# Functors and Monads

## Connections between Programming and Category Theory

Foo Yong Qi

18 December 2023

**ABSTRACT**. In this article, I show that the definitions of functors, monads etc. in the programming sense do not only correspond loosely with the categorical definitions, but are precisely identical if we are in the relevant category. Among other topics, this article shows that the phrase 'a monad is a monoid in the category of endofunctors' is a precise category-theoretic definition of what a monad is, and shows that a monad in the programming sense that satisfies the monad laws (in the programming sense) implies that what we have is also, precisely, a category-theoretic monad.

**Keywords**: category theory, functional programming

## 1 MOTIVATION

If you have done some functional programming before, you would have probably come across and used functors, monads, polymorphic functions, and maybe even monoids. You may also likely have heard that these terms are defined in *category theory*. However, without any knowledge of category theory, sayings like 'a monad is just a monoid in the category of endofunctors, what's the problem?' can be incredibly frustrating, and the connections between functors and monads in the programming sense and those in category theory are not immediately apparent.

Through this article I hope to give readers enough background in category theory to understand that functors, monads etc. in the usual programming sense do not only correspond loosely to those found in category theory, but are indeed exactly the same, i.e. a functor in the programming sense is exactly a functor in some category. However, I shall not cover functional programming fundamentals; these are presumed to be understood and known by the reader (readers who have not acquired sufficient background can do so with the wide variety of resources online). Instead, this article draws equalities between the functional programming constructs we know of, and their category-theoretic definitions. In addition, due to this presumption, the majority of this article starts with the math before showing the correspondence with code.

In this article, I offer to show:

1. The definition of a category, showing that we can assemble types in a programming language into one (Section 2);
2. Functors in the usual programming sense are exactly categorical functors on our category of types (Section 3);
3. Product and function types in the usual programming sense are precisely product and exponential objects in our category of types (Section 4);
4. Monoids in the usual programming sense are precisely monoids in our category of types induced by the categorical product and the unit type (Section 6);
5. Monads in the usual programming sense are precisely monads on our category of types, which are monoids in the category of endofunctors of our category of types, which is a strict monoidal category induced by functor composition and the identity functor (Section 7);

Addresses: **FOO YONG QI**, Email: yongqi@nus.edu.sg, Website: https://yongqi.foo/, *National University of Singapore*.

6. Monads in the usual programming sense that obey the monad laws in the usual programming sense, precisely define monads in the categorical sense (Subsection 7.2).

While I have worked out many of the results myself, none of the definitions, results and observations are my original contribution; at the very least, I am certainly not the first to show them.

## 2 CATEGORIES

To even begin our discussion we must first describe what category theory is. Intuitively, most theories (especially the algebraic ones) study mathematical structures that abstract over things; groups are abstractions of symmetries, and geometric spaces are abstractions of space. Category theory takes things one step further and study abstraction itself.

Effectively the goal of category theory is to observe similar underlying structures between collections of mathematical structures. What is nice about this is that a result from category theory generalizes to all other theories that fit the structure of a category. As such it should be no surprise that computation can be studied in category theory too!

On the other hand, the generality of category theory also makes it incredibly abstract and difficult to understand—this is indeed the case in our very first definition. As such, I will, as much as possible, show you 'concrete' examples of each definition and reason about them if I can. With this in mind, let us start with the definition of a category, as seen in many sources.

**Definition 2.1** (Category)**.** A category $C$ consists of

- a collection of *objects*, $X, Y, Z, \ldots$, denoted $\mathrm{ob}(C)$
- a collection of *morphisms*, $f, g, h, \ldots$, denoted $\mathrm{mor}(C)$

so that:

- Each morphism has specified *domain* and *codomain* objects; when we write $f : X \to Y$, we mean that the morphism $f$ has domain $X$ and codomain $Y$.
- Each object has an *identity morphism* $1_X : X \to X$.
- For any pair of morphisms $f, g$ with the codomain of $f$ equal to the domain of $g$ (i.e. $f$ and $g$ are composable), there exists a *composite morphism* $g \circ f$ whose domain is equal to the domain of $f$ and whose codomain is equal to the codomain of $g$, i.e.

$$f : X \to Y, \quad g : Y \to Z \qquad \rightsquigarrow \qquad g \circ f : X \to Z$$

Composition of morphisms is subject to the two following axioms:

- *Unity*. For any $f : X \to Y$, $f \circ 1_X = 1_Y \circ f = f$.
- *Associativity*. For any composable $f$, $g$ and $h$, $(h \circ g) \circ f = h \circ (g \circ f)$.

As you can see, there is very little describing what a category is, or how to construct one. In category theory, we do not care (that much) about the construction of objects of morphisms; as long as they satisfy the definition of a category, we may work with them in a categorical framework. This allows many different kinds of objects to all assemble into categories.

*Example* 2.1. The category of sets, **Set**, contains sets (like $\mathbb{N}$ and $\{1, 2, 3\}$) as objects, and as morphisms, functions between sets (like $f : \mathbb{R} \to \mathbb{R}, f(x) = x^2 + 2x + 3$). From this example, we can see that there can be more than one morphism between two objects in a category. The identity morphism for each object $\mathbb{A}$ is the function $1_{\mathbb{A}} : \mathbb{A} \to \mathbb{A}$ where $1_{\mathbb{A}}(x) = x$.

Our construction of **Set** indeed forms a category.

Theorem 2.1. ***Set*** *is a category.*

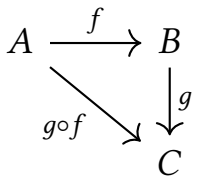
*Proof.* Given objects (sets) $\mathbb{A}$, $\mathbb{B}$ and $\mathbb{C}$ and morphisms (functions) $f : \mathbb{A} \to \mathbb{B}$ and $g : \mathbb{B} \to C$, the composite $g \circ f : \mathbb{A} \to \mathbb{C}$ exists in **Set**, given by $(g \circ f)(x) = g(f(x))$. Similarly, we can see that $(1_{\mathbb{B}} \circ f)(x) = 1_{\mathbb{B}}(f(x)) = f(x)$ and $(f \circ 1_{\mathbb{A}})(x) = f(1_{\mathbb{A}}(x)) = f(x)$, therefore showing that composition is unital. Finally, composition of functions is also associative; suppose we have another morphism $h : \mathbb{C} \to \mathbb{D}$, then $((h \circ g) \circ f)(x) = (h \circ g)(f(x)) = h(g(f(x))$, and $(h \circ (g \circ f))(x) = h((g \circ f)(x)) = h(g(f(x))$ too.

□

As stated earlier, many kinds of objects assemble into categories. *Example* 2.2 gives an example category that has (virtually) nothing to do with **Set**. This category we shall show will be used everywhere in this article.

*Example* 2.2. Suppose some simple types in a type system exist. We can construct a category $\mathcal{T}$ where the objects are types, and the morphisms are functions on those types, i.e. a function from `A` to `B` will be a morphism from `A` to `B` in this category—these are functions of the type `A -> B`. In this category, composition of morphisms is straightforward: if `f :: A -> B` and `g :: B -> C` then its composition is `(g . f) x = g (f x)`. Similarly, for any type `A` the identity morphism is the identity function `id :: A -> A` where `id x = x`. We can show that what we have constructed is indeed a category, by similar proofs of associativity and unity shown in the proof of Theorem 2.1.

Composition in categories can be described by the following *commutative diagram*[1], that is, the following diagram commutes[2]:




In other words, going from object $A$ to $B$ via morphism $f$ then from $B$ to $C$ via $g$ is the same as going from $A$ to $C$ directly via $g \circ f$. Such commutative diagrams will be useful for describing and defining further concepts later.

## 3 FUNCTORS

In mathematics, the relationships between objects are frequently far more interesting than the objects themselves. Of course, we do not just focus on *any* relationship between objects, but of keen

[1]The fact that diagrams are formally defined in category theory blows my mind. Even still, diagrams also assemble into categories!

[2]The identity morphisms are not shown, but they are there!

interest, the *structure preserving* relationships between them, such as group homomorphisms that preserve group structures, or monotonic functions between preordered sets that preserve ordering. In category theory, *functors* are maps between categories that preserve the structure of the domain category, especially the compositions and identities.

**Definition 3.1** (Functor)**.** Let $\mathcal{C}$ and $\mathcal{D}$ be categories. A (*covariant*) *functor* $F : \mathcal{C} \to \mathcal{D}$ consists of:

- An object $F(C) \in \text{ob}(\mathcal{D})$ for each object $C \in \text{ob}(\mathcal{C})$[a].
- A morphism $F(f) : F(C) \to F(D) \in \text{mor}(\mathcal{D})$ for each morphism $f : C \to D \in \text{mor}(\mathcal{C})$.

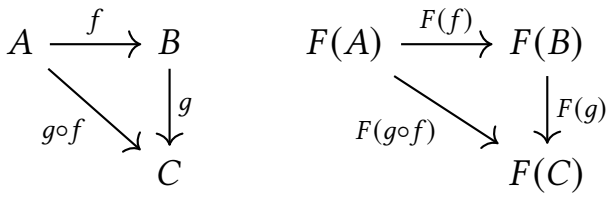
subject to the two *functoriality axioms*:

- For any composable pair of morphisms $f, g \in \text{mor}(\mathcal{C})$, $F(g) \circ F(f) = F(g \circ f)$.
- For each $C \in \text{ob}(\mathcal{C})$, $F(1_C) = 1_{F(C)}$.

in other words, functors respect composition and identities.

[a] We abuse the notation of set membership here. It is not necessary for the collections of objects and morphisms of a category to be sets, as is the case for ob(**Set**).

