in \text{ob}(\mathbf{D})$  is the **Terminal object** in the category  $\mathbf{D}$  then the **Constant functor**  $\Delta_t$  is the **Terminal object** in the category  $[\mathbf{C}, \mathbf{D}]$  [6].

**Example 5.5** (Initial object). [**Fun**] Let  $[\mathbf{C}, \mathbf{D}]$  is the functor category between  $\mathbf{C}$  and  $\mathbf{D}$ . If  $i \in \text{ob}(\mathbf{D})$  is the **Initial object** in the category  $\mathbf{D}$  then the **Constant functor**  $\Delta_i$  is the **Initial object** in the category  $[\mathbf{C}, \mathbf{D}]$  [6].

## 5.3 Operations with natural transformations

**Vertical composition** is not the unique way to compose 2 functors. Another option is also possible.

**Definition 5.6** (Horizontal composition). If we have 2 pairs of functors. The first one  $F, G : \mathbf{C} \rightarrow \mathbf{D}$  and another one  $J, K : \mathbf{D} \Rightarrow \mathbf{E}$ . We also have a natural transformation between each pair:  $\alpha : F \rightrightarrows G$  for the first one and  $\beta : J \rightrightarrows K$  for the second one. We can create a new transformation

$$\alpha \star \beta : F \circ J \rightrightarrows G \circ K$$

that is called *horizontal composition*. Note that we use a special symbol  $\star$  for the composition.

**Remark 5.7** (Bifunctor in the category of functors). If we have the same pair of functors as in definition 5.6 then we can consider the functors as **Objects** of 3 categories:  $\mathcal{A} = [\mathbf{C}, \mathbf{D}]$ ,  $\mathcal{B} = [\mathbf{D}, \mathbf{E}]$  and  $\mathcal{C} = [\mathbf{C}, \mathbf{E}]$

We can construct a **Bifunctor**  $\otimes : \mathcal{A} \times \mathcal{B} \Rightarrow \mathcal{C}$  where for each pair of objects  $F \in \text{ob}(\mathcal{A})$ ,  $J \in \text{ob}(\mathcal{B})$  we get another object from  $\mathcal{C}$ . We used the ordinary functor's composition as the operation for objects mapping. I.e.

$$\otimes : F \times G \rightarrow F \circ G \in \text{ob}(\mathcal{C}).$$

The bifunctor is not just a map for objects. There is also a map between morphisms. Thus if we have 2 **Morphisms**:  $\alpha : F \rightarrow G$  and  $\beta : J \rightarrow K$  then we can construct the following mapping

$$\otimes : \alpha \times \beta \rightarrow \alpha \star \beta \in \text{hom}(\mathcal{C}).$$

As result we just introduced mapping  $\otimes$  as a **Bifunctor** in the category of functors.

**Definition 5.8** (Left whiskering). If we have 3 categories **B, C, D**, **Functors**  $F, G : \mathbf{C} \Rightarrow \mathbf{D}$ ,  $H : \mathbf{B} \rightarrow \mathbf{C}$  and **Natural transformation**  $\alpha : F \rightrightarrows G$  then we can construct a new natural transformations:

$$\alpha H : F \circ H \rightrightarrows G \circ H$$

that is called *left whiskering* of functor and natural transformation [16].

**Definition 5.9** (Right whiskering). If we have 3 categories **C, D, E**, **Functors**  $F, G : \mathbf{C} \Rightarrow \mathbf{D}$ ,  $H : \mathbf{D} \rightarrow \mathbf{E}$  and **Natural transformation**  $\alpha : F \rightrightarrows G$  then we can construct a new natural transformations:

$$H\alpha : H \circ F \rightrightarrows H \circ G$$

that is called *right whiskering* of functor and natural transformation [16].

**Definition 5.10** (Identity natural transformation). If  $F : \mathbf{C} \Rightarrow \mathbf{D}$  is a **Functor** then we can define *identity natural transformation*  $\mathbf{1}_{F \rightrightarrows F}$  that maps any **Object**  $a \in \text{ob}(\mathbf{C})$  into **Identity morphism**  $\mathbf{1}_{F(a) \rightarrow F(a)} \in \text{hom}(\mathbf{D})$ .

**Remark 5.11** (Whiskering). With **Identity natural transformation** we can redefine **Left whiskering** and **Right whiskering** via **Horizontal composition** as follows.

For left whiskering:

$$\alpha H = \alpha \star \mathbf{1}_{H \rightrightarrows H} \quad (5.2)$$

For right whiskering:

$$H\alpha = \mathbf{1}_{H \rightrightarrows H} \star \alpha \quad (5.3)$$

## 5.4 Polymorphism and natural transformation

Polymorphism plays a certain role in programming languages. Category theory provides several facts about polymorphic functions which are very important.

**Definition 5.12** (Parametrically polymorphic function). Polymorphism is *parametric* if all function instances behave uniformly i.e. have the same realization. The functions which satisfy the parametric polymorphism requirements are parametrically polymorphic.

**Definition 5.13** (Ad-hoc polymorphism). Polymorphism is *ad-hoc* if the function instances can behave differently dependently on the type they are being instantiated with.

**Theorem 5.14** (Reynolds). *Parametrically polymorphic functions are Natural transformations*

*Proof.* TBD

□

### 5.4.1 Hask category

In Haskell most of functions are [Parametrically polymorphic functions](#)<sup>3</sup>.

**Example 5.15** (Parametrically polymorphic function). **[Hask]** Consider the following function

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:xs) = Just x
```

The function is parametrically polymorphic and by [Reynolds](#) ([Theorem 5.14](#)) is [Natural transformation](#) (see [fig. 5.6](#)).

Therefore from the definition of the natural transformation ([5.1](#)) we have `fmap f . safeHead = safeHead . fmap f`. I.e. it does not matter if we initially apply `fmap f` and then `safeHead` to the result or initially `safeHead` and then `fmap f`.

The statement can be verified directly. For empty list we have

```
fmap f . safeHead []
-- equivalent to
fmap f Nothing
-- equivalent to
Nothing
```

from other side

---

<sup>3</sup>really in the run-time the functions are not [Parametrically polymorphic functions](#)



Figure 5.6: Haskell parametric polymorphism as a natural transformation

```
safeHead . fmap f []
-- equivalent to
safeHead []
-- equivalent to
Nothing
```

For a non empty list we have

```
fmap f . safeHead (x:xs)
-- equivalent to
fmap f (Just x)
-- equivalent to
Just (f x)
```

from other side

