Category Theory by Example

Ivan Murashko, Alexey Radkov, Marat Minshin

May 22, 2026

Contents

Notations

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

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

αH Left whiskering

α, β Natural transformation (Greek small letters)

$\alpha : F \dot{\to} G$ Natural transformation (arrow with dot)

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

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

$\mathrm{Hom}_C(-, a)$ Contravariant Hom functor

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

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

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

C++ **C++** category

$\mathbf{C_M}$ Kleisli category

Cat **Cat** category

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

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

FdHilb **FdHilb** category

Hask **Hask** category

Proof **Proof** category

Rel **Rel** category

Scala **Scala** category

Set **Set** category

cod f Codomain

$\Delta \downarrow F$ Category of cones to F

Δ_c Constant functor

dom f Domain

$\exists$ exists

$\forall$ for all

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

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

$\mathbf{1}_{a \to a}$ Identity morphism

$\mathbf{1}_{F \dot{\to} F}$ Identity natural transformation

Im f Image of the function f

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

$|A|$ Cardinality of a Set A

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

$a \cong b$ there is an Isomorphism between a and b. The exact isomorphism does not matter in the case

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

$a \oplus b$ Sum

$a \times b$ Product

a, b Objects (Latin small letters)

a^b Exponential

$curry(f)$ Currying

$eq\,(f, g)$ Equalizer

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

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

$F \downarrow \Delta$ Category of co-cones from F

F, G Functor (capital Latin letter)

f, g, h Morphism (Arrow) (Latin small letter)

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

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

$f : a \to b$ Morphism (simple arrow)

$H\alpha$ Right whiskering

$P \implies Q$ Implication

TBD To Be Defined (later)

Introduction

You just looked at yet another introduction to Category Theory. The subject mostly consists of a lot of definitions that are related to each other. We wrote the book to collect all the definitions in one place to be checked and updated easily in the future when we decide to refresh our knowledge about the field of mathematics. Thus the book was written mostly for our category theory study purposes, but we shall appreciate it if somebody else finds it useful.

The topics (chapters) cover base definitions (Object, Morphism and Category) as well as more advanced ones (Functor, Natural transformation, Monad) and also include important results from category theory such as Yoneda's lemma (see chapter 9) and Curry-Howard-Lambek correspondence (see chapter 3). The chapter 10 gives an introduction to topos theory i.e. just another view of Sets.

There are a lot of examples in each chapter. The examples cover different category theory application areas. We assume that the reader is familiar with the corresponding area and the examples can be skipped otherwise. I.e. anyone can choose suitable examples for themselves.

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

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

- Haskell: [9]
- Scala: [10]
- C++: [8]

The examples from physics are related to quantum mechanics that is the best known to us. For the examples we were inspired by the Bob Coecke article [1].

There is also additional material related to abstract algebra (see chapter A) taken from [16]. The material describes the different math constructions used in the book.

The text is distributed under **Creative Common Public License** (see the text of the license at the end of the book) i.e. any reader has the right to copy, store, modify, distribute or build upon the book as long as the original authors are mentioned in the derivative products. The initial text of the book can be found at [17].

Chapter 1

Base definitions

1.1 Definitions

The chapter provides you with base definitions that will be widely used in the book later. We will give definitions for Object, Morphism, Category and also provide you with several important examples, such as the **Set** category.

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 an 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).

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

$$f : a \to b$$

is called a *morphism*. A morphism assumes a direction i.e. one Object (a) is called the *source* or Domain and another one (b) the *target* or Codomain.

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

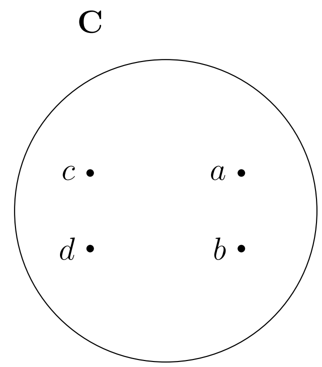

Figure 1.1: Class of objects $\mathrm{ob}(\mathbf{C}) = \{a, b, c, d\}$

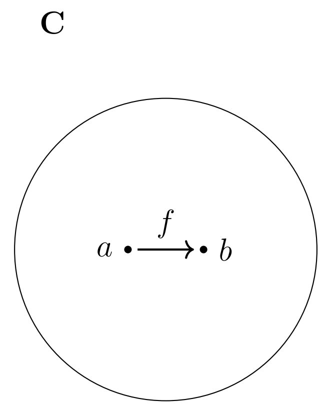

Figure 1.2: Morphism (arrow) $f : a \to b$

Definition 1.5 (Arrow)**.** Morphisms are also called *Arrows* (see fig. 1.2).

The important remark about morphisms is below.

Remark 1.6 (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 $f_1 : a \to b$ and $f_2 : a \to b$ are two 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 two functions $f_1(x) = x, f_2(x) = -x$ define two different orderings on $\mathbb{R}$ and as a result have not to be considered as the same Morphisms.

Definition 1.7 (Domain)**.** Given a Morphism $f : a \to b$, the Object a is called *domain* and denoted as $\operatorname{dom} f$.

Definition 1.8 (Codomain)**.** Given a Morphism $f : a \to b$, the Object b is called *codomain* and denoted as $\operatorname{cod} f$.

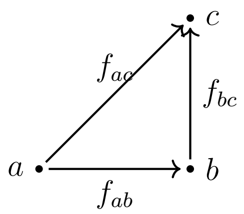

Figure 1.3: Composition $f_{ac} = f_{bc} \circ f_{ab}$

Morphisms have several properties. [1]

Axiom 1.9 (Composition)**.** *If there are 3 Objects a, b, c and 2 Morphisms*

$$f_{ab} : a \to b,$$
$$f_{bc} : b \to c$$

then there exists a Morphism (see fig. 1.3)

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

such that

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

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

Remark 1.10 (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 $f_{ab}(x) \in b$, $f_{ac}(x) \in c$.

Axiom 1.11 (Associativity)**.** *The Morphism Composition (Axiom 1.9) 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 1.12 (Identity morphism)**.** For every Object a there is a special Morphism $\mathbf{1}_{a \to a} : a \to a$ with the following properties: $\forall f_{ba} : b \to a$ (see fig. 1.4)

$$\mathbf{1}_{a \to a} \circ f_{ba} = f_{ba} \tag{1.1}$$

and $\forall f_{ab} : a \to b$ (see fig. 1.5)

$$f_{ab} \circ \mathbf{1}_{a \to a} = f_{ab}. \tag{1.2}$$

This morphism is referred to as *identity morphism*.

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

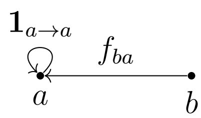

Figure 1.4: Identity morphism property: $\mathbf{1}_{a \to a} \circ f_{ba} = f_{ba}$

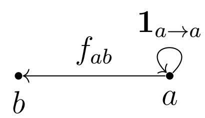

Figure 1.5: Identity morphism property: $f_{ab} \circ \mathbf{1}_{a \to a} = f_{ab}$

Definition 1.13 (Diagram)**.** A *diagram* in a category **C** is a collection of vertices and directed edges where the vertices correspond to the objects of **C** and edges consistently correspond to the morphisms (see 1.6).

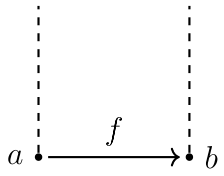

Figure 1.6: Diagram in category **C**. Vertexes a and b are objects in the category **C**, f is a morphism from the category.

Consistently means that for an edge named as f has endpoints labeled a and b, where f is a morphism of **C**, a is Domain of f and b is Codomain of f.

Definition 1.14 (Commutative diagram)**.** A Diagram of category **C** is said to *commute* if all directed paths in the diagram with the same start and endpoint lead to the same result by composition.

Example 1.15. The trivial example of Commutative diagram is Composition (Axiom 1.9) for $f_{ab} = f_{cb} \circ f_{ac}$:

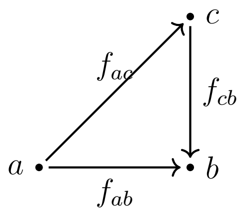

Figure 1.7: Commutative diagram for composition $f_{ab} = f_{cb} \circ f_{ac}$

Remark 1.16 (Class of Morphisms)**.** The Class of Morphisms will be marked as $\hom(\mathbf{C})$ (see fig. 1.8)

Definition 1.17 (Monomorphism)**.** If $\forall g_1, g_2$ the equation

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

leads to

$$g_1 = g_2$$

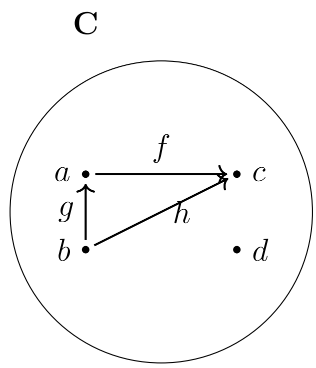

Figure 1.8: Class of morphisms $\hom(\mathbf{C}) = \{f, g, h\}$, where $h = f \circ g$

then f is called *monomorphism* (see fig. 1.9). The monomorphism between a and b is denoted as $f : a \hookrightarrow b$ (see also Injection or "one-to-one" functions).

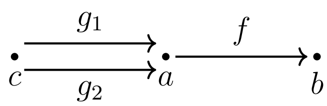

Figure 1.9: Monomorphism $f : a \hookrightarrow b$: $\forall g_1, g_2$: $f \circ g_1 = f \circ g_2$ leads to $g_1 = g_2$

Definition 1.18 (Epimorphism)**.** If $\forall g_1, g_2$ the equation (see fig. 1.10)

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

leads to

$$g_1 = g_2$$

then f is called *epimorphism* (see also Surjection or "onto" functions).

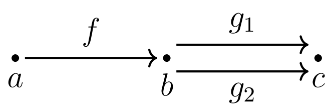

Figure 1.10: Epimorphism f: $g_1 \circ f = g_2 \circ f$ leads to $g_1 = g_2$

Definition 1.19 (Isomorphism)**.** A Morphism $f : a \to b$ is called *isomorphism* if $\exists g : b \to a$ such that $f \circ g = \mathbf{1}_{b \to b}$ and $g \circ f = \mathbf{1}_{a \to a}$. If there is an isomorphism f between objects a and b then it is denoted by $a \cong_f b$.

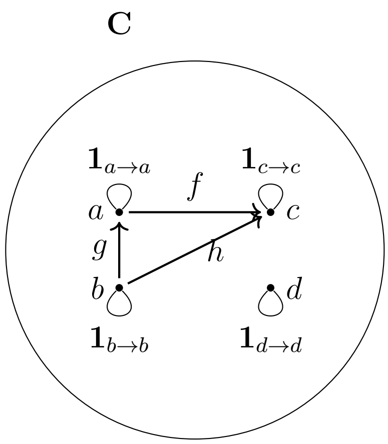

Figure 1.11: Category $\mathbf{C}$. It consists of 4 objects $\mathrm{ob}(\mathbf{C}) = \{a, b, c, d\}$ and 7 morphisms $\mathrm{hom}(\mathbf{C}) = \{f, g, h = f \circ g, \mathbf{1}_{a\to a}, \mathbf{1}_{b\to b}, \mathbf{1}_{c\to c}, \mathbf{1}_{d\to d}\}$

Remark 1.20 (Isomorphism)**.** There can be many different Isomorphisms between 2 Objects.

If there is a unique isomorphism between 2 objects a and b then the objects can be treated as the same object for categorical purposes, although they are not necessarily equal as objects.

1.1.3 Category

Definition 1.21 (Category)**.** A category $\mathbf{C}$ consists of

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

For any Object a there should be unique Identity morphism $\mathbf{1}_{a\to a}$. Any morphism should satisfy Composition (Axiom 1.9) and Associativity (Axiom 1.11) axioms (see example in fig. 1.11).

Definition 1.22 (Set of morphisms)**.** The set of morphisms between objects a and b in the category $\mathbf{C}$ will be denoted as $\mathrm{hom}_{\mathbf{C}}(a, b)$. The set will be denoted as $\mathrm{hom}(a, b)$ if the exact category does not matter.

The Category can be considered as a way to represent a structured data. Objects are the data and Morphisms form the structure that connects the data.

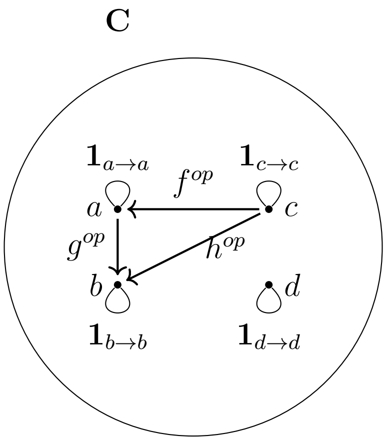

Figure 1.12: Opposite category C^{op} to the category from fig. 1.11 . It consists of 4 objects $\text{ob}(\mathbf{C^{op}}) = \text{ob}(\mathbf{C}) = \{a, b, c, d\}$ and 7 morphisms $\text{hom}(\mathbf{C^{op}}) = \{f^{op}, g^{op}, h^{op} = g^{op} \circ f^{op}, \mathbf{1}_{a\to a}, \mathbf{1}_{b\to b}, \mathbf{1}_{c\to c}, \mathbf{1}_{d\to d}\}$

Definition 1.23 (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 $\text{dom}\, f = a, \text{cod}\, f = b$ (see Domain, Codomain), 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. 1.12)

Remark 1.24. Composition on C^{op} As you can see from fig. 1.12 the Composition (Axiom 1.9) 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 1.25 (Small category)**.** A category $\mathbf{C}$ is called *small* if both $\text{ob}(\mathbf{C})$ and $\text{hom}(\mathbf{C})$ are Sets

Definition 1.26 (Locally small category)**.** A category $\mathbf{C}$ is called *locally small* if for every 2 Objects $a, b \in \text{ob}(\mathbf{C})$ the collection $\text{hom}_{\mathbf{C}}(a, b)$ of all Morphisms from a to b is a Set. The set is called Homset.

Definition 1.27 (Homset)**.** The *homset* $\text{hom}_{\mathbf{C}}(a, b)$ is the Set of Morphisms from a to b in a Locally small category.

Definition 1.28 (Large category)**.** A category $\mathbf{C}$ is not Small category then it is called *large*. The example of large category is **Set** category

Definition 1.29 (Empty category)**.** The category that does not contain any Objects and as result does not contain any Morphisms is called *Empty category* [24].

Definition 1.30 (Trivial category)**.** The category that contains only one Object and only one Morphism (Identity morphism) is called *Trivial category.*

There are several examples of categories below that will also be actively used later in the book:

- **Set** category example: see section 1.2
- Programming languages (Haskell, C++, Scala) examples: see section 1.3
- Quantum mechanics example: see section 1.4

1.2 Set category example

The category of sets is the most important example because it connects our usual knowledge about sets with the category theory.

Definition 1.31 (Set)**.** *Set* is a collection of distinct objects. The objects are called the elements of the set.

The set will be denoted by a capital letter in the book, for instance A. The elements of a set will be denoted by small letters: $a \in A$.

Remark 1.32 (Set)**.** The definition of Set was given above is incomplete. There are several additional axioms should be applied for the complete definition. Different sets of axioms can be used. In our case we consider so called *Zermelo–Fraenkel set theory with the axiom of Choice* or *ZFC* [29]. The system of axioms allows us to avoid different logical paradoxes of the set theory, for instance well known Russell's paradox [28].

Definition 1.33 (Cardinality)**.** The number of elements in the Set A is called *cardinality* and is denoted as $|A|$.

Definition 1.34 (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 1.35 (Binary relation)**.** If A and B are 2 Sets then a subset of the Cartesian product $A \times B$ is called as *binary relation* R between the 2 sets, i.e. $R \subset A \times B$.

Example 1.36 (Binary relation)**.** Example of binary relation is shown on fig. 1.13. There is a relation R between 2 sets A and B. The relation maps a_1 into two different values b_1 and b_2.

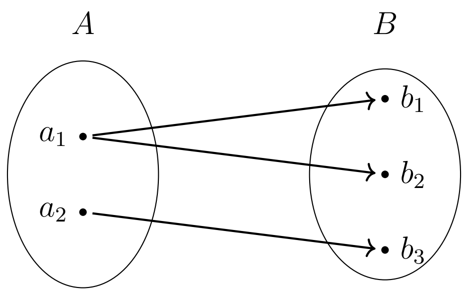

Figure 1.13: Binary relation R between 2 sets A and B. a_1 is mapped into 2 values b_1, b_2

Definition 1.37 (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 $(a, b) \in f$.

Remark 1.38 (Function vs Binary relation)**.** The main difference between Function and Binary relation is that Binary relation allows mapping an argument into more than one value (see fig. 1.13). From other side Function definition does not allow such "multi value".

Definition 1.39 (**Set** category)**.** In the *Set category* we consider a Class of Sets where Objects are the Sets and Morphisms are Functions between the sets. The Identity morphism is the trivial function such that $\forall x \in X : \mathbf{1}_{X \to X}(x) = x$. Composition (Axiom 1.9) is the functions composition (see fig. 1.14)

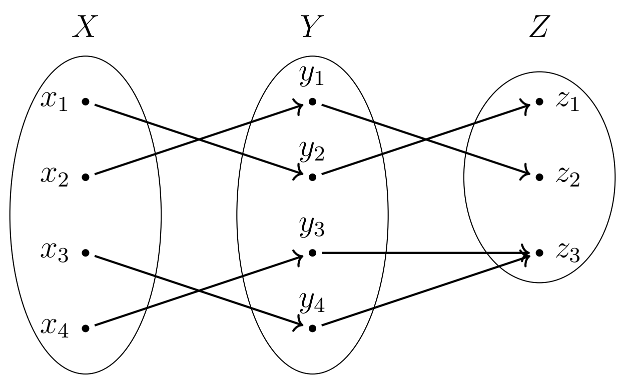

Figure 1.14: Function composition. The function $f_{X\to Z} : X \to Z$ is the composition of 2 functions $f_{X\to Y} : X \to Y$ and $f_{Y\to Z} : Y \to Z$

Remark 1.40 (**Set** category)**.** In general case when we say **Set** category we assume the class of all sets. The class of all sets is not a set itself because famous Russell's paradox [28] can be applied. To avoid such situations we consider a limitation that is applied on our construction as it was mentioned at remark 1.32, especially ZFC [29] is applied. If we take into consideration the limitation then the **Set** category is a Large category.

Definition 1.41 (Singleton)**.** The *singleton* is a Set with only one element.

Example 1.42 (Domain)**.** Given a function $f : X \to Y$, the set X is the domain. I.e. $\operatorname{dom} f = X$

Example 1.43 (Codomain)**.** Given a function $f : X \to Y$, the set Y is the codomain. I.e. $\operatorname{cod} f = Y$

Definition 1.44 (Image)**.** The *image* of a function $f : X \to Y$ is a subset of Codomain Y such that for every element in the subset there is an element in Domain X that maps into the subset:

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

Definition 1.45 (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. 1.15 and 1.19).

Example 1.46 (Surjection)**.** An example of a surjective function is shown in fig. 1.15. Note that the function in the figure is not an Injection. You can find an example of a function that is Surjection as well as Injection (aka Bijection) in fig. 1.19.

