# Category Theory by Example

Ivan Murashko

August 26, 2018

# Contents

# Introduction

There is an introduction to Category Theory. There are a lot of examples in each chapter. The examples covers different category theory application areas. I assume that the reader is familiar with the corresponding area if not the example(s) can be passed. Anyone can choose the most 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. There are Haskell, Scala, C++. The source files for programming languages examples (Haskell, C++, Scala) can be found on github repo [4].

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 $\mathrm{ob}(\mathbf{C})$ (see fig. 1.1).

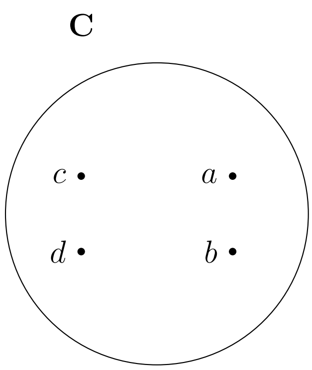


Figure 1.1: Class of objects $\mathrm{ob}(\mathbf{C}) = \{a, b, c, d\}$

### 1.1.2 Morphism

Morphism is a kind of relation between 2 Objects.

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

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

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 called as $\hom(a, b)$.

**Definition 1.5** (Domain)**.** Given a Morphism $f : a \to b$, the Object $a$ is called domain and is denoted as $\operatorname{dom} a$.

**Definition 1.6** (Codomain)**.** Given a Morphism $f : a \to b$, the Object $b$ is called codomain and is denoted as $\operatorname{cod} a$.

Morphisms have several properties. [1]

**Axiom 1.7** (Composition)**.** *If we have 3 Objects $a, b$ and $c$ and 2 Morphisms*

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

*and*

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

*then there exists Morphism*

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

*such that*

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

**Remark 1.8** (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 $f_{ab}(x) \in b$.

[1] The properties don't have any proof and postulated as axioms

**Axiom 1.9** (Associativity). *The Morphisms Composition (Axiom 1.7) s 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 1.10** (Identity morphism). For every Object $a$ we define a special Morphism $\mathbf{1}_a : a \to a$ with the following properties: $\forall f_{ab} : a \to b$

$$\mathbf{1}_a \circ f_{ab} = f_{ab} \tag{1.1}$$

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

$$f_{ba} \circ \mathbf{1}_a = f_{ba}. \tag{1.2}$$

This morphism is called *identity morphism.*

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

**Definition 1.11** (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 endpoints lead to the same result by composition

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

$$\begin{tikzcd}
a \arrow[rr, "f_{ab}" description] \arrow[dr, "f_{ac}" description] & & b \\
& c \arrow[ur, "f_{cb}" description] &
\end{tikzcd}$$

**Remark 1.12** (Class of Morphisms). The Class of Morphisms will be marked as $\hom(\mathbf{C})$ (see fig. 1.2)

**Definition 1.13** (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.*

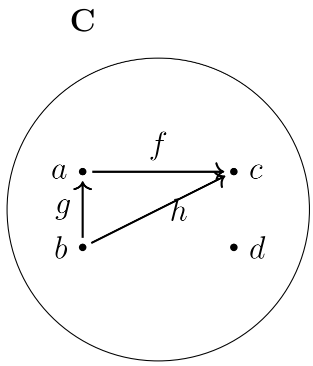


Figure 1.2: Class of morphisms $\hom(\mathbf{C}) = \{f, g, h\}$, where $h = f \circ g$

**Definition 1.14** (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 1.15** (Isomorphism)**.** A Morphism $f : a \to b$ is called isomorphism if $\exists g : b- > a$ such that $f \circ g = \mathbf{1}_a$ and $g \circ f = \mathbf{1}_b$.

**Remark 1.16** (Isomorphism)**.** There are can be many different Isomorphisms between 2 Objects.

### 1.1.3 Category

**Definition 1.17** (Category)**.** A category $\mathbf{C}$ consists of

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

For any Object $a$ there should be unique Identity morphism $\mathbf{1}_a$. Any morphism should satisfy Composition (Axiom 1.7) and Associativity (Axiom 1.9) properties. See fig. 1.3

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

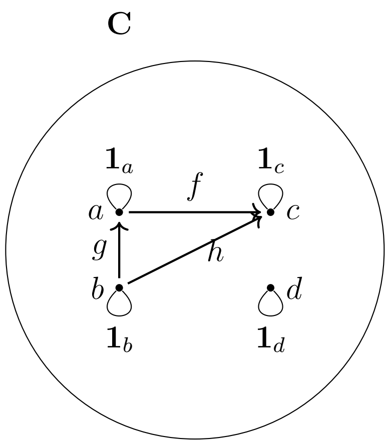


Figure 1.3: Category $\mathbf{C}$. It consists of 4 objects $\text{ob}(\mathbf{C}) = \{a, b, c, d\}$ and 7 morphisms $\text{ob}(\mathbf{C}) = \{f, g, h = f \circ g, \mathbf{1}_a, \mathbf{1}_b, \mathbf{1}_c, \mathbf{1}_d\}$