```
safeHead . fmap f (x:xs)
-- equivalent to
safeHead (f x: fmap f xs )
-- equivalent to
Just ( f x )
```

Using the fact that **fmap f** is an expensive operation if it is applied to the list we can conclude that the second approach is more productive. Such transformation allows compiler to optimize the code. <sup>4</sup>

<sup>4</sup>It is not directly applied to Haskell because it has lazy evaluation that can perform optimization before that one

# Chapter 6

## Monads

Monads are very important for pure functional programming languages such as Haskell. We will start with [Monoid](#) consideration, continue with the formal mathematical definition for monad and will finish with programming languages examples later.

### 6.1 Monoid in Set category

We are going to consider [Monoid](#) in the terms of Set theory and will try to give the definition that is based rather on morphisms then on internal set structure i.e. we will use [Categorical approach](#). Let  $M$  is a set and by the monoid definition (definition [2.17](#))  $\forall m_1, m_2 \in M$  we can define a new element of the set  $\mu(m_1, m_2) \in M$ . Later we will use the following notation for the  $\mu$ :

$$\mu(m_1, m_2) \equiv m_1 \cdot m_2.$$

If the  $(M, \cdot)$  is monoid then the following 2 conditions have to be satisfied. The first one (associativity) declares that  $\forall m_1, m_2, m_3 \in M$

$$m_1 \cdot (m_2 \cdot m_3) = (m_1 \cdot m_2) \cdot m_3. \quad (6.1)$$

The second one (identity presence) says that  $\exists e \in M$  such that  $\forall m \in M$ :

$$m \cdot e = e \cdot m = m. \quad (6.2)$$

#### 6.1.1 Associativity

Lets consider [\(6.1\)](#) in details. We can define  $\mu$  as a [Morphism](#) in the following way

$$\mu : M \times M \rightarrow M,$$

where  $M \times M$  is the [Product](#) ([Example 2.12](#)) in the [Set](#) category. I.e.  $M \times M, M \in \text{ob}(\mathbf{Set})$  and  $\mu \in \text{hom}(\mathbf{Set})$ . Consider other objects of  $\mathbf{Set}$ :  $A = M \times (M \times M)$  and  $A' = (M \times M) \times M$ . They are not the same but there is a trivial [Isomorphism](#) between them  $A \cong_\alpha A'$ , where the isomorphism  $\alpha$  defined as

$$\alpha(x, (y, z)) = ((x, y), z).$$

Consider the action of [Product of morphisms](#)  $\mathbf{1}_{M \rightarrow M} \times \mu$  on  $A$ :

$$[\mathbf{1}_{M \rightarrow M} \times \mu](x, (y, z)) = (\mathbf{1}_{M \rightarrow M}(x), \mu(y, z)) = (x, y \cdot z) \in M \times M$$

i.e.  $\mathbf{1}_{M \rightarrow M} \times \mu : M \times (M \times M) \rightarrow M \times M$ . If we act  $\mu$  on the result then we can obtain:

$$\begin{aligned} \mu([\mathbf{1}_{M \rightarrow M} \times \mu](x, (y, z))) &= \\ &= \mu(\mathbf{1}_{M \rightarrow M}(x), \mu(y, z)) = \\ &= \mu(x, y \cdot z) = x \cdot (y \cdot z) \in M, \end{aligned}$$

i.e.  $\mu \circ [\mathbf{1}_{M \rightarrow M} \times \mu] : M \times (M \times M) \rightarrow M$ .

For  $A'$  we have the following one:

$$\mu \circ [\mu \times \mathbf{1}_{M \rightarrow M}]((x, y), z) = \mu(x \cdot y, z) = (x \cdot y) \cdot z.$$

Monoid associativity requires

$$x \cdot (y \cdot z) = (x \cdot y) \cdot z$$

i.e. the morphisms are shown in [fig. 6.1](#) commute:

$$\mu \circ [\mu \times \mathbf{1}_{M \rightarrow M}] = \mu \circ [\mathbf{1}_{M \rightarrow M} \times \mu] \circ \alpha. \quad (6.3)$$

Very often the isomorphism  $\alpha$  is omitted i.e.

$$M \times (M \times M) = (M \times M) \times M = M^3$$

and the morphism equality [\(6.3\)](#) is written as follow

$$\mu \circ [\mu \times \mathbf{1}_{M \rightarrow M}] = \mu \circ [\mathbf{1}_{M \rightarrow M} \times \mu].$$

The corresponding commutative diagram is shown in [fig. 6.2](#).

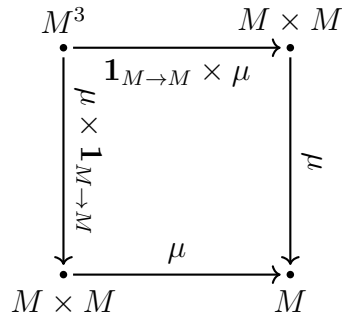
Figure 6.1: Commutative diagram for  $\mu \circ [\mu \times \mathbf{1}_{M \rightarrow M}] = \mu \circ [\mathbf{1}_{M \rightarrow M} \times \mu] \circ \alpha$ .Figure 6.2: Commutative diagram for  $\mu \circ [\mu \times \mathbf{1}_{M \rightarrow M}] = \mu \circ [\mathbf{1}_{M \rightarrow M} \times \mu]$



Figure 6.3: Commutative diagram for  $\mu \circ (\eta \times \mathbf{1}_{M \rightarrow M}) \circ \lambda = \mu \circ (\mathbf{1}_{M \rightarrow M} \times \mu) \circ \rho = \mathbf{1}_{M \rightarrow M}$ .

### 6.1.2 Identity presence

For (6.2) consider a morphism  $\eta$  from [Singleton](#)<sup>1</sup>  $I = \{0\}$  to the special element  $e \in M$  such that  $\forall m \in M : e \cdot m = m \cdot e = m$ . I.e.  $\eta : I \rightarrow M$  and  $e = \eta(0)$ . Consider 2 sets  $B = I \times M$  and  $B' = M \times I$ . We have 2 [Isomorphisms](#):  $B \cong_\lambda M$  and  $B' \cong_\rho M$  such that

$$\lambda(m) = (0, m) \in B = I \times M$$

and

$$\rho(m) = (m, 0) \in B' = M \times I.$$

If we apply the products (see [Product of morphisms](#))  $\eta \times \mathbf{1}_{M \rightarrow M}$  and  $\mathbf{1}_{M \rightarrow M} \times \eta$  on  $B$  and  $B'$  respectively then we get

$$\begin{aligned} [\eta \times \mathbf{1}_{M \rightarrow M}](0, m) &= (e, m), \\ [\mathbf{1}_{M \rightarrow M} \times \eta](m, 0) &= (m, e). \end{aligned}$$

After the application of  $\mu$  on the result we obtain

$$\begin{aligned} \mu([\eta \times \mathbf{1}_{M \rightarrow M}](0, m)) &= \mu(e, m) = e \cdot m, \\ \mu([\mathbf{1}_{M \rightarrow M} \times \eta](m, 0)) &= \mu(m, e) = m \cdot e. \end{aligned}$$

The (6.2) leads to the following equation for morphisms

$$\mu \circ (\eta \times \mathbf{1}_{M \rightarrow M}) \circ \rho = \mu \circ (\mathbf{1}_{M \rightarrow M} \times \mu) \circ \lambda = \mathbf{1}_{M \rightarrow M}$$

or the commutative diagram shown on fig. 6.3.

---

<sup>1</sup> It also is called [\[10, p. 2\]](#) as a one point set



### 6.1.3 Categorical definition for monoid

Before given a formal definition lets look at the operations were used for the construction. The first one is the product of 2 objects:

$$M \times M.$$

We also have 2 pairs of morphisms:

$$\begin{aligned}\mu : M \times M &\rightarrow M, \\ \mathbf{1}_{M \rightarrow M} : M &\rightarrow M.\end{aligned}$$

and

$$\begin{aligned}\eta : I &\rightarrow M, \\ \mathbf{1}_{M \rightarrow M} : M &\rightarrow M.\end{aligned}$$

The pairs can be combined into one using [Product of morphisms](#) as follows:

$$\begin{aligned}\mu \times \mathbf{1}_{M \rightarrow M} : (M \times M) \times M &\rightarrow M \times M, \\ \mathbf{1}_{M \rightarrow M} \times \mu : M \times (M \times M) &\rightarrow M \times M\end{aligned}$$

and

$$\begin{aligned}\eta \times \mathbf{1}_{M \rightarrow M} : I \times M &\rightarrow M \times M, \\ \mathbf{1}_{M \rightarrow M} \times \eta : M \times I &\rightarrow M \times M.\end{aligned}$$

The same structure <sup>2</sup> is used by [Functor](#) and especially by [Bifunctor](#) ([Example 4.20](#)).

Now we are ready to provide the monoid definition in the terms of morphisms.

**Definition 6.1** (Monoid). Consider [Set](#) category  $\mathbf{C}$  with a [Singleton](#)  $t \in \text{ob}(\mathbf{C})$ . The [Cartesian product](#) with [Product of morphisms](#) forms a [Bifunctor](#)  $\times$  (see [example 4.20](#)). The object  $m \in \text{ob}(\mathbf{C})$  is called *monoid* if the following conditions satisfied:

1. there is a [Morphism](#)  $\mu : m \times m \rightarrow m$  in the category
2. there is another morphism  $\eta : t \rightarrow m$  in the category

---

<sup>2</sup>not only objects mapping but also morphisms mapping

3. the morphisms satisfy the following conditions:

$$\mu \circ (\mu \times \mathbf{1}_{M \rightarrow M}) = \mu \circ (\mathbf{1}_{M \rightarrow M} \times \mu) \circ \alpha, \quad (6.4)$$

$$\mu \circ (\eta \times \mathbf{1}_{M \rightarrow M}) \circ \lambda = \mu \circ (\mathbf{1}_{M \rightarrow M} \times \mu) \circ \rho = \mathbf{1}_{M \rightarrow M} \quad (6.5)$$

where  $\alpha$  (associator) is an [Isomorphism](#) between  $m \times (m \times m)$  and  $(m \times m) \times m$ .  $\lambda, \rho$  are other isomorphisms:

$$m \cong_{\lambda} t \times m$$

and

$$m \cong_{\rho} m \times t$$

### 6.1.4 Monoid importance

Why is the concept of [Monoid](#) so important? In pure math it provides the build blocks for important concepts such as Group, Ring, Field [14]. In programming languages it gives more simple and robust concept for software design [3].

We can notice that monoid definition (see definition 2.17) has the same requirements as morphisms in category theory. Moreover the monoid can be viewed as a [Category](#) (see section 2.4).

Monoid provides a closed collection of objects such that if you combine them you will get an object of the same type. This allows to create constructions which are more easy in maintenance. For instance there are 2 possible options to combine objects of type  $A$  in software architecture [3]:

1. **Conventional architecture** assumes that a combination of several objects of type  $A$  will produce a “network” of the objects  $A$  i.e. new type  $B$
2. **Haskell architecture** assumes that a combination of several objects of type  $A$  will produce a new object of the same type  $A$

You can see that in the first case any modification of the base type  $A$  will require changes in the upper-layer class  $B$ . This produce very complex structure of types if objects of type  $B$  will be combined into new type  $C$  etc.

You will not get the problems in the second case because you will be always in a closed collection of objects with the same type  $A$ .

## 6.2 Monoidal category

As we saw in the categorical definition for monoid (see definition 6.1) the category  $\mathbf{C}$  should satisfy several conditions to have an object as monoid. Lets formalise the conditions.

**Definition 6.2** (Monoidal category). A category  $\mathbf{C}$  is called *monoidal category* if it is equipped with a **Monoid** structure i.e. there are

- **Bifunctor**  $\otimes : \mathbf{C} \times \mathbf{C} \Rightarrow \mathbf{C}$  called *monoidal product*
- an **Object**  $e$  called unit object or identity object

The elements should satisfy (up to **Isomorphism**) several conditions. The first one: associativity:

$$a \otimes (b \otimes c) \cong_{\alpha} (a \otimes b) \otimes c,$$

where  $\alpha$  is called associator. The second condition says that  $e$  can be treated as left and right identity:

$$\begin{aligned} a &\cong_{\lambda} e \otimes a, \\ a &\cong_{\rho} a \otimes e, \end{aligned}$$

where  $\lambda, \rho$  are called as left and right unitors respectively.

In the **Set category** we have  $\times$  as the monoidal product (see example 4.20). There is also a morphism  $\eta$  from terminal object  $t$  to  $e$  [5] (see definition 6.1).

**Definition 6.3** (Strict monoidal category). A **Monoidal category**  $\mathbf{C}$  is said to be *strict* if the associator, left and right unitors are all identity morphisms i.e.

$$\alpha = \lambda = \rho = \mathbf{1}_{C \rightarrow C}.$$

**Remark 6.4** (Monoidal product). The monoidal product is a binary operation that specifies the exact monoidal structure. Often it is called as *tensor product* but we will avoid the naming because it is not always the same as the **Tensor product** introduced for **Hilbert spaces**. We also note that the monoidal product is a **Bifunctor**.

### 6.3 Tensor product in Quantum mechanics

**Definition 6.5** (Tensor product). Let  $m, n \in \mathbf{FdHilb}$ . The *tensor product*  $m \otimes n$  is another finite dimensional Hilbert space equipped with a bilinear form

$$\phi : m \times n \rightarrow m \otimes n$$

such that  $\forall a \in \mathbf{FdHilb}$  and for any bilinear

$$f : m \times n \rightarrow a$$

exists only one morphism  $\tilde{f} : m \otimes n \rightarrow a$  such that

$$f = \tilde{f} \circ \phi$$

i.e. the diagram on the fig. 6.4 commutes



Figure 6.4: Commutative diagram for tensor product definition.

**Remark 6.6** (Tensor product). Using the fact that Linear maps are Morphisms in  $\mathbf{FdHilb}$  we can conclude that Tensor product is a Bifunctor.

The tensor product in quantum mechanics is used for representing a system that consists of multiple systems. For instance if we have an interaction between an 2 level atom ( $a$  is excited state  $b$  as a ground state) and one mode light then the atom has its own Hilbert space  $\mathcal{H}_{at}$  with  $|a\rangle$  and  $|b\rangle$  as basis vectors. Light also has its own Hilbert space  $\mathcal{H}_f$  with Fock state  $\{|n\rangle\}$  as the basis.<sup>3</sup> The result system that describes both atom and light is represented as the tensor product  $\mathcal{H}_{at} \otimes \mathcal{H}_f$ .

**Remark 6.7** (Hilbert-Schmidt correspondence). The morphisms of  $\mathbf{FdHilb}$  category have a connection with Tensor product. Consider the so called

---

<sup>3</sup> Really the  $\mathcal{H}_f$  is infinite dimensional Hilbert space and seems to be out of our assumption about  $\mathbf{FdHilb}$  category as a collection of finite dimensional Hilbert spaces only.

Hilbert-Schmidt correspondence for finite dimensional Hilbert spaces i.e. for given  $\mathcal{A}$  and  $\mathcal{B}$  there is a natural isomorphism between the tensor product and [Linear maps](#) (aka morphisms) between  $\mathcal{A}$  and  $\mathcal{B}$ :

$$\mathcal{A}^* \otimes \mathcal{B} \cong \text{hom}(\mathcal{A}, \mathcal{B})$$

where  $\mathcal{A}^*$  - [Dual space](#).

## 6.4 Category of endofunctors

The [Fun category](#) is an example of a category. We can apply additional limitation and consider only [Endofunctors](#) i.e. we will look at the category  $[\mathbf{C}, \mathbf{C}]$  - the category of functors from category  $\mathbf{C}$  to the same category. One of the most popular math definition of a monad is the following: “All told, a monad in  $X$  is just a monoid in the category of endofunctors of  $X$ ” [10, p. 138]. Later we will give an explanation for that one.

We start with the formal definition of category of endofunctors and a tensor product in the category

**Definition 6.8** (Category of endofunctors). Let  $\mathbf{C}$  is a category, then the category  $[\mathbf{C}, \mathbf{C}]$  of functors from category  $\mathbf{C}$  to the same category is called the category of endofunctors. The monoidal product in the category is the functor composition.

**Definition 6.9** (Monad). The monad  $M$  is an [Endofunctor](#) with 2 [Natural transformations](#):

$$\mu : M \circ M \rightarrowtail M \tag{6.6}$$

and

$$\eta : \mathbf{1}_{\mathbf{C} \Rightarrow \mathbf{C}} \rightarrowtail M, \tag{6.7}$$

where  $\mathbf{1}_{\mathbf{C} \Rightarrow \mathbf{C}}$  is [Identity functor](#).

The  $\eta, \mu$  should satisfy the following conditions:

$$\begin{aligned} \mu \circ M\mu &= \mu \circ \mu M, \\ \mu \circ M\eta &= \mu \circ \eta M = \mathbf{1}_{M \rightarrowtail M}, \end{aligned} \tag{6.8}$$

where  $M\mu, M\eta$  - [Right whiskerings](#),  $\mu M, \eta M$  - [Left whiskerings](#),  $\mathbf{1}_{M \rightarrowtail M}$  - [Identity natural transformation](#) for  $M$ . [Vertical composition](#) is used in the equations.

The monad will be denoted later as  $\langle M, \mu, \eta \rangle$ .

**Remark 6.10** (Monad term). The word monad is a concatenation of 2 words: **monoid** and **triad** [10, p. 138]. The first one points to the fact that the object looks like a monoid. The second one says that it is a set of 3 objects (**Endofunctor** and 2 **Natural transformations**) aka triad.

Lets look at the requirements (6.8) more closely. Notice that the functor composition is associative:

$$M \circ (M \circ M) = (M \circ M) \circ M = M^3.$$

Secondly all rewrite it with (5.2) and (5.3) as follows

$$\begin{aligned} \mu \circ (\mathbf{1}_{M \rightarrow M} \star \mu) &= \mu \circ (\mu \star \mathbf{1}_{M \rightarrow M}), \\ \mu \circ (\mathbf{1}_{M \rightarrow M} \star \eta) &= \mu \circ (\eta \star \mathbf{1}_{M \rightarrow M}) = \mathbf{1}_{M \rightarrow M}. \end{aligned} \quad (6.9)$$

Thus we can notice that the pair of operations (composition  $\circ$  and **Horizontal composition**  $\star$ ) forms the bifunctor (see **Bifunctor in the category of functors** (Remark 5.7)).

The morphism  $\mathbf{1}_{M \rightarrow M} \star \mu$  acts on  $M \circ (M \circ M)$  as

$$\mathbf{1}_{M \rightarrow M} \star \mu : M \circ (M \circ M) \rightarrow M \circ (M \otimes M)$$

thus

$$\mu \circ (\mathbf{1}_{M \rightarrow M} \star \mu) : M \circ (M \circ M) \rightarrow M \otimes (M \otimes M).$$

Similarly

$$\mu \circ (\mu \star \mathbf{1}_{M \rightarrow M}) : (M \circ M) \circ M \rightarrow (M \otimes M) \otimes M.$$

I.e. the both morphisms start at the same object  $M^3$  and finish also at the same point. The equality

$$\mu \circ (\mathbf{1}_{M \rightarrow M} \star \mu) = \mu \circ (\mu \star \mathbf{1}_{M \rightarrow M}) \quad (6.10)$$

is similar to the conditions on the fig. 6.2 and can be written as fig. 6.5. Thus if we compare (6.10) and (6.4) then we can say that they are same if we replace  $\star$  sign with  $\times$  one. I.e. in the case we can say that the monad looks like a **Monoid**.

For the identity element consider the same trick: replace in (6.5) tensor product  $\times$  with **Horizontal composition**  $\star$  and morphisms  $\mathbf{1}_{M \rightarrow M}, \rho, \lambda$  with identity natural transformation  $\mathbf{1}_{M \rightarrow M}$ . Thus the equation

$$\mu \circ (\eta \times \mathbf{1}_{M \rightarrow M}) \circ \lambda = \mu \circ (\mathbf{1}_{M \rightarrow M} \times \mu) \circ \rho = \mathbf{1}_{M \rightarrow M}$$

will be replaced with

$$\mu \circ (\eta \star \mathbf{1}_{M \rightarrow M}) = \mu \circ (\mathbf{1}_{M \rightarrow M} \star \mu) = \mathbf{1}_{M \rightarrow M}$$

that is the exact we want to get (see second equation of (6.9)).



Figure 6.5: Monad as monoid in the category of endofunctors.

## 6.5 Monads in programming languages

There are several examples of [Monad](#) implementation in different programming languages:

### 6.5.1 Haskell

**Example 6.11** (Monad). [\[Hask\]](#) In Haskell monad can be defined from [Functor](#) ([Example 4.3](#)) as follows <sup>4</sup>

