

# Category Theory by Example

Ivan Murashko

September 16, 2018



# Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Base definitions</b>                                  | <b>9</b>  |
| 1.1      | Definitions . . . . .                                    | 9         |
| 1.1.1    | Object . . . . .   | 9         |
| 1.1.2    | Morphism . . . . .                                       | 10        |
| 1.1.3    | Category . . . . .                                       | 12        |
| 1.2      | Examples . . . . .                                       | 14        |
| 1.2.1    | <b>Set</b> category . . . . .                            | 14        |
| 1.2.2    | Programming languages . . . . .                          | 18        |
| 1.2.3    | Quantum mechanics . . . . .                              | 22        |
| <b>2</b> | <b>Objects and morphisms</b>                             | <b>27</b> |
| 2.1      | Equality . . . . .                                       | 27        |
| 2.2      | Initial and terminal objects . . . . .                   | 28        |
| 2.3      | Product and sum . . . . .                                | 29        |
| 2.4      | Category as a monoid . . . . .                           | 31        |
| 2.5      | Exponential . . . . .                                    | 31        |
| 2.6      | Programming languages and algebraic data types . . . . . | 32        |
| 2.7      | Examples . . . . .                                       | 32        |
| 2.7.1    | <b>Set</b> category . . . . .                            | 32        |
| 2.7.2    | Programming languages . . . . .                          | 32        |
| 2.7.3    | Quantum mechanics . . . . .                              | 35        |
| <b>3</b> | <b>Functors</b>  | <b>37</b> |
| 3.1      | Definitions . . . . .                                    | 37        |
| 3.2      | Curry-Howard-Lambek correspondence . . . . .             | 39        |
| 3.3      | Examples . . . . .                                       | 39        |
| 3.3.1    | <b>Set</b> category . . . . .                            | 39        |
| 3.3.2    | Programming languages . . . . .                          | 39        |
| 3.3.3    | Quantum mechanics . . . . .                              | 41        |

|          |   |           |
|----------|---|-----------|
| <b>4</b> | <b>Natural transformation</b>                               | <b>43</b> |
| 4.1      | Definitions . . . . .                                       | 43        |
| 4.2      | Operations with natural transformations . . . . .           | 45        |
| 4.3      | Polymorphism and natural transformation . . . . .           | 48        |
| 4.3.1    | <b>Hask</b> category . . . . .                              | 48        |
| <b>5</b> | <b>Monads</b>   | <b>51</b> |
| 5.1      | Monoid in <b>Set</b> category . . . . .                     | 51        |
| 5.2      | Monoidal category . . . . .                                 | 55        |
| 5.3      | Category of endofunctors . . . . .                          | 56        |
| 5.4      | Kleisli category. Monads in programming languages . . . . . | 57        |
| 5.4.1    | Programming languages . . . . .                             | 58        |
| 5.5      | Examples . . . . .  | 58        |
| 5.5.1    | Programming languages . . . . .                             | 58        |
| 5.5.2    | Quantum mechanics . . . . .                                 | 60        |
| <b>6</b> | <b>Yoneda's lemma</b>                                       | <b>61</b> |
| 6.1      | Examples . . . . .  | 61        |
| 6.1.1    | Quantum mechanics . . . . .                                 | 61        |
|          | <b>Index</b>  | <b>63</b> |

# Notations

$[\mathbf{C}, \mathbf{D}]$  **Fun** category (Example 4.2)

$\alpha \circ \beta$  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)

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

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

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

$\text{cod } f$  Codomain

$\text{dom } f$  Domain

$\text{hom}(a, b)$  set of Morphisms between  $a$  and  $b$

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

$1_{\mathbf{C} \Rightarrow \mathbf{C}}$  Identity functor

$1_{a \rightarrow a}$  Identity morphism

$1_{F \rightrightarrows F}$  Identity natural transformation

$\langle M, \mu, \eta \rangle$  Monad

$\mathcal{H}_n$  finite dimensional Hilbert space

$a \cong_f b$  there is an Isomorphism  $f$  between  $a$  and  $b$

$a, b$     **Objects** (Latin small letters)

$F \circ G$  **Functor composition** (circle dot)

$f \circ g$    Morphism composition (circle dot)

$F, G$    **Functor** (capital Latin letter)

$f, g, h$  **Morphism** (Latin small letter)

$F : \mathbf{C} \Rightarrow \mathbf{D}$  **Functor** (double arrow)

$f : a \rightarrow b$  **Morphism** (simple arrow)

$H\alpha$     **Right whiskering**

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 I wrote the book to collect all of them in one place to be easy checked and updated in future when I decide to refresh my knowledge about the field of math. Therefore the book was written mostly for my category theory studying purposes but I 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 and Curry-Howard-Lambek correspondence.

There are a lot of examples in each chapter. The examples cover different category theory application areas. I 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 repository [\[7\]](#).

