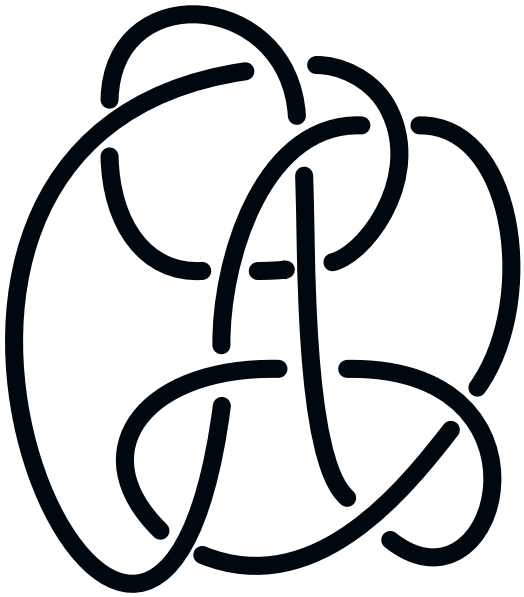

Category Theory and Functional Programming

Code examples in Julia

Davi S. Barreira

Category Theory and Functional Programming

Code examples in Julia

by

Davi Sales Barreira

First edition, 2023

ISBN

Published by

Preface

The goal of these notes is to introduce Category Theory with mathematical rigour, and then show how this theory is used within the Functional Programming paradigm. We use Julia as our programming language to implement our examples. This is specially helpful because Julia is easy to understand and it's *not* made to be "functional first". Thus, we'll have to "do the work" in order to construct the concepts of Functional Programming.

These notes are similar to Milewski [4], but differ in the sense that we are both more mathematically formal and (strangely) more pragmatically inclined. Our examples in Julia aim not only to highlight the category theoretical concept, but also the strength of Functional Programming.

The notes hop over theory and practice. We always start with the theory and then showcase the programming examples and how it ties to Functional Programming. Readers not interested only in the mathematical Category Theory can skip the programming chapters.

Contents

List of Definitions

List of Theorems

Notation

The symbol "⊛" means that such definition or theorem was created by the author, so it should be taken with care.

We usually refer to generic categories as $\mathcal{C}$ and similar uppercase letters with curly font. For named categories (e.g. category of graphs, category of small categories,...) we usually use a bold font with no italic, e.g. $\mathbf{Gr}, \mathbf{SmCat}, \mathbf{Top}$...

1. $\circ$ - Used to symbolize composition of morphisms similar to how we compose functions.
2. $\fatsemi$ - Represents composition, but with orders reversed, i.e. $g \circ f = f \fatsemi g$.
3. $\diamond$ - *Whiskering, prewhiskering, postwhiskering.*
4. $\cong$ - Isomorphism in the context applied (e.g. categorical, set theoretical, etc).
5. $\simeq$ - Equivalence of categories via natural transformations (weaker than isomorphism).

Chapter 1

Category Theory - The Very Basic

The study of Category Theory enables us to view Mathematics from a vantage point, and better understand how the different areas are connected. By looking at the subject from the distance (via Category Theory), we get a glimpse at the connections (and disconnections) between different fields.

Another interesting observation about Category Theory is that it's breaking the theoretical barrier and it's starting to be applied in real world applications. One prominent example is in programming, as highlighted by the book Milewski [4]. The book by Fong and Spivak [3] also focus on the application side of Category Theory.

This chapter introduces the field of Category Theory by presenting the first three components of the theory, i.e. categories, objects, and morphisms.

1.1 Universes, Sets and Classes

This section is a bit technical. The goal is to formally define the difference between a set and a class. This is a nuanced point, but a necessary one if we wish to do things with rigour.

When working on Category Theory, it's common to find universal statements such as "for all topological spaces...". The issue with such statements is that, in a purely set-theoretical sense, we have to know whether such large collection ("all topological spaces") is indeed a set. We might be tempted to say that's true, but it's not so simple. Russel's Paradox of whether there is a set of all sets is an example of when the answer is "no".

Therefore, in order to deal with such issue, we need a way to differentiate between a valid set and an arbitrary collection. Here is where the notion of a Universe comes in.

Definition 1.1.1 (Universe). We say that a set $\mathfrak{U}$ is a universe if[1]

(i) $x \in y$ and $y \in \mathfrak{U}$, then $x \in \mathfrak{U}$;

(ii) $I \in \mathfrak{U}$, and $\forall i \in I, x_i \in \mathfrak{U}$, then $\cup_{i \in I} x_i \in \mathfrak{U}$;

(iii) $x \in \mathfrak{U}$ then $\mathcal{P}(x) \in \mathfrak{U}$, where $\mathcal{P}(x)$ is the power set;

(iv) $x \in \mathfrak{U}$ and $f : x \to y$ is a surjective function, then $y \in \mathfrak{U}$;

(v) $\mathbb{N} \in \mathfrak{U}$.

From this definition, one can prove the following proposition.

Proposition 1.1.2. Let $\mathfrak{U}$ be a universe. Therefore:

(i) $x \in \mathfrak{U}$ and $y \subset x$, then $y \in \mathfrak{U}$;

(ii) $x \in \mathfrak{U}$ and $y \subset x$, then $\{x, y\} \in \mathfrak{U}$;

(iii) $x \in \mathfrak{U}$ and $y \subset x$, then $x \times y \in \mathfrak{U}$;

(iv) $x \in \mathfrak{U}$ and $y \subset x$, then $y^x \in \mathfrak{U}$, where y^x is the set of functions $f : x \to y$.

With this definition, we state the axiom of existence of universes.

Axiom 1.1.1. Every set S belongs to some universe $\mathfrak{U}$.

Definition 1.1.3. For a fixed universe $\mathfrak{U}$, if a set S is an element of $\mathfrak{U}$, then S is called a *small set*.

Talking about "small sets" and "big set" might become daunting, so instead, we use a different convention which is based on Gödel-Bernays theory of sets and classes. This theory states that:

Axiom 1.1.2 (Gödel-Bernays). A class is a set if and only if it belongs to some (other) class.

Note that using the notion of Universes, we can recover Gödel-Bernays theory. For that, use the following definition:

Definition 1.1.4. For a fixed universe $\mathfrak{U}$, we call S a *set* if it's an element of $\mathfrak{U}$, and call S a *class* if it's a subset of $\mathfrak{U}$. A class that is not a set is called a *proper class*.

From now on, whenever we say *set* we are implying *small set* and whenever we say *class* we are implying either small or big sets, following Borceux [2] convention.

[1] Definition from Borceux [2]

1.2 What is a Category?

Let's formally define a category and provide some examples.

Definition 1.2.1 (Category). A category $\mathcal{C} = \langle Ob_{\mathcal{C}}, Mor_{\mathcal{C}} \rangle$ consists of a class of objects $Ob_{\mathcal{C}}$ and a class of morphisms $Mor_{\mathcal{C}}$ satisfying the following conditions:

(i) Every morphism $f \in Mor_{\mathcal{C}}$ is associated to two objects $X, Y \in Ob_{\mathcal{C}}$ which is represented by $f : X \to Y$ or $X \xrightarrow{f} Y$, where $dom(f) = X$ is called the domain of f and $cod(f) = Y$ is the codomain. Moreover, we define $Mor_{\mathcal{C}}(X, Y)$ as

$$Mor_{\mathcal{C}}(X, Y) := \{f \in Mor_{\mathcal{C}} \ : \ X \in dom(f), \ Y \in cod(f)\};$$

(ii) For any three objects $X, Y, Z \in Ob_{\mathcal{C}}$, there exists a composition operator

$$\circ : Mor_{\mathcal{C}}(X, Y) \times Mor_{\mathcal{C}}(Y, Z) \to Mor_{\mathcal{C}}(X, Z),$$

(iii) For each object $X \in Ob_{\mathcal{C}}$ there exists a morfism $id_X \in Mor_{\mathcal{C}}(A, A)$ called the identity.

The composition operator must have the following properties:

(p.1) *Associative*: for every $f \in Mor_{\mathcal{C}}(A, B), g \in Mor_{\mathcal{C}}(B, C), h \in Mor_{\mathcal{C}}(C, D)$ then

$$h \circ (g \circ f) = (h \circ g) \circ f.$$

(p.2) For any $f \in Mor_{\mathcal{C}}(X, Y)$, $g \in Mor_{\mathcal{C}}(Y, X)$,

$$f \circ id_X = f, \quad id_Y \circ g = g.$$

There are many ways to refers to the class of morphisms $Mor_{\mathcal{C}}(X, Y)$, such as $\mathcal{C}(X, Y)$ or $\hom_{\mathcal{C}}(X, Y)$. The reason for this is that this set is sometimes called hom-set. In these notes, we'll use either $Mor_{\mathcal{C}}(X, Y)$ or $\mathcal{C}(X, Y)$ when there is no ambiguity. Also, we'll use dom_f to mean $dom(f)$, and similarly for the codomain.

Another point about conventions. When talking about composition, it's convenient to use the operator $\fatsemi$, which is equivalent to the composition $\circ$, but with the inverted order, i.e. $f \fatsemi g = g \circ f$. The convinience will become clearer once we introduce Hasse diagrams as a way to represent categories.

Definition 1.2.2 (Small Category). A category $\mathcal{C}$ is *small* if $Ob_{\mathcal{C}}$ and $Mor_{\mathcal{C}}$ are sets (small).