**Definition 1.18** (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 $f^{op} \in \text{hom}(\mathbf{C}^{\mathbf{op}})$ has $\text{dom}\, f^{op} = b, \text{cod}\, f^{op} = a$ (see fig. 1.4)

**Remark 1.19.** Composition on $C^{op}$ As you can see from fig. 1.4 the Composition (Axiom 1.7) 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 1.20** (Small category)**.** A category $\mathbf{C}$ is called *small* if both $\text{ob}(\mathbf{C})$ and $\text{hom}(\mathbf{C})$ are Sets

**Definition 1.21** (Large category)**.** A category $\mathbf{C}$ is not Small category then it is called *large*. The example of large category is **Set** category (Example 1.25)

## 1.2 Examples

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

### 1.2.1 Set category

**Definition 1.22** (Set)**.** Set is a collection of distinct object. The objects are called the elements of the set.

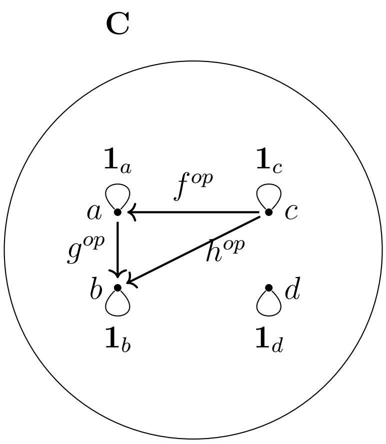


Figure 1.4: Opposite category $C^{op}$ to the category from fig. 1.3 . It consists of 4 objects $\mathrm{ob}(\mathbf{C^{op}}) = \mathrm{ob}(\mathbf{C}) = \{a, b, c, d\}$ and 7 morphisms $\mathrm{hom}(\mathbf{C^{op}}) = \{f^{op}, g^{op}, h^{op} = g^{op} \circ f^{op}, \mathbf{1}_a, \mathbf{1}_b, \mathbf{1}_c,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

**Definition 1.23** (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 1.24** (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 does not allowed "multi value".

**Example 1.25** (**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 $\forall x \in X : \mathbf{1}_[X](x) = x$.

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 [9]can be applied. To avoid such situations we assume the some kind of limitations are applied on our construction, for instance ZFC [10]. 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 1.26** (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" [3]

Contrary in the category theory we initially don't have 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" [3] there.

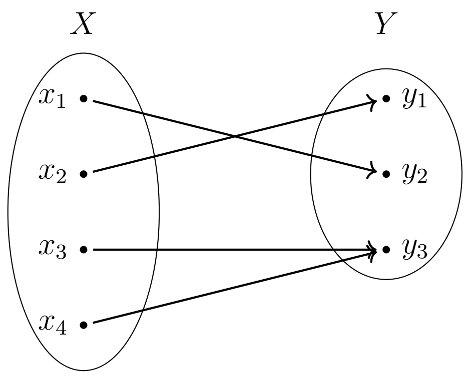


Figure 1.5: A surjective (non-injective) function from domain $X$ to codomain $Y$

**Definition 1.27** (Singleton)**.** The *singleton* is a Set with only one element.

**Definition 1.28** (Domain)**.** Given a function $f : X \to Y$, the set $X$ is the domain.

**Definition 1.29** (Codomain)**.** Given a function $f : X \to Y$, the set $Y$ is the codomain.

**Definition 1.30** (Surjection)**.** The function $f : X \to Y$ 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 1.31** (Surjection vs Epimorphism)**.** Surjection and Epimorphism are related each other. Consider a non-surjective function $f : X \to Y' \subset Y$ (see fig. 1.6). One can conclude that there is not an Epimorphism because $\exists g_1 : Y' \to Y'$ and $g_2 : Y \to Y$ 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 $g_1 = \mathbf{1}_{[}Y'], g_2 = \mathbf{1}_{[}Y]$. 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 **Set** category. Moreover there is a proof [7] of that fact.

**Definition 1.32** (Injection)**.** The function $f : X \to Y$ 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 1.33** (Injection vs Monomorphism)**.** Injection and Monomorphism are related each other. Consider a non-injective function $f : X \to Y$ (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 $f(g_1(a_1)) = y_3 = f(g_2(b_1))$.