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





# 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*.

**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  $\mathbf{C}$  consists of

- **Class** of **Objects**  $\text{ob}(\mathbf{C})$



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 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 **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**

**Definition 1.23** (Large category). A category  $\mathbf{C}$  is not **Small category** then it is called *large*. The example of large category is **Set category**



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  $\text{hom}(C^{op}) = \{f^{op}, g^{op}, h^{op} = g^{op} \circ f^{op}, 1_{a \rightarrow a}, 1_{b \rightarrow b}, 1_{c \rightarrow c}, 1_{d \rightarrow d}\}$

## 1.2 Examples

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

### 1.2.1 Set category

**Definition 1.24** (Set). Set is a collection of distinct object. The objects are called the elements of the set.

**Definition 1.25** (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 1.26** (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 1.27** (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 1.28** (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  $\forall x \in X : 1_{X \rightarrow X}(x) = x$ .



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

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 [13] can be applied. To avoid such situations we consider a limitation that is applied on our construction, for instance ZFC [14]. 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 1.29** (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” [5]

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” [5] there.

**Definition 1.30** (Singleton). The *singleton* is a [Set](#) with only one element.

**Example 1.31** (Domain). Given a function  $f : X \rightarrow Y$ , the set  $X$  is the domain. I.e.  $\text{dom } f = X$

**Example 1.32** (Codomain). Given a function  $f : X \rightarrow Y$ , the set  $Y$  is the codomain. I.e.  $\text{cod } f = Y$

**Definition 1.33** (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 1.34** (Surjection vs Epimorphism). [Surjection](#) and [Epimorphism](#) are related each other. Consider a non-surjective function  $f : X \rightarrow Y' \subset Y$

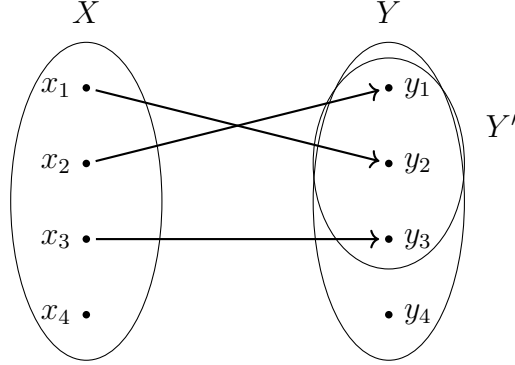


Figure 1.6: A non-surjective function  $f$  from domain  $X$  to codomain  $Y' \subset Y$ .  $\exists g_1 : Y' \rightarrow Y', g_2 : Y \rightarrow Y$  such that  $g_1(Y') = g_2(Y')$ , but as soon as  $Y' \neq Y$  we have  $g_1 \neq g_2$ . Using the fact that  $Y'$  is codomain of  $f$  we got  $g_1 \circ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

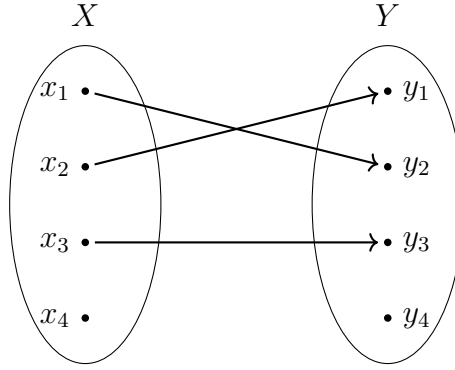


Figure 1.7: A injective (non-surjective) function from domain  $X$  to codomain  $Y$

(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 [11] of that fact.

**Definition 1.35** (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).





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)

**Remark 1.36** (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 [10] of that fact.

**Definition 1.37** (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).

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

### 1.2.2 Programming languages

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 1.44](#))).

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

**Definition 1.40** (Pure function). The function is pure if it's execution give the same results independently from the environment.

**Definition 1.41** (**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 1.42** (**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

### Hask toy category

**Example 1.43** (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 1.44** (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.

### C++ toy category

**Example 1.45** (C++ toy category). We will use the same trick as in [Hask toy category](#) ([Example 1.43](#)) 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).