Definition 1.2.3 (Locally Small Category). A category $\mathcal{C}$ is *locally small* if for any $A, B \in Ob_{\mathcal{C}}$, then $Mor_{\mathcal{C}}(A, B)$ is a set (small).

1.2.1 Examples of Categories

It's very common to represent categories via Hasse Diagrams. In these diagrams, the objects are represented as dots, and the morphisms as arrows. Let's show some examples.

Example 1.2.1 (Category 1). The category **1** consists of $Ob_{\mathbf{1}} := \{A\}$ and $Mor_{\mathbf{1}} = \{id_A\}$. The diagram for such category is shown below.

Figure 1.1: Hasse diagram of category **1**.

Example 1.2.2 (Category 2 and 3). The category **2** consists of $Ob_{\mathbf{2}} := \{A, B\}$ and $Mor_{\mathbf{1}} = \{id_A, id_B, f\}$, where $f : A \to B$. The diagram for such category is shown below.

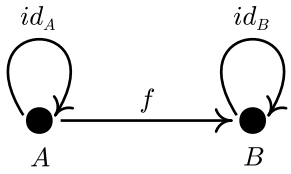

Figure 1.2: Hasse diagram of category **2**.

Since we know that identities are always present in categories, we'll omit them from future diagrams when there is no ambiguity. Thus, the figure below represents the same diagram as Figure 1.2.

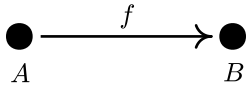

Figure 1.3: Hasse diagram of category **2** omitting identity morphisms.

The category **3** has three morphisms besides the identities, given by f, g and their composition $f \fatsemi g$. The figure below illustrates the category with all it's morphisms.

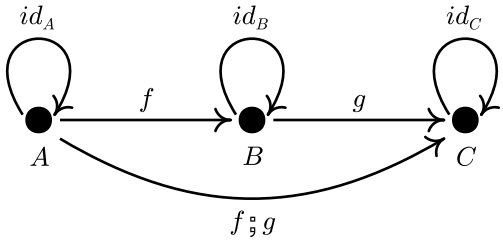

Figure 1.4: Hasse diagram of category **3** showing all morphisms.

Drawing all the morphisms can make the diagram become too crowded, specially as the number of objects and morphisms grows. Hence, we simplify the diagram representation by omitting not only the identity morphisms, but also the compositions. These can always be assumed to exist, since they are a necessary condition for every category. Thus, the figure below represents the same diagram as Figure 1.4.

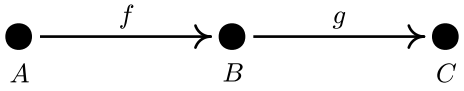

Figure 1.5: Hasse diagram of category **3** omitting identities and compositions.

Example 1.2.3 (Discrete Categories). The discrete category $\underline{\mathbf{N}}$ is the category with N objects and $Mor_{\underline{\mathbf{N}}} := \{id_1, ..., id_N\}$. An example of this category is illustrated below.

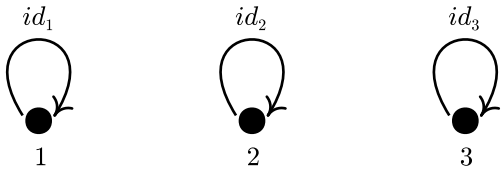

Figure 1.6: Example of category $\underline{\mathbf{3}}$.

Example 1.2.4 (Preorders). A preorder is defined by a tuple $(P, \leq)$, where P is a set of values, such that

(i) For $a, b \in P$, if $a \leq b$ and $b \leq c$, then $a \leq c$;

(ii) For every $a \in P$, $a \leq a$.

We can show that actually, this is a category, which we'll call $\mathcal{P}$, where $Ob_{\mathcal{P}} = P$ and each morphism f represents $a \leq b$, where $cod_f = a$ and $dom_f = b$. One example of preorder is the set of $\mathbb{N}$ equiped with the binary relation $\leq$ which is shown in the diagram below.

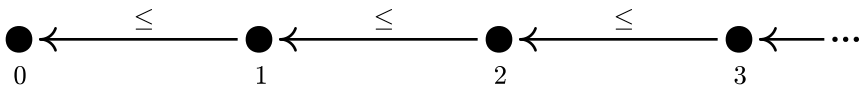

Figure 1.7: Hasse diagram of preorder category of natural numbers.

Note that in preorders, there is at most one morphism between each pair of objects. Thus, categories with such property are often referred as *thin categories* or *preorder category* (in Fong and Spivak [3], the authors call this a *preorder reflection*).

Example 1.2.5 (Monoids). A monoid is a triple $(M, \oplus, e_M)$ where $\oplus : M \times M \to M$ is a binary operation and e_M the neutral element, such that:

1. $a \oplus (b \oplus c) = (a \oplus b) \oplus c$
2. $a \oplus e_M = e_M \oplus a = a$.

Note that $(\mathbb{N} \cup \{0\}, +, 0)$ is a monoid.

Moreover, each single monoid can be defined as a category itself. For monoid $(M, \oplus, e_M)$, define a category $\mathcal{C}$ such that $Ob_{\mathcal{C}} := \{M\}$, and the set of morphisms are the elements of M, i.e. $Mor_{\mathcal{C}} := \{a \in M\}$. Finally, we define the composition operartion as $a \circ b := a \oplus b$ Thus, $(\mathbb{N} \cup \{0\}, +, 0)$ is the category illustrated in Figure 1.8.

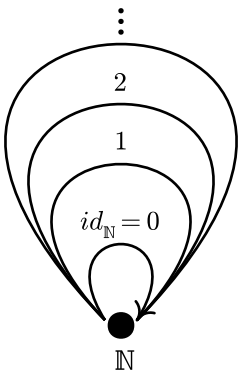

Figure 1.8: Hasse diagram of monoid category of natural numbers.

There are many other examples:

1. **Set** which is the category of sets, where the objects are sets and the morphisms are functions between sets.

2. **Top** is the category where topological spaces are the objects and continuous functions are the morphisms.

3. $\mathbf{Vec}_{\mathbb{F}}$ is the category where vector spaces over field $\mathbb{F}$ are the objects, and linear transformations are the morphisms.

4. **Gr** is the category of directed graphs, where $Ob_{\mathbf{Gr}} := \{\text{Vertex}, \text{Arrow}\}$, and the morphisms are
$$Mor_{\mathbf{Gr}} := \{src, tgt, id_{\text{Vertex}}, id_{\text{Arrow}}\}$$
where $src : \text{Arrow} \to \text{Vertex}$ returns the source vertex for each arrow and $tgt : \text{Arrow} \to \text{Vertex}$ returns the target vertex.

1.2.2 Isomorphism, monomorphism and epimorphism

A very important definition in Category Theory is the notion of isomorphism. In Set Theory, we say that two sets are isomorphic if there is a bijective function between them. Yet, this concept is not restricted to Set Theory and can be generalized in Category Theory as follows:

Definition 1.2.4 (Categorical Isomorphism). Let $\mathcal{C}$ be a category with $X, Y \in Ob_{\mathcal{C}}$ and $f \in Mor_{\mathcal{C}}(X, Y)$.

(i) We say that f is *left invertible* if there exists $f_l \in Mor_{\mathcal{C}}(Y, X)$ such that $f_l \circ f = id_X$;

(ii) We say that f is *right invertible* if there exists $f_r \in Mor_{\mathcal{C}}(Y, X)$ such that $f \circ f_r = id_Y$;

(iii) We say that f is invertible if it's both left and right invertible.

When an invertible morphism exists between X and Y, we say that they are isomorphic.

Proposition 1.2.5. The following properties on inverses are true:

1. If f is an invertible morphism, then the left and right inverses are the same.

2. If f and g are invertible and composable, then $f \circ g$ is also invertible.

Proof. 1. Let f be invertible with left inverse f_l and right inverse f_r. Therefore,
$$f_l \circ id_Y = f_l \circ f \circ f_r = id_X \circ f_r = f_r.$$

2. Let $f : A \to B$ and $g : B \to C$ be invertible and composable, with $f \circ g : A \to C$. There exists the inverses $g^{-1} : C \to B$ and $f^{-1} : B \to A$. Note that $f^{-1} \circ g^{-1} : C \to A$, thus
$$(f^{-1} \circ g^{-1}) \circ (g \circ f) = f^{-1} \circ (g^{-1} \circ g) \circ f = (f^{-1} \circ id_B) \circ f = f^{-1} \circ f = id_A.$$

$$(g \circ f) \circ (f^{-1} \circ g^{-1}) = g \circ (f \circ f^{-1}) \circ g^{-1} = g \circ (id_B \circ g^{-1}) = g \circ g^{-1} = id_B.$$

We conclude that $(g \circ f)^{-1} = f^{-1} \circ g^{-1}$.

□

Although similar to set isomorphism, categorical isomorphism is in a sense more general, and captures our intuition of isomorphism between categories better than the set theoretic case, even when we have finite objects.

Consider the following example. Let $\mathcal{P}$ be the category of Posets, where posets $(P, \leq_p)$ are the objects. Take two objects $P_1, P_2 \in Ob_{\mathcal{P}}$, where $P_1 := \{a, b\}$ with a and b **not** comparable, and $P_2 := \{0, 1\}$ where indeed $0 \leq 1$. The question is whether P_1 is "actually" isomorphic to P_2, and our intuition say that they should not be, since P_1 has two incomparable elements while P_2 has two comparable elements.