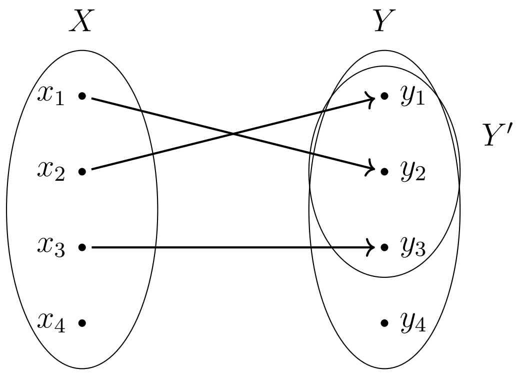


Figure 1.6: A non-surjective function $f$ from domain $X$ to codomain $Y' \subset Y$. $\exists g_1 : Y' \to Y', g_2 : Y \to Y$ such that $g_1(Y') = g_2(Y')$, but as soon as $Y' \neq Y$ we have $g_1 \neq g_2$. Using the fact that $Y'$ is codomain of $f$ we got $g_1 \circ f = g_2 \circ f$. I.e. the function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

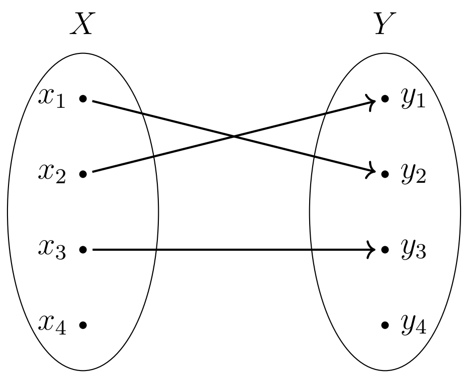


Figure 1.7: A injective (non-surjective) function from domain $X$ to codomain $Y$

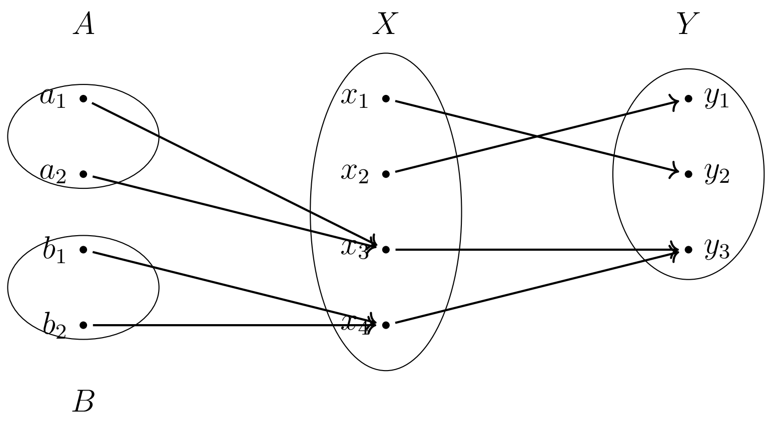


Figure 1.8: A non-injective function $f$ from domain $X$ to codomain $Y$. $\exists g_1 : A \to X, g_2 : B \to X$ such that $g_1 \neq g_2$ but $f \circ g_1 = f \circ g_2$. I.e. the function $f$ is not monomorphism.

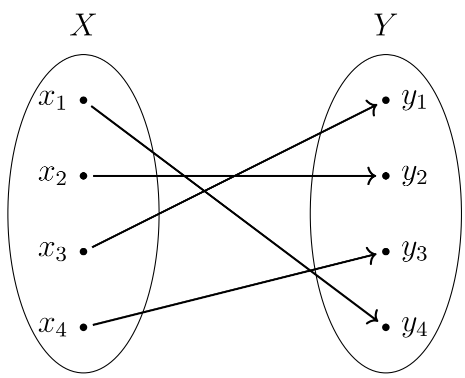


Figure 1.9: An injective and surjective function (bijection)

As result we can say that an Injection is a Monomorphism in **Set** category. Moreover there is a proof [6] of that fact.

**Definition 1.34** (Bijection)**.** The function $f : X \to Y$ is bijective (or one-to-one correspondence) if it is an Injection and a Surjection (see fig. 1.9).

There is a question what's 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 1.35** (Discrete category)**.** Discrete category is a Category where Morphisms are only Identity morphisms.