```
class Functor m => Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

To show how this one can be get we can start from a definition that is similar to the math definition:

```
class Functor m => Monad m where
  return :: a -> m a
  join   :: m (m a) -> m a
```

where **return** can be treated as  $\eta$  ([6.7](#)) and **join** as  $\mu$  ([6.6](#)). In the case the bind operator  $\gg=$  can be implemented as follows

```
(>>=) :: m a -> (a -> m b) -> m b
ma >>= f = join ( fmap f ma )
```

---

<sup>4</sup>real definition is quite different from the presented one

### 6.5.2 C++

The monad in C++ use the functor definition from [Functor](#) ([Example 4.5](#))

```
// from functor.h
template < template< class ...> class M, class A, class B>
M<B> fmap(std::function<B(A)>, M<A>);

// file: monad.h
template < template< class ...> class M, class A>
M<A> pure(A);

template < template< class ...> class M, class A>
M<A> join(M< M<A> >);
```

where **pure** can be treated as  $\eta$  ([6.7](#)) and **join** as  $\mu$  ([6.6](#)). In the case the bind operator can be implemented as follows

```
template < template< class ...> class M, class A, class B>
M<B> bind(std::function< M<B> (A) > f, M<A> a) {
    return join( fmap<>(f, a) );
};
```

### 6.5.3 Scala

**Example 6.12** (Monad). **[Scala]** The monad concept in Scala is more close to formal math definition for [Monad](#). It can be defined as follows <sup>5</sup>

```
trait M[A] {
    def flatMap[B](f: A => M[B]): M[B]
}

def unit[A](x: A): M[A]
```

I.e. **flatMap** can be considered as  $\mu$  and **unit** as  $\eta$ .

TBD

---

<sup>5</sup>real definition is quite different from the presented one



# Chapter 7

## Kleisli category

**Definition 7.1** (Kleisli category). Let  $\mathbf{C}$  is a category,  $M$  is an [Endofunctor](#) and  $\langle M, \mu, \eta \rangle$  is a [Monad](#). Then we can construct a new category  $\mathbf{C}_M$  as follows:

$$\begin{aligned}\text{ob}(\mathbf{C}_M) &= \text{ob}(\mathbf{C}), \\ \text{hom}_{\mathbf{C}_M}(a, b) &= \text{hom}_{\mathbf{C}}(a, M(b))\end{aligned}$$

i.e. objects of categories  $\mathbf{C}$  and  $\mathbf{C}_M$  are the same but morphisms from  $\mathbf{C}_M$  form a subset of morphisms  $\mathbf{C}$ :  $\text{hom}(\mathbf{C}_M) \subset \text{hom}(\mathbf{C})$ . The category is called as *Kleisli category*.

The identity morphism in the Kleisli category is the [Natural transformation](#)  $\eta$  (6.7) defined by the monad  $\langle M, \mu, \eta \rangle$ :

$$\mathbf{1}_{C_M \rightarrow C_M} = \eta$$

**Remark 7.2** (Kleisli category composition). [Kleisli category](#) has a non trivial composition rules. If we have 2 [Morphisms](#) from  $\text{hom}(\mathbf{C}_M)$ :

$$f_M : a \rightarrow b$$

and

$$g_M : b \rightarrow c.$$

The morphisms have correspondent ones in  $\mathbf{C}$ :

$$f : a \rightarrow M(b)$$

and

$$g : b \rightarrow M(c).$$

The composition  $g_M \circ f_M$  gives a new morphism

$$h_M = g_M \circ f_M : a \rightarrow c.$$

The corresponding one from  $\mathbf{C}$  is

$$h : a \rightarrow M(c).$$

It has to be pointed that the compositions in  $\mathbf{C}$  and  $\mathbf{C}_M$  are not the same:

$$g_M \circ f_M \neq g \circ f.$$

[Kleisli category](#) widely spread in programming especially it provides good description for different types of computations, for instance [13, 12]

- **Partiality** i.e. then a function not defined for each input, for instance the following expression is undefined (or partially defined) for  $x = 0$ :  
 $f(x) = \frac{1}{x}$
- **Non-Determinism** i.e. then multiply output are possible
- **Side-effects** i.e. then a function communicates with an environment
- **Exception** i.e. when some input is incorrect and can produce an abnormal result. Therefore it is the same as **Partiality** and will be considered below as the same type of computation.
- **Continuation** i.e. when we need to save the current state of the computation and be able to restore it on demand later
- **Interactive input** i.e. a function that reads data from an input device (keyboard, mouse, etc.)
- **Interactive output** i.e. a function that writes data to an output device (monitor etc.)