### Scala toy category

**Example 1.46** (Scala toy category). We will use the same trick as in [Hask toy category](#) ([Example 1.43](#)) 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)
}
}

```

### 1.2.3 Quantum mechanics

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

**Definition 1.47** (**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 1.48** (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 Hilber space of dimensional  $n$  as  $\mathcal{H}_n$ . Obviously  $\mathcal{H}_1 = \mathbb{C}$ .

**Definition 1.49** (Dual space). Each Hilber space  $\mathcal{H}$  has an associated with it dual space  $\mathcal{H}^*$  that consists of linear functionals

**Example 1.50** (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 1.51** (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 1.52** (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 1.53** (Direct sum of Hilbert spaces). Let  $\mathcal{A}, \mathcal{B}$  are 2 Hilbert 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 1.54** (**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](#).

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

|                 | <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 2.18) | Sum (Example 2.18) | Direct sum of Hilber spaces            |

**Example 1.55** (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 [15]:

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

where  $\omega_R$  - Rabi frequency [15].

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 similar approach as in **Set vs Category** (**Remark 1.29**) : 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 [2]. 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  $\forall x \in \text{ob}(\mathbf{C}) \exists! f_x : i \rightarrow x \in \text{hom}(\mathbf{C})$ .

**Definition 2.5** (Terminal object). Let  $\mathbf{C}$  is a [Category](#), the [Object](#)  $t \in \text{ob}(\mathbf{C})$  is called *terminal object* if  $\forall x \in \text{ob}(\mathbf{C}) \exists! g_x : x \rightarrow t \in \text{hom}(\mathbf{C})$ .

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 2.6** (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](#))*

*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  $\mathbf{1}_{i \rightarrow i} : i \rightarrow i$ . The same is the truth for  $i'$ . Therefore  $u^{-1} \circ u = \mathbf{1}_{i \rightarrow i}$  and  $u \circ u^{-1} = \mathbf{1}_{i' \rightarrow i'}$ . These complete the commutative diagram build and finishes the proof. □



Figure 2.1: Commutative diagram for initial object uniqueness proof



Figure 2.2: Commutative diagram for terminal object uniqueness proof

**Theorem 2.7** (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.  $\square$

## 2.3 Product and sum

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

**Definition 2.8** (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



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



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

$h$  factorizes  $\pi'_{1,2}$ .

If we invert arrows in [Product](#) we will get another object definition that is called sum

**Definition 2.9** (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  $i'_{1,2}$ .

The [Product](#) of objects will provide also a definition for product of morphisms

**Definition 2.10** (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 2.8](#). 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](#)).

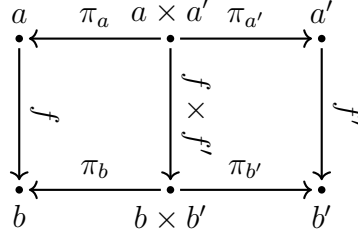


Figure 2.5: Product of morphisms.

## 2.4 Category as a monoid

Consider the following definition from abstract algebra

**Definition 2.11** (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  $\mathbf{1}$  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  $\mathbf{0}$ . I.e. we can see a direct connection with addition in algebra.

## 2.5 Exponential

TBD

**Definition 2.12** (Exponential). TBD

**Definition 2.13** (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  $a^b \in \text{ob}(\mathbf{C})$

## 2.6 Programming languages and algebraic data types

TBD

## 2.7 Examples

### 2.7.1 Set category

**Example 2.14** (Initial object). **[Set]** Note that there is only one function from empty set to any other sets [9] that makes the empty set as the **Initial object** in **Set category**.

**Example 2.15** (Terminal object). **[Set]** **Terminal object** in **Set category** is a set with one element i.e **Singleton**.

**Example 2.16** (Product). **[Set]** The **Product** of two sets  $A$  and  $B$  in **Set category** is defined as a **Cartesian product**:  $A \times B = \{(a, b) | a \in A, b \in B\}$ .

**Definition 2.17** (Disjoint union). Let  $\{A_i : i \in I\}$  be a family of sets indexed by  $I$ . The *disjoint union* [12] 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.

**Example 2.18** (Sum). **[Set]** The **Sum** of two sets  $A$  and  $B$  in **Set category** is defined as **Disjoint union**.

### 2.7.2 Programming languages

In our toy example fig. 1.10 the type `String` is **Initial object** and type `Bool` is the **Terminal object**. From other side there are types in different programming languages that satisfies the definitions of initial and terminal objects.

#### Hask category

**Example 2.19** (Initial object). **[Hask]** If we avoid lazy evaluations in Haskell (see **Haskell lazy evaluation** (Remark 1.44) ) then we can found 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 a object of the type.

There is only one function from the initial object:

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

The function is called absurd because it does absurd action. Nobody can proof 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 opposite direction because it would had been used for the `Void` object creation.

**Example 2.20** (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 2.21** (Product). `[Hask]` The `Product` in `Hask category` keeps a pair and the constructor 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 2.22** (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
```

**C++ category**

**Example 2.23** (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 2.24** (Terminal object). [C++] C++ 17 introduced a special type that keeps only one value - `std::monostate`:

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

**Example 2.25** (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 2.26** (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

### Scala category

**Example 2.27** (Initial object). [Scala] We used a same trick as for [Initial object](#) (Example 2.19) 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 2.28** (Terminal object). [Scala] We used a same trick as for [Terminal object](#) (Example 2.20) 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

### 2.7.3 Quantum mechanics

**Example 2.29** (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 2.30** (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 2.31** (Product). [FdHilb] The [Product](#) in [FdHilb category](#) is a [Direct sum of Hilber spaces](#).

**Example 2.32** (Sum). [FdHilb] The [Sum](#) in [FdHilb category](#) is a [Direct sum of Hilber spaces](#).

TBD



# Chapter 3

## Functors

### 3.1 Definitions

**Definition 3.1** (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. 3.1):

- $\forall a_C \in \text{ob}(\mathbf{C}), \exists a_D \in \text{ob}(\mathbf{D})$  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 3.1: Functor  $F : \mathbf{C} \Rightarrow \mathbf{D}$  definition

**Remark 3.2** (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.

**Definition 3.3** (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 3.4** (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 3.5** (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 3.6** (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

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 3.7** (**Cat** category). The category of small categories (see **Small category**) denoted as **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 3.8** (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:  $\mathbf{1}_{C \times D \rightarrow C \times D} = (\mathbf{1}_{C \rightarrow C}, \mathbf{1}_{D \rightarrow D})$

**Definition 3.9** (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*.

**Definition 3.10** (Terminal object in **Cat** category). Let consider  $\Delta_c$  is a trivial functor 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  $\forall f \in \text{hom}(\mathbf{A}) : \Delta_c f = \mathbf{1}_{c \rightarrow c}$ .