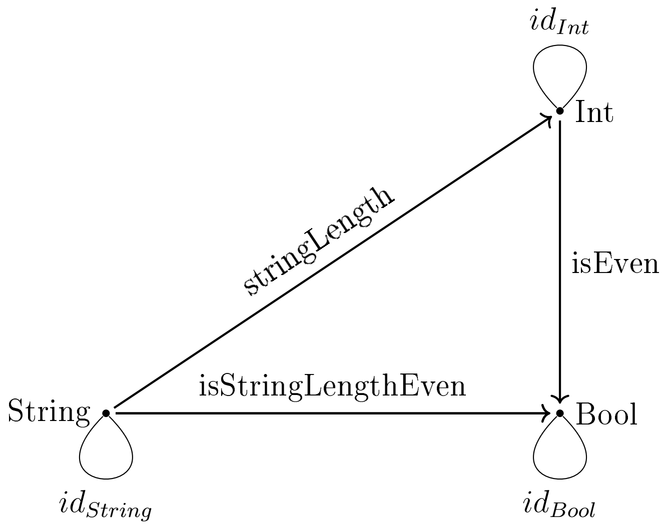


Figure 1.10: Programming language category example. Objects are types: Int, Bool, String. Morphisms are several functions

### 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 function (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 the performance. The laziness can also make category theory axiom invalid (see Haskell lazy evaluation (Remark 1.37) ).

Strictly speaking neither Haskell (pure functional language) nor C++ can be considered as a category in general. As a first approximation the 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.

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).

#### Hask category

**Example 1.36** (**Hask** 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 that 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 1.7)

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

**Remark 1.37** (Haskell lazy evaluation)**.** Each Haskell type has a special value $\perp$. The value presents and lazy evaluations 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++ category

**Example 1.38** (**C++** category)**.** We will use the same trick as in **Hask** category (Example 1.36) 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 category

**Example 1.39** (**Scala** category)**.** We will use the same trick as in **Hask** category (Example 1.36) 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 superposition principle. The **Set** category (Example 1.25) cannot be used for it because it does not satisfied the principle.

A simple modification of the **Set** category can satisfy the principle.

**Example 1.40** (**Rel** category)**.** We will consider a set of sets (same as **Set** category (Example 1.25) ) 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 $\mathbf{C_R}$ - the **Rel** category. $X, Y \in \mathrm{ob}(\mathbf{C_R})$ - 2 sets which consists of different elements. Let $f : X \to X$ - 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 (Example 1.25) ) 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 mechanics.

In the quantum mechanics we say about Hilber spaces.

**Definition 1.41** (Hilbert space)**.** The Hilbert space 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 1.42** (Dual space)**.** Each Hilber space $\mathcal{H}$ has an associated with it so called dual space $\mathcal{H}^*$ that consists of linear functionals

**Example 1.43** (Dirac notation)**.** Consider a so called 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.

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

$\forall |\phi\rangle \in \mathcal{H}$ 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 1.44** (Linear map)**.** The linear map between2 Hilbert spaces $\mathcal{A}$ and $\mathcal{B}$ is a mapping $f : \mathcal{A} \to \mathcal{B}$ 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 $f(a), f(a_{1,2}) \in \mathcal{B}$.

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

| | **Set** | **Rel** | **FdHilb** |
|---|---|---|---|
| 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 2.14) | Sum (Example 2.14) | Direct sum of Hilber spaces |

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

**Definition 1.45** (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

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

**Example 1.46** (**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 (Example 1.40) . The brief relation is described in the table 1.1.

**Definition 1.47** (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. [2] 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

[2] 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.

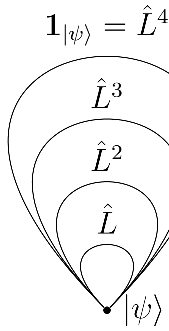


Figure 1.11: Rabi oscillations as a category **R**

between the tensor product and linear maps (aka morphisms) between $\mathcal{A}$ and $\mathcal{B}$:

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

where $\mathcal{A}^*$ - Dual space.

**Example 1.48** (Rabi oscillations)**.** For our example we consider a 2 level atom with states $|a\rangle$ - excited and $|b\rangle$. 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 will be called as **R**. I.e. $\mathrm{ob}(\mathbf{R}) = \mathcal{H}_2\{|\psi\rangle\}$.

The atom interacts with light beam of frequency $\omega = \omega_{ab}$. The state of the system is described by the following equation [11]:

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

where $\omega_R$ - Rabi frequency [11].

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:

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

# 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 for if an Object is a Set. In this case we can say that 2 objects are equal if they contains the same elements. Unfortunately we cannot do the same for default objects as soon as they don't have any internal structure. We can use the same trick as in Set vs Category (Remark 1.26) : if we cannot use "microscope" lets use "telescope" and define the equality of objects and morphisms of a category $\mathbf{C}$ in the terms of whole $\hom(\mathbf{C})$.