If we use the set theoretic definition, we would conclude that they **are** isomorphic, since there is a bijective function between P_1 and P_2. Take for example $f : P_1 \to P_2$ where $f(a) = 0$ and $f(b) = 1$. So the set theoretic isomorphism does not capture what we want. What about the categorical isomorphism? We can prove that this will not be an isomorphism using the categorical definition. Yet, in order to prove this, we need to specify what are the morphisms between the posets, and to do this, we need to define what are functors.

In the same way that set isomorphism is not the same as categorical isomorphism, the notions of injectivity and surjectivity are not equivalent to their categorical counterparts, which are called monomorphism and epimorphism.

Definition 1.2.6 (Monomorphism). Let $\mathcal{C}$ be a category and $f \in Mor_{\mathcal{C}}(A, B)$. We say that f is a monomorphism (or monic), if

$$f \circ g = f \circ h \implies g = h.$$

Definition 1.2.7 (Epimorphism). Let $\mathcal{C}$ be a category and $f \in Mor_{\mathcal{C}}(A, B)$. We say that f is an epimorphism (or epic), if

$$g \circ f = h \circ f \implies g = h.$$

Important! A morphism f can be both epic and monic, without being an isomorphism, which again highlights the difference between this concepts and their set-theoretic counterparts. A counter example for this is the inclusion $\mathbb{Z} \hookrightarrow \mathbb{Q}$ in the category **Ring**. This category has rings as objects and ring homomorphisms as morphisms. Note that the inclusion is a homomorphism from $\mathbb{Z}$ to $\mathbb{Q}$ that is both monic and epic, yet, there is no homomorphism from $\mathbb{Q}$ to $\mathbb{Z}$.

Proposition 1.2.8. The following properties on monomorphism and epimorphism are true:

1. f left-invertible $\implies$ f is monic. The converse is not true.

2. f right-invertible $\implies$ f is epic. The converse is not true.

3. f invertible $\implies$ f is monic and epic. The converse is not true.

4. f monic and right-invertible $\implies$ f is isomorphism.

5. f epic and left-invertible $\implies$ f is isomorphism.

Proof. 1. Note $f : A \to B$ left-invertible implies that there exists a $f_l : B \to A$ such that $f_l \circ f = id$. Hence, for a $g : B \to C$ and $h : B \to C$, if

$$f \circ g = f \circ h,$$

then we have that

$$f_l \circ f \circ g = f_l \circ f \circ h \implies g = h.$$

To show that the converse is false, consider the category **2** (Figure 1.2). Note that $f : A \to B$ is monic, since the only morphism that composes with f is id_A and id_B. Yet, note that f is not left invertible, since there isn't even a morphism from B to A.

2. Use the same argument, but reversing the order of the compositions. For the converse, again consider the same category **2**. Note that $f : A \to B$ is epic, but it's not right invertible.

3. True since invertible means left and right invertible.

4. Since $f : A \to B$ right invertible, then there exists $f_r : B \to A$ such that $f \circ f_r = id_B$. Thus,

$$id_B \circ f = (f \circ f_r) \circ f = f \circ (f_r \circ f) = f \circ id_A \implies f_r \circ f = id_A.$$

5. Same argument. □

Lastly, let's introduce two definitions that are helpful when talking about morphisms.

Definition 1.2.9 (Endomorphism and Automorphism). A morphism is called an endomorphism is the domain and codomain object are the same. If this endomorphism is an isomorphism, then we call it an automorphism.

1.2.3 Zero, Initial and Terminal Objects

Definition 1.2.10 (Zero, Initial and Terminal). Let $\mathcal{C}$ be a category.

1. An object $I \in Ob_{\mathcal{C}}$ is *initial* if for every $A \in Ob_{\mathcal{C}}$, there is exactly one morphism from I to A. Thus, from I to I there is only the identity.

2. An object $T \in Ob_{\mathcal{C}}$ is *terminal* if for every $A \in Ob_{\mathcal{C}}$, there is exactly one morphism from A to T. Thus, from I to I there is only the identity.

3. An object is *zero* if it's both terminal and initial.

Note that in the definitions above, we are defining these objects in terms of existence and uniqueness of morphisms, which is known in category theory as **universal constructions** (more on this later).

Theorem 1.2.11. Every *initial* object is unique up to an isomorphism, i.e. if in a category there are two *initial* objects, then they are isomorphic. Similarly, *terminal* objects are unique up to an isomorphism. Moreover, the isomorphism is unique between initial object, and between terminal objects.

Proof. Let I_1, I_2 be initial. Then, there exists only $f : I_1 \to I_2$ and $g : I_2 \to I_1$. But since $g \circ f : I_1 \to I_1$ is a morphism from the initial object I_1, it must be equal to id_{I_1}. The same for I_2, which implies that f and g are inverses, and thus the objects are isomorphic. Since both f and g are the only morphisms from I_1 and I_2, this also implies that their are the only isomorphism. The same proof works for terminal objects. □

Example 1.2.6 (Terminal and Initial Objects in Set). Without thinking too much, one might assume that in the category **Set** the empty set would be a zero object; but that's not true. In reality, the empty set is the initial object, since $f : \varnothing \to B$ is the only function from the empty set to any other set. Why is this valid?

Remember that in set theory, a function from two sets is defined as a binary relation such that for every $x \in dom_f$, there is a unique $y \in cod_f$, i.e. f is a triple (A, B, G), where $A = dom_f$, $B = cod_f$ and $G \subset A \times B$ such that $\forall x \in dom_f$, there exists a unique $y \in B$, such that $(x, y) \in G$.

Since $dom_f = \varnothing$, we have that $G \subset \varnothing \times B$, but this is actually empty. Why? If $\varnothing \times B$ is not empty, then there exists $(a, b) \in \varnothing \times B$, which is false, since this would imply that $a \in \varnothing$, which contradicts the definition of the empty set that says that it can have no elements (note that $\varnothing \in \varnothing$ is actually false).

With this, we have that $G = \varnothing$, thus, the only possible function from $\varnothing$ to B is $f = (\varnothing, \varnothing, B)$. Which proves that the empty set is initial.

But what about terminal? The empty set actually does not have any morphisms that arrives on it, since there is no function $f : A \to \varnothing$. The terminal sets in **Set** are actually all the singletons (sets with only one element), since for any $\{a\}$, there will be only one function $g : A \to \{a\}$, which is $g(x) = a$.

Another definition we have is that of a *zero* morphism. The idea here is that this morphism must take the elements of an object A to the zero element in B, for example, a for two vector spaces $\mathbb{R}^n$ and $\mathbb{R}^m$, the zero linear transformation $z : \mathbb{R}^n \to \mathbb{R}^m$ should take every vector n-dimensional vector to the $\mathbf{0}$ m-dimensional vector. In Category Theory we do not talk about morphisms according to how they act on the elements, but only in the objects.

So we cannot define z by saying to which element it maps. Yet, there is a way to do this in Category Theory, which gives rise to the *zero* morphism definition.

Definition 1.2.12 (Zero Morphism). Let $\mathcal{C}$ be a category, and 0 be a zero object. A morphism $z : A \to B$ is a zero morphism if there exists two morphisms $f : A \to 0$ and $g : 0 \to B$, such that

$$z = g \circ f.$$

See that this makes intuitive sense. In our example, since we wish to take $v \in \mathbb{R}^n$ to $0 \in \mathbb{R}^m$, we first take all v to the zero object, which in the category of vector spaces will be the zero vector space, i.e. $\{0\}$ the space where $0 \in \mathbb{R}$ is the only element. So now all v are 0. Note that every linear transformation from $\{0\}$ to $\mathbb{R}^m$ must take 0 to $\mathbf{0} \in \mathbb{R}^m$, otherwise, suppose that $g(0) = \mathbf{v} \neq \mathbf{0}$, hence for a scalar α,

$$g(0) = g(\alpha 0) = \mathbf{v} \neq \alpha \mathbf{v} = \alpha g(0).$$

This is a contradiction, since g is a linear transformation.

Theorem 1.2.13. Let $\mathcal{C}$ be a category with zero object 0. Then there exists a unique zero morphism between any two objects.

Proof. Let $A, B \in Ob_{\mathcal{C}}$. By the definition of the zero object, there exists a unique $f : A \to 0$ and $g : 0 \to B$, thus, $g \circ f$ is a zero morphism by definition and is unique, since there is no other f and g with these respective domain and codamain.

Moreover, note that if $z : A \to B$ is our zero morphism and $h : B \to C$, then

$$h \circ z = h \circ (g \circ f) = (h \circ g) \circ f.$$

But, $l = (h \circ g) : 0 \to C$, which means that $l \circ f$ is a zero morphism. The same argument works with a composition from the other direction. This means that compositions with zero morphisms return zero morphisms. □

1.2.4 Understanding Duality

In several fields of Mathematics, one is faced with the informal notion of a dual. Mathematicians define a concept, and call them the dual in some sense, for example, the dual vector space, the dual of an optimization problem, and many more. I always found puzzling what exactly held these things together, i.e. what was the underlying principle that made something a dual of another.