## 7.1 Partiality and Exception

Partial functions and exceptions can be processed via monad be called as Maybe. There will be implementations in different languages below. And the usage example for the following function implementation

$$h(x) = \frac{1}{2\sqrt{x}}.$$

The function is a composition of 3 functions:

$$\begin{aligned} f_1(x) &= \sqrt{x}, \\ f_2(x) &= 2 \cdot x, \\ f_3(x) &= \frac{1}{x} \end{aligned} \tag{7.1}$$

and as result the goal can be implemented as the following composition:

$$h = f_3 \circ f_2 \circ f_1. \tag{7.2}$$

$f_2$  is a [Pure function](#) and defined  $\forall x \in \mathbb{R}$ . The functions  $f_1, f_3$  are partially defined.

### 7.1.1 Haskell example

**Example 7.3** (Maybe monad). [\[Hask\]](#) The Maybe monad can be implemented as follows

```
instance Monad Maybe where
  return = Just
  join Just( Just x) = Just x
  join _ = Nothing
```

Our functions (7.1) can be implemented as follows

```
f1 :: (Ord a, Floating a) => a -> Maybe a
f1 x = if x >= 0 then Just(sqrt x) else Nothing

f2 :: Num a => a -> Maybe a
f2 x = Just (2*x)

f3 :: (Eq a, Fractional a) => a -> Maybe a
f3 x = if x /= 0 then Just(1/x) else Nothing
```

The  $h$  (7.2) is the composition via bind operator:

```
h :: (Ord a, Floating a) => a -> Maybe a
h x = (return x) >>= f1 >>= f2 >>= f3
```

The usage example is the following:

```

*Main> h 4
Just 0.25
*Main> h 1
Just 0.5
*Main> h 0
Nothing
*Main> h (-1)
Nothing

```

### 7.1.2 C++ example

**Example 7.4** (Maybe monad). [C++] The Maybe monad can be implemented as follows

```

template <class A> using Maybe = std::optional<A>;

template < class A, class B>
Maybe<B> fmap(std::function<B(A)> f, Maybe<A> a) {
    if (a) {
        return f(a.value());
    }
    return {};
}

template < class A>
Maybe<A> pure(A a) {
    return a;
}

template < class A>
Maybe<A> join(Maybe< Maybe<A> > a){
    if (a) {
        return a.value();
    }
    return {};
}

```

Our functions (7.1) can be implemented as follows

```

std::function<Maybe<float>(float)> f1 =
    [] (float x) {
        if (x >= 0) {

```

```

        return Maybe<float>(sqrt(x));
    }
    return Maybe<float>();
};

std::function<Maybe<float>(float)> f2 = [](float x) { return 2 * x; };

std::function<Maybe<float>(float)> f3 =
    [](float x) {
        if (x != 0) {
            return Maybe<float>(1 / x);
        }
        return Maybe<float>();
    };
}

```

The  $h$  (7.2) is the composition via bind operator:

```

auto h(float x) {
    Maybe<float> a = pure(x);
    return bind(f3, bind(f2, bind(f1, a)));
};

```

## 7.2 Non-Determinism

The situation when a function returns several values is not applicable for **Set category** but can appear for **Rel category**. From other hand the non standard situation is required for practical applications and as result has to be modeled in programming languages. The **List** monad is used for it.

### 7.2.1 Haskell example

**Example 7.5** (List monad). [Hask]

```

instance Monad [] where
    return x = [x]
    join = concat

```

## 7.3 Side effects and interactive input/output

TBD

## 7.4 Continuation

TBD

# Chapter 8

## Limits

### 8.1 Definitions



Figure 8.1: Diagram of shape  $F : \mathbf{I} \Rightarrow \mathbf{C}$ . Objects  $a_{i,j}^{(I)} \in \text{ob}(\mathbf{I})$  are mapped to  $a_{i,j}^{(C)} \in \text{ob}(\mathbf{C})$ . Morphisms  $g_{ij}^{(I)} \in \text{hom}(\mathbf{I})$  are mapped to  $g_{ij}^{(C)} \in \text{hom}(\mathbf{C})$

**Definition 8.1** (Diagram of shape). Let  $\mathbf{I}$  and  $\mathbf{C}$  are 2 categories. The *diagram of shape  $\mathbf{I}$  in  $\mathbf{C}$*  is a **Functor** (see fig. 8.1)

$$F : \mathbf{I} \Rightarrow \mathbf{C}$$

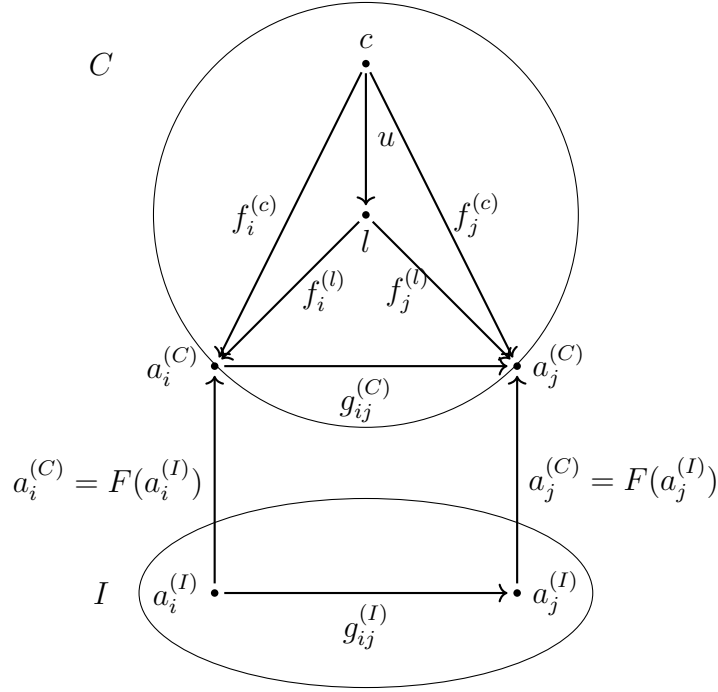
**Definition 8.2** (Index category). Category  $\mathbf{I}$  in the definition 8.1 is called *Index category*.

## 8.1.1 Limit

Figure 8.2: Cone  $\text{cone}(c, f^{(c)})$ 

**Definition 8.3** (Cone). Let  $F$  is a [Diagram of shape  \$\mathbf{I}\$](#)  in  $\mathbf{C}$ . A *cone* to  $F$  is an object  $d \in \text{ob}(\mathbf{C})$  with [Morphisms](#)  $f^c = \left\{ f_i^c : c \rightarrow a_i^{(C)} \right\}$ , where  $a_i^{(C)} = F(a_i^{(I)})$  indexed by objects from  $\mathbf{I}$  (see fig. 8.2). The cone is denoted as  $\text{cone}(c, f^{(c)})$ .



Figure 8.3: Limit cone( $l, f^{(l)}$ )

**Definition 8.4** (Limit). *Limit* of [Diagram of shape](#)  $F : \mathbf{I} \Rightarrow \mathbf{C}$  is a [Cone](#)  $\text{cone}(l, f^{(l)})$  to  $F$  such that for any other cone( $c, f^{(c)}$ ) to  $F$  exists an unique morphism  $u : c \rightarrow l$  such that  $\forall a_i^{(I)} \in \text{ob}(\mathbf{I})$   $f_i^{(l)} \circ u = f_i^{(c)}$  i.e. diagram shown on fig. 8.3 commutes.

If we have 2 objects from  $\mathbf{C}$  ( $c_1, c_2 \in \text{ob}(\mathbf{C})$ ) then we can have a lot of morphisms between the objects which form a set:  $\text{hom}_{\mathbf{C}}(c_1, c_2)$ . There is a subset of  $\text{hom}_{\mathbf{C}}(c_1, c_2)$  that can be called as cone's morphisms.

**Definition 8.5** (Morphisms of cones). Let  $c_1, c_2 \in \text{ob}(\mathbf{C})$  are 2 objects from category  $\mathbf{C}$  and  $\text{cone}(c_1, f^{(c_1)}), \text{cone}(c_2, f^{(c_2)})$  are 2 [Cones](#). The morphism  $m \in \text{hom}_{\mathbf{C}}(c_1, c_2)$  is called as morphism of cones if  $\forall i$

$$f_i^{(c_1)} = f_i^{(c_2)} \circ m,$$

i.e. the morphisms in fig. 8.4 commute.

**Definition 8.6** (Category of cones to  $F$ ). Let  $F$  is a [Diagram of shape](#)  $\mathbf{I}$  in  $\mathbf{C}$ .

TBD

The category of [Cones](#) is denoted as  $\Delta \downarrow F$  [23]



Figure 8.4: Morphism  $m$  between 2 cones  $\text{cone}(c_1, f^{(c_1)})$  and  $\text{cone}(c_2, f^{(c_2)})$

**Remark 8.7** (Category of cones to  $F$ ). Let  $F$  is a [Diagram of shape  \$\mathbf{I}\$  in  \$\mathbf{C}\$](#)  and  $\Delta \downarrow F$  is the [Category of cones to  \$F\$](#) . Then [Limit](#) is [Terminal object](#) in the category.

## 8.1.2 Colimit

Figure 8.5: Co-cone  $\text{cocone}(c, f^{(c)})$ 

**Definition 8.8** (Cocone). Let  $F$  is a [Diagram of shape  \$\mathbf{I}\$](#)  in  $\mathbf{C}$ . A *co-cone* to  $F$  is an object  $d \in \text{ob}(\mathbf{C})$  with [Morphisms](#)  $f^c = \left\{ f_i^c : a_i^{(C)} \rightarrow c \right\}$ , where  $a_i^{(C)} = F(a_i^{(I)})$  indexed by objects from  $\mathbf{I}$  (see fig. 8.5). The co-cone is denoted as  $\text{cocone}(c, f^{(c)})$ .

Figure 8.6: Co-Limit cocone( $l, f^{(l)}$ )

**Definition 8.9** (Colimit). *Co-Limit* of [Diagram of shape](#)  $F : \mathbf{I} \Rightarrow \mathbf{C}$  is a [Cocone](#)  $\text{cocone}(l, f^{(l)})$  to  $F$  such that for any other cocone( $c, f^{(c)}$ ) to  $F$  exists an unique morphism  $u : l \rightarrow c$  such that  $\forall a_i^{(I)} \in \text{ob}(\mathbf{I})$   $u \circ f_i^{(c)} = f_i^{(l)}$  i.e. diagram shown on fig. 8.6 commutes.

**Definition 8.10** (Category of co-cones from  $F$ ). Let  $F$  is a [Diagram of shape](#)  $\mathbf{I}$  in  $\mathbf{C}$ .

TBD

The category of [Cocones](#) is denoted as  $F \downarrow \Delta$  [23]

**Remark 8.11** (Category of co-cones). Let  $F$  is a [Diagram of shape](#)  $\mathbf{I}$  in  $\mathbf{C}$  and  $F \downarrow \Delta$  is the [Category of co-cones from](#)  $F$ . Then [Colimit](#) is [Initial object](#) in the category.