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

Unlike Functions between Sets we don't have any additional info [1] about Morphisms except category theory axioms which the morphisms satisfied [2]. This leads us to the following definition for morphims 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 1.7)
- Associativity (Axiom 1.9)
- Identity morphism: (1.1), (1.2)

[1] 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 $\mathbf{1}_a$. Let $\exists f : a \to a$ such that $f$ is also identity. In the case (1.1) for $f$ as identity gives

$$f \circ \mathbf{1}_a = \mathbf{1}_a.$$

From other side (1.2) for $\mathbf{1}_a$ satisfied

$$f \circ \mathbf{1}_a = f$$

i.e. $f = \mathbf{1}_a$. □

## 2.2 Initial and terminal objects

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

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

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 $\mathbf{C}^{\mathbf{op}}$.

**Theorem 2.6** (Initial object is unique). *Let $\mathbf{C}$ is a category and $i, i' \in \mathrm{ob}(\mathbf{C})$ two Initial objects then there exists an unique Isomorphism $u : i \to i'$ (see Objects equality)*

*Proof.* Consider the following Commutative diagram (see fig. 2.1) □

**Theorem 2.7** (Terminal object is unique). *Let $\mathbf{C}$ is a category and $t, t' \in \mathrm{ob}(\mathbf{C})$ two Terminal objects then there exists an unique Isomorphism $v : t' \to t$ (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. □

$$\begin{tikzcd}
i \arrow[r, "u"] \arrow[dr, "1_i"] & i' \arrow[d, "u^{-1}"] \arrow[dr, "1_{i'}"] & \\
& i \arrow[r, "u"] & i'
\end{tikzcd}$$

Figure 2.1: Commutative diagram for initial object unique proof

$$\begin{tikzcd}
t & t' \arrow[l, "v"'] & \\
& t \arrow[ul, "1_t"] \arrow[u, "v^{-1}"] & t' \arrow[l, "v^{-1}"] \arrow[ul, "1_{t'}"']
\end{tikzcd}$$

Figure 2.2: Commutative diagram for terminal object unique proof

$$\begin{tikzcd}
& c' \arrow[dl, "\pi'_1"'] \arrow[d, "h"] \arrow[dr, "\pi'_2"] & \\
c_1 & c \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & c_2
\end{tikzcd}$$

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

## 2.3 Product and sum

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

**Definition 2.8** (Product)**.** Let we have a category $\mathbf{C}$ and $c_1, c_2 \in \mathrm{ob}(\mathbf{C})$ -two Objects 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 $a = g_a c, b = g_b c$ and the following universal property is satisfied: $\forall c' \in \mathrm{ob}(\mathbf{C})$ and morphisms $\pi'_1 : \pi'_2 c' = c_1, \pi'_2 : \pi'_2 c' = c_2$, 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 $\pi'_{1,2}$.

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

**Definition 2.9** (Sum)**.** Let we have a category $\mathbf{C}$ and $c_1, c_2 \in \mathrm{ob}(\mathbf{C})$ -two Objects 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: $\forall c' \in \mathrm{ob}(\mathbf{C})$ and morphisms $i'_1 : i'_1 x_1 = c', i'_2 : i'_2 x_2 = c'$, 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 $i'_{1,2}$.

## 2.4 Category as monoid

Consider the following definition from abstract algebra

**Definition 2.10** (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: $\forall a, b \in M$: $a \circ b \in M$

$$\begin{tikzcd}
 & c' & \\
c_1 \arrow[ur, "i'_1"] \arrow[r, "i_1"] & c \arrow[u, "h"] & c_2 \arrow[ul, "i'_2"'] \arrow[l, "i_2"']
\end{tikzcd}$$

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

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 firs 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 called as **1** and the operation 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 called as **0**. I.e. we can see a direct connection with addition in algebra.

## 2.5 Exponential

TBD

## 2.6 Programming languages and algebraic data types

TBD

## 2.7 Examples

### 2.7.1 Set category

**Example 2.11** (Initial object)**.** [**Set**] Note that there is only one function from empty set to any other sets [5] that makes the empty set as the Initial object in **Set** category (Example 1.25) .

**Example 2.12** (Terminal object)**.** [**Set**] Terminal object in **Set** category (Example 1.25) is a set with one element i.e Singleton.

**Example 2.13** (Product)**.** [**Set**] The Product of two sets $A$ and $B$ in **Set** category (Example 1.25) is defined as a Cartesian product: $A \times B = \{(a, b)|a \in A, b \in B\}$.

**Example 2.14** (Sum)**.** [**Set**] The Sum of two sets $A$ and $B$ in **Set** category (Example 1.25) is defined as disjoint union [8]. Let $\{A_i : i \in I\}$ be a family of sets indexed by $I$. The disjoint union 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 Ai the element $x$ came from.