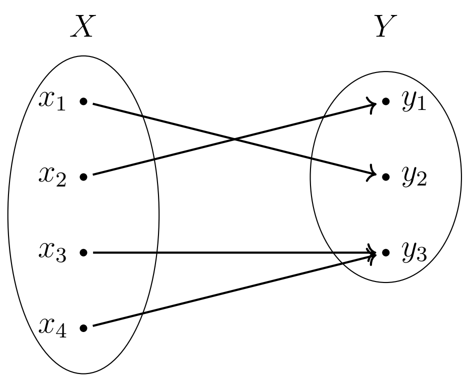

Figure 1.15: A surjective (non-injective) function from domain X to codomain Y

Remark 1.47 (Surjection vs Epimorphism)**.** Surjection and Epimorphism are related each other. Consider a non-surjective function $f : X \to Y$ and let $Y' \subset Y$ be the image of f (see fig. 1.16). One can conclude that f is not an Epimorphism. Let $Z = \{0, 1\}$. We can define two functions $g_1, g_2 : Y \to Z$ such that $g_1(y) = g_2(y)$ for all $y \in Y'$ but $g_1 \neq g_2$ on at least one element of $Y \setminus Y'$. As soon as $f(x) \in Y'$ for all $x \in X$, we always have $g_1(f(x)) = g_2(f(x))$. I.e.

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

but $g_1 \neq g_2$. Thus every Epimorphism in **Set** has to be a Surjection. Conversely, every Surjection is an Epimorphism in the **Set** category. The full proof can be found at [21].

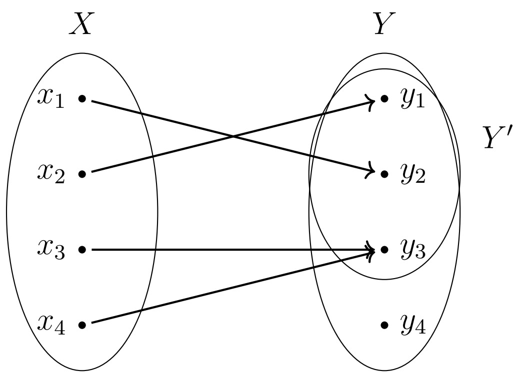

Figure 1.16: Surjection vs epimorphism: A non-surjective function f from domain X to codomain Y. The image of f is $Y' \subset Y$. $\exists g_1, g_2 : Y \to \{0, 1\}$ such that $g_1(y) = g_2(y), \forall y \in Y'$, but $g_1 \neq g_2$. Using the fact that $f(x) \in Y'$ for all $x \in X$ we get $g_1 \circ f = g_2 \circ f$. I.e. the function f is not an epimorphism.

Definition 1.48 (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. 1.17 and 1.19).

Example 1.49 (Injection)**.** An example of an injective function is shown in fig. 1.17. Note that the function in the figure is not a Surjection. You can find an example of a function that is Surjection as well as Injection (aka Bijection) in fig. 1.19.

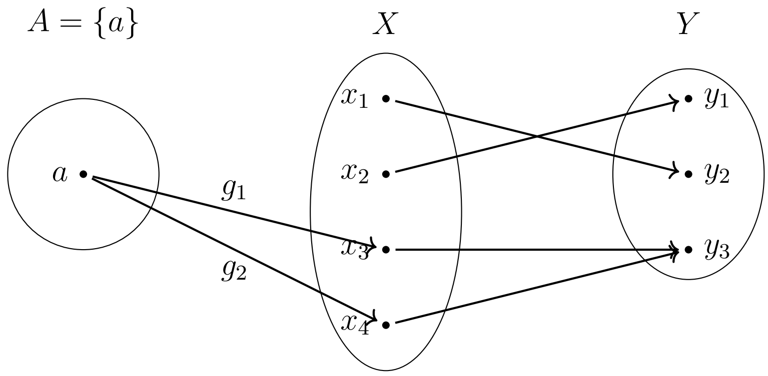

Figure 1.18: A non-injective function f from domain X to codomain Y. $\exists g_1, g_2 : A \to X$ such that $g_1 \neq g_2$ but $f \circ g_1 = f \circ g_2$. I.e. the function f is not a monomorphism.

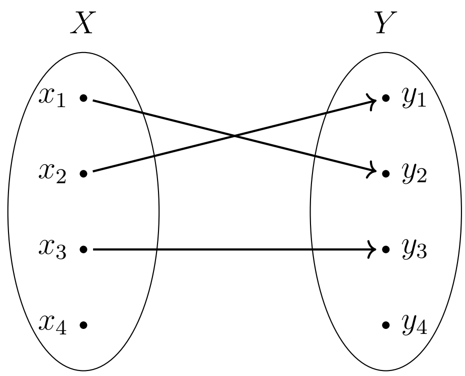

Figure 1.17: A injective (non-surjective) function from domain X to codomain Y

Remark 1.50 (Injection vs Monomorphism)**.** Injection and Monomorphism are related each other. Consider a non-injective function $f : X \to Y$ (see fig. 1.18). One can conclude that it is not a Monomorphism because there are $x_1, x_2 \in X$ such that $x_1 \neq x_2$ but $f(x_1) = f(x_2)$. Let $A = \{a\}$. We can define two functions $g_1, g_2 : A \to X$ such that $g_1(a) = x_1$ and $g_2(a) = x_2$. Thus $g_1 \neq g_2$ but

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

Thus every Monomorphism in **Set** has to be an Injection. Conversely, every Injection is a Monomorphism in the **Set** category. The full proof can be found at [20].

Definition 1.51 (Bijection)**.** The function $f : X \to Y$ is bijective (or "one-to-one" correspondence) if it is an Injection and a Surjection (see fig. 1.19).

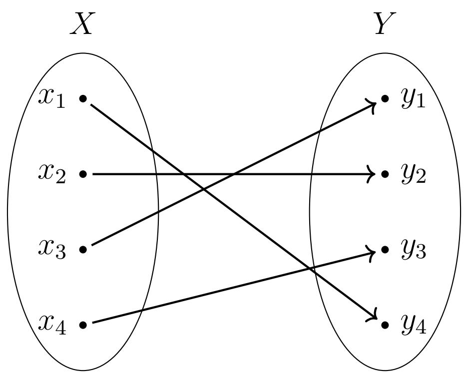

Figure 1.19: An injective and surjective function (bijection)

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

Definition 1.52 (Discrete category)**.** Discrete category is a Category where Morphisms are only Identity morphisms.

1.3 Programming languages examples

Functions are the most important constructions in programming languages. They convert elements [2] of one type into elements of another one.

Definition 1.53 (Pure function)**.** The function is pure if it's execution give the same results independently from the environment.

Categorical view to programming languages assumes 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 1.54) below).

Remark 1.54 (Haskell lazy evaluation)**.** Each Haskell type has a special value $\perp$. The fact that the value and the lazy evaluations are parts of the

[2]We consider variables as the elements in Scala and C++. Constant applicative forms (CAF) are considered as the elements in Haskell

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 [3]

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

i.e. (1.1) and (1.2) are not satisfied.

In the example we used the **seq** function that has the following signature

```
seq :: a -> b -> b
```

i.e. it takes two arguments and returns the second one. It also evaluates the first argument:

$$seq(\perp, b) = \perp,$$
$$seq(a, b) = b.$$

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, not lazy, evaluations (for both Haskell and Scala). Lets take the fact into consideration and define categories for 3 languages

Definition 1.55 (**Hask** category)**.** The objects in the **Hask** category are Haskell types and morphisms are functions. We use only strict (not lazy) evaluations for functions in the category.

Definition 1.56 (**Scala** category)**.** The objects in the **Scala** category are Scala types and morphisms are functions. We don't define functions that have side effects in the category. I.e. the functions are Pure functions. We also use only strict (not lazy) evaluations for functions in the category.

[3] we cannot compare functions in Haskell, but if we could we can get it

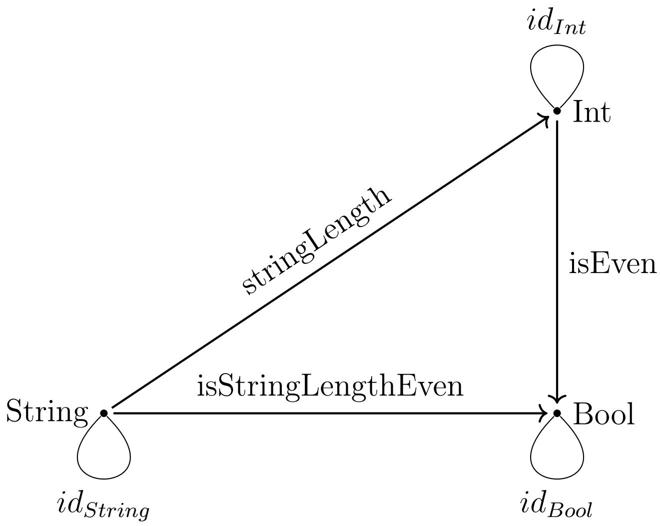

Figure 1.20: Programming language category example. Objects are types: Int, Bool, String. Morphisms are several functions. The diagram assumes $isStringLengthEven = isEven \circ stringLength$.

Definition 1.57 (**C++** category)**.** The objects in the **C++** category are C++ types and morphisms are functions. We don't define functions that have side effects 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 shall look into category with 3 Objects that are types: **Int**, **Bool**, **String**. There are also several Morphisms (functions) between them (see fig. 1.20).

1.3.1 Hask toy category

Example 1.58 (**Hask** toy category)**.** Types in Haskell are considered as Objects. Functions are considered as Morphisms. We are going to implement Category from fig. 1.20.

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 1.9)

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

1.3.2 Scala toy category

Example 1.59 (**Scala** toy category)**.** we shall use the same trick as in **Hask** toy category (Example 1.58) and will assume types in Scala as Objects, functions as Morphisms. We also are going to implement Category from fig. 1.20.

```
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)
    }
  }
```

1.3.3 C++ toy category

Example 1.60 (**C++** toy category)**.** we shall use the same trick as in **Hask toy category** (Example 1.58) and will assume types in C++ as Objects, functions as Morphisms. we shall implement Category from fig. 1.20.

Lets define 2 functions as follows:

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

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

Then we can define composition:

```
// 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 a category until we use pure function (functions without effects).

1.4 Quantum mechanics examples

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

Definition 1.61 (**Rel** category)**.** we shall consider a class of sets (same as **Set** category) i.e. Sets as Objects. Instead of Functions we shall use Binary relations as Morphisms.

The **Rel** category is similar to the finite dimensional Hilbert 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 consist of different elements. Let $f : X \to Y$ - 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 shall get some kind of superposition that is analogous to quantum systems.

In quantum mechanics we use Hilbert spaces that are Vector spaces under Field of complex numbers $\mathbb{C}$.

Definition 1.62 (Hilbert space)**.** The Hilbert space is a complete complex Vector space with an inner product whose values are complex numbers ($\mathbb{C}$).

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

Definition 1.63 (Dual space)**.** Each Hilbert space $\mathcal{H}$ has an associated dual space $\mathcal{H}^*$ that consists of linear functionals.

Example 1.64 (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.

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

$\forall |\phi\rangle \in \mathcal{H}$ 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 1.65 (Linear map)**.** The linear map between 2 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 $a, a_1, a_2 \in \mathcal{A}$ and $f(a), f(a_1), f(a_2) \in \mathcal{B}$.

Remark 1.66 (Linear map)**.** Note that a Linear map does not necessarily preserve the inner product. Let us consider $\mathcal{H}_1 = \mathbb{C}$ with the usual inner product and a linear map $f : \mathbb{C} \to \mathbb{C}$ defined as $f(z) = 2z$. Then

$$\langle f(1)|f(1)\rangle = \langle 2|2\rangle = 4$$

but

$$\langle 1|1\rangle = 1.$$

Thus the equality

$$\langle f(a_1)|f(a_2)\rangle = \langle a_1|a_2\rangle$$

is not a part of the definition of a linear map. It is an additional property.

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

Definition 1.67 (Direct sum of Hilbert spaces)**.** Let $\mathcal{A}, \mathcal{B}$ be 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 1.68 (**FdHilb** category)**.** Most common case in quantum mechanics is the case of quantum states in the finite dimensional Hilbert space. We can consider the class of 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.

Example 1.69 (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. The category in the example will be called as **Rabi**. It has only one Object:

$$\text{ob}(\mathbf{Rabi}) = \{\mathcal{H}_2\}.$$

	Set	**Rel**	**FdHilb**
Object	Set	Set	finite dimensional Hilbert space
Morphism	Function	Binary relation	Linear map
Initial object	empty set	empty set	zero-dimensional Hilbert space
Terminal object	Singleton	empty set	zero-dimensional Hilbert space
Product	Cartesian product	Sum (Example 2.16)	Direct sum of Hilbert spaces
Sum	Sum (Example 2.16)	Sum (Example 2.16)	Direct sum of Hilbert spaces

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

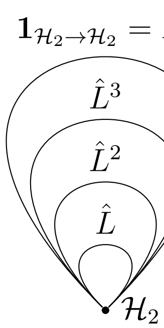

Figure 1.21: Rabi oscillations as a category **Rabi**

The atom interacts with light beam of frequency $\omega = \omega_{ab}$. The state of the system is described by the following equation [33]:

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

where ω_R - Rabi frequency [33].

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 Morphisms:

$$\hom(\mathbf{Rabi}) = \{\mathbf{1}_{\mathcal{H}_2 \to \mathcal{H}_2}, \hat{L}, \hat{L}^2, \hat{L}^3\}.$$

The corresponding orbit of the initial state $|\psi\rangle_0$ is given as follows:

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

1.5 Categorical approach

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" [14]. For instance the definitions for Injection and Surjection are given in the terms of internal objects (sets) structure.

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" [14] there. For the instance in the remark 1.47 we concluded that Epimorphism is a categorical definition for Surjection. The same conclusion for relation between Injection and Monomorphism was made in remark 1.50.

Many constructions can be defined in the 2 different ways: via local (via "microscope") or global (via "telescope") approach. This gives us the following definitions.

Definition 1.70 (Categorical approach)**.** The description of a system (object) via its relations with other systems (objects) will be called as *categorical approach* in the book. This description is an alternative one to an ordinary system description via its internal structure.

Definition 1.71 (Non-categorical approach)**.** The opposite to Categorical approach will be called as *non-categorical approach* in the book. This description is an ordinary system description via its internal structure.

Categorical approach often uses so called *Universal property* to define different constructions. There is an informal definition of the property below

Definition 1.72 (Universal property)**.** In category theory we can highlight constructions when they follow an unique pattern. There can be a lot of such constructions. We pick up the best one via a criteria that can vary for different definitions. The criteria that is used to separate a particular categorical construction from a huge amount of similar ones is called *Universal property*.

Typical examples of the universal property application are Product and Sum definitions. You can find the definitions later in the book (see section 2.3). The Universal property seems to be broadly used in different areas. There are several examples of such Universal property usage (see section 1.5.2) and Categorical approach (see section 1.5.1) below.

1.5.1 Programming languages

There are two basic options possible in programming language. You can use an imperative approach to implement requested functionality or declarative (functional) one. Lets illustrate it on the factorial calculation example.

The factorial can be defined in 2 forms. The first one assumes direct instruction on how to calculate it:

$$n! = \prod_{i=1}^{n} i, \tag{1.3}$$

another one gives you a formal definition for the function:

$$\begin{aligned} n! &= n \cdot (n-1)!, \\ 0! &= 1 \end{aligned} \tag{1.4}$$

Straightforward approach to resolve the task is demonstrated by C++ language in implementing (1.3):

```
int f(int n) {
  if (n < 0) {
    throw std::invalid_argument(
      "the argument has to be greater or equal 0");
  }
  int res = 1;
  for (int i = 1; i <= n; ++i) {
    res *= i;
  }
  return res;
}
```

The solution requires provide all details about the internal structure of the solution i.e. which variables to be used and how to calculate result using them. Thus the approach can be considered as a variation of Non-categorical approach.

Another case assume that the formal definition of the function is provided without any internal details about the implementation. I.e. the (1.4) is used. A functional language such as Haskell is a good candidate for the implementation:

```
-- Factorial
f 0 = 1
f n = n * f (n - 1)
```

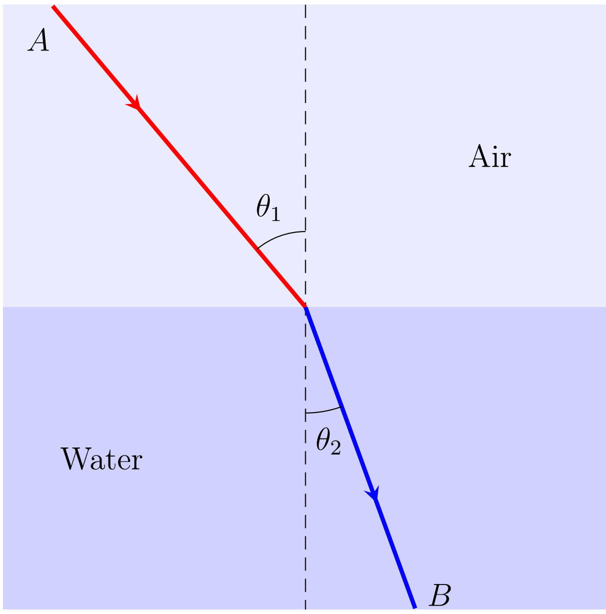

Figure 1.22: Optics refraction as an example of Universal property in optics

1.5.2 Physics

Many physical concepts (may be every one) can be formulated in 2 ways. The first one uses differential equations (aka local approach). The second one uses integral equations (aka global approach). The first case is similar to the "microscope" usage. The second one is the similar to "telescope" approach.

Optics

Optics is another good example of Universal property in physics. In optics it can be reformulated as follows

Remark 1.73 (Universal property)**.** [Optics] A light beam chooses a path that requires the minimal amount of time to path through it.

Good example of the property is shown in fig. 1.22. When the light traverse from point A to point B it does not use the shortest path because a big part of the path will be in water where speed of the light (v_2) is smaller than in air (v_1):

$$v_2 < v_1$$

Thus if it follows the Universal property then it should minimize the path in water that leads to well known Snell's law [32]:

$$\frac{\sin \theta_2}{\sin \theta_1} = \frac{v_2}{v_1} = \frac{n_1}{n_2}.$$

Classical mechanics

we shall consider a motion of a classical mechanical system there. The system consists of n particles. Each particle with number $i \in 1, \dots, n$ has coordinate $q_i \in \mathbb{R}^3$ that changes with time. The set $q_1(t), \dots, q_n(t)$ defines the trajectory. Equations of classical mechanics, that define the trajectory, can be written in different forms. we shall consider Lagrangian form and least action form there. The key point for both is *Lagrangian* that can be written [4] as

$$L = T - U,$$

where T is kinetic energy and U is the potential energy. The Lagrangian is the function of particles positions $\{q_i\}$, velocities $\{\dot{q}_i\}$ and time t. For the case of n particles we can get the following form of Lagrangian:

$$L = L\left(q_1, \dots, q_n, \dot{q}_1, \dots, \dot{q}_n, t\right).$$

The motion equation can be written in the following form:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}_i}\right) = \frac{\partial L}{\partial q_i} \tag{1.5}$$

Example 1.74 (Newton's second law for a particle)**.** Lets consider a single particle motion on the force field $F = -\frac{dU}{dx}$. The kinetic energy is

$$T = \frac{m\dot{x}^2}{2}$$

and Lagrangian

$$L = T - U = \frac{m\dot{x}^2}{2} - U(x).$$

Thus (1.5) gives us

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{x}}\right) = \frac{\partial L}{\partial x}$$

or

$$\frac{d}{dt}\left(m\dot{x}\right) = m\ddot{x} = -\frac{dU}{dx} = F$$

that is the famous Newton's second law for a particle.

The (1.5) is an example of local approach when the knowledge about local properties (this knowledge leads to the differential equation) gives us the motion equation i.e. the equation has been got using a "microscope" there.

[4] non-relativistic case

Another way to investigate the motion is the principle of least action. In the principle we consider all possible trajectories of our system between time points t_1 and t_2. For each trajectory $\mathbf{q}(t) = q_1(t), \dots, q_n(t), t \in [t_1, t_2]$ we can define the following integral

$$S(\mathbf{q}, t_1, t_2) = \int_{t_1}^{t_2} L(\mathbf{q}(t), \dot{\mathbf{q}}(t), t)\, dt, \tag{1.6}$$

that is called as *Action*. The principle of least action states that the trajectory taken by the system between times t_1 and t_2 is the one for which the action is stationary (no change) to first order [31] i.e.

$$\delta S = 0.$$

We can rewrite the principle as follows

$$\begin{aligned} \delta S = \int_{t_1}^{t_2} \delta L(\mathbf{q}(t), \dot{\mathbf{q}}(t), t)\, dt = \\ = \int_{t_1}^{t_2} \left[L(\mathbf{q} + \delta\mathbf{q}, \dot{\mathbf{q}} + \delta\dot{\mathbf{q}}, t) - L(\mathbf{q}, \dot{\mathbf{q}}, t) \right] dt = \\ = \int_{t_1}^{t_2} \left[\frac{\partial L}{\partial \mathbf{q}} \delta\mathbf{q} + \frac{\partial L}{\partial \dot{\mathbf{q}}} \delta\dot{\mathbf{q}} \right] dt = \int_{t_1}^{t_2} \frac{\partial L}{\partial \mathbf{q}} \delta\mathbf{q} dt + \\ + \left. \frac{\partial L}{\partial \dot{\mathbf{q}}} \delta\mathbf{q} \right|_{t_1}^{t_2} - \int_{t_1}^{t_2} \frac{d}{dt} \left(\frac{\partial L}{\partial \dot{\mathbf{q}}} \delta\mathbf{q} \right) dt = \\ = \int_{t_1}^{t_2} \left[\frac{\partial L}{\partial \mathbf{q}} - \frac{d}{dt} \left(\frac{\partial L}{\partial \dot{\mathbf{q}}} \right) \right] \delta\mathbf{q} dt = 0, \end{aligned}$$

that leads to (1.5). Therefore the same motion equation can be used using global approach (via integral over all possible trajectories) or in other words the "telescope" was used there.

Remark 1.75 (Universal property)**.** [Mechanics] The principle of least action can be treated as an universal property that will allow to pick up one object (trajectory) among the set of the similar objects. The same universal properties will appear during the book in a lot of places.

1.5.3 Quantum mechanics

Very interesting example of Categorical approach is provided us by quantum mechanics via path integrals [3]. The question that is asked is the following.

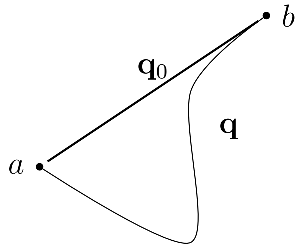

Figure 1.23: Different paths between points a and b. There are a possible trajectory $\mathbf{q}$ and the real one $\mathbf{q}_0$

If we have 2 points a and b, what's probability $P(a, b)$ that a particle moves from a to b? The probability is defined as follows [3]:

$$P(a, b) = |K(a, b)|^2 ,$$

where $K(a, b)$ is a special function defined by all possible paths from a to b as follows

$$K(a, b) = \frac{1}{Z} \int_{\mathbf{q}} e^{\frac{i}{\hbar} S(\mathbf{q})} \mathcal{D}\mathbf{q},$$

where $\mathbf{q}$ is a trajectory from a to b (see fig. 1.23), and S is the action defined by (1.6). For a path x such that $\delta S(\mathbf{q}) > \hbar$ we shall have that a trajectory $\mathbf{q} + \delta\mathbf{q}$ that is similar to $\mathbf{q}$ but gives a completely different action and as result such trajectories will cancel each other.

From other hand the path $\mathbf{q}_0$ such that $\delta S(\mathbf{q}_0) = 0$ will have all other similar trajectories $\mathbf{q}_0 + \delta\mathbf{q}$ will increase each other and as result the $\mathbf{q}_0$ will define the real trajectory for the particle. This is in direct connection with the classical least action principle [31].

Chapter 2

Objects and morphisms

2.1 Equality

The important question is how we can decide whether an object/morphism is equal to another object/morphism. The trivial answer is possible if the Object is a Set. In the case we can say that 2 objects are equal if they contain the equivalent collection of elements. Unfortunately we cannot do the same trick for categorical Objects as soon as they don't have any internal structure, but can use a Categorical approach (see section 1.5) i.e. if we cannot use a "microscope" let's use a "telescope" and define the equality of objects and morphisms of a category $\mathbf{C}$ in terms of the whole $\hom(\mathbf{C})$.

Definition 2.1 (Objects equality and canonical isomorphism)**.** Two Objects a and b in Category $\mathbf{C}$ are equal if they are the same element of the Class $\mathrm{ob}(\mathbf{C})$.

They are isomorphic if there exists a Isomorphism $a \cong_f b$. This also means that there exists an isomorphism $b \cong_g a$. These two Morphisms (f and g) are related to each other via the following equations: $g \circ f = \mathbf{1}_{a \to a}$ and $f \circ g = \mathbf{1}_{b \to b}$.

If there exists a unique isomorphism between a and b, then a and b are canonically isomorphic. In this case the objects can be treated as the same object for categorical purposes, although they are not necessarily equal as objects of $\mathbf{C}$.

Unlike Functions between Sets we don't have any additional info [1] about Morphisms except category theory axioms which the morphisms satisfy [5]. This leads us to the following definition of morphism equality:

[1]for instance info about sets internals. i.e. which elements of the sets are connected by the considered functions

Definition 2.2 (Morphisms equality)**.** Let $f, g : a \to b$ be two Morphisms in Category $\mathbf{C}$. They are equal if they are the same morphism from a to b. If $\mathbf{C}$ is a Locally small category, then this means that they are the same element of the Homset $\hom_{\mathbf{C}}(a, b)$.

In a concrete proof the equality can be derived from the base axioms:

- Composition (Axiom 1.9)
- Associativity (Axiom 1.11)
- Identity morphism: (1.1), (1.2)

or Commutative diagrams that postulate the equality.

As an example let's prove the following theorem.

Theorem 2.3 (Identity is unique)**.** *The Identity morphism is unique.*

Proof. Consider an Object a and its Identity morphism $\mathbf{1}_{a \to a}$. Assume existence of a function $f : a \to a$ such that f is also identity. (1.1), for f as identity, gives us

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

On the other hand (1.2) for $\mathbf{1}_{a \to a}$ is satisfied:

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

i.e.

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

or $f = \mathbf{1}_{a \to a}$. □

2.2 Initial and terminal objects

2.2.1 Initial object

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

Example 2.5 (Initial object)**.** **[Set]** Note that there is only one function from the empty set to any other set [19] that makes the empty set the Initial object in **Set** category.

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

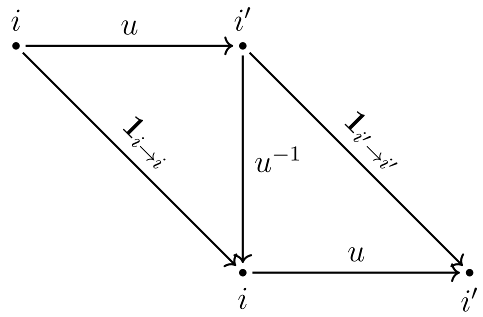

Figure 2.1: Commutative diagram for initial object uniqueness proof

Proof. Consider the following Commutative diagram (see fig. 2.1). Since i is an initial object, $\exists!\, u : i \to i'$. On the other hand i' is also an initial object and therefore $\exists!\, u^{-1} : i' \to i$. Combining them together via composition we can get $u^{-1} \circ u : i \to i$ and $u \circ u^{-1} : i' \to i'$. From the fact that i is initial object one can get that there exists only one morphism $\mathbf{1}_{i \to i} : i \to i$. The same is true for i'. Therefore $u^{-1} \circ u = \mathbf{1}_{i \to i}$ and $u \circ u^{-1} = \mathbf{1}_{i' \to i'}$. This completes the commutative diagram and finishes the proof. □

2.2.2 Terminal object

Definition 2.7 (Terminal object)**.** Let $\mathbf{C}$ is a Category, the Object $t \in \mathrm{ob}(\mathbf{C})$ is called *terminal object* if $\forall x \in \mathrm{ob}(\mathbf{C}) \exists! g_x : x \to t \in \mathrm{hom}(\mathbf{C})$.

Example 2.8 (Terminal object)**.** [**Set**] Terminal object in **Set** category is a set with one element i.e Singleton.

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

Theorem 2.9 (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 and canonical isomorphism)*

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. □

2.2.3 Toy example

Example 2.10 (Toy example)**.** In our toy example fig. 1.20 the type String is Initial object and type Bool is the Terminal object.

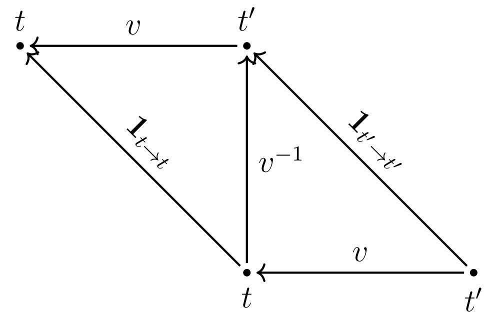

Figure 2.2: Commutative diagram for terminal object uniqueness proof

2.3 Product and sum

2.3.1 Product

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

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

$$\begin{aligned} \pi_1' &= \pi_1 \circ h, \\ \pi_2' &= \pi_2 \circ h. \end{aligned} \tag{2.1}$$

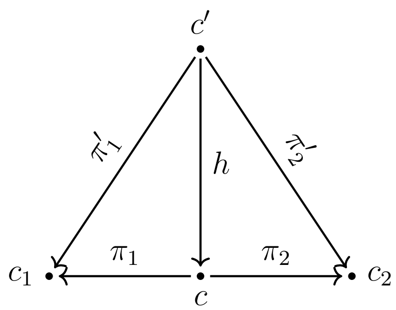

Figure 2.3: Product $c = c_1 \times c_2$. $\forall c', \exists! h \in \text{hom}(\mathbf{C}) : \pi_1' = \pi_1 \circ h, \pi_2' = \pi_2 \circ h$.

In other words h factorizes $\pi_{1,2}'$.

Example 2.12 (Product)**.** [**Set**] Cartesian product: $C = A \times B = \{(a,b)|a \in A, b \in B\}$ is the Product of two sets A and B in **Set** category. We have only one option for $\pi_{1,2}$:

$$\pi_1 : (a,b) \to a \in A,$$
$$\pi_2 : (a,b) \to b \in B.$$

Assume that A and B are non-empty and at least one of them has more than one element. Consider also another candidate: $C' = A \times A \times B \times B = \{(a_1, a_2, b_1, b_2)|a_{1,2} \in A, b_{1,2} \in B\}$. There are different options for π'_1 and π'_2. Lets choose the following ones:

$$\pi'_1 : (a_1, a_2, b_1, b_2) \to a_1 \in A,$$
$$\pi'_2 : (a_1, a_2, b_1, b_2) \to b_2 \in B.$$

We have only one morphism h that satisfied conditions (2.1):

$$h : (a_1, a_2, b_1, b_2) \to (a_1, b_2) \in A \times B$$

that is accordingly with the Product definition for $C = A \times B$.

If C' had been the Product then it would have satisfied the following factorization conditions:

$$\pi_1 = \pi'_1 \circ h',$$
$$\pi_2 = \pi'_2 \circ h', \tag{2.2}$$

where h' would have been an unique morphism. From other side there are a lot of morphisms h' which factorize $\pi_{1,2}$ accordingly (2.2):

$$h' : (a,b) \to (a, \bar{a}, \bar{b}, b),$$

where $\bar{a}$ can be replaced with any element from A and $\bar{b}$ can be replaced with any element of B. Therefore C' can not be considered as the Product of A and B.

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

Definition 2.13 (Product of morphisms)**.** Let **C** is a category and $a, a' \in \text{ob}(\mathbf{C})$ and $b, b' \in \text{ob}(\mathbf{C})$ are 2 pairs of Objects that admit definition 2.11. Consider 2 morphisms that connects the objects: $f : a \to b, f' : a' \to b'$ then we can create a new unique morphism that connects the products: $f \times f' : a \times a' \to b \times b'$ and makes the diagram commute (see fig. 2.4).

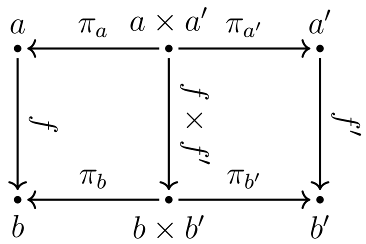

Figure 2.4: Product of morphisms.

2.3.2 Sum

If we invert Arrows in Product we shall got another object definition that is called sum

Definition 2.14 (Sum)**.** Let we have a category **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 **C** $c = c_1 \oplus c_2$ with 2 Morphisms $i_1 : c_1 \to c, i_2 : c_2 \to c$ such that the following Universal property is satisfied: $\forall c' \in \text{ob}(\mathbf{C})$ and morphisms $i'_1 : c_1 \to c', i'_2 : c_2 \to c'$, exists unique morphism h such that the following diagram (see fig. 2.5) commutes, i.e. $i'_1 = h \circ i_1, i'_2 = h \circ i_2$.

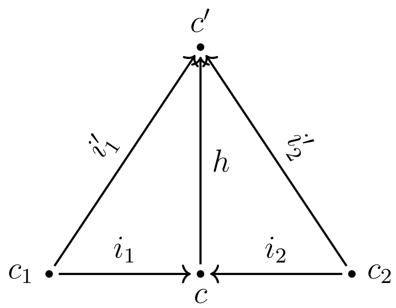

Figure 2.5: Sum $c = c_1 \oplus c_2$. $\forall c', \exists! h \in \text{hom}(\mathbf{C}) : i'_1 = h \circ i_1, i'_2 = h \circ i_2$.

In other words h factorizes $i'_{1,2}$.
The categorical sum is also called as *coproduct*.

Definition 2.15 (Disjoint union)**.** Let $\{A_i : i \in I\}$ be a family of sets indexed by I. The *disjoint union* [25] 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 2.16 (Sum). **[Set]** Disjoint union is the Sum of two sets A and B in **Set** category.

Remark 2.17 (Sum sign). In the book we shall use $\oplus$ as the sign for the categorical Sum. The Disjoint union sign $\sqcup$ is also used [2] as the sign for categorical sum [30].

2.4 Category as a monoid

Consider the following definition from abstract algebra

Definition 2.18 (Monoid). The set of elements M with defined binary operation $\circ$ we shall call as a monoid if the following conditions are satisfied.

1. Closure: $\forall a, b \in M$: $a \circ b \in M$
2. Associativity: $\forall a, b, c \in M$:
$$a \circ (b \circ c) = (a \circ b) \circ c \tag{2.3}$$
3. Identity element: $\exists e \in M$ such that $\forall a \in M$:
$$e \circ a = a \circ e = a \tag{2.4}$$

Example 2.19 (Monoid). Monoid concept is widely spread in math. Especially integer numbers form a monoid under summation operation. They also form another monoid under multiplication operation. The element **0** is used as identity in summation and **1** is used as the identity in multiplication.

Example 2.20 (Monoid). **[Hask]** There is a declaration of **Monoid** in modern **Hask**:

```
class Semigroup m => Monoid m where
    mempty :: m
    mappend :: m -> m -> m
    mappend = (<>)
    mconcat :: [m] -> m
    mconcat = foldr mappend mempty
```

The binary operation is inherited from **Semigroup** and written as **(<>)**, while **mappend** is the compatibility name for the same operation. The identity is **mempty**. As it was mentioned in the Monoid definition (see

[2]but not in the book

definition 2.18), the binary operation should satisfy the associativity (2.3) and identity element (2.4) properties. This is a responsibility of a particular implementation to satisfy the properties. For instance the standard list implementation satisfies them:

```
instance Monoid [a] where
    mappend = (++)
    mempty = []
```

Remark 2.21 (Monoid)**.** The given definition of monoid is based on its internal structure i. e. there is a Non-categorical approach. In section 6.1.3 we shall continue the Monoid concept investigation and will give a Categorical approach of the concept. You can also find there some notes about the concept importance in different areas such as programming languages and math (see section 6.1.4).

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

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

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

Definition 2.22 (Distributive category)**.** A category **C** is *distributive* if [27] it has finite Products and Sums such that $\forall a, b, c \in \mathrm{ob}(\mathbf{C})$ the canonical morphism

$$[\mathbf{1}_{a \to a} \times i_b, \mathbf{1}_{a \to a} \times i_c] : (a \times b) \oplus (a \times c) \to a \times (b \oplus c)$$

is an Isomorphism, where $i_b : b \to b \oplus c$ and $i_c : c \to b \oplus c$ are the sum injections. Also the unique morphism

$$\mathbf{0} \to a \times \mathbf{0}$$

is an Isomorphism, where **0** is the Initial object.

Example 2.23 (Distributive category)**. Set** category is an example [27] of Distributive category

From other hand not all categories which have both product and sum are distributive. One of such example is a category of all groups **Grp** [27] where groups are considered as objects and group homomorphisms as morphisms.

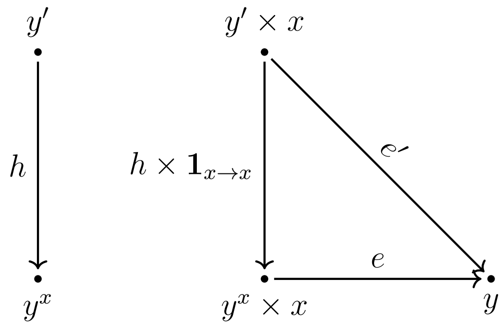

Figure 2.6: Exponential object

2.5 Exponential

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

2.5.1 Definition and examples

Example 2.24 (Homset)**.** Consider 2 sets A and B then the set of functions between the 2 sets forms a new set that is called as Homset and denoted as $\hom(A, B)$. Thus if $A, B \in \text{ob}(\mathbf{Set})$ then $\hom(A, B) \in \text{ob}(\mathbf{Set})$.

The construction of Homset is applied to the **Set** category but not to an arbitrary category because the Homset is a Set and therefore the object in the **Set** category. I.e. if $\mathbf{C}$ is a Locally small category and $a, b \in \text{ob}(\mathbf{C})$ then the Homset $\hom(a, b) \in \text{ob}(\mathbf{Set})$ but for an arbitrary category this collection does not have to be a set. Thus we now want to construct something like to the Homset but that is an object in $\mathbf{C}$. This will be called as the function object. we shall use the universal construction (Universal property) for the object definition.

Definition 2.25 (Exponential)**.** Let $\mathbf{C}$ is a category and $x, y \in \text{ob}(\mathbf{C})$. We also assume that $\mathbf{C}$ allows all Products with x, i.e. $\forall y' \in \text{ob}(\mathbf{C}), \exists y' \times x$. An object y^x together with a Morphism $e : y^x \times x \to y$ is an *exponential object* if for every $y' \in \text{ob}(\mathbf{C})$ and every Morphism $e' : y' \times x \to y$ there exists a unique morphism $h : y' \to y^x$ such that the Commutative diagram shown in fig. 2.6 commutes:

$$e' = e \circ (h \times \mathbf{1}_{x \to x})$$

Example 2.26 (Exponential)**. [Set]** Lets look at the Exponential in **Set**. We want to show that the object corresponds to the function. Really if we want to define a function $f : X \to Y$ then we should look at the Homset $F = \hom(X, Y)$. $f \in F$ - is an element of the Homset. For the function application we have to take the argument $x \in X$ and the function we want to apply $f \in F$. Then we construct the pair $(f, x) \in F \times X$. For the function application we have to call a Morphism $e : F \times X \to Y$. [3] I.e. the application $e(f, x)$ gives us $e(f, x) = y \in Y$ - the function value.

The notation is used for "morphisms (functions) as objects" in the category theory has an explanation provided in the following remark.

Remark 2.27 (Exponential notation)**.** The Homset $\hom(X, Y)$ is often denoted as Y^X. Why the strange notation is used? Lets X is a Singleton i.e. its Cardinality is 1: $|X| = 1$. The set Y has only 2 elements, i.e. its Cardinality is 2: $|Y| = 2$.

Consider a function $f : X \to Y$. How many such functions do we have? There are really 2 functions (see fig. 2.7). One of them f_1 return the first element from Y (y_1) and the other f_2 returns the second one (y_2). The number of functions can be written as 2^1. I.e. one can write for Cardinality of a set of all functions between X and Y as follows

$$\left|Y^X\right| = |Y|^{|X|}. \tag{2.5}$$

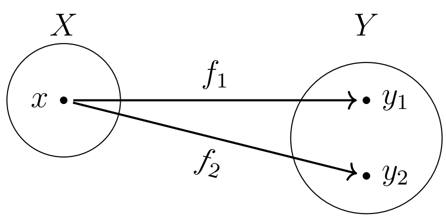

Figure 2.7: $\hom(X, Y)$ consists of 2 elements: $\{f_1, f_2\}$. Thus the cardinality of the homset is 2

Otherwise if we consider a function $g : Y \to X$ then we have only one possible choice for it (see fig. 2.8) : just return the only possible element from X. I.e. $\left|X^Y\right| = 1$ that correlates with (2.5).

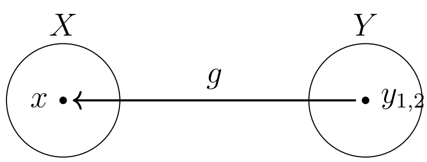

Figure 2.8: $\hom(Y, X)$ consists of 1 element: $\{g\}$. Thus the cardinality of the homset is 1

[3] e from the word "eval"

2.5.2 Currying in Set

In the section we shall be in the **Set** category. The definition of Exponential object is closely related to notion of currying that is defined for Sets as follows.

Definition 2.28 (Currying)**.** Consider a function of 2 arguments:

$$f : (X \times Y) \to Z$$

that maps a pair (Product) of 2 sets X and Y into another set Z. The *currying* constructs a new function

$$h : X \to (Y \to Z)$$

such that the following equation holds

$$h(x)(y) = f(x, y),$$

where $x \in X, y \in Y$. The *currying* will be denoted as $h = curry(f)$.

Remark 2.29 (Currying)**.** If we consider Y^X is a set of functions $f : X \to Y$ then the currying is the Bijection

$$Y^{X_1 \times X_2} \cong \left(Y^{X_2}\right)^{X_1}.$$

Remark 2.30 (Currying and Exponential)**.** If we look in fig. 2.6 then we can notice that

$$curry(e) : y^x \to (x \to y),$$

i.e. currying is used to construct a morphisms from exponential object.

2.5.3 Cartesian closed category

Definition 2.31 (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. $\forall a, b \in \mathrm{ob}(\mathbf{C})$ exists Product $a \times b \in \mathrm{ob}(\mathbf{C})$.
3. $\forall a, b \in \mathrm{ob}(\mathbf{C})$ exists Exponential $a^b \in \mathrm{ob}(\mathbf{C})$

Theorem 2.32 (Cartesian closed category)**.** *If* $\mathbf{C}$ *is a Cartesian closed category with finite Sums then it is a Distributive category.*

Proof. Let $a, b, c, d \in \mathrm{ob}(\mathbf{C})$. We will use the universal property of the Exponential object in the following form for every $y \in \mathrm{ob}(\mathbf{C})$:

$$\hom_{\mathbf{C}}(a \times y, d) \cong \hom_{\mathbf{C}}(y, d^a).$$

Indeed, the definition gives a bijection between morphisms $y \times a \to d$ and morphisms $y \to d^a$. On the other hand, $a \times y$ and $y \times a$ are canonically isomorphic because the Product is defined by a universal property and we can swap the projections.

Let $i_b : b \to b \oplus c$ and $i_c : c \to b \oplus c$ be the sum injections. Then we have morphisms

$$\mathbf{1}_{a\to a} \times i_b : a \times b \to a \times (b \oplus c)$$

and

$$\mathbf{1}_{a\to a} \times i_c : a \times c \to a \times (b \oplus c).$$

Consider an arbitrary object d. We have the following chain of bijections:

$$\begin{aligned}\hom_{\mathbf{C}}(a \times (b \oplus c), d) &\cong \hom_{\mathbf{C}}(b \oplus c, d^a)\\ &\cong \hom_{\mathbf{C}}(b, d^a) \times \hom_{\mathbf{C}}(c, d^a)\\ &\cong \hom_{\mathbf{C}}(a \times b, d) \times \hom_{\mathbf{C}}(a \times c, d).\end{aligned}$$

The first and the last bijections come from the exponential object. The middle bijection comes from the universal property of the Sum. The composed bijection sends a morphism $h : a \times (b \oplus c) \to d$ to the pair

$$(h \circ (\mathbf{1}_{a\to a} \times i_b), h \circ (\mathbf{1}_{a\to a} \times i_c)).$$

Therefore, for every pair of morphisms $f : a \times b \to d$ and $g : a \times c \to d$ there exists a unique morphism $h : a \times (b \oplus c) \to d$ such that

$$f = h \circ (\mathbf{1}_{a\to a} \times i_b)$$

and

$$g = h \circ (\mathbf{1}_{a\to a} \times i_c).$$

Thus $a \times (b \oplus c)$ satisfies the universal property of the sum of $a \times b$ and $a \times c$. The object $(a \times b) \oplus (a \times c)$ satisfies the same universal property by definition. Hence the uniqueness of the sum gives the canonical morphism

$$[\mathbf{1}_{a\to a} \times i_b, \mathbf{1}_{a\to a} \times i_c] : (a \times b) \oplus (a \times c) \to a \times (b \oplus c)$$

as an isomorphism. Therefore

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

It remains to prove the second condition from Distributive category. Let $\mathbf{0}$ be the Initial object. For every $d \in \mathrm{ob}(\mathbf{C})$ we have

$$\hom_{\mathbf{C}}(a \times \mathbf{0}, d) \cong \hom_{\mathbf{C}}(\mathbf{0}, d^a).$$

The last hom-set has exactly one element because $\mathbf{0}$ is initial. Therefore, for every d there exists a unique morphism $a \times \mathbf{0} \to d$. Thus $a \times \mathbf{0}$ is also an initial object. By theorem 2.6, initial objects are canonically isomorphic, and therefore

$$a \times \mathbf{0} \cong \mathbf{0}.$$

Moreover, the unique morphism $\mathbf{0} \to a \times \mathbf{0}$ is the canonical isomorphism between these initial objects. The assumptions already give finite products and finite sums. The two canonical isomorphisms above are exactly the remaining conditions from Distributive category, so $\mathbf{C}$ is distributive. □

2.6 Programming language examples. Type algebra

2.6.1 Hask category

Example 2.33 (Initial object)**.** [**Hask**] If we avoid lazy evaluations in Haskell (see Haskell lazy evaluation (Remark 1.54)) 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 2.34 (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 2.35 (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 2.36 (Sum)**.** **[Hask]** The Sum in **Hask** category defined as follows

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

The typical usage is via pattern matching for instance

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

Example 2.37 (Distributive category)**.** **[Hask]** As soon as **Hask** is a Cartesian closed category then by theorem 2.32 it is a Distributive category i.e. one can conclude that

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

is the same to

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

Example 2.38 (Exponential)**.** **[Hask]** It's not surprisingly that the Exponential in **Hask** is a function object i.e. b^a can be written as **a -> b**.

Example 2.39 (Type algebra)**.** example 2.38 gives interesting results with types manipulations. For instance the type a^{b+c} can be written as

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

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

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

These transformations correspond to the following simple algebraic equation

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

This is also called as *type algebra*.

2.6.2 C++ category

Example 2.40 (Initial object)**.** **[C++]** C++ does not provide a clean standard type that plays the role of an Initial object in our simplified category of types and functions. The type **void** is close to the idea of an empty type, but it cannot be used as an ordinary function argument type, so it does not provide morphisms **void -> A** in the same way as **Void** does in **Hask** category.

If we work in a restricted fragment, we can introduce an uninhabited wrapper type and use it as an approximation:

```
struct Void {
  Void() = delete;
};
```

Since no value of **Void** can be constructed, every pure total function from **Void** to another type is extensionally the same.

Example 2.41 (Terminal object)**.** **[C++]** C++ 17 introduced a special type that keeps only one value - **std::monostate**:

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

Example 2.42 (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 2.43 (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:

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

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;
```

Thus **std::variant<C, D>** plays the role of the sum type $C \oplus D$. The function **factor** corresponds to the universal property of Sum: if we have functions from both alternatives to the same result type, then there is a unique function from the sum type that chooses the correct branch.

2.6.3 Scala category

Example 2.44 (Initial object)**. [Scala]** We used a same trick as for Initial object (Example 2.33) in **Hask** category and define Initial object in **Scala** category as a type without values:

```
Nothing
```

i.e. you cannot construct a value of the type. There is only one function from the initial object:

```
def absurd[A](value: Nothing): A = value
```

Example 2.45 (Terminal object)**. [Scala]** We used a same trick as for Terminal object (Example 2.34) in **Hask** category and define Terminal object in **Scala** category as a type with only one value

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

The only value of the type is written as follows:

```
val unit: Unit = ()
```

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

Example 2.46 (Product)**. [Scala]** The Product in **Scala** category keeps a pair. Scala has a built-in product type as follows:

```
val pair: (Int, Boolean) = (1, true)
val first: Int = pair._1
val second: Boolean = pair._2
```

The values **pair._1** and **pair._2** are the two projectors.

Example 2.47 (Sum)**. [Scala]** The Sum in **Scala** category is represented by **Either**. It keeps a value of the left type or a value of the right type:

```
val left: Either[String, Int] = Left("abc")
val right: Either[String, Int] = Right(4)
```

The universal property of the sum can be written as the following **factor** function:

```
def factor[A, B, C](f: A => C, g: B => C)
                  (either: Either[A, B]): C =
  either match {
    case Left(a) => f(a)
    case Right(b) => g(b)
  }
```

If we have functions from both alternatives to the same result type, then **factor** gives us a unique function from **Either[A, B]** to that result type.

2.7 Quantum mechanics

Example 2.48 (Initial object)**. [FdHilb]** we shall use a Hilbert space of dimension 0 as the Initial object. For every finite dimensional Hilbert space $\mathcal{A}$ there is exactly one linear map $\mathcal{H}_0 \to \mathcal{A}$. It sends the only vector of $\mathcal{H}_0$ to the zero vector of $\mathcal{A}$.

Example 2.49 (Terminal object)**. [FdHilb]** we shall use a Hilbert space of dimension 0 as the Terminal object. For every finite dimensional Hilbert space $\mathcal{A}$ there is exactly one linear map $\mathcal{A} \to \mathcal{H}_0$. It maps every vector from $\mathcal{A}$ to the zero vector of $\mathcal{H}_0$.

Example 2.50 (Product)**. [FdHilb]** The Product in **FdHilb** category is a Direct sum of Hilbert spaces.

Example 2.51 (Sum)**. [FdHilb]** The Sum in **FdHilb** category is a Direct sum of Hilbert spaces.

Example 2.52 (Product and sum)**. [FdHilb]** Let $\mathcal{A}$ and $\mathcal{B}$ be finite dimensional Hilbert spaces. The same object $\mathcal{A} \oplus \mathcal{B}$ is both the Product and the Sum in **FdHilb** category.

For the product structure we use the projections

$$\pi_{\mathcal{A}}(a \oplus b) = a$$

and

$$\pi_{\mathcal{B}}(a \oplus b) = b.$$

For the sum structure we use the injections

$$i_{\mathcal{A}}(a) = a \oplus 0_{\mathcal{B}}$$

and

$$i_{\mathcal{B}}(b) = 0_{\mathcal{A}} \oplus b.$$

The projections satisfy the universal property of the product, and the injections satisfy the universal property of the sum. Thus in **FdHilb** the categorical product and the categorical sum coincide up to canonical isomorphism.

It is worth noting that the tensor product is the usual construction for compound quantum systems, but it is not the categorical product in **FdHilb** category. The categorical product is the direct sum above.

Chapter 3

Curry-Howard-Lambek correspondence

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

3.1 Proof category

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

There are 2 main propositions

Definition 3.2 (True)**.** A true statement is one that is correct, either in all cases or at least in the sample case [22].

and

Definition 3.3 (False)**.** A false statement is one that is not correct [22].

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

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

There is an example of incorrect (false) proposition

$$\exists n \in \mathbb{R} : n^2 < 0.$$

The expression $\forall n \in \mathbb{C} : n^2 \geq 0$ is not a good example of a false proposition in standard math because $\mathbb{C}$ has no standard order for the comparison $\geq$.

Definition 3.5 (Implication)**.** *An implication* is a Proposition of the form $P \implies Q$ i.e. if P then Q [12].

The main logical deduction rule is the following

Definition 3.6 (Modus ponens)**.** If P is true and $P \implies Q$ is true then Q is also true. The rule is often written as [12]

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

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

Definition 3.7 (Proof)**.** *Proof* is a verification [12] of a Proposition by a chain of logical deduction from a base set of axioms.

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

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

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

Table 3.1: Conjunction

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

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

Table 3.2: Disjunction

Operations in Boolean logic follow the distributive law:

$$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c) \tag{3.1}$$

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

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

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

Example 3.11 (Initial object)**.** [**Proof**] The *false* statement can be considered as the initial object because for any other statement there is a proof from the false statement to that one. If we identify proofs with the same entailment or work in a proof-irrelevant setting, this proof is unique.

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

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

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

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

Proof category	Programming language	Cartesian closed category
Proposition/Implication	Type	Object
Proof	Function type	Exponential
Conjunction	Product type	Product
Disjunction	Sum type	Sum
True	unit type	Terminal object
False	botom type	Initial object

Table 3.3: Relation between logic proofs and programming languages

3.2 Linear logic and Linear types

Linear logic [2] is a refinement of classical logic where the usage of a statement becomes important. In classical logic we can use the same statement several times or ignore it. In linear logic a statement can be considered as a resource. If a resource is used in an Implication, then it cannot be used again without an explicit rule that permits copying.

This point of view is useful for computations where values represent real resources. For example, a file handle, a lock, or a communication channel should not be silently duplicated or forgotten. Linear types add the same discipline to a programming language: a value of a linear type has to be used exactly once. Thus the type checker can verify resource usage statically, which is especially useful in concurrent programs.

3.3 Quantum logic and quantum computation

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

We can use the Heisenberg inequality to illustrate the violation of the law. Consider a particle with 2 possible positions range and 1 possible momentum range. The event P is that momentum has range Δp. The event $Q_{1,2}$ is that position is in the corresponding range $\Delta q_{1,2}$ (see fig. 3.1).

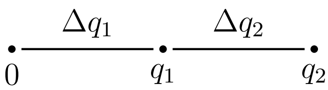

Figure 3.1: Heisenberg inequality. The particle position can be in the following ranges: $[0, q_1]$ (uncertainty is Δq_1), $[q_1, q_2]$ (uncertainty is $\Delta q_2 = \Delta q_1$) or $[0, q_2]$ (uncertainty is $\Delta q = \Delta q_1 + \Delta q_2 = 2\Delta q_1$). We assume that momentum uncertainty is defined as $\Delta p = \frac{\hbar}{2\Delta q_1}$. Thus we have $\Delta p \Delta q_1 = \Delta p \Delta q_1 = \frac{\hbar}{3} < \frac{\hbar}{2}$ and therefore the particle cannot be localised neither inside $[0, q_1]$ or $[q_1, q_2]$. From other side $\Delta p \Delta q = 2\Delta p \Delta q_1 = \frac{2\hbar}{3} > \frac{\hbar}{2}$ and as result the particle can be localized at interval $[0, q_2]$

Heisenberg inequality says

$$\Delta p \Delta q_{1,2} \geq \frac{\hbar}{2}$$

i.e. the following 2 events $P \wedge Q_{1,2}$ are forbidden as soon as (see fig. 3.1):

$$\Delta p \Delta q_1 = \frac{\hbar}{3}$$

and

$$\Delta p \Delta q_2 = \frac{\hbar}{3}$$

i.e.

$$P \wedge Q_1 = P \wedge Q_2 = \text{False}$$

from other side the event $P \wedge (Q_1 \vee Q_2)$ can be True as soon as (see fig. 3.1)

$$\Delta p(\Delta q_1 + \Delta q_2) = \frac{2\hbar}{3} > \frac{\hbar}{2}.$$

Therefore we have distributive law (3.1) violation

$$P \wedge (Q_1 \vee Q_2) \neq (P \wedge Q_1) \vee (P \wedge Q_2).$$

Chapter 4

Functors

4.1 Definitions

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

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

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

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

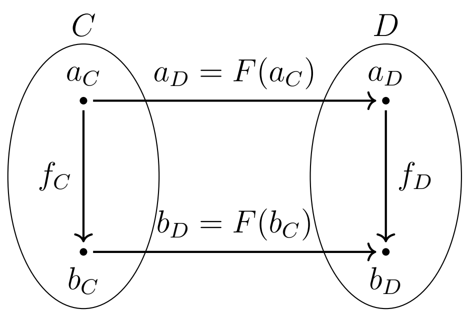

Figure 4.1: Functor $F : \mathbf{C} \Rightarrow \mathbf{D}$ definition

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

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

Programming languages can be considered as a good platform for the functor examples. The functor can be defined in Haskell as follows. [1]

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

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

In Scala it can be defined in the same way.

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

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

In C++ the definition differs.

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

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

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

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

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

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

[1] the real definition is quite different from the current one

The simple usage example is the following.

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

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

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

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

and for every Morphism $f \in \mathrm{hom}(\mathbf{C})$

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

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

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

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

$$H(a) = G(F(a))$$

and for Morphisms as follows:

$$H(f) = G(F(f)).$$

4.2 Cat category

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

Definition 4.10 (**Cat** category)**.** The category of small categories (see Small category) denoted as **Cat** is the Category where objects are small categories and morphisms are Functors between them.