**Definition 3.11** (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*.

**Definition 3.12** (Profunctor). If we have a category  $\mathbf{C}$  then the **Bifunctor**  $\mathbf{C}^{\text{op}} \times \mathbf{C} \Rightarrow \mathbf{C}$  is called *profunctor*.

## 3.2 Curry-Howard-Lambek correspondence

There is an interesting correspondence between computer programs and mathematical proofs.

TBD

## 3.3 Examples

### 3.3.1 Set category

**Example 3.13** (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**  $\times$ .

### 3.3.2 Programming languages

#### Hask category

The functor can be defined in Haskell as follows <sup>1</sup>

**Example 3.14** (Functor). **[Hask]**

---

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

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

**Example 3.15** (Terminal object in **Cat** category). [Hask]

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

**Example 3.16** (Maybe as a functor). [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
```

**Example 3.17** (Contravariant functor). [Hask] TBD

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

**Example 3.18** (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
```

**C++ category**

TBD

**Scala category**

The functor can be defined in Haskell as follows <sup>2</sup>

**Example 3.19** (Functor). [Scala]

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

TBD

---

<sup>2</sup>the real definition is quite different from the current one



**3.3.3 Quantum mechanics**

TBD



# Chapter 4

## Natural transformation

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

### 4.1 Definitions

The natural transformation is not an easy concept compare other one 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  $\mathbf{D}$ :  $a_F = Fa, a_G = Ga \in \text{ob}(\mathbf{D})$  (see fig. 4.1). 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 4.1: Natural transformation: object mapping



Figure 4.2: Natural transformation: morphisms mapping

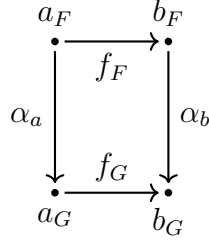


Figure 4.3: Natural transformation: commutative diagram

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 is no such morphisms then the functors can be considered as unrelated each other. Another example if the morphisms are [Isomorphisms](#) then the functors can be considered as equal.

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](#):

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

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

**Definition 4.1** (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**  $\alpha \subset \text{hom}(\mathbf{D})$  that satisfied the following conditions:

- For every **Object**  $a \in \text{ob}(\mathbf{C})$   $\exists \alpha_a \in \text{hom}(F(a), G(a))$  - **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

$$f_G \circ \alpha_a = \alpha_b \circ f_F, \quad (4.1)$$

where  $f_F = F(f), f_G = G(f)$ . In other words the morphisms the morphisms form a **Commutative diagram** shown on the fig. 4.3.

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

## 4.2 Operations with natural transformations

**Example 4.2** (**Fun** category). 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. 4.4). 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 (see fig. 4.5) commutes. The composition defined in the such way is called **Vertical composition**.

The functor category between categories  $\mathbf{C}$  and  $\mathbf{D}$  is denoted as  $[\mathbf{C}, \mathbf{D}]$ .

**Definition 4.3** (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*.



Figure 4.4: Natural transformation vertical composition: object mapping

Figure 4.5: Natural transformation vertical composition: morphism mapping  
- commutative diagram

**Definition 4.4** (Horizontal composition). If we have 2 pairs of functors. The first one  $F, G : \mathbf{C} \rightarrow \mathbf{D}$  and another one  $J, K : \mathbf{D} \Rightarrow \mathbf{E}$ . If we 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 4.5** (Bifunctor in category of functors). If we have the same pair of functors as in definition 4.4 then we can consider the functors as objects of 3 categories:  $\mathcal{A} = [\mathbf{C}, \mathbf{D}]$ ,  $\mathcal{B} = [\mathbf{D}, \mathbf{E}]$  and  $\mathcal{C} = [\mathbf{C}, \mathbf{E}]$

Next we want to 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 got another object from  $\mathcal{C}$ . The operation that is used is an ordinary functor's composition. 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  $\beta : J \rightarrow K$  the we can construct the following mapping

$$\otimes : \alpha \times \beta \rightarrow \alpha \star \beta.$$

As result we have the introduced mapping  $\otimes$  as a bifunctor.

**Definition 4.6** (Left whiskering). If we have 3 categories  $\mathbf{B}, \mathbf{C}, \mathbf{D}$ , **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 [8].

**Definition 4.7** (Right whiskering). If we have 3 categories  $\mathbf{C}, \mathbf{D}, \mathbf{E}$ , **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 [8].

**Definition 4.8** (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 4.9** (Whiskering). With [Identity natural transformation](#) we can redefine [Left whiskering](#) and [Right whiskering](#) via [Horizontal composition](#) as follows.