Fortunately, Category Theory has a very elegant answer. For a given category $\mathcal{C}$, the dual category is denoted by $\mathcal{C}^{op}$ where are the objects are the same, but the morphisms are inverted. This means that $Ob_{\mathcal{C}} = Ob_{\mathcal{C}^{op}}$, and for every $f \in Mor_(A, B)$, we have $f^{op} \in Mor_{\mathcal{C}^{op}}(B, A)$.

This definition gives rise to a very interesting result (observation), called the *Duality Principle*.

Definition 1.2.14 (Dual Property and Dual Statement). We say p^{op} is the dual property for p if for all categories

$$\mathcal{C} \text{ has } p^{op} \iff \mathcal{C}^{op} \text{ has } p.$$

For a statement s about a category $\mathcal{C}$, the dual statement is the same statement, but with regards to $\mathcal{C}^{op}$.

For example, "a category has an initial object if and only if the dual category has a terminal object". In this example, the property of having an initial object is the dual property of having a terminal object, since the above statement is true for any category. What about the dual statement? The dual for the statement "the category $\mathcal{C}$ has an initial object" is "the category $\mathcal{C}^{op}$ has an initial object". Note that the dual statement is not always true. And here is where we get the duality principle.

Definition 1.2.15 (Duality Principle). If a statement s is true for every category, then the dual statement is also true for every category.

Let's digest a bit what this principle states. If we can prove that a certain statement is true for any arbitrary category, then it's dual will also be true without any effort what so ever. Roman et al. [5] gives a nice example of this. We already prove that for any category, if an initial object exists, this initial object is unique up to an isomorphism. Note that this is a statement that is true for any category, so the duality principle applies, i.e. the dual statement is true. And what is the dual statement? That for every $\mathcal{C}^{op}$ the initial object is unique up to an isomorphism. But an initial object in $\mathcal{C}^{op}$ is a terminal object in $\mathcal{C}$. So we have, without any effort, that every terminal object is unique up to an isomorphism.

1.2.5 Categorical Product and Coproduct

In set theory, we are used to the notion of a Cartesian product. Similarly to how we did for isomorphism, the idea of a product can be generalized via Category Theory. Here is how it's done.

Definition 1.2.16 (Span). Let A, B be objects in a category $\mathcal{C}$. A span on A and B is a triple (Z, f, g) where $f : Z \to A$ and $g : Z \to B$ are morphisms in $\mathcal{C}$. This is shown in figure 1.9.

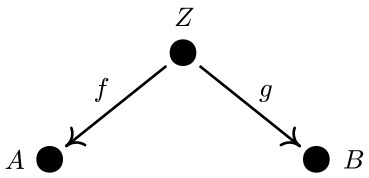

Figure 1.9: Diagrams showcasing a span between A and B.

Definition 1.2.17 (Categorical Product). Let A, B be objects in a category $\mathcal{C}$. A span $(A \times B, \pi_1, \pi_2)$ is called a product between A and B if for every span (Z, f, g) of A and B, there exists a unique morphism $h_{f,g} : Z \to A \times B$ such that $h_{f,g} \fatsemi \pi_1 = f$ and $h_{f,g} \fatsemi \pi_2 = g$. That is the same as saying that the diagram 1.10 commutes.

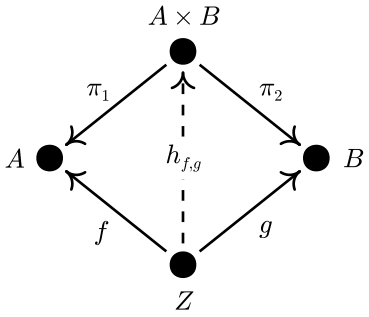

Figure 1.10: Diagrams showcasing the categorical product. Note that the dashed line is intended to highlight that the morphism $h_{f,g}$ is uniquely induced by f and g.

Note that this definition of a product is a **universal construction**, since it's done via existence and uniqueness. Another important aspect to note is that not every pair of objects in a category might have a product associated.

Theorem 1.2.18. For a category $\mathcal{C}$, a pair of objects A and B can have more than one product construction, but if this is the case, then both these constructions will be isomorphic.

Proposition 1.2.19 (Categorical Product vs Set Product). The categorical product generalizes the Cartesian product in set theory.

Proof. Consider the span $(X \times Y, \pi_1, \pi_2)$ where $\pi_1(x, y) = x$ and $\pi_2(x, y) = y$. Thus, for any span (Z, f, g) of A and B, make $h_{f,g}(z) = (f(z), g(z)) \in X \times Y$. This is how the Cartesian product works.

Let's drop the subscript in $h_{f,g}$. Now we have to show that h is a unique morphism, and that $h \fatsemi \pi_1 = f$ and $h \fatsemi \pi_2 = g$.

The second condition is trivially true, just note that

$$\forall z \in Z,\ \pi_1(h(z)) = \pi_1((f(z), g(z))) = f(z) \implies h \fatsemi \pi_1 = f,$$

and the same argument works for π_2 and g.

For uniqueness, consider $h' : Z \to X \times Y$ such that $h' \fatsemi \pi_1 = f$, and $h' \fatsemi \pi_2 = g$. If $h' \neq h$, then there is a $z \in Z$, such that $h(z) = (f(z), g(z)) \neq h'(z)$. Bu then, $\pi_1(h'(z)) \neq f(z)$ or $\pi_2(h'(z)) \neq g(z)$, which is a contradiction.

□

From the definition of a product, it's easy to think of possible dual constructions by just inverting the arrows (morphisms).

Definition 1.2.20 (Cospan). Let A, B be objects in a category $\mathcal{C}$. A cospan on A and B is a triple (Z, f, g) where $f : A \to Z$ and $g : B \to Z$ are morphisms in $\mathcal{C}$. This is shown in Figure 1.11.

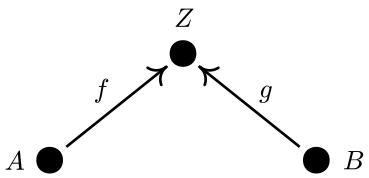

Figure 1.11: Diagrams showcasing a cospan between A and B.

Definition 1.2.21 (Categorical Coproduct). Let A, B be objects in a category $\mathcal{C}$. A cospan $(A+B, i_1, i_2)$ is called a product between A and B if for every span (Z, f, g) of A and B, there exists a unique morphism $h_{f,g} : Z \to A \times B$ such that $i_1 \fatsemi h_{f,g} = f$ and $i_2 \fatsemi h_{f,g} = g$. That is the same as saying that the diagram 1.12 commutes.

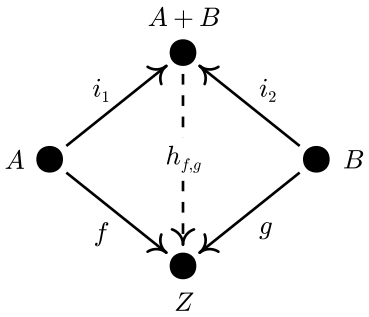

Figure 1.12: Diagrams showcasing the categorical coproduct. Note that the dashed line is intended to highlight that the morphism $h_{f,g}$ is uniquely induced by f and g.

While the categorical product was constructed to generalize the Cartesian set product, the coproduct was constructed in Category Theory, so the question is "to what does the coproduct corresponds in set theory?". The answer is the disjoint union! It's not a coincidence that we used "+" to symbolize it.

The idea of a product construction induced the notion of a product object $A \times B$. Yet, this construction also induces another definition, of the so called (co)product morphism.

Definition 1.2.22 (Product and Coproduct Morphism). Let $A, B, C, D \in Ob_{\mathcal{C}}$, $(A \times B, \pi_1^{A,B}, \pi_2^{A,B})$ and $(C \times D, \pi_1^{C,D}, \pi_2^{C,D})$ two products in the category $\mathcal{C}$, $f : A \to C$ and $g : B \to D$ two morphisms in $\mathcal{C}$. Hence, the morphism induced by $\pi_1^{A,B} \fatsemi f$ and $\pi_2^{A,B} \fatsemi g$ is called product morphism and denoted by $f \times g$, where

$$f \times g := (\pi_1^{A,B} \fatsemi f, \pi_2^{A,B} \fatsemi g) : A \times B \to C \times D.$$

Theorem 1.2.23. The product morphism $f \times g$ is the only morphism in $Mor_{\mathcal{C}}(A \times B, C \times D)$ such that

$$\pi_1^{C,D} \circ (f \times g) = f \circ \pi_1^{A,B}, \quad \pi_2^{C,D} \circ (f \times g) = g \circ \pi_2^{A,B}.$$

The coproduct morphism follows the same definition, but with coproducts.

Finally, one might be wondering what is an actual example of a product morphisms. For sets, it's the intuitive object, e.g. for two functions $f : A \to C$ and $g : B \to D$, the product morphism $f \times g : A \times B \to C \times D$ is just $(f \times g)(x, y) = (f(x), g(y))$.

1.2.6 Cartesian Categories

1.2.7 Pullback, Pushout and Equalizers

Chapter 2

Categories and Functional Programming

Now that we've defined what a category is according to Category Theory, one might wonder how this relates to programming. In this chapter, we introduce Functional Programming, and we show how FP and CT are related.