### 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 2.15** (Initial object)**.** [**Hask**] If we avoid lazy evaluations in Haskell (see Haskell lazy evaluation (Remark 1.37) ) then we can found the following types as candidates for initial and terminal object in haskell. Initial object in **Hask** category (Example 1.36) 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 can be used the following absurd argument: "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 2.16** (Terminal object)**.** **[Hask]** Terminal object (unit) in **Hask** category (Example 1.36) 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 2.17** (Product)**.** **[Hask]** The Product in **Hask** category (Example 1.36) 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 2.18** (Sum)**.** **[Hask]** The Sum in **Hask** category (Example 1.36) 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 2.19** (Initial object)**.** **[C++]** In C++ exists a special type that does not hold any values and as result that cannot be created: `void`. You cannot create an object of that type: you will get a compiler error if you try.

**Example 2.20** (Terminal object)**.** **[C++]** C++ 17 introduced a special type that keeps only one value - `std::monostate`:

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

**Example 2.21** (Product)**.** **[C++]** The Product in **C++ category** (Example 1.38) keeps a pair and the constructor defined as follows

```
namespace std {
  template< class A, class B > struct pair {
    T1 first;
    T2 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 the product.

**Example 2.22** (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 2.23** (Initial object)**. [Scala]** We used a same trick as for Initial object (Example 2.15) and define Initial object in **Scala** category (Example 1.39) as a type without values