## 8.2 Cone as natural transformation

The [Cone](#) can be considered as a [Natural transformation](#). There are 2 functors between categories  $\mathbf{I}$  and  $\mathbf{C}$ . The first one is the [Diagram of shape](#)  $F : \mathbf{I} \Rightarrow \mathbf{C}$ . The second one is the [Constant functor](#):  $\Delta_c : \mathbf{I} \Rightarrow \mathbf{C}$ . The [Natural transformation](#)  $\Delta_c \rightrightarrows F$ , by the definition, is the set of [Morphisms](#)

from  $\mathbf{C}$  with additional relations that are same as conditions defined for the  $\text{Cone}(c, f^{(c)})$ . Therefore we can consider the  $\text{Cone}$  as a  $\text{Natural transformation}$ .

### 8.3 Categorical constructions as limits

Different choice for category  $\mathbf{I}$  gives different types of limits. There are several examples of such constructions below

The empty category will give us the terminal object. The  $\text{Discrete category}$  with 2 elements produces  $\text{Product}$  as the  $\text{Limit}$ .

#### 8.3.1 Initial and terminal objects

If we choose  $\text{Empty category}$  as the  $\text{Index category}$  (see fig. 8.7) then we can get  $\text{Terminal object}$  as  $\text{Limit}$  and  $\text{Initial object}$  as  $\text{Colimit}$ .



Figure 8.7: Index category  $I$  for initial and terminal objects. The category is empty

**Example 8.12** (Limit). [Terminal object] Lets choose  $\text{Empty category}$  as the  $\text{Index category } \mathbf{I}$ .



Figure 8.8: Terminal object  $t$  as a limit

The **Cone** consists from the apex  $c$  only (see fig. 8.8). The **Limit** will be **Terminal object** in the category  $\mathbf{C}$ .

**Example 8.13** (Colimit). [Initial object] Lets choose **Empty category** as the **Index category**  $\mathbf{I}$ .



Figure 8.9: Initial object  $i$  as a colimit

The **Cocone** consists from the apex  $c$  only (see fig. 8.9). The **Colimit** will be **Initial object** in the category  $\mathbf{C}$ .

### 8.3.2 Product and sum

If choose **Discrete category** with 2 objects as the **Index category** (see fig. 8.10) then we can get **Product** as **Limit** and **Sum** as **Colimit**.



Figure 8.10: Index category  $I$  for product and sum. It consists of 2 objects  $a^{(I)}, b^{(I)}$  and 2 trivial (identity) morphisms  $\mathbf{1}_{a^{(I)} \rightarrow a^{(I)}}, \mathbf{1}_{b^{(I)} \rightarrow b^{(I)}}$

**Example 8.14** (Limit). [Product] Lets choose **Discrete category** with 2 objects as the **Index category**  $\mathbf{I}$ .



Figure 8.11: Product as a limit

The [Diagram of shape  \$F\$](#)  gives us the mapping into 2 objects in the category  $\mathbf{C}$  (see fig. 8.11). The [Limit](#) of the [Diagram of shape](#) is the [Product](#) of the 2 objects in the category  $\mathbf{C}$ .

**Example 8.15** (Colimit). [Sum] Lets choose [Discrete category](#) with 2 objects as the [Index category  \$\mathbf{I}\$](#) .



Figure 8.12: Sum as a colimit

The [Diagram of shape  \$F\$](#)  gives us the mapping into 2 objects in the category  $\mathbf{C}$  (see fig. 8.12). The [Colimit](#) of the [Diagram of shape](#) is the [Sum](#) of the 2 objects in the category  $\mathbf{C}$ .

### 8.3.3 Equalizer

If choose a [Category](#) with 2 objects as the [Index category](#) (see fig. 8.13) and 2 [Morphisms](#) connecting one object with another then we can get equalizer as [Limit](#).



Figure 8.13: Index category  $I$  for equalizer. It consists of 2 objects  $a^{(I)}, b^{(I)}$ , 2 trivial (identity) morphisms  $\mathbf{1}_{a^{(I)} \rightarrow a^{(I)}}, \mathbf{1}_{b^{(I)} \rightarrow b^{(I)}}$  and 2 additional morphisms  $f^{(I)}, g^{(I)} \in \text{hom}_{\mathbf{I}}(a^{(I)}, b^{(I)})$



**Definition 8.16** (Equalizer). Lets choose a [Category](#) with 2 objects  $a^{(I)}, b^{(I)}$  and 2 additional morphisms  $f^{(I)}, g^{(I)} \in \text{hom}_{\mathbf{I}}(a^{(I)}, b^{(I)})$  as the [Index category](#)  $\mathbf{I}$  (see fig. 8.13).



Figure 8.14: Equalizer

The [Diagram of shape](#)  $F$  gives us the mapping into 2 objects and 2 morphisms in the category  $\mathbf{C}$ . The [Limit](#) of the [Diagram of shape](#) (see fig. 8.14) is the *equalizer*. The equalizer is denoted as  $eq(f, g)$ .

The meaning of the [Equalizer](#) can be described in the [Set category](#)

**Example 8.17** (Equalizer). [\[Set\]](#) In the [Set category](#) equalizer determines a solution for the following equation

$$f(x) = g(x)$$

We consider a  
TBD



# Chapter 9

## Yoneda's lemma

We will assume that the category  $\mathbf{C}$  to be a [Locally small category](#).

### 9.1 Hom functors

We are going to define the Hom functors. There are 2 hom functors: [Covariant functor](#) and [Contravariant functor](#). For the [Covariant Hom functor](#) we pick up an object  $a$  from  $\mathbf{C}$  and consider the collection of morphisms from  $a$  to an arbitrary object  $x$  from the category. The collection is a [Set](#) as soon as  $\mathbf{C}$  is a [Locally small category](#).

TBD

**Definition 9.1** (Covariant Hom functor). Let  $\mathbf{C}$  is a [Locally small category](#) and  $a \in \text{ob}(\mathbf{C})$ . Consider [Functor](#) from  $\mathbf{C}$  to the [Set category](#) defined by the following rules

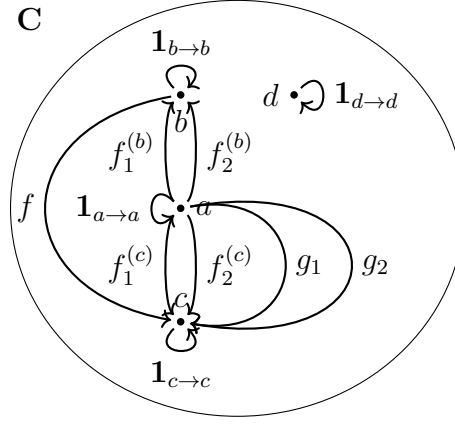
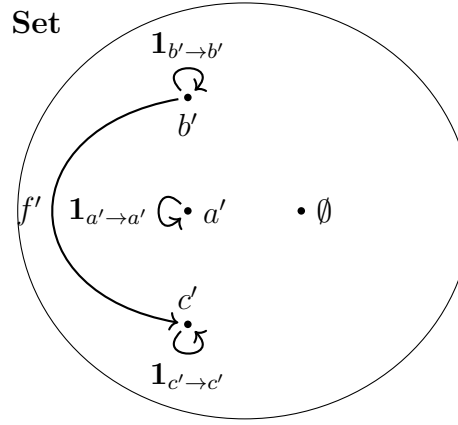
- $\forall x \in \text{ob}(\mathbf{C})$  define an object in the set category:  $\text{hom}_{\mathbf{C}}(a, x) \in \text{ob}(\mathbf{Set})$
- $\forall f : x \rightarrow y \in \text{hom}(\mathbf{C})$  define a function in the set category  $\text{hom}_{\mathbf{C}}(a, f) : \text{hom}_{\mathbf{C}}(a, x) \rightarrow \text{hom}_{\mathbf{C}}(a, y)$  as follows  $\text{hom}_{\mathbf{C}}(a, f) = \{f \circ g | g \in \text{hom}_{\mathbf{C}}(a, x)\}$ .

The *covariant Hom functor* is denoted as  $\text{Hom}_{\mathbf{C}}(a, -)$ .

**Example 9.2** (Covariant Hom functor). Consider category  $\mathbf{C}$  in the fig. [9.1](#). It consists of 4 objects:

$$\text{ob}(\mathbf{C}) = \{a, b, c, d\}.$$

We are going to construct  $\text{Hom}_{\mathbf{C}}(a, -)$  functor and therefore are interested

Figure 9.1: Covariant Hom functor  $\text{Hom}_{\mathbf{C}}(a, -)$  example. Category  $\mathbf{C}$ Figure 9.2: Covariant Hom functor  $\text{Hom}_{\mathbf{C}}(a, -)$  example. Category  $\mathbf{Set}$ 

in the following sets of morphisms:

$$\begin{aligned} \text{hom}_{\mathbf{C}}(a, a) &= \{1_{a \rightarrow a}\}, \\ \text{hom}_{\mathbf{C}}(a, b) &= \{f_1^{(b)}, f_2^{(b)}\}, \\ \text{hom}_{\mathbf{C}}(a, c) &= \{f_1^{(c)}, f_2^{(c)}, g_1, g_2\}, \\ \text{hom}_{\mathbf{C}}(a, d) &= \emptyset. \end{aligned}$$

There is also a single [Morphism](#)  $f$  between  $b$  and  $c$ .

The corresponding objects in the **Set** category is described in the fig. 9.2:

$$\begin{aligned} a' &= \text{hom}_{\mathbf{C}}(a, a) = \{\mathbf{1}_{a \rightarrow a}\}, \\ b' &= \text{hom}_{\mathbf{C}}(a, b) = \{f_1^{(b)}, f_2^{(b)}\}, \\ c' &= \text{hom}_{\mathbf{C}}(a, c) = \{f_1^{(c)}, f_2^{(c)}, g_1, g_2\}, \\ d' &= \text{hom}_{\mathbf{C}}(a, d) = \emptyset. \end{aligned}$$

The  $\text{Hom}_{\mathbf{C}}(a, -)$  does the following mapping between objects:

$$\begin{aligned} a &\Rightarrow \text{hom}_{\mathbf{C}}(a, a) = a', \\ b &\Rightarrow \text{hom}_{\mathbf{C}}(a, b) = b', \\ c &\Rightarrow \text{hom}_{\mathbf{C}}(a, c) = c', \\ d &\Rightarrow \text{hom}_{\mathbf{C}}(a, d) = \emptyset. \end{aligned}$$

The functor maps morphisms in addition to objects. There are mapping for trivial **Identity morphisms**:

$$\begin{aligned} \mathbf{1}_{a \rightarrow a} &\Rightarrow \mathbf{1}_{a' \rightarrow a'}, \\ \mathbf{1}_{b \rightarrow b} &\Rightarrow \mathbf{1}_{b' \rightarrow b'}, \\ \mathbf{1}_{c \rightarrow c} &\Rightarrow \mathbf{1}_{c' \rightarrow c'}, \\ \mathbf{1}_{d \rightarrow d} &\Rightarrow \mathbf{1}_{\emptyset \rightarrow \emptyset}, \end{aligned}$$

and for a single non trivial morphism  $f \Rightarrow f'$  that is defined by the following rules:

$$\begin{aligned} f'(f_1^{(b)}) &= g_1, \\ f'(f_2^{(b)}) &= g_2, \end{aligned}$$

i.e. the **Image** of  $f'$  is a subset of  $\text{hom}_{\mathbf{C}}(a, c)$ :

$$\text{Im } f' \subsetneq \text{hom}_{\mathbf{C}}(a, c).$$

If we revert arrows in the definition 9.1 then we can get a definition for **Contravariant functor** as follows.