We show two diagrams below, where on the left we have a diagram in $\mathcal{C}$ and on the right we have a diagram in $\mathcal{D}$. Given a functor $F : \mathcal{C} \to \mathcal{D}$, the following diagrams commute:




*Example* 3.1. The powerset functor $P : \mathbf{Set} \to \mathbf{Set}$ maps a set $\mathbb{A}$ to its powerset $P(\mathbb{A}) = \mathcal{P}(\mathbb{A})$ and a function $f : \mathbb{A} \to \mathbb{B}$ to $P(f) : P(\mathbb{A}) \to P(\mathbb{B})$ defined by

$$P(f)(\mathbb{X}) = \{f(x) \mid x \in \mathbb{X}\}$$

In other words, $P$ lifts a function of elements of $\mathbb{A}$ into a function of subsets of $\mathbb{A}$.

Theorem 3.1. *$P$ is a functor.*

*Proof.* $P$ respects composition. Suppose we have $f : \mathbb{A} \to \mathbb{B}$ and $g : \mathbb{B} \to \mathbb{C}$, then $P(g \circ f)(\mathbb{X}) = \{(g \circ f)(x) \mid x \in \mathbb{X}\} = \{g(f(x)) \mid x \in \mathbb{X}\}$ and $(P(g) \circ P(f))(\mathbb{X}) = P(g)(\{f(x) \mid x \in \mathbb{X}\}) = \{g(f(x)) \mid x \in \mathbb{X}\}$. $P$ also respects identities. Given $1_{\mathbb{X}} : \mathbb{X} \to \mathbb{X}$ where $1_{\mathbb{X}}(x) = x$, then $P(1_{\mathbb{X}})(\mathbb{X}) = \{1_{\mathbb{X}}(x) \mid x \in \mathbb{X}\} = \{x \mid x \in \mathbb{X}\} = \mathbb{X}$, thus showing that $P(1_{\mathbb{X}}) = 1_{P(\mathbb{X})}$.

□

The powerset functor is one example of an *endofunctor*, which is a functor that has equal domain and codomain categories.

*Example* 3.2. In many languages, the list type is a type constructor that receives a type and produces a list of that type. For example, the `[Int]` type is produced from passing in the `Int` type into the `[]` type constructor. We shall denote the list type constructor as `[_]`, sort of as a

function on types, for example, `[_](Int) = [Int]`.
Furthermore, we can define a higher order function `lmap` that lifts a function on elements to one on a list of those elements, like so:

```
lmap :: (a -> b) -> [a] -> [b]
lmap f [] = []
lmap f (x : xs) = f x : lmap f xs
```

As an example, `lmap length ["abc", "de"]` gives `[3, 2]`.
Then, let $\mathcal{T}$ be the category of types described in *Example* 2.2. We can define an endofunctor $L : \mathcal{T} \to \mathcal{T}$ that maps:

- each object (type) `A` $\in \text{ob}(\mathcal{T})$ to the type $L(\texttt{A}) = \texttt{[\_](A)} = \texttt{[A]}$
- each morphism (function) `f :: A -> B` $\in \text{mor}(\mathcal{T})$ to the function $L(\texttt{f}) =$ `lmap f :: [A] -> [B]`.

The functoriality of $L$ should be straightforward to verify.

In many programming texts, a type constructor (together with its implementation of `lmap`) is a functor if we have:

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

It should be immediately clear that our definition of `lmap` satisfies them. Also, you should notice that the functor laws described in the usual programming sense is precisely what is needed to define a categorical functor in $\mathcal{T}$. As such, we can define any arbitrary functor (in the programming sense) that maps types via a type constructor and lifts function on types into functions on the types after applying the type constructor. As long as this functor satisfies the functor laws, this specifies a functor on $\mathcal{T}$! This is precisely the motivation for the **`Functor`** typeclass in Haskell:

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

Here, the type constructor `f` is a **`Functor`** when it is equipped with a way to lift functions via `fmap` (subject to the functoriality axioms). Since the list type constructor is already a **`Functor`**, it provides a definition of `fmap` that is identical to `lmap` which we defined earlier. We can even define our own type constructors and allow them to be **`Functor`**s by providing their definitions of `fmap` as long as they respect composition and identities. I show an example of defining our own functor below:

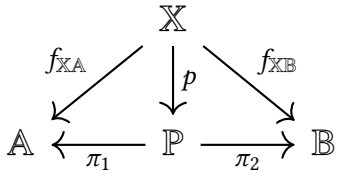
```
1  -- Our own type constructor
2  data Box a = Box a deriving Show
3
4  -- fmap definition
5  instance Functor Box where
6      fmap f (Box x) = Box $ f x
7
8  main :: IO ()
9  main = do
10     print $ fmap (+ 1) (Box 3) -- Box 4
```

The definition of a category does not necessarily preclude any particular object from being a part of a category; as such, it stands to reason that categories themselves can assemble into a category[3]. In such a category, the objects are categories themselves, and the morphisms between categories are functors between them. The identities for each category $\mathcal{C}$, denoted $1_{\mathcal{C}}$, are their corresponding identity functor (mapping each object and morphism to themselves) and composition of morphisms is defined by the composition of functors. The composition of functors $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{E}$ is $G \circ F : \mathcal{C} \to \mathcal{E}$ such that for each object $X$ in $\mathcal{C}$ we have $G(F(X))$ in $\mathcal{E}$, and for each morphism $f$ in $\mathcal{C}$ we have $G(F(f))$ in $\mathcal{E}$. Associativity and unity of functor composition should be relatively straightforward to show.

## 4 UNIVERSAL PROPERTIES

In many instances we want to characterize an object with some unique property in relation to other objects in a category via morphisms, without needing to deal with the details of some particular construction. This allows us to discover results of these objects without needing to repeat the same proofs in different categories. This is what is known as a universal property.

Before defining universal properties, we shall look at some examples of them first. Suppose we are in **Set** and we have sets $\mathbb{A}$ and $\mathbb{B}$. We would like to find some set $\mathbb{P}$ and functions $\pi_1 : \mathbb{P} \to \mathbb{A}$ and $\pi_2 : \mathbb{P} \to \mathbb{B}$ such that, for all sets $\mathbb{X}$ and functions $f_{\mathbb{XA}} : \mathbb{X} \to \mathbb{A}$ and $f_{\mathbb{XB}} : \mathbb{X} \to \mathbb{B}$, there exists a function $p : \mathbb{X} \to \mathbb{P}$ so that $\pi_1 \circ p = f_{\mathbb{XA}}$ and $\pi_2 \circ p = f_{\mathbb{XB}}$. In simple terms, we are looking for $\mathbb{P}$, $\pi_1$ and $\pi_2$ that allows $\mathbb{P}$ to be a 'common pit stop', or in other words, there will exist $p$ that encodes the data of both $f_{\mathbb{XA}}$ and $f_{\mathbb{XB}}$. As a commutative diagram, given objects $\mathbb{A}$ and $\mathbb{B}$, we want to find object $\mathbb{P}$ and morphisms $\pi_1$ and $\pi_2$ such that for all objects $\mathbb{X}$ and morphisms $f_{\mathbb{XA}}$ and $f_{\mathbb{XB}}$, there exists $p$ so that the following diagram commutes:




It turns out that the cartesian product of $\mathbb{A}$ and $\mathbb{B}$, i.e. $\mathbb{A} \times \mathbb{B}$, is a construction of $\mathbb{P}$:

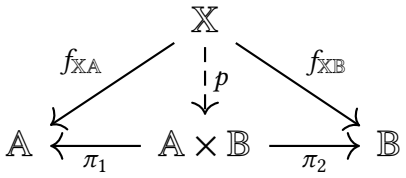
$$\mathbb{A} \times \mathbb{B} = \{(a, b) \mid a \in \mathbb{A}, b \in \mathbb{B}\}$$

and the following functions are constructions of $\pi_1$ and $\pi_2$: $\pi_1(a, b) = a$ and $\pi_2(a, b) = b$. This is so that given functions $f_{\mathbb{XA}} : \mathbb{X} \to \mathbb{A}$ and $f_{\mathbb{XB}} : \mathbb{X} \to \mathbb{B}$, $p : \mathbb{X} \to \mathbb{P}$ would be the function $p(x) = (f_{\mathbb{XA}}(x), f_{\mathbb{XB}}(x))$. The diagram above commutes as $(\pi_1 \circ p)(x) = \pi_1(f_{\mathbb{XA}}(x), f_{\mathbb{XB}}(x)) = f_{\mathbb{XA}}(x)$, and $(\pi_2 \circ p)(x) = \pi_2(f_{\mathbb{XA}}(x), f_{\mathbb{XB}}(x)) = f_{\mathbb{XB}}(x)$.

In fact, notice that given our construction of $\mathbb{P}$, $\pi_1$ and $\pi_2$, $p$ is unique. Suppose $p$ is not unique, and there is another morphism $p' : \mathbb{X} \to \mathbb{P}$ such that $\pi_1 \circ p' = f_{\mathbb{XA}}$ and $\pi_2 \circ p' = f_{\mathbb{XB}}$ where $p \neq p'$. This means that $p'(x) = (y, z)$ where either $y \neq f_{\mathbb{XA}}(x)$ or $z \neq f_{\mathbb{XB}}(x)$. We also know that $\pi_1(y, z) = y$ and $\pi_2(y, z) = z$. As such, either $(\pi_1 \circ p')(x) = y \neq f_{\mathbb{XA}}(x)$ or $(\pi_1 \circ p')(x) = z \neq f_{\mathbb{XB}}(x)$ so either $\pi_1 \circ p' \neq f_{\mathbb{XA}}$ or $\pi_2 \circ p' \neq f_{\mathbb{XB}}$, which is a contradiction.

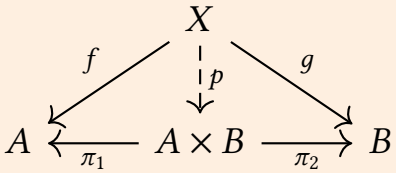
We can now re-draw our commutative diagram, where dashed arrows represent a unique morphism:

[3]By Russell's paradox we cannot have a category of *all* categories—this is the *quasicategory* **CAT**. However, there does exist the category **Cat**, the category of all *small* categories, which are categories where the collection of its morphisms forms a set. **Cat** is not an object of itself, because it is not small.