```
sealed trait Void
```

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

**Example 2.24** (Terminal object)**. [Scala]** We used a same trick as for Terminal object (Example 2.16) and define Terminal object in **Scala** category (Example 1.39) 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 2.25** (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 2.26** (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 2.27** (Product)**. [FdHilb]** The Product in **FdHilb** category (Example 1.46) is a Direct sum of Hilber spaces.

**Example 2.28** (Sum)**. [FdHilb]** The Sum in **FdHilb** category (Example 1.46) 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 $F : \mathbf{C} \to \mathbf{D}$ between the categories is called *functor* is it preserves the internal structure (see fig. 3.1):

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

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

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

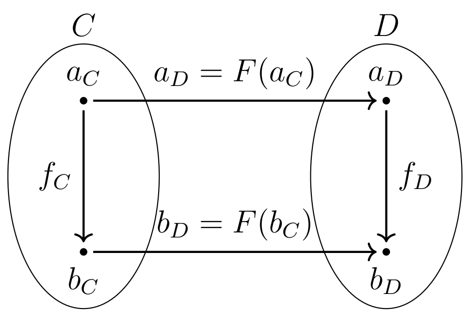


Figure 3.1: Functor $F : \mathbf{C} \to \mathbf{D}$ definition

**Remark 3.2** (Functor)**.** When we say that functor preserve internal structure means that 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** (Category Composition)**.** TBD

**Definition 3.4** (Category Identity)**.** TBD

**Definition 3.5** (**Cat** category)**.** TBD

As an extension of Cartesian product is used so called Category product

**Definition 3.6** (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 $c \in \mathrm{ob}(\mathbf{C})$ and $d \in \mathrm{ob}(\mathbf{D})$
- Morphisms are the pair $(f, g)$ where $f \in \mathrm{hom}(\mathbf{C})$ and $g \in \mathrm{hom}(\mathbf{D})$
- Composition (Axiom 1.7) 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: $\mathbf{1}_{C \times D} = (\mathbf{1}_C, \mathbf{1}_D)$

**Definition 3.7** (Bifunctor)**.** Bifunctor is a Functor whose Domain is a Category Product.

**Definition 3.8** (Terminal object in **Cat** category)**.** Let consider $\Delta_c$ is a trivial functor from Category $\mathbf{A}$ to category $\mathbf{C}$ such that $\forall a \in \mathrm{ob}(\mathbf{A}) : \Delta_c a = c$ -fixed object in $\mathbf{C}$ and $\forall f \in \mathrm{hom}(\mathbf{A}) : \Delta_c f = \mathbf{1}_c$.

**Definition 3.9** (Contravariant functor)**.** If we have a categories $\mathbf{C}$ and $\mathbf{D}$ then the Functor $\mathbf{C^{op}} \to \mathbf{D}$ is called *contravariant functor*.

**Definition 3.10** (Profunctor)**.** If we have a category $\mathbf{C}$ then the Bifunctor $\mathbf{C^{op}} \times \mathbf{C} \to \mathbf{C}$ is called *profunctor*.

## 3.2 Curry-Howard-Lambek correspondence

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

TBD

## 3.3 Monoidal category

**Definition 3.11** (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: $\forall a, b \in M\colon\ a \circ b \in M$
2. Associativity: $\forall a, b, c \in M\colon\ a \circ (b \circ c) = (a \circ b) \circ c$
3. Identity element: $\exists e \in M$ such that $\forall a \in M\colon\ e \circ a = a \circ e = a$

TBD

## 3.4 Examples

### 3.4.1 Set category

TBD

### 3.4.2 Programming languages

#### Hask category

TBD

**Example 3.12** (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 3.13** (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 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 3.14** (Contravariant functor)**.** [**Hask**] TBD

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

**Example 3.15** (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**

TBD

### 3.4.3 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 $F : \mathbf{C} \to \mathbf{D}$ and $G : \mathbf{C} \to \mathbf{D}$. If we have an Object $a \in \mathrm{ob}(\mathbf{C})$ then it will be translated by different functors into different objects of category $\mathbf{D}$: $a_F = Fa, a_G = Ga \in \mathrm{ob}(\mathbf{D})$ (see fig. 4.1). There are 2 options possible

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

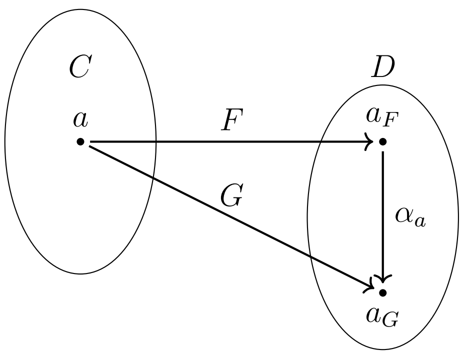


Figure 4.1: Natural transformation: object mapping

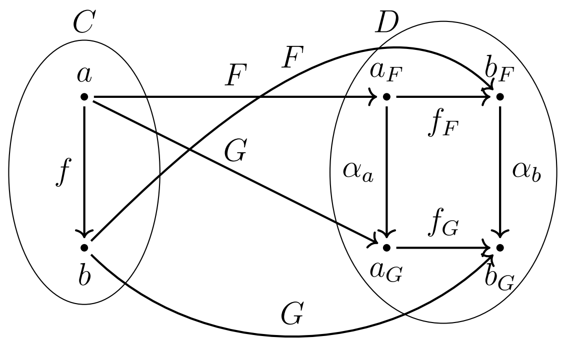


Figure 4.2: Natural transformation: morphisms mapping

We can of course to create an artificial morphism that connects the objects but if we use *natural* morphisms [1] 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 $f \in \hom(a, b)$. 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: $\alpha_a, \alpha_b, f_F, f_G \in \hom(\mathbf{D})$. It is natural to impose additional conditions on the morphisms especially that they form a Commutative diagram:

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