2.1 Introduction to Functional Programming

According to Widman [6], there are three "main" programming paradigms: imperative programming, Object-Oriented Programming (OOP), and Functional Programming (FP). Imperative programming is *plain* programming, in the sense that you define variables, and functions, and mutate it's values. OOP models programs via objects. An object is from a class, and it has a state and methods. Thus, the value mutation is usually encapsulated by in the state of an object. Finally, FP tries to eradicate value mutation. Once a value is assigned to a variable, this should not change. This might seem strange, but the idea is that value mutation is usually a source of bugs and complications, thus, by limiting it, we can try to make our code more robust. This property is usually called "immutability". Besides immutability, Widman [6] lists four other core tenants of Functional Programming, which are referential transparency, pure functions, higher order functions and lazy evaluations.

In a sense, Functional Programming is the most unusual of the three approaches, as value mutation is a very natural way of thinking when coding. Yet, by avoiding value mutation, functions in FP have a closer resemblance to what we call functions in mathematics (more specifically in Set Theory).

The definition that we gave for Functional Programming was pragmatic. One important aspect missing from such definition was the necessity of working with types. Types are at

the core of how FP relates to Category Theory.

2.1.1 Is Julia an FP language?

As stated in the preface, we are going to use Julia as the language to code our examples. Julia is *not* as FP oriented as languages such as Haskell and Scala. This helps people that are not used to FP. Also, the language enforces types and is *not* Object-Oriented, which eases the process of porting FP concepts. In summary, Julia stands close enough to FP without the often unusual syntax of highly FP oriented languages.

2.1.2 Julia's Type System

These notes do not intend to introduce the Julia programming language, as there are already plenty of resources on the subject. Yet, it is worthy to say some brief words on how Julia deals with types.

In Julia, types can be concrete or abstract. Only concrete type can be instantiated, while abstract types are ways of grouping these types. For example, `Int` and `Float64` are concrete types, while `Number` is an abstract type that contains both, i.e. `Int` and `Float64` are subtypes of `Number`. We can check whether a type is a subtype of another type via the `<:` operator, as shown below.

```
julia> Int <: Number, Float64 <: Number
(true, true)
```

Types can be enforced when calling arguments in a functions, e.g. `f(x::Int, y::String)`. Yet, the output type of a function is not enforced. This type enforcing allows for a feature called multiple dispatch. In Julia, a function can be defined several times by varying only the arguments, both quantity and types. Thus, we can have something like:

```
f(x::Int) = x^2
f(x::Int,y) = (x^2, y)
f(x::String) = "a string!"
f(x::Int, y::Int) = x + y

julia> f(2)
4
julia> f(2,"ok")
(4,"ok")
julia> f("test")
a string!
julia> f(1,1)
2
```

Note that the same function `f` was defined several times, one for each dispatching argument. Each of these instances is called a method of the function `f`. When we don't define the type of the argument, Julia uses the type `Any`, which means any type. If a function has a method with a more specific type, the compiler tries to call the more specialized function. In our example, the method `f(x::Int, y::Int)` was more specialized than `f(x::Int, y)`. This idea of a more specialized method is possible due to the fact that types have a hierarchy as we have shown in the example with types `Int`, `Float64` and `Numbers`.

Besides the default types that Julia provides, we can create new types. There are several ways of doing this, such as structs. By creating new types, we can define functions that take variables with these new types as arguments and dispatch on them.

```
struct Point2D
    x::Real
    y::Real
end

julia> p = Points2D(1,1)
Point2D(1, 1)
julia> p.x
1
julia> getproperty(p, :x)
1
```

Structs are by default immutable, which is good for FP. Yet, we can define mutable structs by simply adding the word "mutable" before the "struct".

Here is a more interesting example of how to use structs, type hierarchy and multiple-dispatch:

```
abstract type Shape end

struct Point3D
    x::Real
    y::Real
    z::Real
end

struct Square <: Shape
    center::Point3D
    length::Real
    Square(center, length) = length ≥ 0 ? new(center, length) : error(
   "length should be greater or equal than 0.")
end

struct Circle <: Shape
    center::Point3D
    radius::Real
```

```
    Circle(center, radius) = radius ≥ 0 ? new(center, radius) : error(
   "radius should be greater or equal than 0.")
end

area(s::Shape) = 0.0
area(s::Circle) = π * s.radius^2
area(s::Square) = length^2

julia> s = Square(10,Point3D(0,0,0));
julia> area(s)
100
```

In our example, we set the default area of a shape to zero by defining our `area(s::Shape)` equal to `0.0`. For the shapes that we know how to compute the area, we write the formulas (e.g. square and circle). Note that we wrote `Square <: Shape` to indicate that both of our structs are subtypes of `Shape`. Note how Julia is *not* Object-Oriented. The type `Square` and the type `Circle` do not have methods themselves. We instead define functions that dispatch on the desired type. This illustrates how Julia is somewhat FP oriented, but not all the way, since we don't have output type enforcing like in, for example, Haskell.

2.1.3 A First Example of Immutability

As we've said, FP has as a core principle the idea of immutable variables. If we consider a For-Loop, in Python, a simplefor i in range(0,10):print(i) will create a variable `i` that mutates by taking first value 0 then 1, and so on. In Julia, the default behaviour of a For-Loop is different. Julia enforces local scope, and creates local variables that are then destroyed. This is more like an FP style.

```
for i in 1:4
    m = 1
end
```

Instead of using a For-Loop, a functional way of doing this would be by defining a function and recursively calling it.

```
function myloop(i::Int)
    if i > 5
        return
    end
    println(i)
    return myloop(i+1)
end

julia> myloop(0)
```

```
0
1
2
3
4
5
```

2.1.4 Side Effects, Pure Functions and Referential Transparency

Consider a function `f`. A function produces a side effect if it affects anything outside of it's scope. For example:

```
x = []
function f!(x)
    push!(x,1)
end

julia> f!(x);
julia> x
Any[1]
```

In Julia, we usually write an exclamation to the end of the name of a function to indicate that this function has side effects. Note that this is *just* a notation. The exclamation point does not modify the function. A function without side effects is called *pure*.

Another important aspect is "referencial transparency". This means that for a given input `x` a function should always return the same output `y`. This seems logical, but there are clear examples that break this. For example:

```
julia> rand(1)
1-element VectorFloat64:
 0.024980886821626025
```

Note that, in Mathematics, what we define as a function usually follows these two principles, i.e. mathematical functions are pure and referentially transparent.

2.1.5 Higher Order Functions

A high order function is a function that receives functions as input or that returns a function as output. In FP, we want functions to be "first-class citizens", meaning that we can pass them to variables and to other functions just like a regular value. Here is one example:

```
square() = nums -> map(x->x^2, nums)

julia> square()(10)
100
```

The function `square` when called actually returns a function. Another similar consists in

```
square_plus_one = function(x)
    x^2 + 1
end

julia> square_plus_one(1)
2

square_plus_two = x->x^2+2

julia> square_plus_two(2)
6
```

2.1.6 Lazy Evaluation

Another relevant concept is the one about lazy evaluation. In Julia we don't natively have lazy evaluation, on the contrary, our code is *eagerly* evaluated. This means that once we call a function, it evaluates all the parameters. Of course, we can alter our code to try to make it lazy. Consider the following example:

```
imap = Iterators.map # version of `map` that returns an iterable
take = Iterators.take # returns the `n` first values of an iterable.
squarelazy(nums) = imap(x->x+1,nums)

julia> x = 1:10;
julia> squarelazy(x);
julia> x
1:10
```

Note that our function did not evaluate x. Lastly, we use the `collect` function to actually evaluate our iterator.

```
julia> collect((squarelazy ∘ take)(x,2))
2-element Vector{Int64}:
 2
 3
```

2.2 Categories in Programming

The emphasis of FP in controlling side-effects, having referentially transparent functions and enforcing types makes programming functions similar to functions in Set Theory. We know that in Category Theory we have a category **Set**, where sets are objects and functions are morphisms. Using **Set** as an analogy, we can try to define a category **Prog**(L) where types are objects and pure referentially transparent functions are morphisms. Thus, the analogy between **Prog**(L) and **Set** is clear. Each type corresponds to a set and each programming function corresponds to a set-theoretic function. The L in **Prog**(L), represents a Functional Programming language. This is necessary since each programming language might have a different collection of types.

Note that this correspondence of **Prog**(L) and **Set** is not perfect. Functions in programming have an internal algorithm that defines how to go from the input to the output, while functions in Set Theory are simply another set. Thus, two functions might always produce the same value for the same input, but with very distinct algorithms. Another possible issue is that even if pure and referentially transparent, a function might still be non-terminating.

Hence, to make our correspondence precise, we need more simplifying assumptions about the behaviour of our programming language. First, we equate our programming functions to mathematical functions by establishing that two programming functions `f(x::T)` and `g(x::T)` are "isomorphic" (the same) if they are denotationally the same, i.e. if for every `x::T` we have `f(x) = g(x)`. Thus, we are restricted to pure, referentially transparent and terminating functions, plus this equivalence relation.

Let's give a definition for our category.

Definition 2.2.1 (Prog(L)). Category **Prog**(L) is a subcategory of **Set** where programming types are the objects and correspond to a set. The morphisms are pure referentially transparent and terminating functions, which correspond to a function in **Set**, i.e. two programming functions are the same in **Prog**(L) if they correspond to the same function in **Set**.

As we've pointed out, Julia does not enforce functions to be pure or referentially transparent. Julia also does check if two composing functions match input and output types. Thus, this is left for us programmers to enforce. If the functions we define follow these theoretical assumptions (i.e. our functions are pure, referentially transparent, terminate, and only compose with functions that match domain and codomain), then we can assume that we are working in a subcategory of **Set**.