This property we have described completely characterizes the *categorical product* of two objects in **Set**. We can in fact generalize the notion of a product of two objects in any arbitrary category.

**Definition 4.1** (Product)**.** Fix category $C$. Given objects $A$ and $B$, the *product* of $A$ and $B$, denoted $A \times B$, equipped with morphisms $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$, is such that for all objects $X$ and morphisms $f : X \to A$ and $g : X \to B$, there exists a unique morphism $p : X \to A \times B$ (denoted $\langle f, g \rangle$) following diagram commutes:



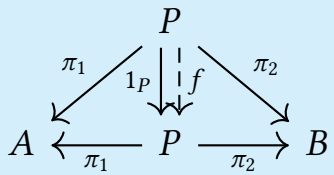

The *universality* of $A \times B$ stems from the fact that there exists exactly one $p$, i.e. $p$ exists and is unique, thus denoted $\langle f, g \rangle$. This effectively gives rise to some notion of uniqueness of the product of $A$ and $B$ in any arbitrary category.

**Definition 4.2** (Isomorphism)**.** Fix category $C$. Given objects $A$ and $B$, $f : A \to B$ is an *isomorphism* is there exists $g : B \to A$ such that $g \circ f = 1_A$ and $f \circ g = 1_B$. If there exists an isomorphism between $A$ and $B$, we say that $A$ and $B$ are *isomorphic*, i.e $A \cong B$.

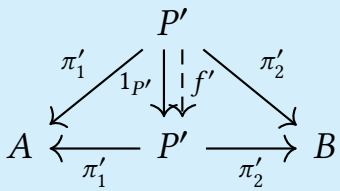
Isomorphisms are important in mathematics because they group objects together that have essentially the same properties despite their different representations. That means that what we discover about one object will also be true of the other. For example, if two groups $G$ and $G'$ are isomorphic, then showing $G$ is abelian tells us immediately that $G'$ is also abelian; showing $G'$ is cyclic tells us immediately that $G$ is also cyclic. Two objects being isomorphic means that the two objects are essentially the same.

Theorem 4.1. *Fix category $C$. Given objects $A$ and $B$, if both $P$ and $P'$ are products of $A$ and $B$, then $P \cong P'$.*

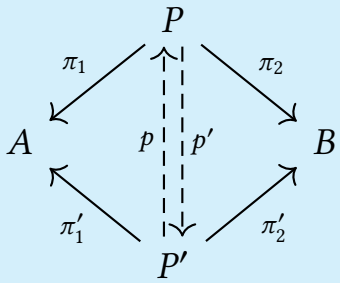
*Proof.* By **Definition 4.1**, because $P$ is a product, there exists a unique morphism $f$ such that the following diagram commutes:



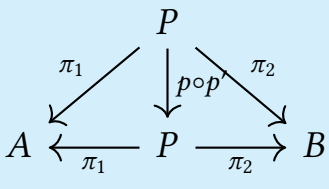

We already know by definition of the identity morphism on $P$ that $\pi_1 = \pi_1 \circ 1_P$ and $\pi_2 = \pi_2 \circ 1_P$. Since $f$ is *unique*, it must be the case that $f$ is precisely $1_P$. This shows that *any* morphism from $P$ to $P$ that satisfies this commutative diagram must be equal to $1_P$.
Similarly, since $P'$ is also a product, the following diagram commutes:

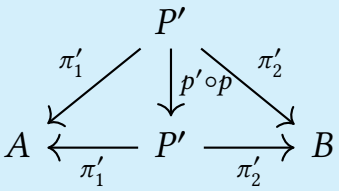



We can argue similarly to show any morphism from $P'$ to $P'$ that satisfies the commutative diagram above must be equal to $1_{P'}$.
Now, again by **Definition** 4.1, the following diagram commutes:




Collapsing the diagram, once going from $P$ to $P'$ then back, and the other going from $P'$ to $P$ and back, gives us two commutative diagrams:




Combining these diagrams with the first two diagrams above shows us that $p \circ p' = 1_P$ and $p' \circ p = 1_{P'}$, which implies that $p$ and $p'$ are isomorphisms, and therefore $P \cong P'$. In fact, $P$ and $P'$ are isomorphic with a *unique* isomorphism $p$ and $p'$.

□

This gives further insight as to the *universality* of the categorical product of two objects: that if two objects $P$ and $P'$ have the universal property of being a product of two other objects $A$ and $B$, then there is a unique isomorphism between $P$ and $P'$—i.e. the product of $A$ and $B$ is unique up to a unique isomorphism. As such, when speaking of a product of $A$ and $B$, we can describe it as *the* product of $A$ and $B$, which we shall denote as $A \times B$.

*Example* 4.1. Suppose we are in the category of types $\mathcal{T}$ and we have types `A` and `B`. Then, the type `(A, B)`, together with projections `fst' :: (A, B) -> A` and `snd' :: (A, B) -> B` where `fst' (a, b) = a` and `snd' (a, b) = b`, is the product of types `A` and `B`. That means that given a type `X` and two functions `f :: X -> A` and `g :: X -> B`, we can construct a unique function `p :: X -> (A, B)` given by `p x = (f x, g x)` so that `fst' . p == f` and `snd' . p == g`:

```haskell
fst' :: (Int, Char) -> Int
fst' (a, b) = a
snd' :: (Int, Char) -> Char
snd' (a, b) = b

f :: String -> Int
```

```haskell
f = length
g :: String -> Char
g = head

p :: String -> (Int, Char)
p x = (f x, g x)

main = do
    let x = "hello"
    print $ f x          -- 5
    print $ (fst' . p) x -- 5
    print $ g x          -- 'h'
    print $ (snd' . p) x -- 'h'
```

Before we look at another universal property, we shall provide a definition of the product of morphisms, which is similar to what we have seen earlier.

**Definition 4.3** (Product Morphism)**.** Suppose we are in a category $C$ with pairs of objects $A, A'$ and $B, B'$ admitting binary products:

$$A \xleftarrow{p_1} A \times A' \xrightarrow{p_2} A'$$

$$B \xleftarrow{q_1} B \times B' \xrightarrow{q_2} B'$$

and further suppose we have morphisms $f : A \to B$ and $f' : A' \to B'$. Then, the product morphism of $f$ and $f'$, denoted $f \times f'$, is the unique morphism that makes the following diagram commute:

$$\begin{array}{ccccc} A & \xleftarrow{p_1} & A \times A' & \xrightarrow{p_2} & A' \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle f \times f'} & & \downarrow{\scriptstyle f'} \\ B & \xleftarrow{q_1} & B \times B' & \xrightarrow{q_2} & B' \end{array}$$

We can see from this diagram that the product of morphisms relates closely to the unique morphism obtained from the definition of the product of objects, i.e. $f \times f' = \langle f \circ p_1, f' \circ p_2 \rangle$.

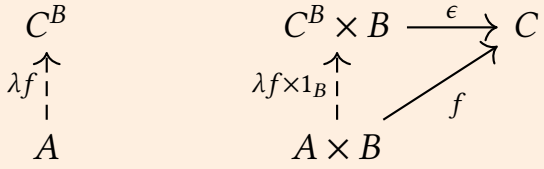
*Example* 4.2. Forming product morphisms in $\mathcal{T}$ is very similar to obtaining the product of two objects of a type. Let us suppose we have `f :: a -> b` and `g :: a' -> b'`. Then we can form the product of these two functions; this must be a function from `(a, a')` to `(b, b')`. It can be defined in the most obvious way:

```haskell
prod' :: (a -> b) -> (a' -> b') -> (a, a') -> (b -> b')
prod' f g (a, a') = (f a, g a')
```

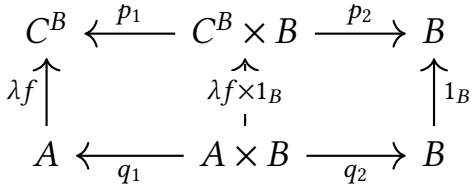
From this definition, we can now define the following universal property.

**Definition 4.4** (Exponential Object)**.** Suppose we are in category $C$ with objects $B$ and $C$, and $C$ contains all binary products with $B$ (i.e., for all objects $X$ in $C$, the product $X \times B$ exists).

Then, the exponential object, denoted $C^B$, equipped with morphism $\epsilon : C^B \times B \to B$, is an object such that for any object $A$ and morphism $f : A \times B \to C$, there exists a unique morphism $\lambda f : A \to C^B$ (called the *transpose* of $f$) that makes the following diagram commute:




We show the product morphism of $\lambda f$ with $1_B$ in the commutative diagram below for clarity:




The uniqueness of $\lambda f$ and $1_B$, and the uniqueness of product morphisms imply that $\lambda f \times 1_B$ is also unique.

*Example* 4.3. Suppose we are in **Set** and we have sets $\mathbb{B}$ and $\mathbb{C}$. The set of all functions from $\mathbb{B}$ to $\mathbb{C}$ given by

$$\mathbb{C}^{\mathbb{B}} = \{f \mid f : \mathbb{B} \to \mathbb{C}\}$$

together with the function $\epsilon : \mathbb{C}^{\mathbb{B}} \to \mathbb{C}$ given by

$$\epsilon(f, b) = f(b)$$

is the exponential object $\mathbb{C}^{\mathbb{B}}$.
Suppose we have a function $g : A \times B \to C$. Then, the transpose of $g$ can be given by $\lambda g(a)(b) = g(a, b)$. This construction uniquely (up to a unique isomorphism) characterizes the exponential set of $\mathbb{B}$ and $\mathbb{C}$.