We can construct an extension of Cartesian product as follows

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

- Objects are the pairs (c, d) where $c \in \text{ob}(\mathbf{C})$ and $d \in \text{ob}(\mathbf{D})$
- Morphisms are the pair (f, g) where $f \in \text{hom}(\mathbf{C})$ and $g \in \text{hom}(\mathbf{D})$
- Composition (Axiom 1.9) 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 objectwise as follows: $\mathbf{1}_{(c,d)\to(c,d)} = (\mathbf{1}_{c\to c}, \mathbf{1}_{d\to d})$

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

Example 4.13 (Initial object)**.** [**Cat**] Empty category is the Initial object in **Cat** category [24].

Example 4.14 (Terminal object)**.** [**Cat**] Trivial category is the Terminal object in **Cat** category.

The good example can be found in **Hask** category.

Example 4.15 (Constant functor)**.** [**Hask**]

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

4.3 Contravariant functor

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

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

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

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

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

The contravariant functor should follow the following laws

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

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

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

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

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

```
strgt10 :: Predicate String
```

In the case the Contravariant functor helps.

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

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

4.4 Bifunctors

Definition 4.19 (Bifunctor)**.** Bifunctor is a Functor whose Domain is a Category Product. I.e. if $\mathbf{C_1}, \mathbf{C_2}, \mathbf{D}$ are 3 categories then the Functor $F : \mathbf{C_1} \times \mathbf{C_2} \Rightarrow \mathbf{D}$ is called *bifunctor*.

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

Example 4.21 (Either as a bifunctor)**.** **[Hask]** In **Hask** the **Either** type constructor has 2 arguments, and both arguments can be mapped independently.

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

class Bifunctor p where
  bimap :: (a -> c) -> (b -> d) -> p a b -> p c d

instance Bifunctor Either where
  bimap f _ (Left a) = Left (f a)
  bimap _ g (Right b) = Right (g b)
```

Thus the first function changes the **Left** value and the second function changes the **Right** value.

Definition 4.22 (Profunctor)**.** If we have categories **C** and **D** then the Bifunctor

$$\mathbf{C}^{\mathbf{op}} \times \mathbf{D} \Rightarrow \mathbf{Set}$$

is called *profunctor*. It is contravariant by the first argument and covariant by the second one.

Example 4.23 (Profunctor)**.** **[Hask]** In **Hask** we can think about a profunctor as a type constructor with 2 arguments. The first argument is contravariant and the second one is covariant.

```
class Profunctor p where
  dimap :: (a' -> a) -> (b -> b') -> p a b -> p a' b'

instance Profunctor (->) where
  dimap f g h = g . h . f
```

The function arrow is the simplest example. If $h : a \to b$, then **dimap** converts it into $a' \to b'$ by composing it with $f : a' \to a$ and $g : b \to b'$.

Chapter 5

Natural transformation

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

5.1 Definitions

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

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

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

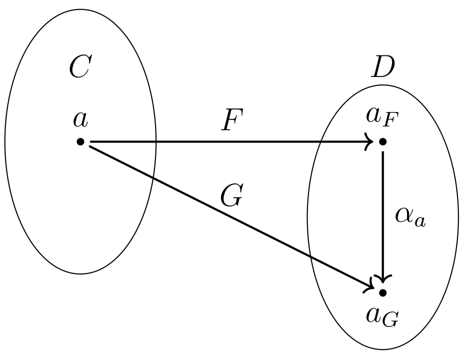

Figure 5.1: Natural transformation: object mapping

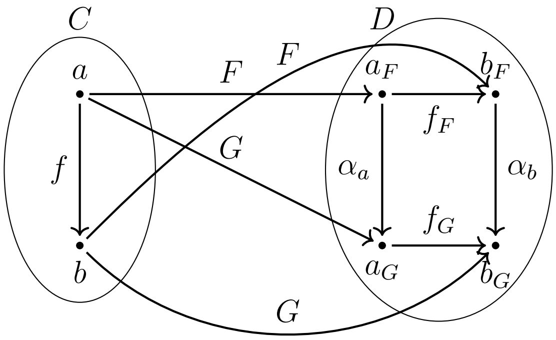

Figure 5.2: Natural transformation: morphisms mapping

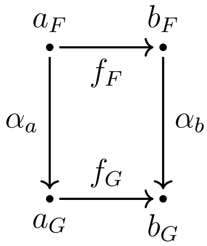

Figure 5.3: Natural transformation: commutative diagram

We can of course 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 to each other. As an opposite example, if there are no such morphisms then the functors can be considered as unrelated each other.

The functor is not just the object mapping but also the morphisms mapping. If we have 2 objects a and b in the category $\mathbf{C}$ then we potentially can have a morphism $f \in \hom_{\mathbf{C}}(a, b)$. 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: $\alpha_a, \alpha_b, f_F, f_G \in \hom(\mathbf{D})$. It is natural to impose additional conditions on the morphisms especially that they form a Commutative diagram (see fig. 5.3):

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

Definition 5.1 (Natural transformation)**.** Let F and G be 2 Functors from category $\mathbf{C}$ to the category $\mathbf{D}$. A *natural transformation* from F to G is a

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

family of Morphisms

$$\alpha = \{\alpha_a : F(a) \to G(a) \mid a \in \mathrm{ob}(\mathbf{C})\}$$

in category $\mathbf{D}$ that satisfies the following conditions:

- For every Object $a \in \mathrm{ob}(\mathbf{C})$ the family has exactly one component $\alpha_a \in \mathrm{hom}_{\mathbf{D}}(F(a), G(a))$. The morphism α_a is called the component of the natural transformation.

- For every morphism $f \in \mathrm{hom}(\mathbf{C})$ that connects 2 objects a and b, i.e. $f \in \mathrm{hom}_{\mathbf{C}}(a, b)$, the corresponding components α_a and α_b should satisfy the following condition

$$f_G \circ \alpha_a = \alpha_b \circ f_F, \tag{5.1}$$

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

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

Definition 5.2 (Natural isomorphism)**.** The Natural transformation $\alpha : F \dot{\to} G$ is called *natural isomorphism* if all morphisms $\alpha \subset \mathrm{hom}(\mathbf{D})$ are Isomorphisms in $\mathbf{D}$

5.2 Category of functors

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

To define a category we need to define composition operation that satisfied Composition (Axiom 1.9) , identity morphism and verify Associativity (Axiom 1.11) .

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

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

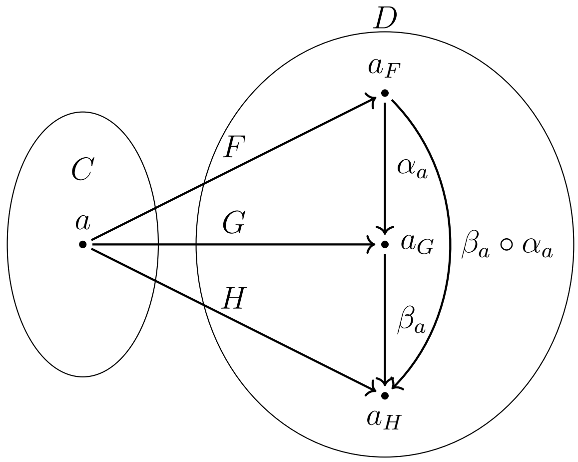

Figure 5.4: Natural transformation vertical composition: object mapping

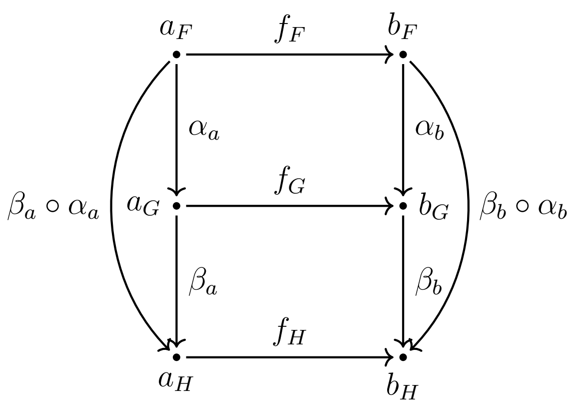

Figure 5.5: Natural transformation vertical composition: morphism mapping - commutative diagram

Definition 5.3 (Vertical composition)**.** Let F, G, H are functors between categories $\mathbf{C}$ and $\mathbf{D}$. Also we have $\alpha : F \dot{\to} G, \beta : G \dot{\to} H$ - natural transformations. We can compose the α and β as follows

$$\beta \circ \alpha : F \dot{\to} H.$$

This composition is called *vertical composition*.

Definition 5.4 (**Fun** category)**.** Let $\mathbf{C}$ and $\mathbf{D}$ are 2 categories. The category that contains functors $F : \mathbf{C} \Rightarrow \mathbf{D}$ as objects and Natural transformation as morphisms is called as *functor category*. The morphism composition is the Vertical composition in the category. The *functor category* between categories $\mathbf{C}$ and $\mathbf{D}$ is denoted as $[\mathbf{C}, \mathbf{D}]$.

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

Example 5.5 (Terminal object)**.** [**Fun**] Let $[\mathbf{C}, \mathbf{D}]$ is the functor category between $\mathbf{C}$ and $\mathbf{D}$. If $t \in \text{ob}(\mathbf{D})$ is the Terminal object in the category $\mathbf{D}$ then the Constant functor Δ_t is the Terminal object in the category $[\mathbf{C}, \mathbf{D}]$ [7].

Example 5.6 (Initial object)**.** [**Fun**] Let $[\mathbf{C}, \mathbf{D}]$ is the functor category between $\mathbf{C}$ and $\mathbf{D}$. If $i \in \text{ob}(\mathbf{D})$ is the Initial object in the category $\mathbf{D}$ then the Constant functor Δ_i is the Initial object in the category $[\mathbf{C}, \mathbf{D}]$ [7].

5.3 Operations with natural transformations

Vertical composition is not the unique way to compose 2 Natural transformations. Another option is also possible.

Definition 5.7 (Horizontal composition)**.** If we have 2 pairs of functors. The first one $F, G : \mathbf{C} \to \mathbf{D}$ and another one $J, K : \mathbf{D} \Rightarrow \mathbf{E}$. We also have a natural transformation between each pair: $\alpha : F \dot{\to} G$ for the first one and $\beta : J \dot{\to} K$ for the second one. We can create a new transformation

$$\alpha \star \beta : J \circ F \dot{\to} K \circ G$$

that is called *horizontal composition*. For every object $a \in \text{ob}(\mathbf{C})$ its component is defined as follows:

$$(\alpha \star \beta)_a = \beta_{G(a)} \circ J(\alpha_a) = K(\alpha_a) \circ \beta_{F(a)}.$$

The equality of the 2 expressions follows from naturality of β. Note that we use a special symbol $\star$ for the composition.

Remark 5.8 (Bifunctor in the category of functors)**.** If we have the same pair of functors as in definition 5.7 then we can consider the functors as Objects of 3 categories: $\mathcal{A} = [\mathbf{C}, \mathbf{D}], \mathcal{B} = [\mathbf{D}, \mathbf{E}]$ and $\mathcal{C} = [\mathbf{C}, \mathbf{E}]$

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

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

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

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

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

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

$$\alpha H : F \circ H \dot{\to} G \circ H$$

that is called *left whiskering* of functor and natural transformation [18].

Definition 5.10 (Right whiskering)**.** If we have 3 categories $\mathbf{C}, \mathbf{D}, \mathbf{E}$, Functors $F, G : \mathbf{C} \Rightarrow \mathbf{D}$, $H : \mathbf{D} \to \mathbf{E}$ and Natural transformation $\alpha : F \dot{\to} G$ then we can construct a new natural transformations:

$$H\alpha : H \circ F \dot{\to} H \circ G$$

that is called *right whiskering* of functor and natural transformation [18].

Definition 5.11 (Identity natural transformation)**.** If $F : \mathbf{C} \Rightarrow \mathbf{D}$ is a Functor then we can define *identity natural transformation* $\mathbf{1}_{F \dot{\to} F}$ that maps any Object $a \in \text{ob}(\mathbf{C})$ into Identity morphism $\mathbf{1}_{F(a) \to F(a)} \in \text{hom}(\mathbf{D})$.

Remark 5.12 (Whiskering)**.** With Identity natural transformation we can redefine Left whiskering and Right whiskering via Horizontal composition as follows.

For left whiskering:

$$\alpha H = \mathbf{1}_{H \dot{\to} H} \star \alpha \tag{5.2}$$

For right whiskering:

$$H\alpha = \alpha \star \mathbf{1}_{H \dot{\to} H} \tag{5.3}$$

5.4 Polymorphism and natural transformation

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

Definition 5.13 (Parametrically polymorphic function)**.** Polymorphism is *parametric* if all function instances behave uniformly i.e. have the same realization. The functions which satisfy the parametric polymorphism requirements are parametrically polymorphic.

Definition 5.14 (Ad-hoc polymorphism)**.** Polymorphism is *ad-hoc* if the function instances can behave differently dependently on the type they are being instantiated with.

Theorem 5.15 (Reynolds)**.** *Let* $\mathbf{T}$ *be a category of types and total functions. Let* $F, G : \mathbf{T} \Rightarrow \mathbf{T}$ *be covariant type constructors whose relational interpretations map the graph relation of every function* $f : a \to b$ *to the graph relations of* $F(f)$ *and* $G(f)$*. A Parametrically polymorphic function*

$$p : \forall a.\ F(a) \to G(a)$$

defines a Natural transformation $p : F \dot{\to} G$*.*

Proof. For every type a the function instance $p_a : F(a) \to G(a)$ is a Morphism in $\mathbf{T}$. Therefore we already have the components that are required by Natural transformation.

It remains to verify the naturality condition. Let $f : a \to b$ be a morphism in $\mathbf{T}$. Reynolds' abstraction theorem [23] states that a parametrically polymorphic function preserves relations between type instances. We apply this statement to the graph relation of f:

$$x\ R_f\ y \iff f(x) = y.$$

By the assumption on F and G, this graph relation is mapped to the graph relations of $F(f)$ and $G(f)$. Thus, if $x \in F(a)$, then x and $F(f)(x)$ are related by the relation obtained from R_f by the type constructor F. Parametricity of p implies that $p_a(x)$ and $p_b(F(f)(x))$ are related by the relation obtained from R_f by the type constructor G. But this relation is the graph relation of $G(f)$. Thus

$$G(f)(p_a(x)) = p_b(F(f)(x)).$$

Since this equality holds for every $x \in F(a)$, we get

$$G(f) \circ p_a = p_b \circ F(f).$$

This is exactly the naturality condition from (5.1). Therefore the family of components $p = \{p_a\}_{a \in \mathrm{ob}(\mathbf{T})}$ is a Natural transformation. □

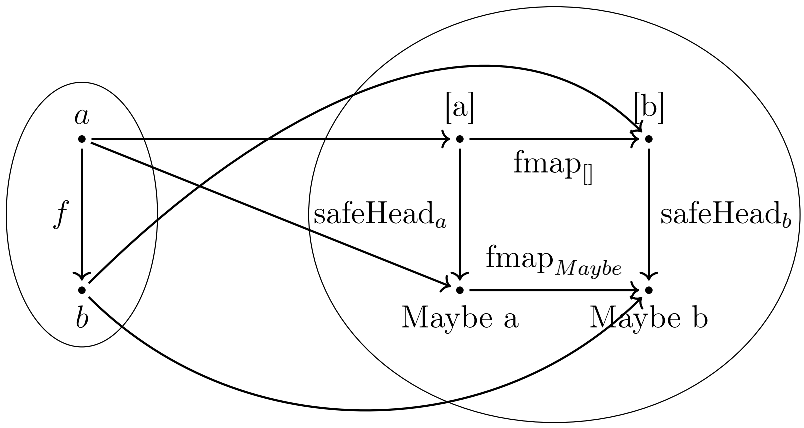

Figure 5.6: Haskell parametric polymorphism as a natural transformation

5.4.1 Hask category

Many pure polymorphic functions in Haskell are Parametrically polymorphic functions [2].

Example 5.16 (Parametrically polymorphic function)**.** [**Hask**] Consider the following function

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

The function is parametrically polymorphic and by Reynolds (Theorem 5.15) is Natural transformation (see fig. 5.6).

Therefore from the definition of the natural transformation (5.1) we have

```
fmap f . safeHead = safeHead . fmap f
```

I.e. it does not matter if we initially apply **fmap f** and then **safeHead** to the result or initially **safeHead** and then **fmap f**.

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

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

[2]This statement ignores Haskell values such as **undefined**, **seq**, and other non-parametric extensions.

On the 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)
```

On the 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]

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

Chapter 6

Monads

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

6.1 Monoid in Set category

In the section 2.4 we considered the definition and importance of the Monoid concept. The definition was given in terms of internal structure i.e. the ordinary and not Categorical approach was used. Now we are going to consider Monoid in terms of Set theory but will try to give the definition that is based rather on morphisms than on internal set structure i.e. we shall use Categorical approach. Let M be a set and by the monoid definition (definition 2.18) $\forall m_1, m_2 \in M$ we can define a new element of the set $\mu(m_1, m_2) \in M$. Later we shall use the following notation for the μ:

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

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

$$m_1 \cdot (m_2 \cdot m_3) = (m_1 \cdot m_2) \cdot m_3. \tag{6.1}$$

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

$$m \cdot e = e \cdot m = m. \tag{6.2}$$

6.1.1 Associativity

Let's consider (6.1) in detail. We can define μ as a Morphism in the following way

$$\mu : M \times M \to M,$$

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

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

Consider the action of Product of morphisms $\mathbf{1}_{M\to M} \times \mu$ on A:

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

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

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

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

For A' we have the following one:

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

Monoid associativity requires

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

i.e. the morphisms shown in fig. 6.1 commute:

$$\mu \circ [\mu \times \mathbf{1}_{M\to M}] = \mu \circ [\mathbf{1}_{M\to M} \times \mu] \circ \alpha. \tag{6.3}$$

Very often the isomorphism α is omitted i.e.

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

and the morphism equality (6.3) is written as follow

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

The corresponding commutative diagram is shown in fig. 6.2.

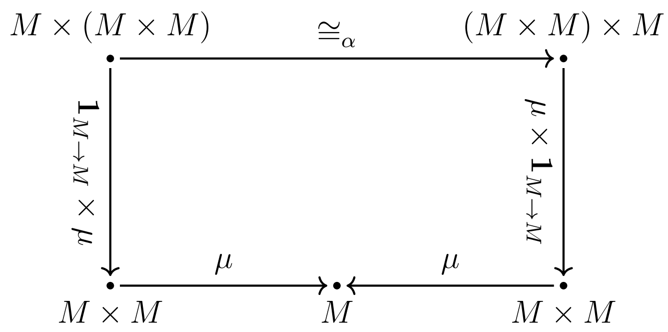

Figure 6.1: Commutative diagram for $\mu \circ [\mu \times \mathbf{1}_{M\to M}] = \mu \circ [\mathbf{1}_{M\to M} \times \mu] \circ \alpha$.