Once we stablish that we are in a category, we have successfully connected the world of programming and the world of mathematics. This means that we can now use mathematical tools to prove properties about our programs. We can also import concepts from Category Theory and use it for programming purposes. In fact, Functional Programming does just that. Many software patterns of Functional Programming are based on Category Theory. Functors, Monoids and Monads are some of the many examples.

2.2.1 Exploring Prog(J)

Our category **Prog**(L) is actually not a single category, but a whole family of categories that vary depending on the underlying language L. A very simple example would be a language S that only has types `Integer` and `String`. Thus, the category **Prog**(S) would be the subcategory of **Set** where the objects are **Z** and the set of all possible strings, together with all functions acting on these objects. Thus, as our underlying programming language introduces more types, the size of **Prog**(L) also grows.

Note 2.2.1. The types of a language are not the objects of **Prog**(L), instead, the objects of **Prog**(L) are sets that have a corresponding type in language L. The same is true for functions, which are mathematical functions in **Prog**(L) and which have one or many counterparts in L. Using this correspondence language is daunting, hence, instead we'll just say that a type `T` is an object of **Prog**(L), and that a function `f(x::T)` is a morphism in **Prog**(L). Just remember that whenever we are in **Prog**(L), the actual objects and morphisms are sets.

In the Julia programming, the number of types far exceeds those in S. Thus, the category **Prog**(J) of Julia programming is way larger than **Prog**(S). How large is it? This is not trivial to answer. As we've mentioned, Julia has many ways for user to construct new types, e.g. struct, product types, union types. In order to fully describe **Prog**(J), we would need to inspect all the primitive types available in the language, plus all the possibilities of constructing new types. Since this is not our goal, we'll instead showcase just some relevant properties of **Prog**(J).

First, we know that **Set** has both an initial and terminal objects, which are the empty set and the singleton sets, respectively. In **Prog**(J) we also have both initial and terminal objects. The initial object is type `::Union{}`. This type is also known as `Base.Bottom`, and it's a subtype of every type in Julia, including itself. It behaves like $\varnothing$, in the sense that $\varnothing \subset A$ for any set A including $\varnothing$.

```
julia> Union{} <: Int, Union{} <: Nothing, Union{} <: Union{}
(true, true, true)
```

Note that a function $f : \varnothing \to A$ cannot ever be called, since we cannot provide an element of $\varnothing$. In the same way, a function `f(x::Union{})` cannot be called, because there is no instance of type `Union{}`.

For the terminal object, the `Nothing` type is a singleton type, which has value `nothing`. So is the type `Tuple{}`, which only has `()` as an element. We can also define our own terminal types using a struct:

```
struct Terminal end

julia> Base.issingletontype(Terminal)
True
```

The only element of our type `Terminal` is `Terminal()`.

In **Set**, we have the product and the co-product (disjoint union) of sets. The same is true for types in Julia. The `Tuple{Type1, Type2}` is the product of `Type1` and `Type2`, while `Union{Type1,Type2}` is the co-proudct. Hence, **Prog**(J) is also complete and co-complete.

Chapter 3

Functors

FUNCTORS PRESERVE ISOMORPHISMS? CHECK,WRITE ABOUT Another central definition in Category Theory is that of Functors. While morphisms relate objects inside a category, a Functor establishes a relation between categories, thus, it's one level higher in terms of abstraction.

3.1 What is a Functor?

Let's formally define a Functor.

Definition 3.1.1 (Functor). Let $\mathcal{C}$ and $\mathcal{D}$ be two categories. A functor $F : \mathcal{C} \to \mathcal{D}$ is a pair of mappings with the following properties:

(i) a mapping between objects

$$F : Ob_{\mathcal{C}} \to Ob_{\mathcal{D}},$$

where for each $A \in Ob_{\mathcal{C}}$, $F(A) \in Ob_{\mathcal{D}}$.

(ii) a mapping between morphisms

$$F : Mor_{\mathcal{C}} \to Mor_{\mathcal{D}},$$

where there are two possibilities:

(a) **Covariant Functor**, in which

$$F : Mor_{\mathcal{C}}(A, B) \to Mor_{\mathcal{D}}(F(A), F(B)),$$

hence for a morphism $f : A \to B$, then $F(f) : F(A) \to F(B)$.

(b) **Contravariant Functor**, in which

$$F : Mor_{\mathcal{C}}(A, B) \to Mor_{\mathcal{D}}(F(B), F(A)),$$

hence for a morphism $f : A \to B$, then $F(f) : F(B) \to F(A)$.

(iii) Identities morphisms are preserved, i.e. for $A \in Ob_{\mathcal{C}}$

$$F(id_A) = id_{F(A)}.$$

(iv) Compositions are preserved, i.e. for $f \in Mor_{\mathcal{C}}(A, B)$ and $g \in Mor_{\mathcal{C}}(B, C)$,

(a) For a **Covariant Functor**,

$$F(g \circ f) = F(g) \circ F(f).$$

(b) For a **Contravariant Functor**,

$$F(g \circ f) = F(f) \circ F(g).$$

It's common for authors to refer to covariant functors only as functors, i.e. whenever someone say that F is a functor, it might be implied that it's a covariant functor. We'll also use this convention whenever it's not ambiguous, and we'll always. Also, we'll sometimes use FA to mean $F(A)$.

Lemma 3.1.2 (Contravariant Functor). The definition 3.1.1 of covariant functor is equivalent to saying that F is a (covariant) functor from $\mathcal{C}^{\text{op}}$ to $\mathcal{D}$.

Proof. To prove this, we need to show that for any contravariant functor $F : \mathcal{C} \to \mathcal{D}$ defined as in 3.1.1, there is an equivalent functor $F' : \mathcal{C}^{op} \to \mathcal{D}$, and vice-versa.

For every $A \in Ob_{\mathcal{C}}$, make $F(A) = F'(A)$. This is fine, since $A \in Ob_{\mathcal{C}} \iff A \in Ob_{\mathcal{C}^{op}}$. Next, for $f \in Mor_{\mathcal{C}}(A, B)$, make $Ff = F'f^{op}$, where f^{op} is the reversed morphism f. Again, this is well defined since $f \in Mor_{\mathcal{C}}(A, B) \iff f^{op} \in Mor_{\mathcal{C}^{op}}(B, A)$.

Lastly, note that

$$(g \circ f)^{op} = f^{op} \circ g^{op},$$

hence:

$$F(g \circ f) = F'(f^{op} \circ g^{op}) \iff Ff \circ Fg = F'f^{op} \circ F'g^{op}.$$

Thus, we showed that F is a contravariant functor according to 3.1.1 if and only if F' is a covariant functor from $\mathcal{C}^{\text{op}}$ to $\mathcal{D}$, where F and F' are the same up to an isomorphism from $\mathcal{C}$ to $\mathcal{C}^{op}$. □

Again, the use of diagrams may help understand what is going on. The figure below illustrates the identity and composition preservation of Functors.

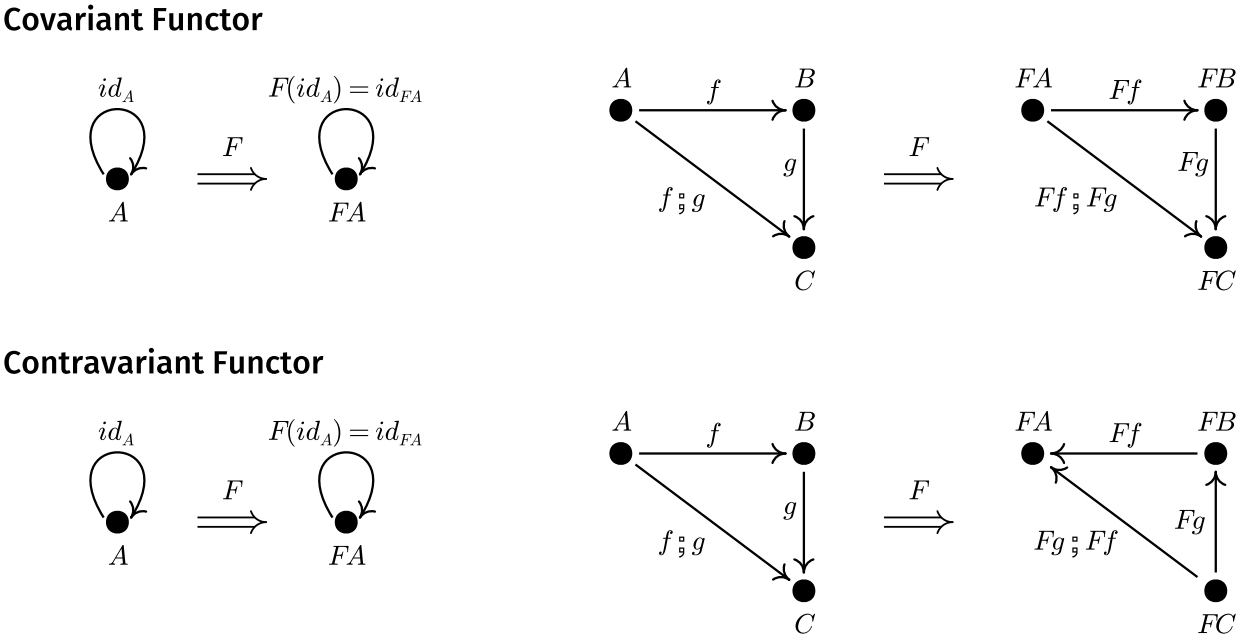

Figure 3.1: Diagrams showcasing the properties of Functors.