*Example* 4.4. Similar to our earlier example, suppose we are in the category of types $\mathcal{T}$ and we have types `B` and `C`. Then, the function type `B -> C` together with a function `eval' :: (B -> C, B) -> C` (recall that `(A, B)` is the product of types `A` and `B`) given by `eval' (f, b) = f b` is the exponential object $C^B$. That means that given a function of `g :: (A, B) -> C`, we can define a new function `gT :: A -> B -> C` given by `gT a = \b -> g (a, b)` so that `g (a, b) == (eval' . (prod' gT id)) (a, b)`. You should notice that `gT` is the *curried* equivalent of `g`.

```haskell
-- Char -> String is the exponential String^Char
eval' :: (Char -> String, Char) -> String
eval' (f, s) = f s

-- g repeats a character some number of times
g :: (Int, Char) -> String
g (0, c) = []
g (i, c) = c : g (i - 1, c)

-- id is the identity function (for all types)
```

```haskell
-- id x = x

-- gT is the transpose of g
gT :: Int -> Char -> String
gT i c = g (i, c)

main = do
    let a = 5
    let b = 'a'
    print $ g (a, b)                        -- "aaaaa"
    print $ gT a b                          -- "aaaaa"
    print $ (eval' . (prod' gT id)) (a, b) -- "aaaaa"
```

We have seen some examples of universal properties which show that these properties are not unique to a particular category, but instead can be described in any arbitrary category. The formal definition of a universal property is not exactly necessary for understanding later sections, but is good to know. We define universal properties and connect them to products and exponentials in Appendix A. Perhaps the most pertinent to our discussion is the fact that universal properties like products and exponentials can be described on any particular category, including categories with categories as objects.

## 5 NATURAL TRANSFORMATIONS

In categories, sometimes morphisms do not depend in an essential way on the particular objects they relate. For example, our definition of the projection function on products `fst'` and `snd'` operate on the pair `(Int, Char)`, but these functions can operate on a pair of *any* two types `a` and `b` and be defined identically. These are the polymorphic functions `fst :: (a, b) -> a` and `snd :: (a, b) -> b` which can be described as a family or collection of morphisms, one of each for every product type `(A, B)`. Such a family is known as a *natural transformation*, which we will define in this section, and we shall also give some intuition of why we can describe natural transformations as being morphisms between functors.

Suppose in **Set** we have objects $\mathbb{A}$, $\mathbb{B}$ and their product $\mathbb{A} \times \mathbb{B}$. The swap function $\text{swap}_{\mathbb{A},\mathbb{B}}(a, b) = (b, a)$ maps $\mathbb{A} \times \mathbb{B}$ to $\mathbb{B} \times \mathbb{A}$. Notice that for *any* objects $\mathbb{C}$ and $\mathbb{D}$ there is also its own swap function $\text{swap}_{\mathbb{C},\mathbb{D}} : \mathbb{C} \times \mathbb{D} \to \mathbb{D} \times \mathbb{C}$ defined as the more generic $\text{swap}(c, d) = (d, c)$. The swap function does not depend on the particular objects it swaps, since it is defined in the same way for any pair of objects. Such a definition allows swap to preserve composition. For any functions $f : \mathbb{A} \to \mathbb{C}$ and $g : \mathbb{B} \to \mathbb{D}$, the following diagram commutes:

$$\begin{array}{ccc} \mathbb{A} \times \mathbb{B} & \xrightarrow{\text{swap}_{\mathbb{A},\mathbb{B}}} & \mathbb{B} \times \mathbb{A} \\ \downarrow{\scriptstyle f \times g} & & \downarrow{\scriptstyle g \times f} \\ \mathbb{C} \times \mathbb{D} & \xrightarrow[\text{swap}_{\mathbb{C},\mathbb{D}}]{} & \mathbb{D} \times \mathbb{C} \end{array}$$

As we can see, swapping does not depend on the objects it acts on. As such, we can define swap as an entire collection of functions, one for each product. Since this collection preserves composition of morphisms, we can now begin to build some intuition on such collections as morphisms between functors.

**Definition 5.1** (Product Category)**.** For a category $C$, the *product category* $C \times C$ has:

- for all objects $a, b \in \text{ob}(C)$, object $(a, b) \in \text{ob}(C \times C)$
- for all morphisms $f, g \in \text{mor}(C)$, morphism $(f, g) \in \text{mor}(C \times C)$

where composition of morphisms is defined componentwise[a].

[a]This definition of product categories generalizes to a product of two categories, and is indeed a construction of the product of two objects (categories) in **Cat**.

Now we suppose $C$ has products. Then there is a product functor $P : C \times C \to C$ given by $P(A, B) = A \times B$ for objects $(A, B)$ and $P(f, g) = f \times g$ for morphisms $(f, g)$, and a swapped product functor $S : C \times C \to C$ given by $S(A, B) = B \times A$ and $S(f, g) = g \times f$. We can now see that for every morphism $(f : A \to C, g : B \to D)$ in $C \times C$ the following diagram commutes in $C$:

$$\begin{array}{ccc} P(A,B) & \xrightarrow{\text{swap}_{A,B}} & S(A,B) \\ \Big\downarrow{\scriptstyle P(f,g)} & & \Big\downarrow{\scriptstyle S(f,g)} \\ P(C,D) & \xrightarrow[\text{swap}_{C,D}]{} & S(C,D) \end{array}$$

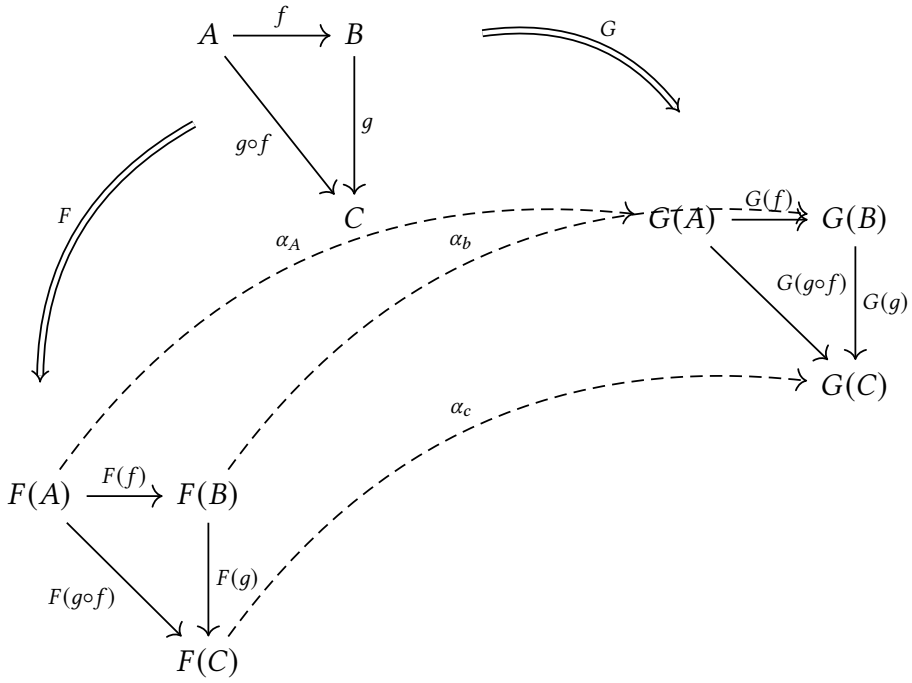
This generalization of swap as a family of morphisms between the images of two functors that preserve functoriality of $P$ and $S$ is precisely a *natural transformation* from $P$ to $S$, which we shall define as the following:

**Definition 5.2** (Natural Transformation)**.** Suppose we have categories $C$ and $\mathcal{D}$ and a parallel pair of functors $F : C \to \mathcal{D}$ and $G : C \to \mathcal{D}$. A *natural transformation* $\alpha : F \Rightarrow G$ is a family of morphisms (forming the *components* of $\alpha$) $\alpha_C : F(C) \to G(C)$ for all $C \in \text{ob}(C)$, such that for all morphisms $f : A \to B$ in $C$ the following diagram commutes, or by saying that the family of morphisms $\alpha_C$ is *natural* in $C$:

$$\begin{array}{ccc} F(A) & \xrightarrow{\alpha_A} & G(A) \\ \Big\downarrow{\scriptstyle F(f)} & & \Big\downarrow{\scriptstyle G(f)} \\ F(B) & \xrightarrow[\alpha_B]{} & G(B) \end{array}$$

Essentially, natural transformations are functoriality-preserving maps between parallel functors.

A natural transformation $\alpha$ with components $\alpha_a$, $\alpha_b$ and $\alpha_c$ can be depicted with the following diagram.



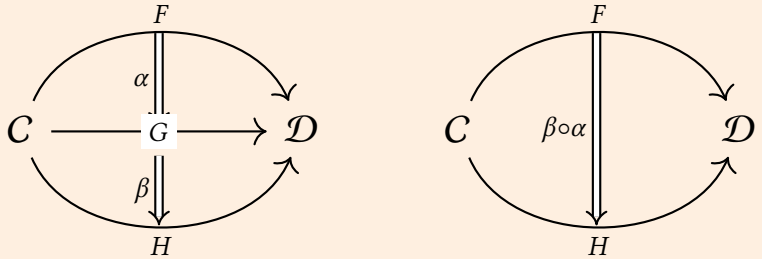

## 5.1 Composition of Natural Transformations

Natural transformations can be composed in multiple ways, many of which are pertinent to our discussion. The simplest among them, which is used to define the other forms of composition, is known as *vertical composition*, which is composed componentwose.