TBD

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

- For every Object $a \in \mathrm{ob}(\mathbf{C})$ $\exists \alpha_a \in \hom(F(a), G(a))$ - Morphism in category $\mathbf{D}$. The morphism $\alpha_a$ is called the component of the natural transformation.

- For every morphism $f \in \hom(\mathbf{C})$ that connects 2 objects $a$ and $b$ , i.e.

[1] the word natural means that already existent morphisms from category $\mathbf{D}$ are used

$$\begin{tikzcd} a_F \arrow[r, "f_F"'] \arrow[d, "\alpha_a"'] & b_F \arrow[d, "\alpha_b"] \\ a_G \arrow[r, "f_G"] & b_G \end{tikzcd}$$

Figure 4.3: Natural transformation: commutative diagram

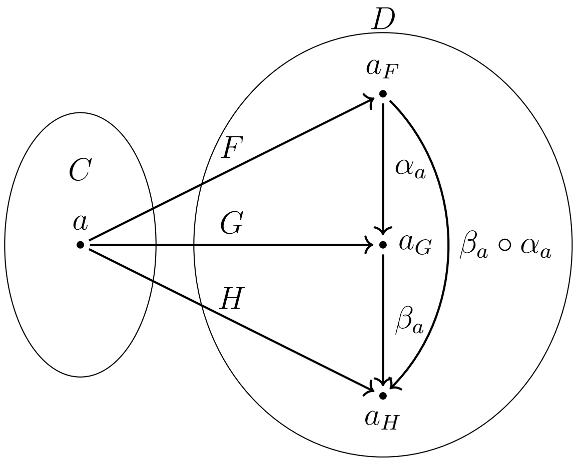


Figure 4.4: Natural transformation composition: object mapping

$f \in \mathrm{hom}(a, b)$ the corresponding components of the natural transformation $\alpha_a, \alpha_b \in \alpha$ should satisfy the following conditions

$$f_G \circ \alpha_a = \alpha_b \circ f_F, \tag{4.1}$$

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 for the natural transformation between fanctors $F$ and $G$: $\alpha : F \dot{\to} G$.

**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 1.7) , identity morphism and verify Associativity (Axiom 1.9) .

$$
\begin{tikzcd}
a_F \arrow[r, "f_F"] \arrow[d, "\alpha_a"] \arrow[dd, bend right=60, "\beta_a \circ \alpha_a"'] & b_F \arrow[d, "\alpha_b"] \arrow[dd, bend left=60, "\beta_b \circ \alpha_b"] \\
a_G \arrow[r, "f_G"] \arrow[d, "\beta_a"] & b_G \arrow[d, "\beta_b"] \\
a_H \arrow[r, "f_H"] & b_H
\end{tikzcd}
$$

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

For the composition consider 2 Natural transformations $\alpha, \beta$ and consider how they act on an object $a \in \mathrm{ob}(\mathbf{C})$ (see fig. 4.4). We always can construct the composition $\beta_a \cdot \alpha_a$ i.e. we can define the composition of natural transformations $\alpha, \beta$ as $\beta \cdot \alpha = \{\beta_a \circ \alpha_a | a \in \mathrm{ob}(\mathbf{C})\}$, note that we use $\cdot$ and not $\circ$ for the composition definition.

The natural transformation is not just object mapping but also morphism mapping. We will require that all morphisms (see fig. 4.5) commutes.

TBD

## 4.2 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 4.3** (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 4.4** (Ad-hoc polymorphism)**.** Polymorphism is parametric if the function instances can behave differently dependently on the type they are being instantiated with.

**Theorem 4.5** (Reynolds)**.** *Parametrically polymorphic functions are Natural transformations*

*Proof.* TBD ☐

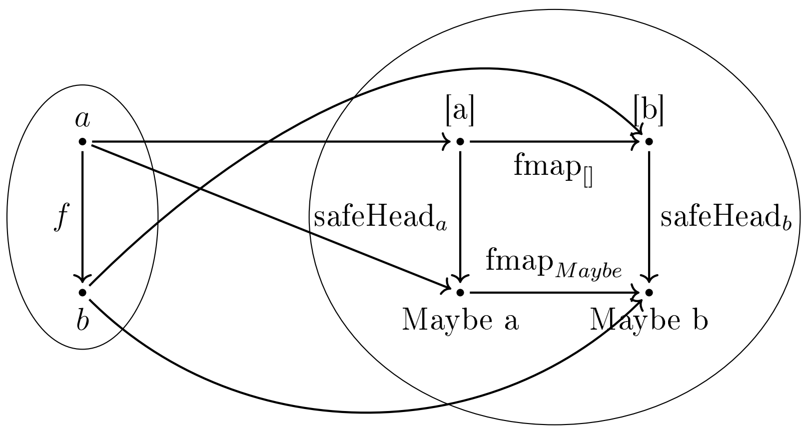


Figure 4.6: Haskell parametric polymorphism as a natural transformation

### 4.2.1 Hask category

In Haskell the most functions are Parametrically polymorphic functions [2].

**Example 4.6** (Parametrically polymorphic function)**.** [**Hask**] Consider the following function

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

The function is parametricaly polymorphic and by Reynolds (Theorem 4.5) is Natural transformation (see fig. 4.6).

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

[2]really in the run-time the functions are not Parametrically polymorphic functions

```
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. [3]

## 4.3 Examples

### 4.3.1 Set category

TBD

### 4.3.2 Programming languages

TBD

[3] It is not directly applied to Haskell because it lazy evaluation that can perform optimization before that one

# Chapter 5

# Monads

TBD

# Index

# 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), 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] 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.

[4] Murashko, I. Category theory. — https://github.com/ivanmurashko/articles/tree/master/cattheory/src. — 2018.

[5] ProofWiki. Empty mapping is unique / ProofWiki. — 2018. — https://proofwiki.org/wiki/Empty_Mapping_is_Unique.

[6] ProofWiki. Injection iff monomorphism in category of sets / ProofWiki. — 2018. — https://proofwiki.org/wiki/Injection_iff_Monomorphism_in_Category_of_Sets.

[7] ProofWiki. Surjection iff epimorphism in category of sets / ProofWiki. — 2018. — https://proofwiki.org/wiki/Surjection_iff_Epimorphism_in_Category_of_Sets.

[8] 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.

[9] 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.

[10] 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%93Fraenkel_set_theory&oldid=852467638.

[11] Мурашко И. В. Квантовая оптика / Мурашко И. В. — 2018. — https://github.com/ivanmurashko/lectures/blob/master/pdfs/qo.pdf.