For left whiskering:

$$\alpha H = \alpha \star \mathbf{1}_{H \rightarrow H} \quad (4.2)$$

For right whiskering:

$$H\alpha = \mathbf{1}_{H \rightarrow H} \star \alpha \quad (4.3)$$

### 4.3 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 4.10** (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 4.11** (Ad-hoc polymorphism). Polymorphism is parametric if the function instances can behave differently dependently on the type they are being instantiated with.

**Theorem 4.12** (Reynolds). *Parametrically polymorphic functions are Natural transformations*

*Proof.* TBD □

#### 4.3.1 Hask category

In Haskell the most functions are [Parametrically polymorphic functions](#)<sup>2</sup>.

**Example 4.13** (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 4.12](#)) is [Natural transformation](#) (see [fig. 4.6](#)).

---

<sup>2</sup>really in the run-time the functions are not [Parametrically polymorphic functions](#)





Figure 4.6: Haskell parametric polymorphism as a natural transformation

From the definition of the natural transformation we have (4.1) therefore `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

```
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>3</sup>

---

<sup>3</sup>It is not directly applied to Haskell because it lazy evaluation that can perform optimization before that one

# Chapter 5

## 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.

### 5.1 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 than on internal set structure. Let  $M$  is a set and by the monoid definition (definition [2.11](#))  $\forall m_1, m_2 \in M$  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  $\forall m_1, m_2, m_3 \in M$

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

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

$$m \cdot e = e \cdot m = m. \tag{5.1}$$

Lets start with the first one we can define  $\mu$  as [Morphism](#) in the following way  $\mu : M \times M \rightarrow M$  where  $M \times M$  is [Product](#) ([Example 2.16](#)) in [Set category](#). I.e.  $M, M \times M \in \text{ob}(\mathbf{Set})$  and  $\mu \in \text{hom}(\mathbf{Set})$ . Consider another 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

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

$$\begin{array}{ccc}
M \times (M \times M) & \xrightarrow{\alpha} & (M \times M) \times M \\
\downarrow \eta \times \eta \times \eta & & \downarrow \eta \times \eta \times \eta \\
M \times M & \xrightarrow{\mu} & M
\end{array}$$

Figure 5.1: Commutative diagram for  $\mu \circ (\mu \times \mathbf{1}_{M \rightarrow M}) = \mu \circ (\mathbf{1}_{M \rightarrow M} \times \mu) \circ \alpha$ .

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 we will get:

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

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 the corresponding morphisms commute:

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

This corresponds the diagram is at fig. 5.1.

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 is written as follow

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

$$\begin{array}{ccc}
M^3 & \xrightarrow{\quad} & M \times M \\
\downarrow \eta \times \mathbf{1} & \searrow \mathbf{1}_{M \rightarrow M} \times \mu & \downarrow \eta \\
M \times M & \xrightarrow{\quad \mu \quad} & M
\end{array}$$

Figure 5.2: Commutative diagram for  $\mu \circ (\mu \times \mathbf{1}_{M \rightarrow M}) = \mu \circ (\mathbf{1}_{M \rightarrow M} \times \mu)$ .

The corresponding commutative diagram is shown on fig. 5.2.

For (5.1) consider a morphism  $\eta$  from a one point set  $I = \{0\}$  to a 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  $B' \cong_\rho M$  where the isomorphisms are defined as follow

$$\lambda(0, m) = m$$

and

$$\rho(m, 0) = m.$$

If we apply **Product of morphisms**  $\eta \times \mu$  and  $\mu \times \eta$  on  $B$  and  $B'$  respectively then we get

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

If we apply  $\mu$  on the result then we get

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

The (5.1) 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 show on fig. 5.3.

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

$$M \times M.$$



Figure 5.3: Commutative diagram for  $\mu \circ (\eta \times \mathbf{1}_{M \rightarrow M}) \circ \lambda = \mu \circ (\mathbf{1}_{M \rightarrow M} \times \mu) \circ \rho = \mathbf{1}_{M \rightarrow M}$ .

We also have 2 morphisms:

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

and

$$\mathbf{1}_{M \rightarrow M} : M \rightarrow M.$$

The 2 morphisms were combined into one using [Product of morphisms](#) for instance:

$$\mu \times \mathbf{1}_{M \rightarrow M} : (M \times M) \times M \rightarrow M \times M$$

The same structure <sup>1</sup> is used by [Functor](#) and especially by [Bifunctor](#) ([Example 3.13](#)).

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