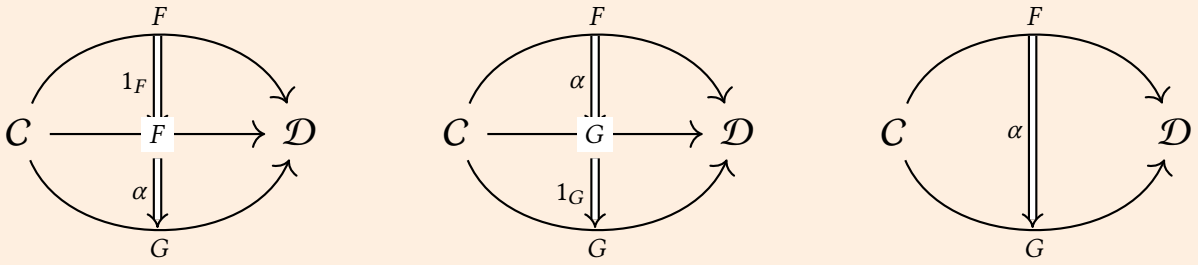
**Definition 5.3** (Vertical Composition)**.** Suppose we have categories $\mathcal{C}$ and $\mathcal{D}$ and parallel functors $F, G, H : \mathcal{C} \to \mathcal{D}$ with natural transformations $\alpha : F \Rightarrow G$ and $\beta : G \Rightarrow H$. Then the *vertical composition* of $\alpha$ and $\beta$ denoted $\beta \circ \alpha : F \Rightarrow H$ is composition done component-wise: i.e. $(\beta \circ \alpha)_X = \beta_X \circ \alpha_X$:

$$X \xrightarrow{f} Y \qquad \begin{array}{ccc} F(X) & \xrightarrow{F(f)} & F(Y) \\ {\scriptstyle \alpha_X}\downarrow & & \downarrow{\scriptstyle \alpha_Y} \\ G(X) & \xrightarrow{G(f)} & G(Y) \\ {\scriptstyle \beta_X}\downarrow & & \downarrow{\scriptstyle \beta_Y} \\ H(X) & \xrightarrow[H(f)]{} & H(Y) \end{array} \quad \begin{array}{l} (\beta\circ\alpha)_X \\ \\ \\ \\ \\ \end{array} \quad \begin{array}{l} \\ \\ \\ \\ \\ \end{array} \quad (\beta\circ\alpha)_Y$$

Or depicted as a globular diagram:




Vertical composition is associative and unital; the identity natural transformation on a functor $F$ is given by $(1_F)_X = 1_{F(X)}$ and is natural in $X$:



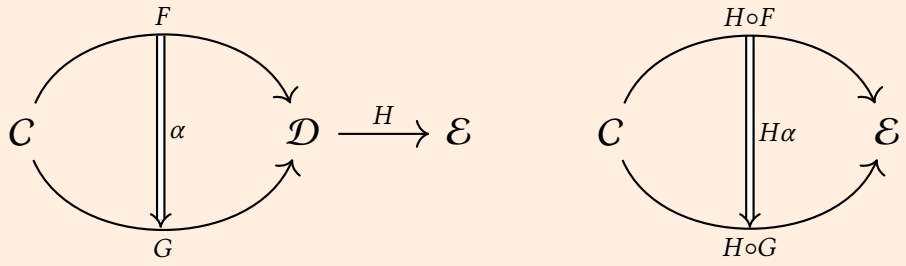

To describe another way of composing natural transformations, we need to define a binary operation between a functor and a natural transformation, known as *whiskering*.

**Definition 5.4** (Whiskering)**.** Suppose we have parallel functors $F, G : \mathcal{C} \to \mathcal{D}$ and a natural transformation $\alpha : F \Rightarrow G$, and another functor $H : \mathcal{D} \to \mathcal{E}$. Then, whiskering $\alpha$ with $H$, denoted $H\alpha$, is the resulting natural transformation $H\alpha : H \circ F \Rightarrow H \circ G$ and $(H\alpha)_X = H(\alpha_X)$ where $\circ$ describes functor composition:

$$X \xrightarrow{\quad f \quad} Y$$

$$\begin{array}{ccc} F(X) & \xrightarrow{F(f)} & F(Y) \\ {\scriptstyle \alpha_X}\downarrow & & \downarrow{\scriptstyle \alpha_Y} \\ G(X) & \xrightarrow[G(f)]{} & G(Y) \end{array} \qquad \begin{array}{ccc} H(F(X)) & \xrightarrow{H(F(f))} & H(F(Y)) \\ {\scriptstyle (H\alpha)_X}\downarrow & & \downarrow{\scriptstyle (H\alpha)_Y} \\ H(G(X)) & \xrightarrow[H(G(f))]{} & H(G(Y)) \end{array}$$

Depicted as a globular diagram:

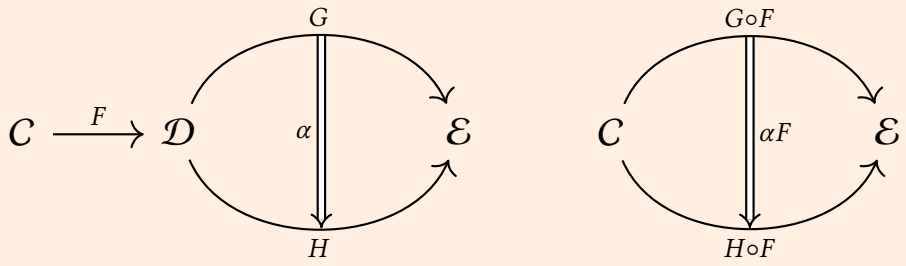



Alternatively if we have a functor $F : \mathcal{C} \to \mathcal{D}$ and parallel functors $G, H : \mathcal{D} \to \mathcal{E}$ with natural transformation $\alpha : G \Rightarrow H$ then we get the natural transformation $\alpha F : (G \circ F) \Rightarrow (H \circ F)$ where $(\alpha F)_X = \alpha_{F(X)}$:

$$X \xrightarrow{\quad f \quad} Y$$

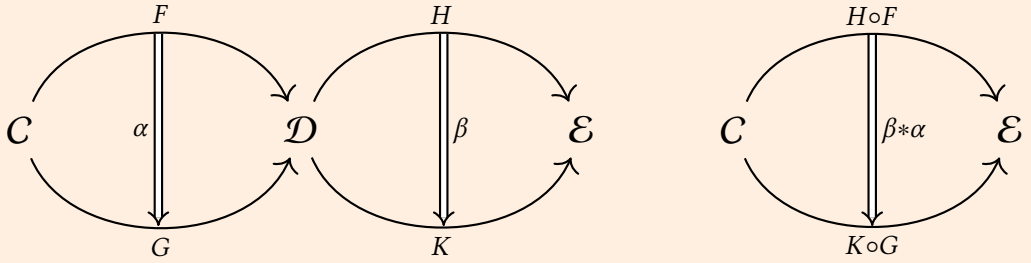
$$\begin{array}{ccc} F(X) & \xrightarrow{F(f)} & F(Y) \end{array} \qquad \begin{array}{ccc} G(F(X)) & \xrightarrow{G(F(f))} & G(F(Y)) \\ {\scriptstyle (\alpha F)_X}\downarrow & & \downarrow{\scriptstyle (\alpha F)_Y} \\ H(F(X)) & \xrightarrow[H(F(f))]{} & H(F(Y)) \end{array}$$

And as a globular diagram:

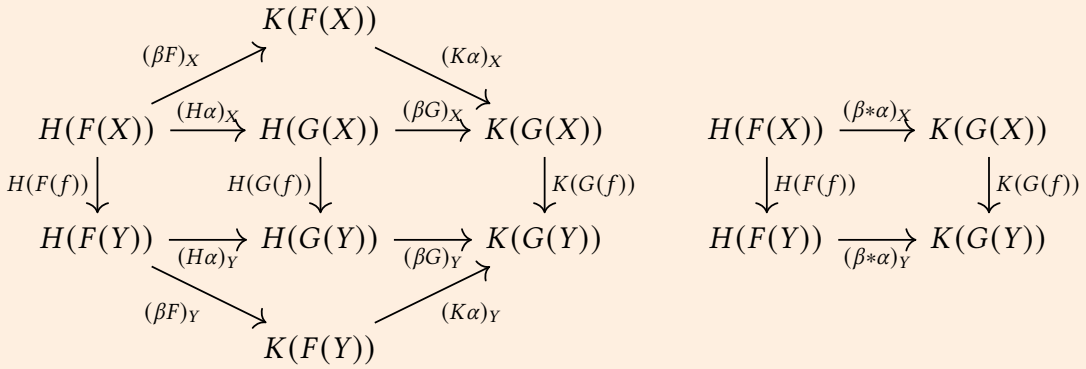



Whiskering allows us to define *horizontal composition* succinctly.

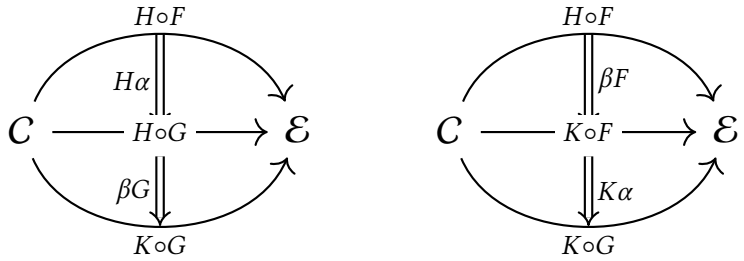
**Definition 5.5** (Horizontal Composition)**.** Suppose we have parallel functors $F, G : \mathcal{C} \to \mathcal{D}$ and $H, K : \mathcal{D} \to \mathcal{E}$, and two natural transformations $\alpha : F \Rightarrow G$ and $\beta : H \Rightarrow K$. The *horizontal composition* of $\alpha$ and $\beta$, denoted $\beta * \alpha : H \circ F \Rightarrow K \circ G$, is given by $\beta * \alpha = \beta G \circ H\alpha = K\alpha \circ \beta F$. This is most easily shown with a globular diagram:




Alternatively, with the following commutative diagrams:

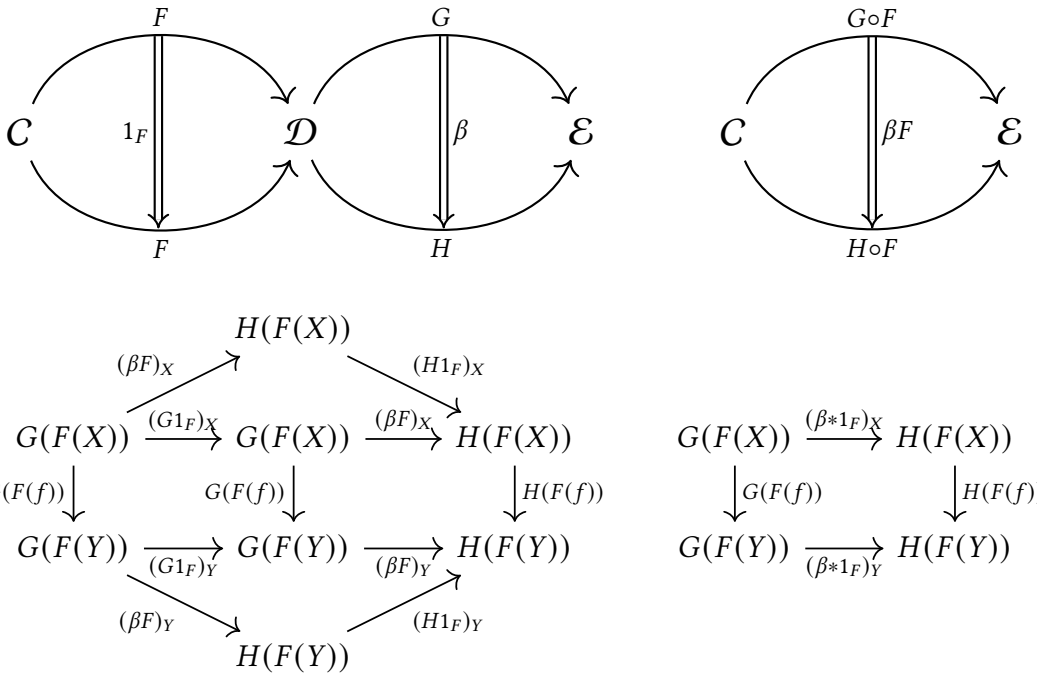



The following globular diagrams help us understand the correspondence of horizontal composition with vertical composition and whiskering:

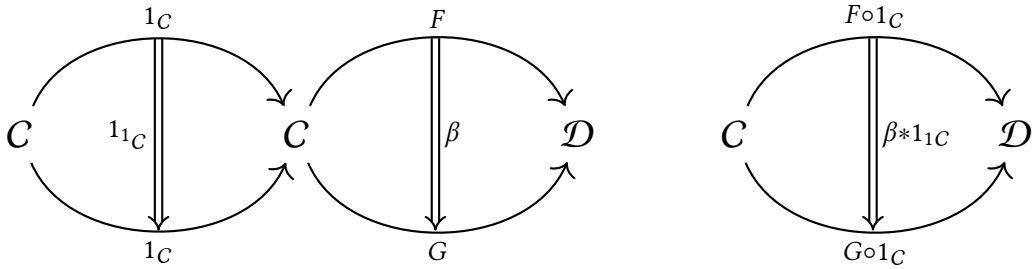



Horizontal composition is also associative and unital. However, take note that the identity, unlike with vertical composition, is in general, not the identity natural transformation on any functor $F$ i.e. $1_F$ which contains the family of morphisms $(1_F)_X = 1_{F(X)}$; if we have $F : \mathcal{C} \to \mathcal{D}$ (parallel to itself) and its identity natural transformation $1_F : F \Rightarrow F$ horizontally composed with a natural

transformation $\beta : G \Rightarrow H$ between two parallel functors $G, H : \mathcal{D} \to \mathcal{E}$, we get $\beta F : G \circ F \Rightarrow H \circ F$, which is simply whiskering:




Instead, the identity of horizontal composition is the identity natural transformation of the identity functor of a category. Given category $\mathcal{C}$, its identity functor $1_\mathcal{C}$ maps all objects and morphisms to themselves, i.e. $1_\mathcal{C}(X) = X$ and $1_\mathcal{C}(f) = f$ for all objects $X$ and morphisms $f$ in $\mathcal{C}$ (this is the identity morphism of an object $\mathcal{C}$ in the category of categories). The identity natural transformation of $1_\mathcal{C}$, i.e. $1_{1_\mathcal{C}}$ simply contains the identity morphisms in $\mathcal{C}$, i.e. $(1_{1_\mathcal{C}})_X = 1_{1_\mathcal{C}(X)} = 1_X$ for each object $X$ in $\mathcal{C}$. This is indeed the identity for horizontal composition. If we have $\beta : F \Rightarrow G$ where $F, G : \mathcal{C} \to \mathcal{D}$, then $\beta * 1_{1_\mathcal{C}} : F \circ 1_\mathcal{C} \Rightarrow G \circ 1_\mathcal{C}$ will be defined as $\beta * 1_{1_\mathcal{C}} = \beta 1_\mathcal{C} \circ F 1_{1_\mathcal{C}}$. As per the definition of whiskering, we have $(\beta 1_\mathcal{C})_X = \beta_{1_\mathcal{C}(X)} = \beta_X$ so $\beta 1_\mathcal{C} = \beta$, and $(F 1_{1_\mathcal{C}})_X = F((1_{1_\mathcal{C}})_X) = F(1_{1_\mathcal{C}(X)}) = F(1_X) = 1_{F(X)} = (1_F)_X$ so $F 1_{1_\mathcal{C}}$ is the identity natural transformation $1_F$, which we know is an identity for vertical composition with $\beta$, i.e. $\beta * 1_{1_\mathcal{C}} = \beta \circ 1_F = \beta$:




With similar arguments you can show that $1_{1_\mathcal{D}} * \beta = \beta$.

As stated earlier, the identity natural transformation on the identity functor on a category is precisely the family of identity morphisms in a category. In our category of types $\mathcal{T}$, $(1_{1_\mathcal{T}})_A = 1_A$, which is the identity function `id :: a -> a` given by `id x = x`.

## 5.2 Correspondence with Polymorphic Functions

*Example* 5.1. Suppose we have two functorial type constructors: non-empty lists, and boxes.

```
-- NonEmpty List type
data NEL a = C a (NEL a) | L a
-- For printing NELs, not important
instance Show a => Show (NEL a) where
  show ls = show $ toList ls where
    toList (L a) = [a]
    toList (C a t) = a : toList t
-- Box type
data Box a = Box a deriving Show
-- Functor instances
instance Functor NEL where
  fmap f (L a) = L $ f a
  fmap f (C a t) = C (f a) (fmap f t)
instance Functor Box where
  fmap f (Box a) = Box $ f a
```

Letting $F : \mathcal{T} \to \mathcal{T}$ be the `NEL` functor, and $G : \mathcal{T} \to \mathcal{T}$ be the `Box` functor, we have the following commutative diagrams describing the action of $F$ and $G$ in the category of types $\mathcal{T}$:

$$\begin{array}{ccc} A & F(A) & G(A) \\ \downarrow{\scriptstyle f} & \downarrow{\scriptstyle F(f)} & \downarrow{\scriptstyle G(f)} \\ B & F(B) & G(B) \end{array}$$

Now let us define a function `toBox` that receives a nonempty list of integers and puts its first element in a box:

```
toBox :: NEL Int -> Box Int
toBox (L a) = Box a
toBox (C a t) = Box a
```

We can see that this function is a single morphism from `NEL Int` to `Box Int`. Clearly, this definition should not be restricted to the `Int` type argument since the same definition applies to all types `a`:

```
toBox :: NEL a -> Box a
toBox (L a) = Box a
toBox (C a t) = Box a
```

This allows `toBox` to be natural in all types `a`:

$$\begin{array}{ccc} A & F(A) & \xrightarrow{\texttt{toBox}_A} G(A) \\ \downarrow{\scriptstyle f} & \downarrow{\scriptstyle F(f)} & \downarrow{\scriptstyle G(f)} \\ B & F(B) & \xrightarrow[\texttt{toBox}_B]{} G(B) \end{array}$$

```haskell
main :: IO ()
main = do
    let ls = C "abc" (C "de" (L "f"))
    print $ fmap length ls         -- [3,2,1]
    print $ fmap length (toBox ls) -- Box 3
    print $ toBox $ fmap length ls -- Box 3
```

Naturality of `toBox` should be easy to show, since its definition does not depend on what the type `a` is.

### 5.3 Functor Categories

In fact, functors also assemble into categories. Such a category has functors as objects and natural transformations between them as morphisms.

**Definition 5.6** (Functor Category)**.** Suppose we have categories $\mathcal{C}$ and $\mathcal{D}$. The *functor category* $\mathcal{D}^{\mathcal{C}}$ has as objects, functors $F : \mathcal{C} \to \mathcal{D}$ and as morphisms, natural transformations $\alpha : F \Rightarrow G$. Composition of morphisms is defined as vertical composition of natural transformations, and the identity morphism of any object $F : \mathcal{C} \to \mathcal{D}$ is its identity natural transformation $1_F$.

To build some intuition for later sections, given category $\mathcal{C}$ we shall define the category of endofunctors of $\mathcal{C}$ to be $\mathcal{C}^{\mathcal{C}}$, containing all endofunctors $F : \mathcal{C} \to \mathcal{C}$. Notice that the domain and codomain of these functors are equal, so they can be composed with themselves, i.e. since $F \circ F : \mathcal{C} \to \mathcal{C}$ is also an endofunctor of $\mathcal{C}$, so $F \circ F$ is also an object in $\mathcal{C}^{\mathcal{C}}$. We also know that functor composition is associative, i.e. $((F \circ F) \circ F)(X) = (F \circ (F \circ F))(X) = F(F(F(X)))$ for all objects (and morphisms) $X$ of $\mathcal{C}$, so we shall denote $F \circ F$ and $F \circ F \circ F$ as $F^2$ and $F^3$ respectively (all of these functors are objects in $\mathcal{C}^{\mathcal{C}}$).