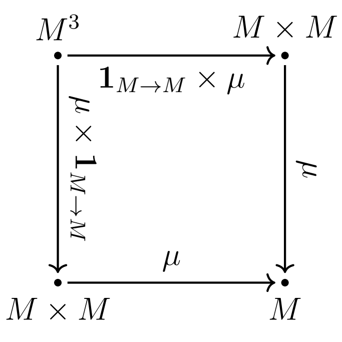

Figure 6.2: Commutative diagram for $\mu \circ [\mu \times \mathbf{1}_{M\to M}] = \mu \circ [\mathbf{1}_{M\to M} \times \mu]$

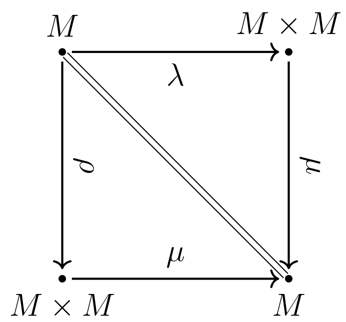

Figure 6.3: Commutative diagram for $\mu \circ (\eta \times \mathbf{1}_{M\to M}) \circ \lambda = \mu \circ (\mathbf{1}_{M\to M} \times \eta) \circ \rho = \mathbf{1}_{M\to M}$.

6.1.2 Identity presence

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

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

and

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

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

$$[\eta \times \mathbf{1}_{M\to M}]\,(0, m) = (e, m),$$
$$[\mathbf{1}_{M\to M} \times \eta]\,(m, 0) = (m, e).$$

After the application of μ on the result we obtain

$$\mu\left([\eta \times \mathbf{1}_{M\to M}]\,(0, m)\right) = \mu\,(e, m) = e \cdot m,$$
$$\mu\left([\mathbf{1}_{M\to M} \times \eta]\,(m, 0)\right) = \mu\,(m, e) = m \cdot e.$$

The (6.2) leads to the following equation for morphisms

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

or the commutative diagram shown on fig. 6.3.

[1] It also is called [11, p. 2] as a one point set

6.1.3 Categorical definition for monoid

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

$$M \times M.$$

We also have 2 pairs of morphisms:

$$\mu : M \times M \to M,$$
$$\mathbf{1}_{M \to M} : M \to M.$$

and

$$\eta : I \to M,$$
$$\mathbf{1}_{M \to M} : M \to M.$$

The pairs can be combined into one using Product of morphisms as follows:

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

and

$$\eta \times \mathbf{1}_{M \to M} : I \times M \to M \times M,$$
$$\mathbf{1}_{M \to M} \times \eta : M \times I \to M \times M.$$

The same structure [2] is used by Functor and especially by Bifunctor (Example 4.20) .

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

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

1. there is a Morphism $\mu : m \times m \to m$ in the category
2. there is another morphism $\eta : t \to m$ in the category

[2] not only objects mapping but also morphisms mapping

3. the morphisms satisfy the following conditions:

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

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

   where α (associator) is an Isomorphism between $m \times (m \times m)$ and $(m \times m) \times m$. λ, ρ are other isomorphisms:

$$m \cong_\lambda t \times m$$

   and

$$m \cong_\rho m \times t$$

6.1.4 Monoid importance

Why is the concept of Monoid so important? In pure math it provides the build blocks for important concepts such as Group, Ring, Field [16]. In programming languages it gives more simple and robust concept for software design [4].

We can notice that monoid definition (see definition 2.18) has the same requirements as morphisms in category theory. Moreover the monoid can be viewed as a Category (see section 2.4).

Monoid provides a closed collection of objects such that if you combine them you will get an object of the same type. This allows to create constructions which are more easy in maintenance. For instance there are 2 possible options to combine objects of type A in software architecture [4, 13]:

1. **Conventional architecture** assumes that a combination of several objects of type A will produce a "network" of the objects A i.e. new type B

2. **Haskell architecture** assumes that a combination of several objects of type A will produce a new object of the same type A

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

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

6.2 Monoidal category

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

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

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

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

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

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

$$a \cong_\lambda e \otimes a,$$
$$a \cong_\rho a \otimes e,$$

where λ, ρ are called as left and right unitors respectively.

In the **Set** category we have $\times$ as the monoidal product (see example 4.20). There is also a morphism η from terminal object t to e [6] (see definition 6.1).

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

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

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

6.3 Tensor product in Quantum mechanics

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

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

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

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

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

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

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

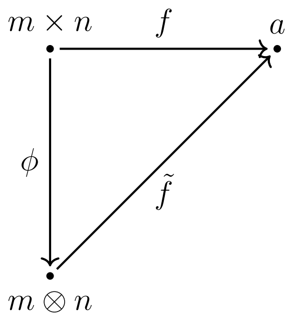

Figure 6.4: Commutative diagram for tensor product definition.

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

The tensor product in quantum mechanics is used for representing a system that consists of multiple systems. For instance if we have an interaction between an 2 level atom (a is excited state b as a ground state) and one mode light then the atom has its own 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. [3] The result system that describes both atom and light is represented as the tensor product $\mathcal{H}_{at} \otimes \mathcal{H}_f$.

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

[3] 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.

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 $\mathcal{B}$:

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

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

6.4 Category of endofunctors

The **Fun** category is an example of a category. We can apply additional limitation and consider only Endofunctors i.e. we shall look at the category $[\mathbf{C}, \mathbf{C}]$ - the category of functors from category $\mathbf{C}$ to the same category. One of the most popular math definition of a monad is the following: "All told, a monad in X is just a monoid in the category of endofunctors of X"[11, p. 138]. Later we shall give an explanation for that one.

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

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

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

$$\mu : M \circ M \dot{\to} M \tag{6.6}$$

and

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

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

The η, μ should satisfy the following conditions:

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

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

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

Remark 6.10 (Monad term)**.** The word monad is a concatenation of 2 words: **mon**oid and tri**ad** [11, p. 138]. The first one points to the fact that the object looks like a monoid. The second one says that it is a set of 3 objects (Endofunctor and 2 Natural transformations) aka triad.

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

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

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

$$\begin{aligned} \mu \circ (\mathbf{1}_{M \dot{\to} M} \star \mu) = \mu \circ (\mu \star \mathbf{1}_{M \dot{\to} M}), \\ \mu \circ (\mathbf{1}_{M \dot{\to} M} \star \eta) = \mu \circ (\eta \star \mathbf{1}_{M \dot{\to} M}) = \mathbf{1}_{M \dot{\to} M}. \end{aligned} \tag{6.9}$$

Thus we can notice that the pair of operations (composition $\circ$ and Horizontal composition $\star$) forms the bifunctor (see Bifunctor in the category of functors (Remark 5.8)).

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

$$\mathbf{1}_{M \dot{\to} M} \star \mu : M \circ (M \circ M) \dot{\to} M \circ M$$

thus

$$\mu \circ (\mathbf{1}_{M \dot{\to} M} \star \mu) : M \circ (M \circ M) \dot{\to} M.$$

Similarly

$$\mu \circ (\mu \star \mathbf{1}_{M \dot{\to} M}) : (M \circ M) \circ M \dot{\to} M.$$

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

$$\mu \circ (\mathbf{1}_{M \dot{\to} M} \star \mu) = \mu \circ (\mu \star \mathbf{1}_{M \dot{\to} M}) \tag{6.10}$$

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

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

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

will be replaced with

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

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

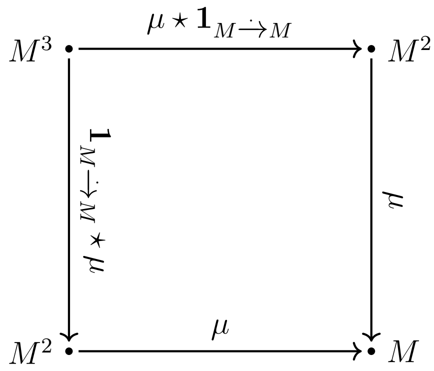

Figure 6.5: Monad as monoid in the category of endofunctors.

6.5 Monads in programming languages

There are several examples of Monad implementation in different programming languages:

6.5.1 Haskell

Example 6.11 (Monad)**. [Hask]** In Haskell monad can be defined from Functor (Example 4.3) as follows [4]

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

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

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

where **return** can be treated as η (6.7) and **join** as μ (6.6). In the case the bind operator »= can be implemented as follows

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

As an application of the last example lets consider the next one

Example 6.12 (Monad)**. [Hask]**

[4] real definition is quite different from the presented one

```
joinHeads :: [[a]] -> [a]
joinHeads ys = ys >>= \xs -> [head xs]
```

The usage example is the following

```
> joinHeads ["a123", "b123", "c123"]
"abc"
```

As one can see the fmap was applyed first with the result

```
> fmap (\xs -> [head xs]) ["a123", "b123", "c123"]
["a","b","c"]
```

and finally the join will get us the required result - the list contained the first elements of internal lists.

6.5.2 C++

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

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

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

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

where **pure** can be treated as η (6.7) and **join** as μ (6.6). In the case the bind operator can be implemented as follows

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

6.5.3 Scala

Example 6.13 (Monad)**.** [**Scala**] The monad concept in Scala is close to the bind form of Monad. It can be defined as follows [5]

[5] real definition is quite different from the presented one

```
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 the bind operator and **unit** as η. If we use the mathematical notation then **flatMap** can be expressed as follows:

$$ma.\mathit{flatMap}(f) = \mu(\mathit{fmap}(f, ma)),$$

where $ma : M[A]$ and $f : A \to M[B]$. Thus Scala uses the same monad structure, but exposes it through a method that is convenient for programming languages.

Chapter 7

Kleisli category

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

$$\text{ob}(\mathbf{C_M}) = \text{ob}(\mathbf{C}),$$
$$\text{hom}_{\mathbf{C_M}}(a, b) = \text{hom}_{\mathbf{C}}(a, M(b))$$

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

The identity morphism in the Kleisli category is the Natural transformation component $\eta_a : a \to M(a)$ (6.7) defined by the monad $\langle M, \mu, \eta \rangle$:

$$\mathbf{1}_{a \to a}^{C_M} = \eta_a.$$

If $f_M : a \to b$ and $g_M : b \to c$ are represented in $\mathbf{C}$ by $f : a \to M(b)$ and $g : b \to M(c)$, then their composition in $\mathbf{C_M}$ is represented by

$$\mu_c \circ M(g) \circ f : a \to M(c).$$

Remark 7.2 (Kleisli category composition)**.** Kleisli category has non-trivial composition rules. If we have 2 Morphisms from $\text{hom}(\mathbf{C_M})$:

$$f_M : a \to b$$

and

$$g_M : b \to c.$$

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

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

and

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

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

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

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

$$h = \mu_c \circ M(g) \circ f : a \to M(c).$$

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

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

Kleisli category is widely spread in programming, especially it provides good description for different types of computations, for instance [15, 14]:

- **Partiality** i.e. when a function is 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. when multiple outputs are possible
- **Side-effects** i.e. when a function communicates with an environment
- **Exception** i.e. when some input is incorrect and can produce an abnormal result. Therefore it is the same as **Partiality** and will be considered below as the same type of computation.
- **Continuation** i.e. when we need to save the current state of the computation and be able to restore it on demand later
- **Interactive input** i.e. a function that reads data from an input device (keyboard, mouse, etc.)
- **Interactive output** i.e. a function that writes data to an output device (monitor etc.)

7.1 Partiality and Exception

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

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

The function is a composition of 3 functions:

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

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

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

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

7.1.1 Haskell example

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

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

Our functions (7.1) can be implemented as follows

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

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

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

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

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

The usage example is the following:

```
*Main> h 4
Just 0.25
*Main> h 1
```

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

7.1.2 C++ example

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

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

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

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

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

Our functions (7.1) can be implemented as follows

```
std::function<Maybe<float>(float)> f1 =
    [](float x) {
      if (x >= 0) {
        return Maybe<float>(sqrt(x));
      }
      return Maybe<float>();
```

```
    };

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

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

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

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

7.2 Non-Determinism

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

7.2.1 Haskell example

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

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

7.3 Side effects and interactive input/output

TBD

7.4 Continuation

TBD

Chapter 8

Limits

8.1 Definitions

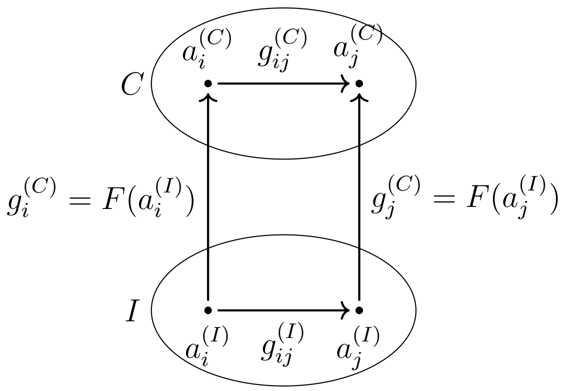

Figure 8.1: Diagram of shape $F : \mathbf{I} \Rightarrow \mathbf{C}$. Objects $a_{i,j}^{(I)} \in \text{ob}(\mathbf{I})$ are mapped to $a_{i,j}^{(C)} \in \text{ob}(\mathbf{C})$. Morphisms $g_{ij}^{(I)} \in \text{hom}(\mathbf{I})$ are mapped to $g_{ij}^{(C)} \in \text{hom}(\mathbf{C})$

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

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

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

8.1.1 Limit

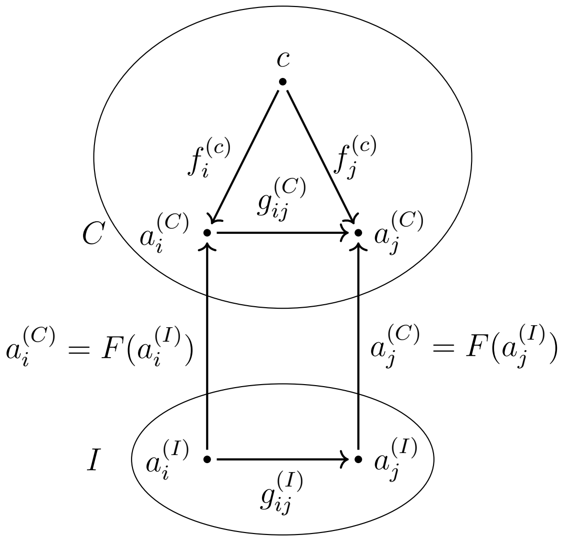

Figure 8.2: Cone cone$(c, f^{(c)})$

Definition 8.3 (Cone)**.** Let F be a Diagram of shape $\mathbf{I}$ in $\mathbf{C}$. A *cone* to F is an object $c \in \mathrm{ob}(\mathbf{C})$ with Morphisms $f^c = \left\{ f_i^c : c \to a_i^{(C)} \right\}$, where $a_i^{(C)} = F(a_i^{(I)})$ indexed by objects from $\mathbf{I}$ (see fig. 8.2). For every $g_{ij}^{(I)} : a_i^{(I)} \to a_j^{(I)}$ with $g_{ij}^{(C)} = F(g_{ij}^{(I)})$ the following condition holds:

$$g_{ij}^{(C)} \circ f_i^{(c)} = f_j^{(c)}.$$

The cone is denoted as cone$(c, f^{(c)})$.

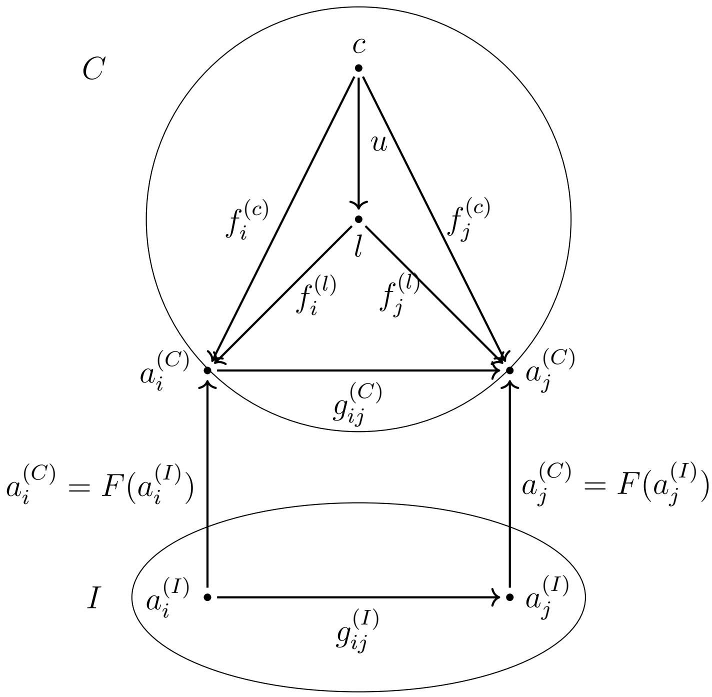

Figure 8.3: Limit cone$(l, f^{(l)})$

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

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

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

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

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

Definition 8.6 (Category of cones to F)**.** Let F be a Diagram of shape $\mathbf{I}$ in $\mathbf{C}$. The objects of the category are Cones cone$(c, f^{(c)})$ to F. A morphism from cone$(c_1, f^{(c_1)})$ to cone$(c_2, f^{(c_2)})$ is a Morphism $m : c_1 \to c_2$ in $\mathbf{C}$ such that $\forall i$

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

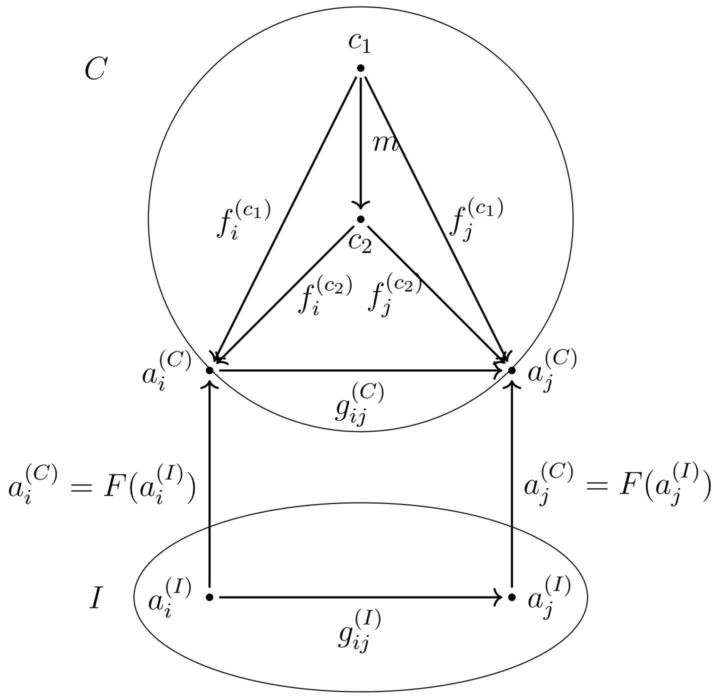

Figure 8.4: Morphism m between 2 cones $\mathrm{cone}(c_1, f^{(c_1)})$ and $\mathrm{cone}(c_2, f^{(c_2)})$

The identity morphism is the identity morphism of the cone's apex, and the composition is inherited from **C**.