Example 3.1.1 (Power set functors). An example of (covariant) functor is the functor $P : \mathbf{Set} \to \mathbf{Set}$, which sends a set A to it's power set 2^A, and sends functions (the morphisms in the case of the category of sets) to their image set, i.e. for $f : A \to B$, we have $Ff : 2^A \to 2^B$ such that for $X \in 2^A$ then $Ff(X) = \{f(x) : x \in X\}$.

An example of **contravariant** functor is the inverse image functor $F : \mathbf{Set} \to \mathbf{Set}$, which sends a set A to it's power set 2^A, but sends f to the inverse image, i.e. $Ff(Y) = \{x \in A : f(x) \in Y\}$. Note that the inverse image satisfy the contravariant property

$$F(f \circ g) = (f \circ g)^{-1} = g^{-1} \circ f^{-1}.$$

3.2 Category of Small Categories

One might realize that functors are acting on categories in a very similar way as morphisms do to objects. Indeed, we can define a functor composition.

Definition 3.2.1 (Functor composition). For two functors $F : \mathcal{C} \Rightarrow \mathcal{D}$ and $G : \mathcal{D} \Rightarrow \mathcal{E}$, then $G \circ F$ is a functor from $\mathcal{C}$ to $\mathcal{E}$ where

(i) For any $A \in Ob_{\mathcal{C}}$, $G \circ F(A) = G(F(A))$,

(ii) For any $f \in Mor_{\mathcal{C}}$, $G \circ F(f) = G(F(f))$.

We can also define an identity functor $I : \mathcal{C} \Rightarrow \mathcal{C}$, where $F(A) = A$ and $F(f) = f$.

Therefore, we might wonder whether there exists a category of all categories where objects are categories and morphisms are functors. The answer is "no". Similar to the set of all sets, it can be proven that this category does not exists. Yet, the category of all *small categories* does.

Remember, a small category is one where both morphisms and objects are sets. With this, let's prove our first theorem.

Theorem 3.2.2 (Category of Small Categories). Let $Ob_{\mathbf{SmCat}}$ be small categories and $Mor_{\mathbf{SmCat}}$ be functors. This constitutes a category.

Proof. To prove this, we'll use the fact that in Gödel-Bernays class set theory, the axiom 1.1.2 implies what is called a *comprehension scheme*.

Proposition 3.2.3 (Comprehension Scheme from Borceux [2]). If $\phi(x_1, ..., x_n)$ is a formula that the quantification occurs on set variables, then there exists a class A such that

$$(x_1, ..., x_n) \in A \iff \phi(x_1, ..., x_n).$$

Note that

$$\mathcal{C} := \langle Ob_{\mathcal{C}}, Mor_{\mathcal{C}} \rangle \in \mathbf{SmCat} \iff \mathcal{C} \text{ "is a category".}$$

Since every $\mathcal{C}$ is small, then the formula to check whether $\mathcal{C}$ is a category iterates over set variables, i.e. $Ob_{\mathcal{C}}$ and $Mor_{\mathcal{C}}$. Thus, the Comprehension Scheme proposition guarantees that **SmCat** exists. □

3.3 Types of Functors

Before we go on to provide examples of functors, let's present a way to classify different functors.

Definition 3.3.1 (Faithful, Full, Fully Faithful and Embedding). Here we follow Roman et al. [5]. Let F be a functor between categories $\mathcal{C}$ and $\mathcal{D}$.

1. F is **faithful** if for every $A, B \in Ob_{\mathcal{C}}$, $F : Mor_{\mathcal{C}}(A, B) \to Mor_{\mathcal{C}}(FA, FB)$ is injective.

2. F is **full** if for every $A, B \in Ob_{\mathcal{C}}$, $F : Mor_{\mathcal{C}}(A, B) \to Mor_{\mathcal{C}}(FA, FB)$ is surjective.

3. F is **fully faithful** if for every $A, B \in Ob_{\mathcal{C}}$, $F : Mor_{\mathcal{C}}(A, B) \to Mor_{\mathcal{C}}(FA, FB)$ is bijective.

4. F is an **embedding** of $\mathcal{C}$ in $\mathcal{D}$ if F is fully faithful and $F : Ob_{\mathcal{C}} \to Ob_{\mathcal{D}}$ is injective.

3.4 Subcategories

Definition 3.4.1 (Subcategory). Let $\mathcal{D}$ be a category. We say that $\mathcal{C}$ is a subcategory of $\mathcal{D}$ if $Ob_{\mathcal{C}} \subset Ob_{\mathcal{D}}$ and $Mor_{\mathcal{C}} \subset Mor_{\mathcal{D}}$, such that $\mathcal{C}$ is a category.

If for every $A, B \in Ob_{\mathcal{C}}$, we have that $Mor_{\mathcal{D}}(A, B) = Mor_{\mathcal{C}}(A, B)$, then $\mathcal{C}$ is a *full subcategory*.

From the definition of a functor, one might wonder whether for any functor $F : \mathcal{C} \Rightarrow \mathcal{D}$, the image $F(\mathcal{C})$ is a subcategory of $\mathcal{D}$, i.e. if $\langle Ob_{F(\mathcal{C})}, Mor_{F(\mathcal{C})} \rangle$ is a category where

$$Ob_{F(\mathcal{C})} := \{F(A) : A \in Ob_{\mathcal{C}}\}, \quad Mor_{F(\mathcal{C})} := \{F(f) : f \in Mor_{\mathcal{C}}\}.$$

The answer is no.

3.5 Relevant Examples of Functors

Now that we know what a functor is, let's showcase some relevant examples that might be useful to someone applying Category Theory to another field.

Example 3.5.1 (Power Set Functor). The power set functor $\mathcal{P} : \mathbf{Set} \to \mathbf{Set}$ takes a set to it's power set and a function $f : A \to B$ to the image function $Imf : \mathcal{P}(A) \to \mathcal{P}(B)$, i.e. for a subset $S \subset A$ in the domain, returns $f(S) := \{f(x) \ : x \in S\}$.

Another even more relevant example is the *contravariante* power set functor $F : \mathbf{Set} \to \mathbf{Set}$ that takes A to $\mathcal{P}(A)$ and f to the inverse image f^{-1}.

Example 3.5.2 (Identity Functor). This does what one might expect from the name. The identity functor is $1_{\mathcal{C}} : \mathcal{C} \to \mathcal{C}$, such that $1_{\mathcal{C}}(A) = A \in Ob_{\mathcal{C}}$ and $1_{\mathcal{C}}(f) = f \in Mor_{\mathcal{C}}$.

Example 3.5.3 (Inclusion Functor). For a subcategory $\mathcal{S}$ of $\mathcal{C}$, the inclusion functor is $I_{\mathcal{C}} : \mathcal{S} \to \mathcal{C}$, such that $I_{\mathcal{C}}(A \in Ob_{\mathcal{S}}) = A \in Ob_{\mathcal{C}}$ and $I_{\mathcal{C}}(f \in Mor_{\mathcal{S}}) = f \in Mor_{\mathcal{C}}$.

3.6 Formalizing Diagrams

We've drawn many diagrams representing either categories or portions of categories. Up until now, this has been done in a informal way. Yet, it's possible to define such diagrams

rigorously. One of the reasons that formal diagrams are useful is that they allow us to talk about portions of some category $\mathcal{C}$ in a rigorous manner.

Definition 3.6.1 (Diagram). A $\mathcal{I}$-diagram (or just diagram) D of a category $\mathcal{C}$ is a functor $D : \mathcal{I} \to \mathcal{C}$, where $\mathcal{I}$ is called the index category.

The definition above is pretty much a "placeholder", i.e. any functor can be seen as a diagram since there is no conditions placed on the index category. It's common to say that D is an I-shaped diagram. The reason for this is that the functor D maps the objects and morphisms of $\mathcal{C}$ in the I category. This can be better understood from the figure below.

(Figure here of I shaped)

We define a *path* in D to be an n-tuple of morphisms in D where the codomain of the morphism matches the domain following morphism, e.g. for $f, g, h \in Mor_{\mathcal{I}}$, we have a path (Df, Dg, Dh) if the codomain of Df mathces the domain of Dg and the codomain of Dg matches the domain of Dh. The length of a path is equal to the number of morphisms. For a path $(Df_1, Df_2, ..., Df_n)$, the composition along such path is a morphism $Df_1 \fatsemi ... \fatsemi Df_n$.

We can now formally define a commutative diagram.

Definition 3.6.2 (Commutative Diagram). A diagram $D : \mathcal{I} \to \mathcal{C}$ is said to be commutative if for every pair of objects $D(A), D(B) \in Ob_{\mathcal{C}}$, every composition along paths from $D(A)$ to $D(B)$ are equal.

From the definition above, we say that the diagram shown below commutes if $h = f \fatsemi g$.

Aesthetic differences when drawing diagrams. If you've picked up any book on Category Theory, you'll note that almost every drawing of a diagram uses the label of a node instead of the black circle as we've done here. This is just a matter of aesthetics, yet, authors sometimes draw the black circles when they want to emphasize the "directed graph" nature of the diagram, i.e. they want to highlight that this diagram can be seen as a directed graph.