**Definition 9.3** (Contravariant Hom functor). Let  $\mathbf{C}$  is a **Locally small category** and  $a \in \text{ob}(\mathbf{C})$ . Consider **Functor** from  $\mathbf{C}$  to the **Set** category defined by the following rules

- $\forall x \in \text{ob}(\mathbf{C})$  define an object in the set category:  $\text{hom}_{\mathbf{C}}(x, a) \in \text{ob}(\mathbf{Set})$
- $\forall h : x \rightarrow y \in \text{hom}(\mathbf{C})$  define a function in the set category  $\text{hom}_{\mathbf{C}}(h, a) : \text{hom}_{\mathbf{C}}(y, a) \rightarrow \text{hom}_{\mathbf{C}}(x, a)$  as follows  $\text{hom}_{\mathbf{C}}(h, a) = \{g \circ h | g \in \text{hom}_{\mathbf{C}}(y, a)\}$ .



Figure 9.3: Contravariant Hom functor  $\text{Hom}_D(-, a)$  example. Category **D**

The *contravariant Hom functor* is denoted as  $\text{Hom}_C(-, a)$ .

**Example 9.4** (Contravariant Hom functor). Consider category **D** in the fig. 9.3. It is similar to the category **C** from example 9.2 and has the same set of objects and morphisms but all morphisms are reverted i.e.  $D = \mathbf{C}^{op}$ . Therefore the category consists of 4 objects:

$$\text{ob}(\mathbf{D}) = \{a, b, c, d\}.$$

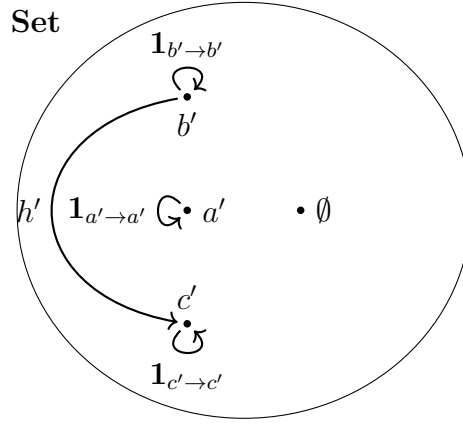
We are going to construct  $\text{Hom}_D(-, a)$  functor and therefore are interested in the following sets of morphisms:

$$\begin{aligned} \text{hom}_{\mathbf{D}}(a, a) &= \{1_{a \rightarrow a}\}, \\ \text{hom}_{\mathbf{D}}(b, a) &= \{h_1^{(b)}, h_2^{(b)}\}, \\ \text{hom}_{\mathbf{D}}(c, a) &= \{h_1^{(c)}, h_2^{(c)}, g_1, g_2\}, \\ \text{hom}_{\mathbf{D}}(d, a) &= \emptyset. \end{aligned}$$

There is also a single [Morphism](#)  $h$  between  $c$  and  $b$ .

The corresponding objects in the [Set category](#) is described in the fig. 9.4:

$$\begin{aligned} a' &= \text{hom}_{\mathbf{D}}(a, a) = \{1_{a \rightarrow a}\}, \\ b' &= \text{hom}_{\mathbf{D}}(b, a) = \{f_1^{(b)}, f_2^{(b)}\}, \\ c' &= \text{hom}_{\mathbf{D}}(c, a) = \{f_1^{(c)}, f_2^{(c)}, g_1, g_2\}, \\ d' &= \text{hom}_{\mathbf{D}}(d, a) = \emptyset. \end{aligned}$$

Figure 9.4: Contravariant Hom functor  $\text{Hom}_D(-, a)$  example. Category **Set**

The  $\text{Hom}_D(-, a)$  does the following mapping between objects:

$$\begin{aligned} a &\Rightarrow \text{hom}_{\mathbf{D}}(a, a) = a', \\ b &\Rightarrow \text{hom}_{\mathbf{D}}(b, a) = b', \\ c &\Rightarrow \text{hom}_{\mathbf{D}}(c, a) = c', \\ d &\Rightarrow \text{hom}_{\mathbf{D}}(d, a) = \emptyset. \end{aligned}$$

The functor maps morphisms in addition to objects. There are mapping for trivial **Identity morphisms**:

$$\begin{aligned} 1_{a \rightarrow a} &\Rightarrow 1_{a' \rightarrow a'}, \\ 1_{b \rightarrow b} &\Rightarrow 1_{b' \rightarrow b'}, \\ 1_{c \rightarrow c} &\Rightarrow 1_{c' \rightarrow c'}, \\ 1_{d \rightarrow d} &\Rightarrow 1_{\emptyset \rightarrow \emptyset}, \end{aligned}$$

and for a single non trivial morphism  $h \Rightarrow h'$  that is defined by the following rules:

$$\begin{aligned} h'(h_1^{(b)}) &= g_1, \\ h'(h_2^{(b)}) &= g_2, \end{aligned}$$

i.e. the **Image** of  $h'$  is a subset of  $\text{hom}_{\mathbf{D}}(c, a)$ :

$$\text{Im } h' \subsetneq \text{hom}_{\mathbf{D}}(c, a).$$

## 9.2 Yoneda's lemma

TBD

## 9.3 Examples

### 9.3.1 Quantum mechanics

Flori interpretation of quantum mechanics

TBD



# Index

- C** ++ category, 7, 40
  - definition, 23
- C** ++ toy category
  - example, 25
- Cat** category, 7
  - definition, 49
- FdHilb** category, 7, 41
  - definition, 28
- Fun** category, 7, 57, 69
  - definition, 57
- Hask** category, 7, 38, 39, 41
  - definition, 23
- Hask** toy category
  - example, 24
- Hask** toy category example, 25, 26
- Proof** category, 8, 45
  - definition, 45
- Rel** category, 8, 28, 77
  - definition, 27
- Scala** category, 8, 41
  - definition, 23
- Scala** toy category
  - example, 26
- Set** category, 8, 18, 27, 32, 34–36, 62, 65, 67, 77, 89, 91, 93, 94
  - definition, 19
- Ad-hoc polymorphism
  - definition, 59
- Associativity axiom, 17, 31, 55
  - declaration, 15
- Associator, 67
  - definition, 67
- Bifunctor, 51, 52, 57, 58, 65, 67, 68
  - Set** example, 51
  - definition, 51
- Bifunctor example, 65
- Bifunctor in the category of functors
  - remark, 57
- Bifunctor in the category of functors remark, 70
- Bijection
  - definition, 20
- Binary relation, 18, 27, 29
  - definition, 18
- Cartesian closed category, 37, 39, 45
  - definition, 37
- Cartesian closed category theorem
  - declaration, 37
- Cartesian product, 29, 34, 51, 65
  - definition, 19
- Categorical approach, 31, 61
  - definition, 19
- Category, 7, 11, 17, 23–26, 31, 32, 49, 50, 66, 88, 89

- Fun** example, 57
- Set**, 19
  - definition, 16
  - dual, 17
  - large, 18, 19
  - locally small, 18
  - opposite, 17
  - small, 17
- Category of co-cones
  - remark, 84
- Category of co-cones from  $F$ , 8, 84
  - definition, 84
- Category of cones to  $F$ , 8, 82
  - definition, 81
  - remark, 82
- Category of endofunctors
  - definition, 69
- Category Product, 51
  - definition, 50
- Class, 13, 15, 17
  - definition, 13
- Class of Morphisms
  - remark, 15
- Class of Objects
  - remark, 13
- Cocone, 7, 84, 86
  - definition, 83
- Codomain, 8, 19, 20
  - definition, 14
  - example, 19
- Colimit, 84–86, 88
  - definition, 83
  - Initial object example, 86
  - Sum example, 87
- Commutative diagram, 32, 33, 37, 54, 55
  - definition, 15
- Composition
  - opposite category, 17
  - remark, 15
- Composition axiom, 15, 17, 24, 31, 50, 55
  - declaration, 14
- Cone, 7, 81, 84–86
  - definition, 80
- Conjunction, 45
  - definition, 44
- Constant functor, 8, 57, 84
  - Hask** example, 50
  - definition, 50
- Contravariant functor, 50, 51, 91, 93
  - Hask** example, 50
  - definition, 50
- Contravariant Hom functor, 7
  - definition, 93
  - example, 94
- Covariant functor, 50, 91
  - definition, 50
- Covariant Hom functor, 7, 91
  - definition, 91
  - example, 91
- Diagram of shape, 80–84, 87–89
  - definition, 79
- Dirac notation
  - example, 28
- Direct sum of Hilber spaces, 29, 41
  - definition, 28
- Discrete category, 23, 85–87
  - definition, 23
- Disjoint union, 35
  - definition, 34
- Disjunction, 45
  - definition, 44
- Distributive category, 36, 37, 39, 45
  - Hask** example, 39
  - definition, 36
  - example, 36
- Domain, 8, 19, 20, 51