**Definition 5.1** (Monoid). Consider [Set category](#)  $\mathbf{C}$  with a [Singleton](#)  $t \in \text{ob}(\mathbf{C})$ . The [Cartesian product](#) with [Product of morphisms](#) form a [Bifunctor](#)  $\times$  (see [example 3.13](#)). 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  $\eta : t \rightarrow m$
3. the morphisms satisfy the following conditions:

$$\begin{aligned} \mu \circ (\mu \times \mathbf{1}_{M \rightarrow M}) &= \mu \circ (\mathbf{1}_{M \rightarrow M} \times \mu) \circ \alpha, \\ \mu \circ (\eta \times \mathbf{1}_{M \rightarrow M}) \circ \lambda &= \mu \circ (\mathbf{1}_{M \rightarrow M} \times \mu) \circ \rho = \mathbf{1}_{M \rightarrow M} \end{aligned} \quad (5.2)$$

---

<sup>1</sup>not only objects mapping but also morphisms mapping

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

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

and

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

## 5.2 Monoidal category

As we saw in the categorical definition for monoid (see definition [5.1](#)) the category  $\mathbf{C}$  should satisfy several conditions to have an object as monoid. Lets formalise the conditions.

**Definition 5.2** (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: associativity:

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

where  $\alpha$  is called associator.  $e$  can be treated as left and right identity:

$$e \otimes a \cong_{\lambda} a,$$

$$a \otimes e \cong_{\rho} a,$$

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

In the [Set category](#) we have  $\times$  as the monoidal product (see example [3.13](#)). There is a morphism  $\eta$  from terminal object  $t$  to  $e$  [[3](#)] (see definition [5.1](#)).

**Definition 5.3** (Strict monoidal category). A [Monoidal category](#)  $\mathbf{C}$  is said to be strict if the associator, left unitor and right unitors are all identity morphisms i.e.

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

**Remark 5.4** (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](#) as it is introduced for [Hilbert spaces](#). Note that the monoidal product is [Bifunctor](#).

### 5.3 Category of endofunctors

The **Fun category** (Example 4.2) is an example of a category. We can apply additional limitation and consider only **Endofunctors** i.e. we will look at the category  $[\mathbf{C}, \mathbf{C}]$  - 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  $X$ ”[4]. 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 5.5** (Category of endofunctors). Let  $\mathbf{C}$  is a category, then  $[\mathbf{C}, \mathbf{C}]$  - category 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 5.6** (Monad). The monad  $M$  is an **Endofunctor** with 2 **Natural transformations**:

1.  $\eta : \mathbf{1}_{\mathbf{C} \Rightarrow \mathbf{C}} \rightarrow M$
2.  $\mu : M \circ M \rightarrow M$

where  $\mathbf{1}_{\mathbf{C} \Rightarrow \mathbf{C}}$  is **Identity functor**.

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

$$\begin{aligned} \mu \circ M\mu &= \mu \circ \mu M, \\ \mu \circ M\eta &= \mu \circ \eta M = \mathbf{1}_{M \rightarrow M}, \end{aligned} \tag{5.3}$$

where  $M\mu, M\eta$  - **Right whiskerings**,  $\mu M, \eta M$  - **Left whiskerings**,  $\mathbf{1}_{M \rightarrow M}$  - **Identity natural transformation** for  $M$ . **Vertical composition** is used in the equations.

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

Lets look at the requirements (5.3) 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 (4.2) and (4.3) as follows

$$\begin{aligned} \mu \circ (\mathbf{1}_{M \rightarrow M} \star \mu) &= \mu \circ (\mu \star \mathbf{1}_{M \rightarrow M}), \\ \mu \circ (\mathbf{1}_{M \rightarrow M} \star \eta) &= \mu \circ (\eta \star \mathbf{1}_{M \rightarrow M}) = \mathbf{1}_{M \rightarrow M}. \end{aligned} \tag{5.4}$$





Figure 5.4: Monad as monoid in the category of endofunctors.

Thus we can notice that the pair of operations (composition  $\circ$  and  $\star$ ) forms the bifunctor (see [Bifunctor in category of functors](#) ([Remark 4.5](#))).

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

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

thus

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

Similarly

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

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

$$\mu \circ (1_{M \rightarrow M} \star \mu) = \mu \circ (\mu \star 1_{M \rightarrow M}) \quad (5.5)$$

is similar to the conditions on the fig. 5.2 and can be rewritten on fig. 5.4. Thus if we compare (5.5) and (5.2) 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 and replace TBD

## 5.4 Kleisli category. Monads in programming languages

**Definition 5.7** (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  $\mathbf{C}_M$  that

is called as *Kleisli category* 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  $\mathbf{C}_M$ :  $\text{hom}(\mathbf{C}_M) \subset \text{hom}(\mathbf{C})$ .

### 5.4.1 Programming languages

*Kleisli category* widely spread in programming especially it provides good description for different types of computations, for instance [6, 5]