The category of Cones is denoted as $\Delta \downarrow F$ [26]

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

8.1.2 Colimit

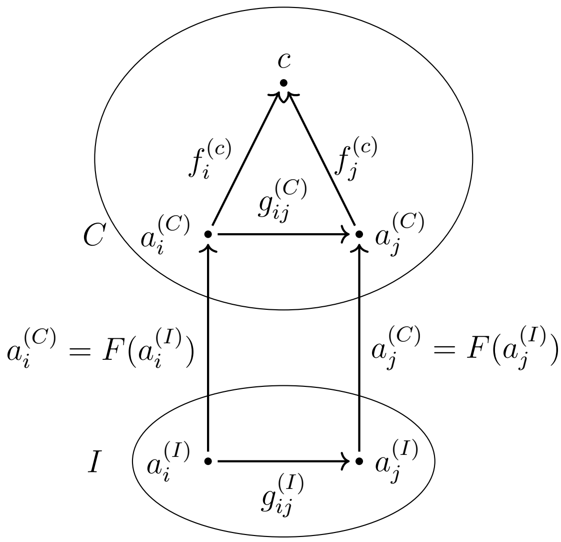

Figure 8.5: Co-cone cocone$(c, f^{(c)})$

Definition 8.8 (Cocone)**.** Let F be a Diagram of shape $\mathbf{I}$ in $\mathbf{C}$. A *co-cone* to F is an object $c \in \text{ob}(\mathbf{C})$ with Morphisms $f^c = \left\{f_i^c : a_i^{(C)} \to c\right\}$, where $a_i^{(C)} = F(a_i^{(I)})$ indexed by objects from $\mathbf{I}$ (see fig. 8.5). For every $g_{ij}^{(I)} : a_i^{(I)} \to a_j^{(I)}$ with $g_{ij}^{(C)} = F(g_{ij}^{(I)})$ the following condition holds:

$$f_j^{(c)} \circ g_{ij}^{(C)} = f_i^{(c)}.$$

The co-cone is denoted as cocone$(c, f^{(c)})$.

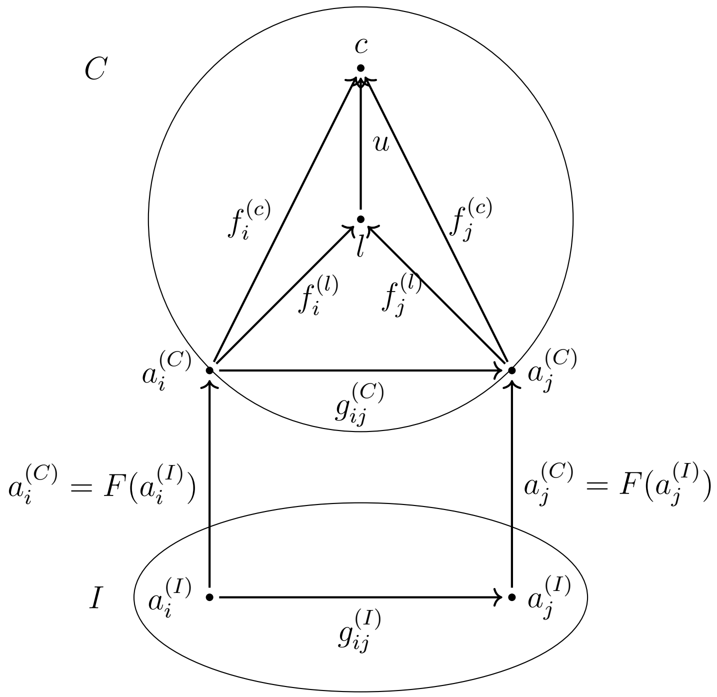

Figure 8.6: Co-Limit cocone$(l, f^{(l)})$

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

Definition 8.10 (Category of co-cones from F)**.** Let F be a Diagram of shape $\mathbf{I}$ in $\mathbf{C}$. The objects of the category are Cocones cocone$(c, f^{(c)})$ from F. A morphism from cocone$(c_1, f^{(c_1)})$ to cocone$(c_2, f^{(c_2)})$ is a Morphism $m : c_1 \to c_2$ in $\mathbf{C}$ such that $\forall i$

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

The identity morphism is the identity morphism of the co-cone's apex, and the composition is inherited from $\mathbf{C}$.

The category of Cocones is denoted as $F \downarrow \Delta$ [26]

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

8.2 Cone as natural transformation

The Cone can be considered as a Natural transformation. There are 2 functors between categories **I** and **C**. The first one is the Diagram of shape $F : \mathbf{I} \Rightarrow \mathbf{C}$. The second one is the Constant functor: $\Delta_c : \mathbf{I} \Rightarrow \mathbf{C}$. The Natural transformation $\Delta_c \dot{\to} F$, by the definition, is the set of Morphisms from **C** with additional relations that are same as conditions defined for the Cone $\mathrm{cone}(c, f^{(c)})$. Therefore we can consider the Cone as a Natural transformation.

8.3 Categorical constructions as limits

Different choice for category **I** gives different types of limits. There are several examples of such constructions below

The empty category will give us the terminal object. The Discrete category with 2 elements produces Product as the Limit.

8.3.1 Initial and terminal objects

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

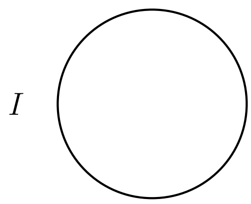

Figure 8.7: Index category I for initial and terminal objects. The category is empty

Example 8.12 (Limit)**.** [Terminal object] Lets choose Empty category as the Index category **I**.

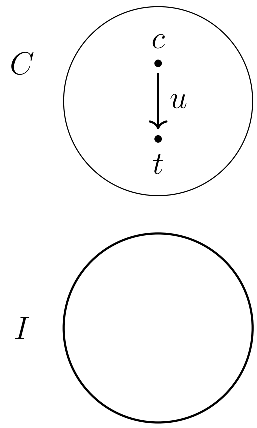

Figure 8.8: Terminal object t as a limit

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

Example 8.13 (Colimit)**.** [Initial object] Lets choose Empty category as the Index category **I**.

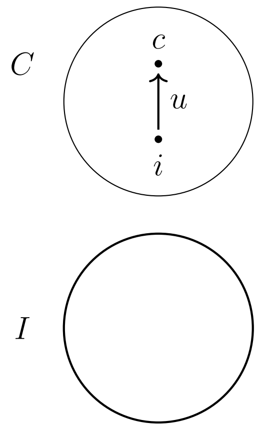

Figure 8.9: Initial object i as a colimit

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

8.3.2 Product and sum

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

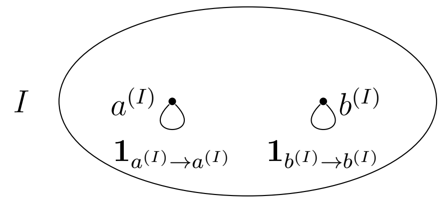

Figure 8.10: Index category I for product and sum. It consists of 2 objects $a^{(I)}, b^{(I)}$ and 2 trivial (identity) morphisms $\mathbf{1}_{a^{(I)} \to a^{(I)}}, \mathbf{1}_{b^{(I)} \to b^{(I)}}$

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

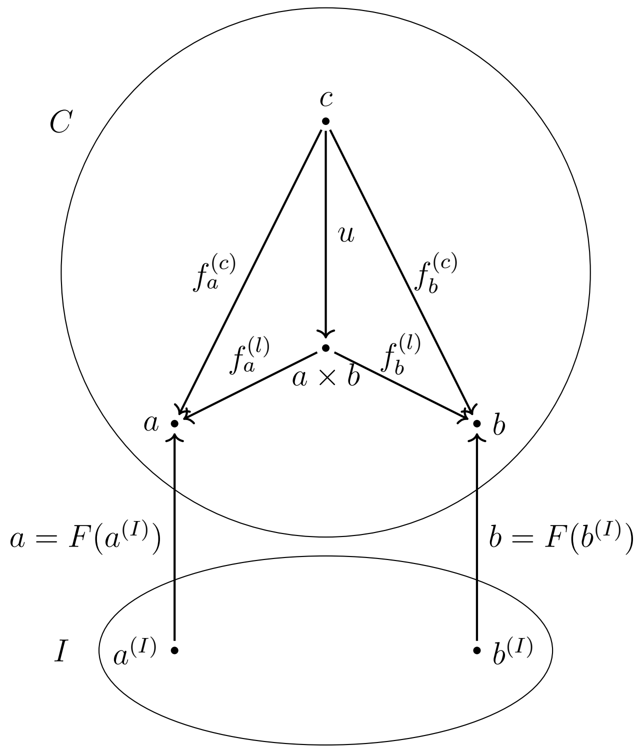

Figure 8.11: Product as a limit

The Diagram of shape F gives us the mapping into 2 objects in the category **C** (see fig. 8.11). The Limit of the Diagram of shape is the Product of the 2 objects in the category **C**.

Example 8.15 (Colimit)**.** [Sum] Lets choose Discrete category with 2 objects as the Index category **I**.

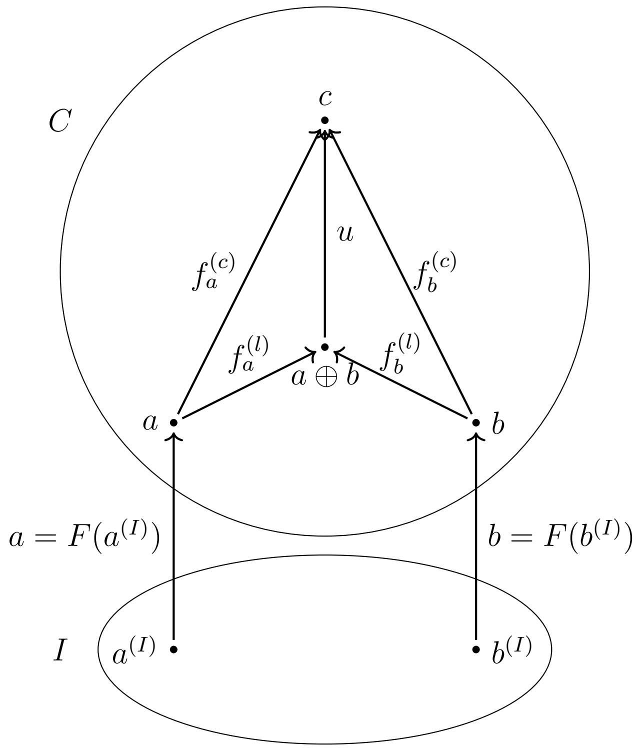

Figure 8.12: Sum as a colimit

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

8.3.3 Equalizer

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

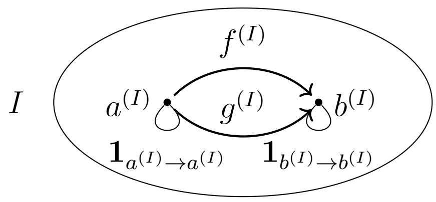

Figure 8.13: Index category I for equalizer. It consists of 2 objects $a^{(I)}, b^{(I)}$, 2 trivial (identity) morphisms $\mathbf{1}_{a^{(I)} \to a^{(I)}}, \mathbf{1}_{b^{(I)} \to b^{(I)}}$ and 2 additional morphisms $f^{(I)}, g^{(I)} \in \hom_{\mathbf{I}}\left(a^{(I)}, b^{(I)}\right)$

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

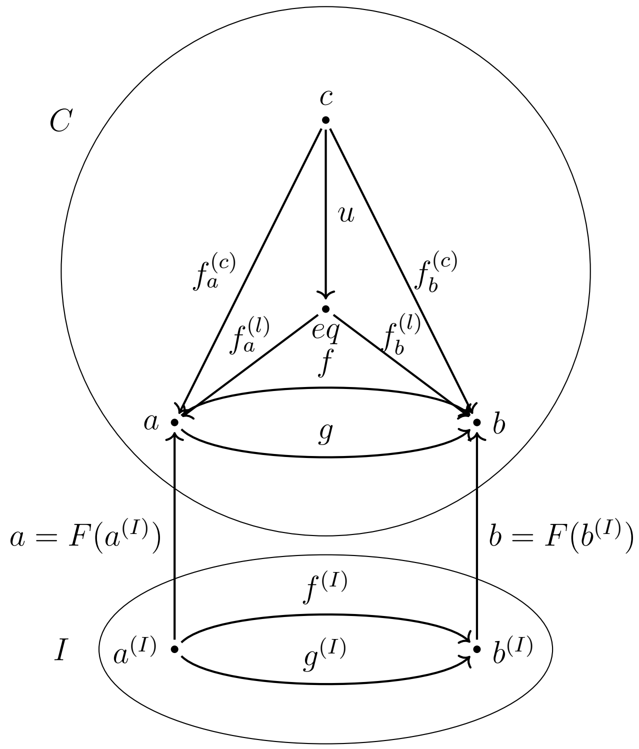

Figure 8.14: Equalizer

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

The meaning of the Equalizer can be described in the **Set** category

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

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

Let us consider 2 Sets A and B and 2 Functions $f, g : A \to B$. The equalizer is a subset of A that contains all elements where both functions have the same value:

$$eq(f, g) = \{x \in A \mid f(x) = g(x)\}.$$

The cone morphism $f_a^{(l)} : eq(f, g) \to A$ is the inclusion map. Thus if we have any other set C and a function $h : C \to A$ such that $f \circ h = g \circ h$, then every value $h(c)$ belongs to $eq(f, g)$. Therefore there exists a unique function $u : C \to eq(f, g)$ such that

$$f_a^{(l)} \circ u = h.$$

This is exactly the universal property of the Limit for the equalizer diagram.

Chapter 9

Yoneda's lemma

Yoneda lemma is a fact about so-called Hom functors. We shall start with the definition and examples for both Covariant Hom functor and Contravariant Hom functor. The definition and examples for Yoneda lemma will be provided after that. In the chapter we shall assume that the category **C** is a Locally small category.

9.1 Hom functors

We are going to define the Hom functors. There are 2 hom functors: Covariant functor and Contravariant functor. For the Covariant Hom functor we pick up an object a from **C** and consider the collection of morphisms from a to an arbitrary object x from the category. The collection is a Set as soon as **C** is a Locally small category. Therefore we can associate a set (object from **Set** category) with the object x from the category C.

The same approach is used for Contravariant Hom functor. But in the case we consider the set of morphisms from an arbitrary object x to the picked object a i.e. we reverse Arrows in this case.

9.1.1 Covariant Hom functor

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

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

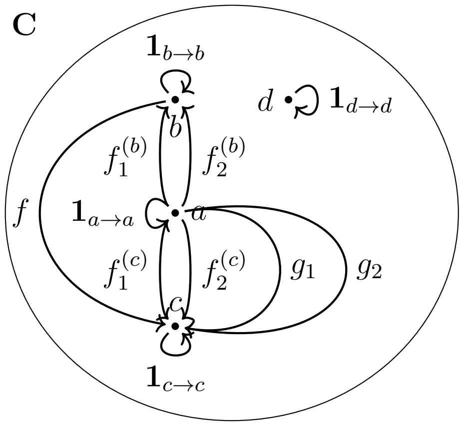

Figure 9.1: Covariant Hom functor $\text{Hom}_{\mathbf{C}}(a, -)$ example. Category **C**

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

Example 9.2 (Covariant Hom functor)**.** Consider category **C** in the fig. 9.1. It consists of 4 objects:

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

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

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

There is also a single Morphism f between b and c.

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

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

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

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

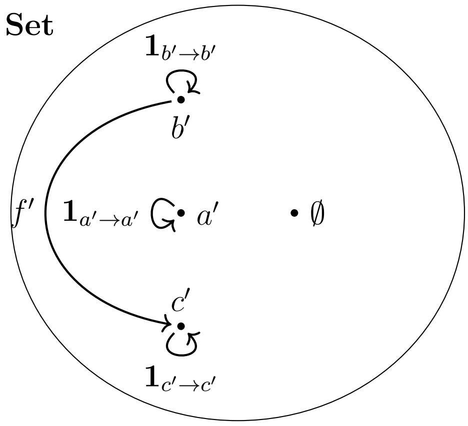

Figure 9.2: Covariant Hom functor $\mathrm{Hom}_{\mathbf{C}}(a, -)$ example. Category **Set**

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

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

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

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

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

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

9.1.2 Contravariant Hom functor

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

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

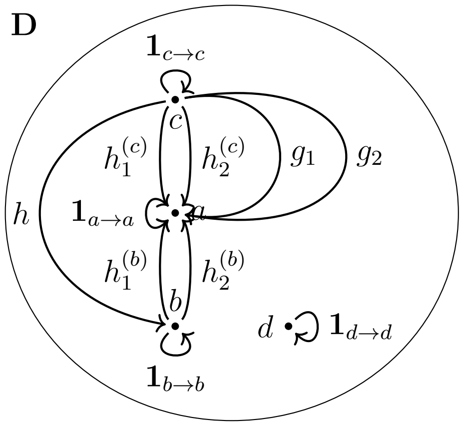

Figure 9.3: Contravariant Hom functor $\mathrm{Hom}_D(-, a)$ example. Category **D**

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

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

From the definition of Contravariant functor follows that we can get it simply reverting Arrows in the initial category. Lets do it for example 9.2 as follows

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

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

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

$$\begin{aligned} \mathrm{hom}_{\mathbf{D}}(a, a) = \{\mathbf{1}_{a\to a}\}, \\ \mathrm{hom}_{\mathbf{D}}(b, a) = \{h_1^{(b)}, h_2^{(b)}\}, \\ \mathrm{hom}_{\mathbf{D}}(c, a) = \{h_1^{(c)}, h_2^{(c)}, g_1 = h_1^{(b)} \circ h, g_2 = h_2^{(b)} \circ h\}, \\ \mathrm{hom}_{\mathbf{D}}(d, a) = \emptyset. \end{aligned}$$

There is also a single Morphism h between c and b.

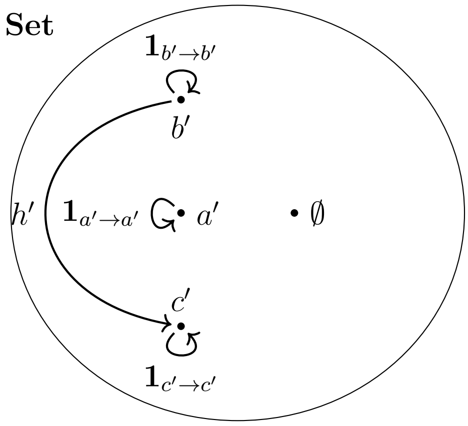

Figure 9.4: Contravariant Hom functor $\mathrm{Hom}_D(-, a)$ example. Category **Set**

The corresponding objects in the **Set** category is described in the fig. 9.4:

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

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

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

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

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

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

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

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

$$\operatorname{Im} h' \subsetneq \hom_{\mathbf{D}}(c, a).$$

9.1.3 Representable functor

Definition 9.5 (Representable functor)**.** Let $\mathbf{C}$ is a Locally small category. The functor $F : \mathbf{C} \Rightarrow \mathbf{Set}$ is called *representable* if it is naturally isomorphic (see Natural isomorphism) to $\operatorname{Hom}_{\mathbf{C}}(a, -)$ for some object $a \in \operatorname{ob}(\mathbf{C})$.

Representation of F is a pair (a, α) where

$$\alpha : \operatorname{Hom}_{\mathbf{C}}(a, -) \dot{\to} F$$

is a Natural isomorphism.

Example 9.6 (Representable functor)**.** **[Hask]** Consider a Representable functor F. we shall mark it as a small letter **f** in the example. [1] Representable functor is defined by a pair: (a, α) where a is the object from $\mathbf{C}$ and α is a Natural isomorphism. The first condition for a can be written as follows in **Hask** category

```
type Rep f :: *
```

where **Rep f** is the type a that represent our functor f.