A natural question might be to ask, what is a morphism from $F$ to $F^2$? For example, we know that functor composition is unital with the identity functor on the (co)domain category, i.e. $1_{\mathcal{C}} \circ F = F \circ 1_{\mathcal{C}} = F$, thus with a natural transformation $\alpha : 1_{\mathcal{C}} \Rightarrow F$, we can construct two natural transformations from $F$ to $F^2$, $F\alpha : F \circ 1_{\mathcal{C}} \Rightarrow F^2$ and $\alpha F : 1_{\mathcal{C}} \circ F \Rightarrow F^2$. Notice that $\alpha F \circ \alpha = F\alpha \circ \alpha$:[4]

$$\begin{array}{ccc} A & \xrightarrow{\alpha_A} & F(A) \\ \downarrow{\scriptstyle f} & & \downarrow{\scriptstyle F(f)} \\ B & \xrightarrow{\alpha_B} & F(B) \end{array} \qquad \begin{array}{ccc} A & \xrightarrow{\alpha_A} & F(A) \\ \downarrow{\scriptstyle \alpha_A} & & \downarrow{\scriptstyle F(\alpha_A)} \\ F(A) & \xrightarrow{\alpha_{F(A)}} & F(F(A)) \end{array} \qquad \begin{array}{ccc} 1_{\mathcal{C}} & \xrightarrow{\alpha} & F \\ \alpha \Downarrow & & \Downarrow \alpha F \\ F & \xrightarrow[F\alpha]{} & F^2 \end{array}$$

*Example* 5.2. Suppose we have, as a natural transformation in $\mathcal{T}$, the polymorphic function `pairList` which receives an object and puts two of them in a list:

```haskell
pairList :: a -> [a]
pairList a = [a, a]
```

Letting $L$ be our list functor, we let $\alpha L$ be `pairList` itself (applied to objects of type `[a]`) and $L\alpha$ be `fmap pairList`. Then, we can see that $\alpha L \circ \alpha = L\alpha \circ \alpha$, i.e. `fmap pairList . pairList` and `pairList . pairList` is the same polymorphic function:

```haskell
print $ fmap pairList . pairList $ 2 -- [[2,2],[2,2]]
print $ pairList . pairList $ 2 -- [[2,2],[2,2]]
```

[4] In general, we cannot claim that $\alpha F \circ \alpha = F\alpha \circ \alpha$ implies $\alpha F = F\alpha$. This is only true when $\alpha$ is epic (the categorical notion of surjective).

Since we are in the category of (endo)functors, another question might be to ask, what is an isomorphism of functors? By definition, this would be two natural transformations that when composes, gives the identity functor. This is known as a *natural isomorphism*.

**Definition 5.7.** Suppose we have functors $F, G : \mathcal{C} \to \mathcal{D}$. The natural transformation $\alpha : F \Rightarrow G$ is a *natural isomorphism* if the two conditions are met (the two conditions are equal):

- Each component $\alpha_X : F(X) \to G(X)$ in $\mathcal{D}$ is an isomorphism.
- There exists $\beta : G \Rightarrow F$ such that $\beta \circ \alpha = 1_F$ and $\alpha \circ \beta = 1_G$.

If there exists a natural isomorphism between $F$ and $G$, they are said to be naturally isomorphic, i.e. $F \cong G$.

Let us show that the two conditions are equal.

Lemma 5.1. *Suppose we have parallel functors $F, G : \mathcal{C} \to \mathcal{D}$ and a natural transformation $\alpha : F \Rightarrow G$. If each component $\alpha_X : F(X) \to G(X)$ is an isomorphism, then there exists $\beta : G \Rightarrow F$ such that $\beta \circ \alpha = 1_F$ and $\alpha \circ \beta = 1_G$.*

*Proof.* Since each component $\alpha_X : F(X) \to G(X)$ is an isomorphism, each has an isomorphism $\beta_X : G(X) \to F(X)$ such that $\alpha_X \circ \beta_X = 1_{G(X)}$ and $\beta_X \circ \alpha_X = 1_{F(X)}$:

$$\begin{array}{ccc} F(X) & \underset{\beta_X}{\overset{\alpha_X}{\rightleftarrows}} & G(X) \\ {\scriptstyle F(f)}\downarrow & & \downarrow{\scriptstyle G(f)} \\ F(Y) & \underset{\beta_Y}{\overset{\alpha_Y}{\rightleftarrows}} & G(Y) \end{array}$$

Clearly, these morphisms assemble into a natural transformation $\beta : G \Rightarrow F$. Composition of these natural transformations show $(\alpha \circ \beta)_X = \alpha_X \circ \beta_X = 1_{G(X)} = (1_G)_X$, and $(\beta \circ \alpha)_X = \beta_X \circ \alpha_X = 1_{F(X)} = (1_F)_X$.

□

Lemma 5.2. *Suppose we have parallel functors $F, G : \mathcal{C} \to \mathcal{D}$ and a natural transformation $\alpha : F \Rightarrow G$. If there exists $\beta : G \Rightarrow F$ such that $\beta \circ \alpha = 1_F$ and $\alpha \circ \beta = 1_G$, then each component $\alpha_X : F(X) \to G(X)$ is an isomorphism.*

*Proof.* Clearly, we have, for each $X$, $(\beta \circ \alpha)_X = (1_F)_X$ so $\beta_X \circ \alpha_X = 1_{F(X)}$, and $(\alpha \circ \beta)_X = (1_G)_X$ so $\alpha_X \circ \beta_X = 1_{G(X)}$, which shows that each component $\alpha_X$ is an isomorphism.

□

Theorem 5.3. *Suppose we have parallel functors $F, G : \mathcal{C} \to \mathcal{D}$ and a natural transformation $\alpha : F \Rightarrow G$. Each component $\alpha_X : F(X) \to G(X)$ is an isomorphism if and only if there exists $\beta : G \Rightarrow F$ such that $\beta \circ \alpha = 1_F$ and $\alpha \circ \beta = 1_G$.*

*Proof.* By Lemma 5.1 and Lemma 5.2.

□

*Example* 5.3. In the category of types $\mathcal{T}$, `((A, B), C)` is isomorphic to `(A, (B, C))` for all types `A`, `B` and `C`, given by the following functions:

```
-- isomorphism
tripleIso :: (a, (b, c)) -> ((a, b), c)
tripleIso (a, (b, c)) = ((a, b), c)

-- inverse of isomorphism
tripleIso' :: ((a, b), c) -> (a, (b, c))
tripleIso' ((a, b), c) = (a, (b, c))
```

This isomorphism is natural in `A`, `B` and `C`. This can be expressed as the natural isomorphism `tripleIso` between two functors $F : \mathcal{T} \times \mathcal{T} \times \mathcal{T} \to \mathcal{T}$ and $G : \mathcal{T} \times \mathcal{T} \times \mathcal{T} \to \mathcal{T}$ given by $F(\mathtt{A}, \mathtt{B}, \mathtt{C}) = (\mathtt{A}, (\mathtt{B}, \mathtt{C}))$ and $G(\mathtt{A}, \mathtt{B}, \mathtt{C}) = ((\mathtt{A}, \mathtt{B}), \mathtt{C})$.

By this point, we should have built up enough intuition behind describing a natural family of morphisms as a natural transformation between functors, and how they can be seen as morphisms between functors. Correspondingly, a natural family of isomorphisms in a category is a natural isomorphism, and they can likewise be seen as an isomorphism of functors. Natural isomorphisms also help us to loosen our notion of 'equivalence' of categories. In the category of (small) categories, we get an isomorphism of categories $F : \mathcal{C} \to \mathcal{D}$ where there exists a functor $G : \mathcal{D} \to \mathcal{C}$ such that $F \circ G = 1_{\mathcal{D}}$ and $G \circ F = 1_{\mathcal{C}}$, in other words, $\mathcal{C} \cong \mathcal{D}$. However, such an isomorphism is too strict to categorize an 'equivalence' of categories. Instead, we can define a *natural equivalence* that replaces the equal sign earlier with natural isomorphisms, i.e. $\mathcal{C}$ and $\mathcal{D}$ are *naturally equivalent* if there exists functors $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{C}$ such that $F \circ G \cong 1_{\mathcal{D}}$ and $G \circ F \cong 1_{\mathcal{C}}$. The natural equivalence of these categories is denoted $\mathcal{C} \simeq \mathcal{D}$.

## 6 MONOIDS

Monoids are also algebraic structures, which is usually defined as such[5]:

**Definition 6.1** (Monoid (algebraic))**.** A *monoid* $(M, \cdot, e)$ is a set $M$ endowed with a binary operator $\cdot : M \times M \to M$ and an identity element $e \in M$ subject to:

- *Unity.* $e \cdot x = x \cdot e = x$ for all $x \in M$.
- *Associativity.* $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ for all $x, y, z \in M$.

*Example* 6.1. $(\mathbb{N}, +, 0)$ and $(\mathbb{N}, \times, 1)$ are monoids.

We would, as always, like to generalize the notion of monoids to other categories. Let us attempt to generalize the monoid $(\mathbb{N}, +, 0)$. We can see that a monoid has a set $\mathbb{N}$, which is an object in **Set**. We also have a binary operation $\cdot$, which we can model as a function on the cartesian product of $\mathbb{N}$ with itself, to $\mathbb{N}$, i.e. $\cdot : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ given by $\cdot(x, y) = x + y$. The identity element can be seen as a function $\epsilon : I \to \mathbb{N}$ from any singleton set to $\mathbb{N}$. For example, let the singleton be $I = \{1\}$. Then we can define $\epsilon : I \to \mathbb{N}$ be given by $\epsilon(x) = 0$.