- definition, [14](#)
  - example, [19](#)
- Dual space, [28](#), [69](#)
  - definition, [27](#)
- Empty category, [50](#), [85](#), [86](#)
  - definition, [18](#)
- Endofunctor, [49](#), [69](#), [70](#), [73](#)
  - definition, [49](#)
- Epimorphism, [20](#)
  - definition, [16](#)
- Equalizer, [8](#), [89](#)
  - Set** example, [89](#)
  - definition, [89](#)
- Exponential, [8](#), [37](#), [39](#), [45](#)
  - Hask** example, [39](#)
  - Set** example, [37](#)
  - definition, [37](#)
- False, [45](#)
  - definition, [43](#)
- Field, [66](#)
- Function, [19](#), [27](#), [29](#), [31](#), [51](#)
  - definition, [18](#)
- Functor, [8](#), [9](#), [11](#), [48–51](#), [53](#), [55](#),  
[58](#), [65](#), [79](#), [91](#), [93](#)
  - C++** example, [48](#)
  - Hask** example, [48](#)
  - Scala** example, [48](#)
  - definition, [47](#)
  - remark, [48](#)
- Functor composition, [8](#), [49](#)
  - definition, [49](#)
- Functor example, [71](#), [72](#)
- Group, [66](#)
- Haskell lazy evaluation
  - remark, [25](#)
- Haskell lazy evaluation remark,  
[23](#), [38](#)
- Hilbert space, [8](#), [28](#), [29](#), [67](#), [68](#)
  - definition, [27](#)
- Hilbert-Schmidt correspondence
  - remark, [68](#)
- Hom set
  - example, [36](#)
- Hom set example, [36](#), [37](#)
- homset, [18](#)
- Horizontal composition, [7](#), [58](#), [70](#)
  - definition, [57](#)
- Identity functor, [8](#), [49](#), [69](#)
  - definition, [49](#)
  - remark, [49](#)
- Identity is unique theorem, [15](#)
  - declaration, [32](#)
- Identity morphism, [8](#), [15](#), [17–19](#),  
[23–26](#), [31](#), [32](#), [49](#), [58](#), [93](#),  
[95](#)
  - definition, [15](#)
- Identity natural transformation, [8](#),  
[58](#), [69](#)
  - definition, [58](#)
- Image, [8](#), [93](#), [95](#)
  - definition, [19](#)
- Implication, [9](#), [45](#)
  - definition, [44](#)
- Index category, [85–89](#)
  - definition, [79](#)
- Initial object, [29](#), [32](#), [33](#), [36](#), [38](#),  
[41](#), [45](#), [50](#), [57](#), [84–86](#)
  - C++** example, [39](#)
  - Cat** example, [50](#)
  - FdHilb** example, [41](#)
  - Fun** example, [57](#)
  - Hask** example, [38](#)
  - Proof** example, [45](#)
  - Scala** example, [41](#)
  - Set** example, [32](#)
  - Colimit example, [86](#)
  - definition, [32](#)
- Initial object example, [41](#)

- Initial object is unique theorem
  - declaration, 32
- Injection, 20
  - definition, 20
- Injection vs Monomorphism
  - remark, 20
- Isomorphism, 8, 16, 31–33, 62, 64, 66, 67
  - definition, 16
  - remark, 16
- Kleisli category, 7, 73, 74
  - definition, 73
- Kleisli category composition
  - remark, 73
- Large category, 19
  - definition, 18
- Left unitor, 67
  - definition, 67
- Left whiskering, 7, 58, 69
  - definition, 58
- Limit, 82, 85–89
  - definition, 80
  - Product example, 86
  - Terminal object example, 85
- Linear map, 28, 29, 68, 69
  - definition, 28
  - remark, 28
- List monad
  - Hask** example, 77
- Locally small category, 91, 93
  - definition, 18
- Maybe as a bifunctor
  - Hask** example, 51
- Maybe monad
  - C + +** example, 76
  - Hask** example, 75
- Modus ponens
  - definition, 44
- Monad, 8, 11, 71–73
  - Hask** example, 71
  - Scala** example, 72
  - definition, 69
- Monad term
  - remark, 70
- Monoid, 36, 61, 66, 67, 70
  - definition, 35, 65
- Monoidal product
  - definition, 67
- Monoidal category, 67
  - definition, 67
- Monoidal product
  - remark, 67
- Monomorphism, 9, 20
  - definition, 16
- Morphism, 8, 9, 11, 14–19, 23–29, 31, 34, 37, 45, 48–50, 53, 55, 57, 58, 61, 65, 68, 73, 80, 83, 84, 88, 92, 94
  - C + +** example, 25
  - FdHilb** category, 28
  - Fun** example, 57
  - Hask** example, 24
  - Rel** category, 27
  - Scala** example, 26
  - Set** category, 19
  - definition, 14
  - remark, 14
- Morphisms equality
  - definition, 31
- Morphisms of cones
  - definition, 81
- Natural transformation, 7, 11, 55, 57–59, 69, 70, 73, 84, 85
  - definition, 54
  - Horizontal composition, 57
  - Vertical composition, 55
- Object, 8, 11, 13–19, 23–29, 31, 32, 34–36, 40, 45, 48–50, 53, 55, 57, 58, 67

- C** ++ example, 25
- FdHilb** category, 28
- Fun** example, 57
- Hask** example, 24
- Rel** category, 27
- Scala** example, 26
- Set** category, 19
  - definition, 13
- Objects equality, 32, 33
  - definition, 31
- Opposite category, 7, 17, 32, 33
  - definition, 17
- Parametric polymorphism, 59
- Parametrically polymorphic
  - function, 59
  - Hask** example, 59
  - definition, 59
- Product, 8, 29, 34–38, 40, 41, 45, 85–87
  - C** ++ example, 40
  - FdHilb** example, 41
  - Hask** example, 38
  - Proof** example, 45
  - Set** example, 34
  - definition, 34
  - Limit example, 86
- Product example, 62
- Product of morphisms, 51, 62, 64, 65
  - definition, 35
- Profunctor
  - Hask** example, 52
  - definition, 52
- Proof, 45
  - definition, 44
- Proposition, 44, 45
  - definition, 43
  - example, 43
- Pure function, 23, 75
  - definition, 23
- Rabi oscillations
  - example, 29
- Reynolds theorem, 59
  - declaration, 59
- Right unitor, 67
  - definition, 67
- Right whiskering, 9, 58, 69
  - definition, 58
- Ring, 66
- Set, 14, 18, 19, 23, 27, 29, 31, 91
  - definition, 18
- Set of morphisms, 7
  - definition, 17
- Set vs Category
  - remark, 19
- Set vs Category remark, 31
- Singleton, 27, 29, 32, 64, 65
  - definition, 19
- Small category, 18, 49
  - definition, 17
- Strict monoidal category
  - definition, 67
- Sum, 8, 29, 35–37, 39–41, 45, 86, 88
  - C** ++ example, 40
  - FdHilb** example, 41
  - Hask** example, 39
  - Proof** example, 45
  - Set** example, 35
  - Colimit example, 87
  - definition, 34
- Sum example, 29
- Surjection, 20
  - definition, 20
- Surjection vs Epimorphism
  - remark, 20
- Tensor product, 67, 68
  - definition, 68
  - remark, 68

- Terminal object, [29](#), [32](#), [33](#), [36](#), [37](#),  
[41](#), [45](#), [50](#), [57](#), [82](#), [85](#), [86](#)
  - C** ++ example, [40](#)
  - Cat** example, [50](#)
  - FdHilb** example, [41](#)
  - Fun** example, [57](#)
  - Hask** example, [38](#)
  - Proof** example, [45](#)
  - Scala** example, [41](#)
  - Set** example, [32](#)
  - definition, [32](#)
  - Limit example, [85](#)
- Terminal object example, [41](#)
- Terminal object is unique theorem
  - declaration, [33](#)
- Toy example
  - example, [33](#)
- Trivial category, [50](#)
  - definition, [18](#)
- True, [45](#)
  - definition, [43](#)
- Type algebra
  - example, [39](#)
- Vertical composition, [7](#), [55](#), [57](#), [69](#)
  - definition, [55](#)
- Whiskering
  - remark, [58](#)

# Bibliography

- [1] Coecke, B. Introducing categories to the practicing physicist / Bob Coecke. — 2008. — <https://arxiv.org/abs/0808.1032>.
- [2] Di Cosmo, R. Linear logic. — 2016. — <https://plato.stanford.edu/archives/win2016/entries/logic-linear/>.
- [3] Gonzalez, G. Scalable program architectures. — 2014. — <http://www.haskellforall.com/2014/04/scalable-program-architectures.html>.
- [4] ([https://math.stackexchange.com/users/142355/david\\_myers](https://math.stackexchange.com/users/142355/david_myers)), D. M. How should i think about morphism equality? — Mathematics Stack Exchange. — URL:<https://math.stackexchange.com/q/1346167> (version: 2015-07-01). <https://math.stackexchange.com/q/1346167>.
- [5] ([https://math.stackexchange.com/users/232/qiaochu\\_yuan](https://math.stackexchange.com/users/232/qiaochu_yuan)), Q. Y. Is the identity functor the terminal object of the category of endofunctors on  $\mathcal{C}$ ? — Mathematics Stack Exchange. — URL:<https://math.stackexchange.com/q/6318> (version: 2010-10-08). <https://math.stackexchange.com/q/6318>.
- [6] (<https://math.stackexchange.com/users/455216/enkidu>), E. Terminal and initial objects in the category of functors. — Mathematics Stack Exchange. — URL:<https://math.stackexchange.com/q/3033845> (version: 2018-12-10). <https://math.stackexchange.com/q/3033845>.
- [7] Ivan Murashko Alexey Radkov, M. C++ examples for category theory by example book. — 2018. — <https://github.com/CatTheoryByExample/cpp-examples>.
- [8] Ivan Murashko Alexey Radkov, M. Haskell examples for category theory by example book. — 2018. — <https://github.com/CatTheoryByExample/hs-examples>.

- [9] Ivan Murashko Alexey Radkov, M. Scala examples for category theory by example book. — 2018. — <https://github.com/CatTheoryByExample/scala-examples>.
- [10] Lane, S. Categories for the Working Mathematician / S.M. Lane. Graduate Texts in Mathematics. — Springer New York, 1998. — <https://books.google.ru/books?id=eBvhyc4z8HQC>.
- [11] Leighton, T. What is a proof? — <http://web.mit.edu/neboat/Public/6.042/proofs.pdf>.
- [12] Milewski, B. Category Theory for Programmers / B. Milewski. — Bartosz Milewski, 2018. — <https://github.com/hmemcpy/milewski-ctfp-pdf/releases/download/v0.7.0/category-theory-for-programmers.pdf>.
- [13] Moggi, E. Notions of computation and monads / Eugenio Moggi // Inf. Comput. — 1991. — Vol. 93, no. 1. — P. 55–92. — <http://fsl.cs.illinois.edu/pubs/moggi-1991-ic.pdf>.
- [14] Murashko, I. Lectures notes in introduction to galois theory by eka-  
terina amerik. — 2016-2019. — <https://github.com/ivanmurashko/courseragalois>.
- [15] Murashko, I. Category theory by example. — 2018. — <https://github.com/ivanmurashko/articles/tree/master/cattheory>.
- [16] nLab authors. whiskering. — <http://ncatlab.org/nlab/show/whiskering>. — 2018. — sep. — Revision 11.
- [17] ProofWiki. Empty mapping is unique / ProofWiki. — 2018. — [https://proofwiki.org/wiki/Empty\\_Mapping\\_is\\_Unique](https://proofwiki.org/wiki/Empty_Mapping_is_Unique).
- [18] ProofWiki. Injection iff monomorphism in category of sets / ProofWiki. — 2018. — [https://proofwiki.org/wiki/Injection\\_iff\\_Monomorphism\\_in\\_Category\\_of\\_Sets](https://proofwiki.org/wiki/Injection_iff_Monomorphism_in_Category_of_Sets).
- [19] ProofWiki. Surjection iff epimorphism in category of sets / ProofWiki. — 2018. — [https://proofwiki.org/wiki/Surjection\\_iff\\_Epimorphism\\_in\\_Category\\_of\\_Sets](https://proofwiki.org/wiki/Surjection_iff_Epimorphism_in_Category_of_Sets).
- [20] Quist, M. Writing and classifying true, false and open statements in math. — <https://study.com/academy/lesson/writing-and-classifying-true-false-and-open-statements-in-math.html>.