The second condition for Natural isomorphism requires 2 Natural transformations:

$$\begin{aligned} \text{tabulate} &: \operatorname{Hom}_{\mathbf{C}}(a, -) \dot{\to} F, \\ \text{index} &: F \dot{\to} \operatorname{Hom}_{\mathbf{C}}(a, -). \end{aligned}$$

In Haskell the 2 functions can be written as follows

```
tabulate :: (Rep f -> x) -> f x
index :: f x -> Rep f -> x
```

From Reynolds (Theorem 5.15) we know that such functions are Natural transformations and therefore can be 2 parts of the required Natural isomorphism α. Combining these conditions together we can obtain the following definition for Representable functor in **Hask** category

```
class Representable f where
    type Rep f :: *
    tabulate :: (Rep f -> x) -> f x
    index :: f x -> Rep f -> x
```

[1] There is a requirement from Haskell to use small but not capital letter for it.

Consider the following type as a concrete example of the Representable functor

```
data Pair a = P a a
```

The representation type for **Pair** is **Bool**

```
instance Representable Pair where
  type Rep Pair = Bool

  index :: Pair a -> (Bool -> a)
  index (P x _) False = x
  index (P _ y) True  = y

  tabulate :: (Bool -> a) -> Pair a
  tabulate generate = P (generate False) (generate True)
```

Remark 9.7 (Functor logarithm)**.** Consider the category **Set**. Set of morphisms $\hom_{\mathbf{Set}}(a, x)$ is the same as the Exponential object x^a, i.e.

$$\hom_{\mathbf{Set}}(a, x) \cong x^a.$$

More generally, the same notation makes sense in a category where the exponential object x^a is defined and interpreted as the object of morphisms from a to x. Therefore, in such context, formally, we can write

$$\mathrm{Hom}_{\mathbf{C}}(a, -) \cong (-)^a.$$

If functor F is a Representable functor then

$$F \cong \mathrm{Hom}_{\mathbf{C}}(a, -) \cong (-)^a.$$

Thus we can define the logarithm operation for a Representable functor as follows

$$\log F = a.$$

9.2 Yoneda's lemma

Lemma 9.8 (Yoneda)**.** *Let* **C** *is a Locally small category and* F *is a functor from* **C** *to* **Set** *i.e.*

$$F \in \mathrm{ob}([\mathbf{C}, \mathbf{Set}])$$

and also we have

$$\mathrm{Hom}_{\mathbf{C}}(a, -) \in \mathrm{ob}([\mathbf{C}, \mathbf{Set}]).$$

Then

$$\hom_{[\mathbf{C},\mathbf{Set}]}(\mathrm{Hom}_{\mathbf{C}}(a, -), F) \cong F(a)$$

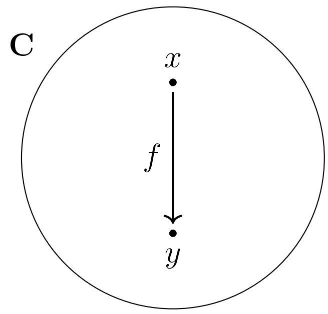

Figure 9.5: Category **C**. We look at 2 objects x and y and a morphism f between them

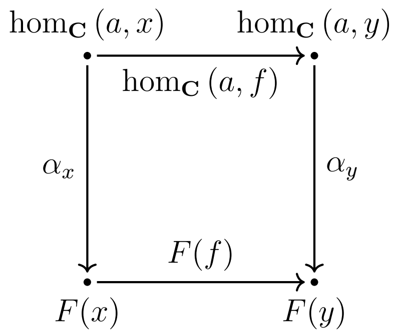

Figure 9.6: Commutative diagram for components of natural transformation α_x and α_y

Proof. Lets start with 2 objects x, y from category $\mathbf{C}$ and a morphism f between the 2 objects fig. 9.5.

Functor $\mathrm{Hom}_{\mathbf{C}}(a, -)$ maps x into $\mathrm{hom}_{\mathbf{C}}(a, x)$ and y into $\mathrm{hom}_{\mathbf{C}}(a, y)$. Functor F maps the 2 objects into $F(x)$ and $F(y)$ respectively. There is a Natural transformation α between the functors. I.e. $\alpha \in \mathrm{hom}_{[\mathbf{C}, \mathbf{Set}]}(\mathrm{Hom}_{\mathbf{C}}(a, -), F)$. We are interested in 2 components of the natural transformations:

$$\alpha_x : \mathrm{hom}_{\mathbf{C}}(a, x) \to F(x)$$

and

$$\alpha_y : \mathrm{hom}_{\mathbf{C}}(a, y) \to F(y).$$

The components of natural transformation should satisfy the naturality conditions (5.1) i.e. the commutative diagram fig. 9.6 should commute.

We can replace object x with a in $\mathrm{hom}_{\mathbf{C}}(a, x)$. The result set $\mathrm{hom}_{\mathbf{C}}(a, a)$ should contain Identity morphism $\mathbf{1}_{a \to a}$. Lets look how the morphism is mapped by the commutative diagram from fig. 9.6.

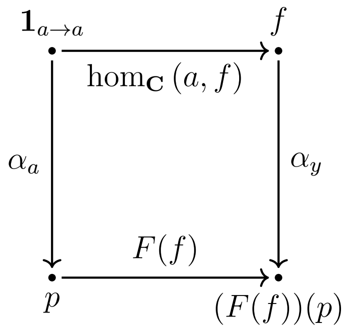

Figure 9.7: Mapping for $\mathbf{1}_{a\to a}$. The identity morphism is mapped into $p \in F(a)$ i.e. $p = \alpha_a(\mathbf{1}_{a\to a})$

As we can see in fig. 9.7, morphism α_a pick up an element p of the set $F(a)$. There is an arbitrary element that is determined by α_a. All others elements in fig. 9.7 is completely determined by the choice of p. From definition 9.1 we have

$$\hom_{\mathbf{C}}(a, f) = \{f \circ g | g \in \hom_{\mathbf{C}}(a, x)\}$$

i.e. if $g = \mathbf{1}_{a\to a}$ then

$$\hom_{\mathbf{C}}(a, f)(\mathbf{1}_{a\to a}) = f \circ \mathbf{1}_{a\to a} = f.$$

From other side we have mapping $F(f) : p \to q$ where $q = (F(f))(p)$. I.e. if we pick an arbitrary object $y \in \mathrm{ob}(\mathbf{C})$ when we can pick a morphism $f : a \to y$. This leads to the definition for an arbitrary component α_y of Natural transformation α as soon as only one component α_a is defined:

$$\alpha_y(f) = (F(f))(p).$$

Therefore from only one element $p \in F(a)$ we can got the Natural transformation $\alpha \in \hom_{[\mathbf{C},\mathbf{Set}]}(\mathrm{Hom}_{\mathbf{C}}(a, -), F)$. We also can go in other direction i.e. $\alpha_a(\mathbf{1}_{a\to a})$ will gives as an element p from the set $F(a)$. □

Chapter 10

Topos

Every Set can be considered from a categorical point of view (see Categorical approach) i.e. every set can be considered as a category. On the other hand not every category can be considered as a set. *Toposes* are categories that have all the properties required to be a set.

TBD

Appendices

Appendix A

Abstract algebra

A.1 Groups

Definition A.1 (Group)**.** Let us have a set of elements G with a defined binary operation $\circ$ that satisfies the following properties.

1. Closure: $\forall a, b \in G$: $a \circ b \in G$
2. Associativity: $\forall a, b, c \in G$: $a \circ (b \circ c) = (a \circ b) \circ c$
3. Identity element: $\exists e \in G$ such that $\forall a \in G$: $e \circ a = a \circ e = a$
4. Inverse element: $\forall a \in G$ $\exists a^{-1} \in G$ such that $a \circ a^{-1} = e$

In this case $(G, \circ)$ is called a group.

Therefore the group is a Monoid with inverse element property.

Example A.2 (Group $\mathbb{Z}/2\mathbb{Z}$)**.** Consider a set of 2 elements: $G = \{0, 1\}$ with the operation $\circ$ defined by the table A.1.

The identity element is 0 i.e. $e = 0$. Inverse element is the element itself because $\forall a \in G$: $a \circ a = 0 = e$.

Definition A.3 (Abelian group)**.** Let us have a Group $(G, \circ)$. The group is called an Abelian or commutative if $\forall a, b \in G$ it holds $a \circ b = b \circ a$.

$\circ$	0	1
0	0	1
1	1	0

Table A.1: Cayley table for $\mathbb{Z}/2\mathbb{Z}$

A.2 Rings and Fields

A.2.1 Rings

Definition A.4 (Ring)**.** Consider a set R with 2 binary operations defined. The first one $\oplus$ (addition) and the elements of R form an Abelian group under this operation. The second one is $\odot$ (multiplication) and the elements of R form a Monoid under the operation. The two binary operations are connected to each other via the following distributive law:

- Left distributivity: $\forall a, b, c \in R$: $a \odot (b \oplus c) = a \odot b \oplus a \odot c$
- Right distributivity: $\forall a, b, c \in R$: $(a \oplus b) \odot c = a \odot c \oplus b \odot c$

  The identity element for $(R, \oplus)$ is denoted as 0 (additive identity). The identity element for $(R, \odot)$ is denoted as 1 (multiplicative identity).

  The inverse element to a in $(R, \oplus)$ is denoted as $-a$

In this case $(R, \oplus, \odot)$ is called a ring.

The Ring is a generalization of the concept of integer numbers.

Example A.5 (Ring of integers $\mathbb{Z}$)**.** The set of integer numbers $\mathbb{Z}$ forms a Ring under $+$ and $\cdot$ operations i.e. addition $\oplus$ is $+$ and multiplication $\odot$ is $\cdot$. Thus for integer numbers we have the following Ring: $(\mathbb{Z}, +, \cdot)$

A.2.2 Fields

Definition A.6 (Field)**.** The ring $(R, \oplus, \odot)$ is called a field if $(R \setminus \{0\}, \odot)$ is an Abelian group.

The inverse element to a in $(R \setminus \{0\}, \odot)$ is denoted as a^{-1}

Example A.7 (Field $\mathbb{Q}$)**.** Note that $\mathbb{Z}$ is not a field because an inverse does not exist for every integer number. But if we consider a set of fractions $\mathbb{Q} = \{a/b \mid a \in \mathbb{Z}, b \in \mathbb{Z} \setminus \{0\}\}$ then it will be a field.

The inverse element to a/b in $(\mathbb{Q} \setminus \{0\}, \cdot)$ will be b/a.

A.3 Linear algebra

Definition A.8 (Vector space)**.** Let F is a Field. The set V is called as vector space under F if the following conditions are satisfied

1. We have a binary operation $V \times V \to V$ (addition): $(x, y) \to x + y$ with the following properties:

   (a) $x + y = y + x$
   (b) $(x + y) + z = x + (y + z)$
   (c) $\exists 0 \in V$ such that $\forall x \in V : x + 0 = x$
   (d) $\forall x \in V \exists - x \in V$ such that $x + (-x) = x - x = 0$

2. We have a binary operation $F \times V \to V$ (scalar multiplication) with the following properties
   (a) $1_F \cdot x = x$
   (b) $\forall a, b \in F, x \in V\colon\ a \cdot (b \cdot x) = (ab) \cdot x.$
   (c) $\forall a, b \in F, x \in V\colon\ (a + b) \cdot x = a \cdot x + b \cdot x$
   (d) $\forall a \in F, x, y \in V\colon\ a \cdot (x + y) = a \cdot x + a \cdot y$

Index

Bibliography

[1] Coecke, B. Introducing categories to the practicing physicist.

[2] Di Cosmo, R., and Miller, D. Linear logic, 2016.

[3] Feynman, R., Hibbs, A., and Styer, D. *Quantum Mechanics and Path Integrals*. Dover Books on Physics. Dover Publications, 2010.

[4] Gonzalez, G. Scalable program architectures, 2014.

[5] (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).

[6] (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).

[7] (https://math.stackexchange.com/users/455216/enkidu), E. Terminal and initial objects in the category of functors. Mathematics Stack Exchange. URL:https://math.stackexchange.com/q/3033845 (version: 2018-12-10).

[8] Ivan Murashko, Alexey Radkov, M. C++ examples for category theory by example book, 2018.

[9] Ivan Murashko, Alexey Radkov, M. Haskell examples for category theory by example book, 2018.

[10] Ivan Murashko, Alexey Radkov, M. Scala examples for category theory by example book, 2018.

[11] Lane, S. *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer New York, 1998.

[12] LEIGHTON, T., AND RUBINFELD, R. What is a proof?

[13] LEX SHEEHAN. What's the significance of monoids?, 2017.

[14] MILEWSKI, B. *Category Theory for Programmers*. Bartosz Milewski, 2018.

[15] MOGGI, E. Notions of computation and monads. *Inf. Comput. 93*, 1 (1991), 55–92.

[16] MURASHKO, I. Lectures notes in introduction to galois theory by ekaterina amerik, 2016-2019.

[17] MURASHKO, I. Category theory by example, 2018.

[18] NLAB AUTHORS. whiskering. http://ncatlab.org/nlab/show/whiskering, sep 2018. Revision 11.

[19] PROOFWIKI. Empty mapping is unique.

[20] PROOFWIKI. Injection iff monomorphism in category of sets.

[21] PROOFWIKI. Surjection iff epimorphism in category of sets.

[22] QUIST, M. Writing and classifying true, false and open statements in math.

[23] REYNOLDS, J. C. Types, abstraction and parametric polymorphism. In *IFIP Congress 1983* (1983), North-Holland, pp. 513–523.

[24] USER84563 (HTTPS://MATH.STACKEXCHANGE.COM/USERS/301130/USER84563). Does an(/the?) empty category exist? Mathematics Stack Exchange. URL:https://math.stackexchange.com/q/1997015 (version: 2016-11-03).

[25] WIKIPEDIA. Disjoint union — wikipedia, the free encyclopedia, 2017. [Online; accessed 13-April-2017].

[26] WIKIPEDIA CONTRIBUTORS. Cone (category theory) — Wikipedia, the free encyclopedia, 2018. [Online; accessed 10-November-2018].

[27] WIKIPEDIA CONTRIBUTORS. Distributive category — Wikipedia, the free encyclopedia, 2018. [Online; accessed 14-October-2018].

[28] WIKIPEDIA CONTRIBUTORS. Russell's paradox — Wikipedia, the free encyclopedia, 2018. [Online; accessed 29-July-2018].

[29] Wikipedia contributors. Zermelo–fraenkel set theory — Wikipedia, the free encyclopedia, 2018. [Online; accessed 29-July-2018].

[30] Wikipedia contributors. Coproduct — Wikipedia, the free encyclopedia, 2019. [Online; accessed 5-May-2019].

[31] Wikipedia contributors. Principle of least action — Wikipedia, the free encyclopedia, 2019. [Online; accessed 30-May-2019].

[32] Wikipedia contributors. Snell's law — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Snell%27s_law&oldid=908247962, 2019. [Online; accessed 20-August-2019].

[33] Мурашко И. В. *Квантовая оптика*. 2018.

Creative Commons Legal Code

Attribution-NonCommercial 3.0 Unported

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

License

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

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

1. Definitions

(a) **"Adaptation"** means a work based upon the Work, or upon the Work and other pre-existing works, such as a translation, adaptation, derivative work, arrangement of music or other alterations of a literary or artistic work, or phonogram or performance and includes cinematographic adaptations or any other form in which the Work may be recast, transformed, or adapted including in any form recognizably derived from the original, except that a work that constitutes a Collection will not be considered an Adaptation for the purpose of this License. For the avoidance of doubt, where the Work is a musical work, performance or phonogram, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered an Adaptation for the purpose of this License.

(b) **"Collection"** means a collection of literary or artistic works, such as encyclopedias and anthologies, or performances, phonograms or broadcasts, or other works or subject matter other than works listed in Section 1(f) below, which, by reason of the selection and arrangement of their contents, constitute intellectual creations, in which the Work is included in its entirety in unmodified form along with one or more other contributions, each constituting separate and independent works in themselves, which together are assembled into a collective whole. A work that constitutes a Collection will not be considered an Adaptation (as defined above) for the purposes of this License.

(c) **"Distribute"** means to make available to the public the original and copies of the Work or Adaptation, as appropriate, through sale or other transfer of ownership.

(d) **"Licensor"** means the individual, individuals, entity or entities that offer(s) the Work under the terms of this License.

(e) **"Original Author"** means, in the case of a literary or artistic work, the individual, individuals, entity or entities who created the Work or if no individual or entity can be identified, the publisher; and in addition (i) in the case of a performance the actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret or otherwise perform literary or artistic works or expressions of folklore; (ii) in the case of a phonogram the producer being the person or legal entity who first fixes the sounds of a performance or other sounds; and, (iii) in the case of broadcasts, the organization that transmits the broadcast.

(f) **"Work"** means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same

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

(a) to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections;

(b) to create and Reproduce Adaptations provided that any such Adaptation, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original Work. For example, a translation could be marked "The original work was translated from English to Spanish," or a modification could indicate "The original work has been modified.";

(c) to Distribute and Publicly Perform the Work including as incorporated in Collections; and,

(d) to Distribute and Publicly Perform Adaptations.

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

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

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

(b) You may not exercise any of the rights granted to You in Section 3 above in any manner that is primarily intended for or directed toward commercial advantage or private monetary compensation. The exchange of the Work for other copyrighted works by means of digital file-sharing or otherwise shall not be considered to be intended for or directed toward commercial advantage or private monetary compensation, provided there is no payment of any monetary compensation in connection with the exchange of copyrighted works.

(c) If You Distribute, or Publicly Perform the Work or any Adaptations or Collections, You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or Licensor designate another party or parties (e.g., a sponsor institute, publishing entity, journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and, (iv) consistent with Section 3(b), in the case of an Adaptation, a credit identifying the use of the Work in the Adaptation (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). The credit required by this Section 4(c) may be implemented in any reasonable manner; provided, however, that in the case of a Adaptation or Collection, at a minimum such credit will appear, if a credit for all contributing authors of the Adaptation or Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original

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

(d) For the avoidance of doubt:

i. **Non-waivable Compulsory License Schemes**. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;

ii. **Waivable Compulsory License Schemes**. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License if Your exercise of such rights is for a purpose or use which is otherwise than noncommercial as permitted under Section 4(b) and otherwise waives the right to collect royalties through any statutory or compulsory licensing scheme; and,

iii. **Voluntary License Schemes**. The Licensor reserves the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License that is for a purpose or use which is otherwise than noncommercial as permitted under Section 4(c).

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

5. Representations, Warranties and Disclaimer

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

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

7. Termination

(a) This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Adaptations or Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

(b) Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

(a) Each time You Distribute or Publicly Perform the Work or a Collection, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

(b) Each time You Distribute or Publicly Perform an Adaptation, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

(c) If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

(d) No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

(e) This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

(f) The rights granted under, and the subject matter referenced, in this License were drafted utilizing the terminology of the Berne Convention for the Protection of Literary and Artistic Works (as amended on September 28, 1979), the Rome Convention of 1961, the WIPO Copyright Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996 and the Universal Copyright Convention (as revised on July 24, 1971). These rights and subject matter take effect in the relevant jurisdiction in which the License terms are sought to be enforced according to the corresponding provisions of the implementation of those treaty provisions in the applicable national law. If the standard suite of rights granted under applicable copyright law includes additional rights not granted under this License, such additional rights are deemed to be included in the License; this License is not intended to restrict the license of any rights under applicable law.