- **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. TBD
- **Exception** i.e. TBD
- **Continuation** i.e. TBD
- **Interactive input** i.e. TBD
- **Interactive output** i.e. TBD

TBD

## 5.5 Examples

### 5.5.1 Programming languages

Haskell

**Example 5.8** (Monad). [**Hask**] In Haskell monad can be defined from **Functor** (Example 3.14) as follows <sup>2</sup>

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

---

<sup>2</sup>real definition is quite different from the presented one

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  $\eta$  and `join` as  $\mu$ . In the case the bind operator `>>=` can be implemented as follows

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

**Example 5.9** (Maybe monad). [Hask] Consider the monad definition that is most close to math definition first:

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

the Maybe monad can be implemented as follows

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

TBD

C++

TBD

Scala

**Example 5.10** (Monad). [Scala] The monad concept is Scala is more close to formal math definition for `Monad`. It can be defined as follows <sup>3</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>3</sup>real definition is quite different from the presented one

### 5.5.2 Quantum mechanics

**Definition 5.11** (Tensor product). TBD

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 Hilber space  $\mathcal{H}_{at}$  with  $|a\rangle$  and  $|b\rangle$  as basis vectors. Light also has its own Hilber space  $\mathcal{H}_f$  with Fock state  $\{|n\rangle\}$  as the basis.<sup>4</sup> The result system that describes both atom and light is represented as the tensor product  $\mathcal{H}_{at} \otimes \mathcal{H}_f$ .

The morphisms of **FdHilb** category have a connection with [Tensor product](#). Consider the so called 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](#).

TBD

---

<sup>4</sup> Really the  $\mathcal{H}_f$  is infinite dimensional Hilber space and seems to be out of our assumption about **FdHilb** category as a collection of finite dimensional Hilber spaces only.

# Chapter 6

## Yoneda's lemma

TBD

### 6.1 Examples

#### 6.1.1 Quantum mechanics

Flori interpretation of quantum mechanics

TBD



# Index

- C++** category, 34
  - definition, 18
- C++** toy category
  - example, 20
- Cat** category
  - definition, 38
- FdHilb** category, 35
  - definition, 23
- Fun** category
  - example, 45
- Fun** category example, 5, 56
- Hask** category, 32, 33, 35
  - definition, 18
- Hask** toy category
  - example, 19
- Hask** toy category example, 20, 21
- Rel** category, 23
  - definition, 22
- Scala** category, 35
  - definition, 18
- Scala** toy category
  - example, 21
- Set** category, 13, 22, 32, 51, 54, 55
  - definition, 14
- Ad-hoc polymorphism
  - definition, 48
- Associativity axiom, 13, 27, 45
  - declaration, 11
- Associator, 55
  - definition, 55
- Bifunctor, 39, 47, 54, 55
  - Set** example, 39
  - definition, 39
- Bifunctor example, 54
- Bifunctor in category of functors
  - remark, 47
- Bifunctor in category of functors
  - remark, 57
- Bijection
  - definition, 17
- Binary relation, 14, 22, 24
  - definition, 14
- Cartesian closed category
  - definition, 31
- Cartesian product, 24, 32, 39, 54
  - definition, 14
- Category, 5, 7, 13, 18–21, 27, 28, 38, 39
  - Fun** example, 45
  - Set**, 14
  - definition, 12
  - dual, 13
  - large, 13, 15
  - opposite, 13
  - small, 13
- Category of endofunctors
  - definition, 56
- Category Product, 39

- definition, 38
- Class, 9, 11–13
  - definition, 9
- Class of Morphisms
  - remark, 11
- Class of Objects
  - remark, 9
- Codomain, 5, 16
  - definition, 10
  - example, 15
- Commutative diagram, 28, 44, 45
  - definition, 11
- Composition
  - opposite category, 13
  - remark, 11
- Composition axiom, 11, 13, 19, 27, 39, 45
  - declaration, 10
- Contravariant functor
  - Hask** example, 40
  - definition, 39
- Dirac notation
  - example, 23
- Direct sum of Hilber spaces, 24, 35
  - definition, 23
- Discrete category, 18
  - definition, 18
- Disjoint union, 32
  - definition, 32
- Domain, 5, 16, 39
  - definition, 10
  - example, 15
- Dual space, 23, 60
  - definition, 22
- Endofunctor, 38, 56, 57
  - definition, 38
- Epimorphism, 15, 16
  - definition, 12
- Exponential, 31
  - definition, 31
- Function, 14, 22, 24, 27, 39
  - definition, 14
- Functor, 6, 7, 38–40, 43, 45, 47, 54
  - Hask** example, 39
  - Scala** example, 40
  - definition, 37
  - remark, 38
- Functor composition, 6, 38
  - definition, 38
- Functor example, 58
- Haskell lazy evaluation
  - remark, 20
- Haskell lazy evaluation remark, 18, 32
- Hilbert space, 5, 23, 24, 55
  - definition, 22
- Horizontal composition, 5, 48
  - definition, 47
- Identity functor, 5, 38, 56
  - definition, 38
  - remark, 38
- Identity is unique theorem, 11
  - declaration, 28
- Identity morphism, 5, 11, 13, 14, 18–21, 27, 28, 38, 47
  - definition, 11
- Identity natural transformation, 5, 48, 56
  - definition, 47
- Initial object, 24, 28, 31, 32, 35