- [21] user84563 (<https://math.stackexchange.com/users/301130/user84563>). Does an(/the?) empty category exist? — Mathematics Stack Exchange. — URL:<https://math.stackexchange.com/q/1997015> (version: 2016-11-03). <https://math.stackexchange.com/q/1997015>.
- [22] Wikipedia. Disjoint union — wikipedia, the free encyclopedia. — 2017. — [Online; accessed 13-April-2017]. [https://en.wikipedia.org/w/index.php?title=Disjoint\\_union&oldid=774047863](https://en.wikipedia.org/w/index.php?title=Disjoint_union&oldid=774047863).
- [23] Wikipedia contributors. Cone (category theory) — Wikipedia, the free encyclopedia. — 2018. — [Online; accessed 10-November-2018]. [https://en.wikipedia.org/w/index.php?title=Cone\\_\(category\\_theory\)&oldid=827711162](https://en.wikipedia.org/w/index.php?title=Cone_(category_theory)&oldid=827711162).
- [24] Wikipedia contributors. Distributive category — Wikipedia, the free encyclopedia. — 2018. — [Online; accessed 14-October-2018]. [https://en.wikipedia.org/w/index.php?title=Distributive\\_category&oldid=851490594](https://en.wikipedia.org/w/index.php?title=Distributive_category&oldid=851490594).
- [25] Wikipedia contributors. Russell's paradox — Wikipedia, the free encyclopedia. — 2018. — [Online; accessed 29-July-2018]. [https://en.wikipedia.org/w/index.php?title=Russell%27s\\_paradox&oldid=852430810](https://en.wikipedia.org/w/index.php?title=Russell%27s_paradox&oldid=852430810).
- [26] Wikipedia contributors. Zermelo–fraenkel set theory — Wikipedia, the free encyclopedia. — 2018. — [Online; accessed 29-July-2018]. [https://en.wikipedia.org/w/index.php?title=Zermelo%E2%80%9393Fraenkel\\_set\\_theory&oldid=852467638](https://en.wikipedia.org/w/index.php?title=Zermelo%E2%80%9393Fraenkel_set_theory&oldid=852467638).
- [27] Мурашко И. В. Квантовая оптика / Мурашко И. В. — 2018. — <https://github.com/ivanmurashko/lectures/blob/master/pdfs/qo.pdf>.



# Creative Commons Legal Code

## Attribution-NonCommercial 3.0 Unported

CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM AND DOES NOT PROVIDE LEGAL SERVICES. DISTRIBUTION OF THIS LICENSE DOES NOT CREATE AN ATTORNEY-CLIENT RELATIONSHIP. CREATIVE COMMONS PROVIDES THIS INFORMATION ON AN “AS-IS” BASIS. CREATIVE COMMONS MAKES NO WARRANTIES REGARDING THE INFORMATION PROVIDED, AND DISCLAIMS LIABILITY FOR DAMAGES RESULTING FROM ITS USE.

### *License*

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE (“CCPL” OR “LICENSE”). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

#### **1. Definitions**

- (a) **“Adaptation”** means a work based upon the Work, or upon the Work and other pre-existing works, such as a translation, derivative work, arrangement of music or other alterations of a literary or artistic work, or phonogram or performance and includes cinematographic adaptations or any other form in which the Work may be recast,

transformed, or adapted including in any form recognizably derived from the original, except that a work that constitutes a Collection will not be considered an Adaptation for the purpose of this License. For the avoidance of doubt, where the Work is a musical work, performance or phonogram, the synchronization of the Work in timed-relation with a moving image (“synching”) will be considered an Adaptation for the purpose of this License.

- (b) **“Collection”** means a collection of literary or artistic works, such as encyclopedias and anthologies, or performances, phonograms or broadcasts, or other works or subject matter other than works listed in Section 1(f) below, which, by reason of the selection and arrangement of their contents, constitute intellectual creations, in which the Work is included in its entirety in unmodified form along with one or more other contributions, each constituting separate and independent works in themselves, which together are assembled into a collective whole. A work that constitutes a Collection will not be considered an Adaptation (as defined above) for the purposes of this License.
- (c) **“Distribute”** means to make available to the public the original and copies of the Work or Adaptation, as appropriate, through sale or other transfer of ownership.
- (d) **“Licensor”** means the individual, individuals, entity or entities that offer(s) the Work under the terms of this License.
- (e) **“Original Author”** means, in the case of a literary or artistic work, the individual, individuals, entity or entities who created the Work or if no individual or entity can be identified, the publisher; and in addition
  - (i) in the case of a performance the actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret or otherwise perform literary or artistic works or expressions of folklore;
  - (ii) in the case of a phonogram the producer being the person or legal entity who first fixes the sounds of a performance or other sounds; and,
  - (iii) in the case of broadcasts, the organization that transmits the broadcast.
- (f) **“Work”** means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same

nature; a dramatic or dramatico-musical work; a choreographic work or entertainment in dumb show; a musical composition with or without words; a cinematographic work to which are assimilated works expressed by a process analogous to cinematography; a work of drawing, painting, architecture, sculpture, engraving or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.

- (g) **“You”** means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.
- (h) **“Publicly Perform”** means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place individually chosen by them; to perform the Work to the public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.
- (i) **“Reproduce”** means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

**2. Fair Dealing Rights.** Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

**3. License Grant.** Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpet-

ual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

- (a) to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections;
- (b) to create and Reproduce Adaptations provided that any such Adaptation, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original Work. For example, a translation could be marked “The original work was translated from English to Spanish,” or a modification could indicate “The original work has been modified.”;
- (c) to Distribute and Publicly Perform the Work including as incorporated in Collections; and,
- (d) to Distribute and Publicly Perform Adaptations.

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. Subject to Section 8(f), all rights not expressly granted by Licensor are hereby reserved, including but not limited to the rights set forth in Section 4(d).

**4. Restrictions.** The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

- (a) You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be made subject to the terms of this License. If You create a Collection,

upon notice from any Licensor You must, to the extent practicable, remove from the Collection any credit as required by Section 4(c), as requested. If You create an Adaptation, upon notice from any Licensor You must, to the extent practicable, remove from the Adaptation any credit as required by Section 4(c), as requested.

- (b) You may not exercise any of the rights granted to You in Section 3 above in any manner that is primarily intended for or directed toward commercial advantage or private monetary compensation. The exchange of the Work for other copyrighted works by means of digital file-sharing or otherwise shall not be considered to be intended for or directed toward commercial advantage or private monetary compensation, provided there is no payment of any monetary compensation in connection with the exchange of copyrighted works.
- (c) If You Distribute, or Publicly Perform the Work or any Adaptations or Collections, You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or Licensor designate another party or parties (e.g., a sponsor institute, publishing entity, journal) for attribution (“Attribution Parties”) in Licensor’s copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and, (iv) consistent with Section 3(b), in the case of an Adaptation, a credit identifying the use of the Work in the Adaptation (e.g., “French translation of the Work by Original Author,” or “Screenplay based on original Work by Original Author”). The credit required by this Section 4(c) may be implemented in any reasonable manner; provided, however, that in the case of a Adaptation or Collection, at a minimum such credit will appear, if a credit for all contributing authors of the Adaptation or Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original

Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

(d) For the avoidance of doubt:

- i. **Non-waivable Compulsory License Schemes.** In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;
- ii. **Waivable Compulsory License Schemes.** In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License if Your exercise of such rights is for a purpose or use which is otherwise than non-commercial as permitted under Section 4(b) and otherwise waives the right to collect royalties through any statutory or compulsory licensing scheme; and,
- iii. **Voluntary License Schemes.** The Licensor reserves the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License that is for a purpose or use which is otherwise than noncommercial as permitted under Section 4(c).

(e) Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Adaptations or Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author's honor or reputation. Licensor agrees that in those jurisdictions (e.g. Japan), in which any exercise of the right granted in Section 3(b) of this License (the right to make Adaptations) would be deemed to be a distortion, mutilation, modification or other derogatory action prejudicial to the Original Author's honor and reputation, the Licensor will waive or not assert, as appropriate, this Section, to the fullest extent permitted by the applicable national law, to enable You



to reasonably exercise Your right under Section 3(b) of this License (right to make Adaptations) but not otherwise.

### **5. Representations, Warranties and Disclaimer**

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

**6. Limitation on Liability.** EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

### **7. Termination**

- (a) This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Adaptations or Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.
- (b) Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

### **8. Miscellaneous**

- (a) Each time You Distribute or Publicly Perform the Work or a Collection, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.
- (b) Each time You Distribute or Publicly Perform an Adaptation, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.
- (c) If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.
- (d) No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.
- (e) This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.
- (f) The rights granted under, and the subject matter referenced, in this License were drafted utilizing the terminology of the Berne Convention for the Protection of Literary and Artistic Works (as amended on September 28, 1979), the Rome Convention of 1961, the WIPO Copyright Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996 and the Universal Copyright Convention (as revised on July 24, 1971). These rights and subject matter take effect in the relevant jurisdiction in which the License terms are sought to be enforced according to the corresponding provisions of the implementation of those treaty provisions in the applicable national law. If the standard suite of rights granted under applicable copyright law includes additional rights not granted under this License, such additional rights are deemed to be included in the License; this License is not intended to restrict the license of any rights under applicable law.

## Creative Commons Notice

Creative Commons is not a party to this License, and makes no warranty whatsoever in connection with the Work. Creative Commons will not be liable to You or any party on any legal theory for any damages whatsoever, including without limitation any general, special, incidental or consequential damages arising in connection to this license. Notwithstanding the foregoing two (2) sentences, if Creative Commons has expressly identified itself as the Licensor hereunder, it shall have all rights and obligations of Licensor.

Except for the limited purpose of indicating to the public that the Work is licensed under the CCPL, Creative Commons does not authorize the use by either party of the trademark “Creative Commons” or any related trademark or logo of Creative Commons without the prior written consent of Creative Commons. Any permitted use will be in compliance with Creative Commons’ then-current trademark usage guidelines, as may be published on its website or otherwise made available upon request from time to time. For the avoidance of doubt, this trademark restriction does not form part of the License.

Creative Commons may be contacted at <https://creativecommons.org/>.