Therefore, the choice of how to draw these diagram comes down to aesthetic preferences. In these notes, we've opted for drawing the black dots.

Bibliography

[1] Paolo Aluffi. *Algebra: chapter 0*, volume 104. American Mathematical Soc., 2021.

[2] Francis Borceux. *Handbook of categorical algebra: volume 1, Basic category theory*, volume 1. Cambridge University Press, 1994.

[3] Brendan Fong and David I Spivak. *An invitation to applied category theory: seven sketches in compositionality*. Cambridge University Press, 2019.

[4] Bartosz Milewski. *Category theory for programmers*. Blurb, 2018.

[5] Steven Roman et al. *An Introduction to the Language of Category Theory*, volume 6. Springer, 2017.

[6] J. Widman. *Learning Functional Programming*. O'Reilly Media, 2022. ISBN 9781098111700. URL https://books.google.com.br/books?id=_c-AEAAAQBAJ.

Appendix A

Abstract Algebra

Mostly based on Aluffi [1].

A.1 Initial Definitions for Groups

Definition A.1.1 (Groups). Consider the triple $(G, \cdot, e)$, where G is a set, $\cdot : G \times G \to G$ is the product mapping and $e \in G$ is the identity element. This triple is a group if:

1. (Associativity): $a \cdot (b \cdot c) = (a \cdot b) \cdot c$ for every $a, b, c \in G$;
2. (Identity): $a \cdot e = e \cdot a = a$ for every $a \in G$;
3. (Inverse): For every $a \in G$ there exists $a^{-1} \in G$ such that $a \cdot a^{-1} = a^{-1} \cdot a = a$;

When there is no ambiguity, we call the set G a group omitting the product and neutral element.

Whenever it's not ambiguous, we omit the product operator, thus, $g \cdot h \equiv gh$.

Definition A.1.2 (Abelian Group). A group $(G, \cdot, e)$ is *Abelian* if besides the group properties (i.e. associativity, identity and inverse) it's also commutative, i.e. $a \cdot b = b \cdot a$ for every $a, b \in G$.

Example A.1.1. Note that $(\mathbb{R}, +, 0)$ is an Abelian Group. In this case, a^{-1} is usally denoted as $-a$. The triple $(\mathbb{R} \setminus \{0\}, \cdot, 1)$ is also an Abelian Group.

An example of non-Abelian group would be the set of invertible matrices from $\mathbb{R}^n$ to $\mathbb{R}^n$, with $\cdot$ as matrix composition, e.g. $A \cdot B = AB$. Since every matrix considered is invertible and we have the identity matrix as our identity element, then we indeed have a non-Abelian group, since the matrix product is not commutative.

Proposition A.1.3 (Group Cancellation). Let $(G, \cdot)$ be a group. Therefore:

$$fa = ha \implies f = h, \qquad af = ah \implies f = h$$

Proof. If $fa = ha$, then $faa^{-1} = ha^a - 1 \implies f = h$. □

Definition A.1.4 (Subgroup Generated). Let $(G, \cdot, e)$ be a group. We say that $S \subset G$ is a subgroup of G if $(S, \cdot, e)$ is a group. For $A \subset G$, $\mathrm{Gr}(A)$ is called the subgroup generated by A, and it's the smallest subgroup of G containing A, i.e. $\cap_{\alpha\in\Gamma} S_\alpha$ where $\{S_\alpha\}_{\alpha\in\Gamma}$ are all the sets that are subgroups of G. It's easy to prove that such set is indeed a subgroup.

For a singleton $\{g\}$, we define $\mathrm{Gr}(g) := \{g^n \ : \ n \in \mathbb{Z}\}$, where $g^0 = e$, and g^n is the product of n copies of g, while g^{-n} is the product of n copies of $-g$.

Definition A.1.5 (Cyclic Group). If a group G is equal to $\mathrm{Gr}(g)$ for some $g \in G$, then we say that G is cyclic.

Definition A.1.6 (Order of Groups). The order of a finite group G is the number of elements of G. An element $g \in G$ has *finite order* if $g^n = e$ for $n \in \mathbb{N}$. The order of g is then the smallest n such that $g^n = e$.

Definition A.1.7 (Homomorphism and Isomorphism). Let $(G, \cdot_G, e_G)$ and $(H, \cdot_H, e_H)$ be two groups. A function $\theta : G \to H$ is a homomorphism between G and H if $\theta(g_1 \cdot_G g_2) = \theta(g_1) \cdot_H \theta(g_2)$ for every $g_1, g_2 \in G$.

If θ is bijective, then we say that θ is an isomorphism.

Definition A.1.8 (Normal / Self-conjugate). Let K be a subgroup of G. We say that K is *normal*, or *self-conjugate*, if $gkg^{-1} \in K$ for every $g \in G$.

A.2 Groups and Category Theory

Remember that in Category Theory we have a notion of isomorphism that generalizes set isomorphism (i.e. bijective function between sets).

Definition A.2.1 (Automorphism). Let A be an object of a category $\mathcal{C}$. An automorphism is an isomorphism from A to itself. The set[1] of automorphism of A is denoted by $\mathrm{Aut}_{\mathcal{C}}(A)$.

Definition A.2.2 (Grupoid and Groups). A groupoid is a category where every morphism is an isomorphism. Hence, a group is a groupoid category with a single object G. We denote **Grp** as the catogory of groups. In similar fashion, we can define **Ab** as the category of abelian groups, where the only difference is that the objects are abelian groups.

[1] Remember that a $Hom(A, A)$ is guaranteed to be a set if the category is locally small.

Note that this definition is equivalent to our definition of a group in algebraic terms. Why? Because every morphism is equivalent to an element of G, and the morphism composition does the part of the product operator. Also, note that every category has an identity morphism, thus, $id_G \equiv e$ our neutral element. Since every morphism is an isomorphism, this means that for every $g \in Hom(G, G)$, there is a $g^{-1} \in Hom(G, G)$ such that $g \circ g^{-1} = id_G = e$.

In pure categorical terms. Let $\mathcal{C}$ be a locally small category and $G \in \mathcal{C}$, i.e. an object of $\mathcal{C}$.

A.3 Rings and Modules

Let's begin by remembering the concept of a monoid. A monoid $(M, \cdot, e)$ is a set M, together with the binary operator $\cdot : M \times M \to M$ and the identity element e. Besides, $\cdot$ is associative.

Definition A.3.1 (Ring). A ring $(R, \cdot, +)$ is an abelian group $(R, +)$ together with a monoid $(R, \cdot)$, with the property of distributivity, i.e. $a \cdot (b + c) = a \cdot b + a \cdot c$ for every $a, b, c \in R$.

One usually denotes the identity of $(R, +)$ by 0_R and the identity of $(R, \cdot)$ by 1_R. The reason is clear, since these are the corresponding identities for the usual sum and multiplication of numbers.[2]

Based on this definition, one can prove that:

Proposition A.3.2. Let $(R, \cdot, +)$ be a ring. Therefore:

$$0 \cdot r = 0 = r \cdot 0.$$

Note that in this definition, the $+$ operator has much more stated properties, e.g. there are inverse elements, there is commutativity. The $\cdot$ has more freedom. For example, we are not requiring for an inverse to exist, and neither commutativity. Which leads to the following definition.

Definition A.3.3 (Commutative Ring). A ring $(R, \cdot, +)$ is commutative if $a \cdot b = b \cdot a$ for every $a, b \in R$.

Now, we want to slowly increment the properties of these algebraic concepts in order to construct our usual suspects, e.g. $\mathbb{N}$ $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{Q}$ and $\mathbb{C}$.

Definition A.3.4 (Zero-Divisor). Let $(R, \cdot, +)$ be a ring. We say that $a \in R$ is a left-zero-divisor if there exists $b \neq 0 \in R$ such that $ab = 0$. Analogously, we define a right-zero-divisor.

[2]Note that sometimes this is called a unital ring. Another definition of ring would be to consider $(R, \cdot, +)$ as a semigroup instead of a monoid, which would imply the inexistence of an identity element. A ring that is not unital is sometimes called a Rng.

Note that $0 \in R$ is a zero-divisor of every ring R **with the exception** of the *zero-ring* case. The zero-ring is the ring where R is a singleton set. Hence, since there is only one element, there is no element such that $ab = 0$ for $b \neq 0$, since no such b exists.

Example A.3.1 ($\mathbb{Z} \setminus n\mathbb{Z}$). Let n be a positive integer.

Definition A.3.5 (R-Module). An abelian group $(M, \oplus)$ is called a module over a ring $(R, +, \cdot)$ if there is a map (often called scalar multiplication) where:

$$* : R \times M \to M,$$

such that for all $r, r' \in R$ and $m, m' \in M$ we have

(i) $0_R * m = 0_M$;

(ii) $1_R * m = m$;

(iii) $(r + r') * m = r * m \oplus r' * m$;

(iv) $r * (m \oplus m') = r * m \oplus r * m'$;

(v) $(r \cdot r') * m = r * (r' * m)$.

We also call this an R-Module M.

Definition A.3.6 (R-Algebra). An R-Algebra M is an R-Module M together with a bilinear map $M \times M \to M$.

Note that a vector space V over $\mathbb{R}$ with an inner product is an example of R-algebra.