  - C++** example, 34
  - FdHilb** example, 35
  - Hask** example, 32
  - Scala** example, 35
  - Set** example, 32
  - definition, 28
- Initial object example, 35
- Initial object is unique theorem



- declaration, 28
- Injection, 17
  - definition, 16
- Injection vs Monomorphism
  - remark, 17
- Isomorphism, 5, 12, 27–29, 44, 51, 53, 55
  - definition, 12
  - remark, 12
- Kleisli category, 5, 58
  - definition, 57
- Large category, 15
  - definition, 13
- Left unitor, 55
  - definition, 55
- Left whiskering, 5, 48, 56
  - definition, 47
- Linear map, 23, 24
  - definition, 23
  - remark, 23
- Maybe as a functor
  - Hask** example, 40
- Maybe monad
  - Hask** example, 59
- Monad, 5, 7, 57, 59
  - Hask** example, 58
  - Scala** example, 59
  - definition, 56
- Monoid, 31, 51, 55, 57
  - definition, 31, 54
- Monoidal product
  - definition, 55
- Monoidal category, 55
  - definition, 55
- Monoidal product
  - remark, 55
- Monomorphism, 17
  - definition, 12
- Morphism, 5–7, 10–15, 18–24, 27, 29, 30, 38, 39, 43, 45, 47, 51, 54
  - C++** example, 20
  - FdHilb** category, 23
  - Fun** example, 45
  - Hask** example, 19
  - Rel** category, 22
  - Scala** example, 21
  - Set** category, 14
  - definition, 10
  - remark, 10
- Morphisms equality
  - definition, 27
- Natural transformation, 5, 7, 45, 47, 48, 56
  - definition, 44
  - Horizontal composition, 47
  - Vertical composition, 45
- Object, 6, 7, 9–14, 18–24, 27–30, 34, 38, 43, 45, 47, 55
  - C++** example, 20
  - FdHilb** category, 23
  - Fun** example, 45
  - Hask** example, 19
  - Rel** category, 22
  - Scala** example, 21
  - Set** category, 14
  - definition, 9
- Objects equality, 28, 29
  - definition, 27
- Opposite category, 5, 13, 28, 29
  - definition, 13
- Parametric polymorphism, 48
- Parametrically polymorphic
  - function, 48
  - Hask** example, 48
  - definition, 48
- Product, 24, 30–35

- C++** example, 34
- FdHilb** example, 35
- Hask** example, 33
- Set** example, 32
  - definition, 29
- Product example, 51
- Product of morphisms, 39, 52–54
  - definition, 30
- Profunctor
  - Hask** example, 40
  - definition, 39
- Pure function, 18
  - definition, 18
- Rabi oscillations
  - example, 24
- Reynolds theorem, 48
  - declaration, 48
- Right unitor, 55
  - definition, 55
- Right whiskering, 6, 48, 56
  - definition, 47
- Set**, 10, 14, 15, 17, 22, 24, 27
  - definition, 14
- Set of morphisms, 5
  - definition, 13
- Set vs Category
  - remark, 15
- Set vs Category remark, 27
- Singleton, 22, 24, 32, 54
  - definition, 15
- Small category, 13, 38
  - definition, 13
- Strict monoidal category
  - definition, 55
- Sum, 24, 31–35
  - C++** example, 34
  - FdHilb** example, 35
  - Hask** example, 33
  - Set** example, 32
  - definition, 30
- Sum example, 24
- Surjection, 15–17
  - definition, 15
- Surjection vs Epimorphism
  - remark, 15
- Tensor product, 55, 60
  - definition, 60
- Terminal object, 24, 28, 29, 31, 32, 35
  - C++** example, 34
  - FdHilb** example, 35
  - Hask** example, 33
  - Scala** example, 35
  - Set** example, 32
  - Cat** category, 39
  - definition, 28
- Terminal object example, 35
- Terminal object in **Cat** category
  - Hask** example, 40
  - definition, 39
- Terminal object is unique theorem
  - declaration, 29
- Vertical composition, 5, 45, 56
  - definition, 45
- Whiskering
  - remark, 48

# Bibliography

- [1] Coecke, B. Introducing categories to the practicing physicist / Bob Coecke. — 2008. — <https://arxiv.org/abs/0808.1032>.
- [2] ([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>.
- [3] ([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  $c$ ? — Mathematics Stack Exchange. — URL:<https://math.stackexchange.com/q/6318> (version: 2010-10-08). <https://math.stackexchange.com/q/6318>.
- [4] MacLane, S. Categories for the Working Mathematician / Saunders MacLane. — New York: Springer-Verlag, 1971. — P. ix+262. — Graduate Texts in Mathematics, Vol. 5.
- [5] 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>.
- [6] 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>.
- [7] Murashko, I. Category theory. — <https://github.com/ivanmurashko/articles/tree/master/cattheory/src>. — 2018.
- [8] nLab authors. whiskering. — <http://ncatlab.org/nlab/show/whiskering>. — 2018. — Sep. — Revision 11.
- [9] ProofWiki. Empty mapping is unique / ProofWiki. — 2018. — [https://proofwiki.org/wiki/Empty\\_Mapping\\_is\\_Unique](https://proofwiki.org/wiki/Empty_Mapping_is_Unique).

- [10] 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).
- [11] 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).
- [12] 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).
- [13] 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).
- [14] 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%9993Fraenkel\\_set\\_theory&oldid=852467638](https://en.wikipedia.org/w/index.php?title=Zermelo%E2%80%9993Fraenkel_set_theory&oldid=852467638).
- [15] Мурашко И. В. Квантовая оптика / Мурашко И. В. — 2018. — <https://github.com/ivanmurashko/lectures/blob/master/pdfs/qo.pdf>.