[5] **Definition** 6.1 should appear eerily similar to the definition of a category, shown in **Definition** 2.1. As such, we can quite easily model this set-theoretic monoid as a category: A *monoid* is a category with one object. To understand this characterization, allow $M$ to be the only object in a categorical monoid $C$, and $\cdot$ be the composition of morphisms and $1_M$ be the identity. Then, we can see that this category fits the monoid axioms, i.e. $1_M \circ f = f \circ 1_M = f$ for all morphisms $f$ in $C$, and $f \circ (g \circ h) = (f \circ g) \circ h$ for all morphisms $f, g, h$ in $C$.

Now, observe the following:

- For all sets $A, B$ and $C$ we can see that $A \times (B \times C)$ and $(A \times B) \times C$ is isomorphic, given by $f(a,(b,c)) = ((a,b),c)$ and $f^{-1}((a,b),c) = (a,(b,c))$. This gives us a natural isomorphism $\alpha_{A,B,C}$ that associates the cartesian product of sets.
- For all sets $A$, we can see that $I \times A$ is isomorphic to $A$, given by $f(i,a) = a$ and $f^{-1}(a) = (1,a)$ (recall that 1 is the only element in $I$). This gives us a natural isomorphism $\lambda_A$ that shows an isomorphism between $A$ and $I \times A$.
- Similarly, for all sets $A$, we can see that $A \times I$ is isomorphic to $A$, given by $f(a,i) = a$ and $f^{-1}(a) = (a,1)$. This gives us a natural isomorphism $\rho_A$ that shows an isomorphism between $A$ and $A \times I$.

Based on these observations, the following diagrams commute. The first diagram shows that for $a, b, c \in \mathbb{N}$, $a + b \in \mathbb{N}$ and $a + (b + c) = (a + b) + c$, while the second shows that $a + 0 = 0 + a = a$ where $0 \in \mathbb{N}$:

$$\begin{array}{ccccc}
\mathbb{N} \times (\mathbb{N} \times \mathbb{N}) & \xrightarrow{\alpha_{\mathbb{N},\mathbb{N},\mathbb{N}}} & (\mathbb{N} \times \mathbb{N}) \times \mathbb{N} & \xrightarrow{\cdot \times 1_{\mathbb{N}}} & \mathbb{N} \times \mathbb{N} \\
\downarrow{\scriptstyle 1_{\mathbb{N}} \times \cdot} & & & & \downarrow{\scriptstyle \cdot} \\
\mathbb{N} \times \mathbb{N} & & \xrightarrow{\qquad \cdot \qquad} & & \mathbb{N}
\end{array}$$

$$\begin{array}{ccccc}
I \times \mathbb{N} & \xrightarrow{\epsilon \times 1_{\mathbb{N}}} & \mathbb{N} \times \mathbb{N} & \xleftarrow{1_{\mathbb{N}} \times \epsilon} & \mathbb{N} \times I \\
& {\scriptstyle \lambda_{\mathbb{N}}} \searrow & \downarrow{\scriptstyle \cdot} & \swarrow {\scriptstyle \rho_{\mathbb{N}}} & \\
& & \mathbb{N} & &
\end{array}$$

Another question to ask is, in what categories do monoids arise? Let us observe that in **Set**, together with the cartesian product $\times$ and a singleton set $I$, the following diagrams commute:

$$\begin{array}{ccccc}
A \times (B \times (C \times D))) & \xrightarrow{\alpha_{A,B,C\times D}} & (A \times B) \times (C \times D) & \xrightarrow{\alpha_{A\times B,C,D}} & ((A \times B) \times C) \times D \\
\downarrow{\scriptstyle 1_A \times \alpha_{B,C,D}} & & & & \uparrow{\scriptstyle \alpha_{A,B,C} \times 1_D} \\
A \times ((B \times C) \times D) & & \xrightarrow[\alpha_{A,B\times C,D}]{\qquad\qquad} & & (A \times (B \times C)) \times D
\end{array}$$

$$\begin{array}{ccc}
A \times (I \times B) & \xrightarrow{\qquad \alpha_{A,I,B} \qquad} & (A \times I) \times B \\
& {\scriptstyle 1_A \times \lambda_B} \searrow \quad \swarrow {\scriptstyle \rho_A \times 1_B} & \\
& A \times B &
\end{array}$$

Let us finally generalize these observations to define *monoids in a monoidal category*. First, we generalize the cartesian product $\times$ to a *monoidal product* $\otimes$ that associate up to natural isomorphisms $\alpha$, $\lambda$ and $\rho$. This gives the definition of a *monoidal category*:

**Definition 6.2** (Monoidal Category)**.** A monoidal category is a category $C$ equipped with:

- A bifunctor $\otimes : C \times C \to C$ known as the *monoidal product*

- An object $I$ in $C$ known as the *monoidal unit*

Such that the monoidal product is associative and unital up to natural isomorphism, via three natural isomorphisms:

- The *associator* $\alpha_{A,B,C} : A \otimes (B \otimes C) \cong (A \otimes B) \otimes C$.
- The *left identity* on $I$, $\lambda_A : I \otimes A \cong A$.
- The *right identity* on $I$, $\rho_A : A \otimes I \cong A$.

This data is subject to the condition that for all objects $A, B, C$ and $D$, the following diagrams commute:

$$\begin{array}{ccccc} A \otimes (B \otimes (C \otimes D))) & \xrightarrow{\alpha_{A,B,C\otimes D}} & (A \otimes B) \otimes (C \otimes D) & \xrightarrow{\alpha_{A\otimes B,C,D}} & ((A \otimes B) \otimes C) \otimes D \\ \downarrow{\scriptstyle 1_A \otimes \alpha_{B,C,D}} & & & & \uparrow{\scriptstyle \alpha_{A,B,C} \otimes 1_D} \\ A \otimes ((B \otimes C) \otimes D) & & \xrightarrow[\alpha_{A,B\otimes C,D}]{} & & (A \otimes (B \otimes C)) \otimes D \end{array}$$

$$\begin{array}{ccc} A \otimes (I \otimes B) & \xrightarrow{\alpha_{A,I,B}} & (A \otimes I) \otimes B \\ & {\scriptstyle 1_A \otimes \lambda_B} \searrow \quad \swarrow {\scriptstyle \rho_A \otimes 1_B} & \\ & A \otimes B & \end{array}$$

This gives rise to the (most general) definition of a *monoid* in a monoidal category.

**Definition 6.3.** A *monoid* $(M, \mu, \epsilon)$ in a monoidal category $(C, \otimes, I)$ consists of:

- An object $M$ in $C$.
- A morphism for *multiplication* $\mu : M \otimes M \to M$.
- A *unit* morphism $\epsilon : I \to M$

such that the following diagrams commute:

$$\begin{array}{ccccc} M \otimes (M \otimes M) & \xrightarrow{\alpha_{M,M,M}} & (M \otimes M) \otimes M & \xrightarrow{\mu \otimes 1_M} & M \otimes M \\ \downarrow{\scriptstyle 1_M \otimes \mu} & & & & \downarrow{\scriptstyle \mu} \\ M \otimes M & & \xrightarrow[\mu]{} & & M \end{array}$$

$$\begin{array}{ccccc} I \otimes M & \xrightarrow{\epsilon \otimes 1_M} & M \otimes M & \xleftarrow{1_M \otimes \epsilon} & M \otimes I \\ & {\scriptstyle \lambda_M} \searrow & \downarrow{\scriptstyle \mu} & \swarrow {\scriptstyle \rho_M} & \\ & & M & & \end{array}$$

The diagrams given in these two definitions are a generalization of the ones for the monoid $(\mathbb{N}, +, 0)$ replacing $\mathbb{N}$ with $M$, $\times$ with $\otimes$, and $\cdot$ with $\mu$.

*Example* 6.2. $(\mathbb{N}, +, e)$ where $+(x, y) = x + y$ and $e(x) = 0$ is a monoid in $(\mathbf{Set}, \times, \{1\})$.

*Example* 6.3. The unit type `()` has only one object `()` inhabiting it. Then, $(\mathcal{T}, \times, ())$ is a monoidal category. Recall the `tripleIso` natural isomorphism showing associativity, and the `prod'` function that creates the product of two functions:

```
tripleIso :: (a, (b, c)) -> ((a, b), c)
tripleIso (a, (b, c)) = ((a, b), c)
prod' :: (a -> a') -> (b -> b') -> (a, b) -> (a', b')
prod' f g (x, y) = (f x, g y)
```

Let us now show an example of what follows from the commutativity of the pentagon diagram:

```
main :: IO ()
main = do
  let x = (1, ("a", (2.0, 'b')))
  print $ tripleIso $ tripleIso x -- (((1,"a"),2.0),'b')
  print $ prod' tripleIso id $ tripleIso $ prod' id tripleIso $ x
  -- (((1,"a"),2.0),'b')
```

Also, with the natural isomorphisms describing left and right identities:

```
leftId :: ((), a) -> a
leftId ((), a) = a
rightId :: (a, ()) -> a
rightId (a, ()) = a
```

The following follows from the commutativity of the triangle:

```
main :: IO ()
main = do
  let x = (1, ((), "hello"))
  print $ prod' id leftId $ x -- (1,"hello")
  print $ prod' rightId id $ tripleIso x -- (1,"hello")
```

Then, (`String`, `concat'`, `emptyString`) is a monoid in our monoidal category, where the `concat'` function concatenates two strings, and the `emptyString` function produces the empty string (list) from the unit object:

```
concat' :: (String, String) -> String
concat' (a, b) = a ++ b
emptyString :: () -> String
emptyString x = ""
```

An example of the result of commutativity of the monoid pentagon follows:

```
main :: IO ()
main = do
  let x = ("a", ("b", "c"))
  print $ concat' $ prod' concat' id $ tripleIso x -- "abc"
  print $ concat' $ prod' id concat' $ x           -- "abc"
```

An example of the result of the commutativity of the monoid triangle follows:

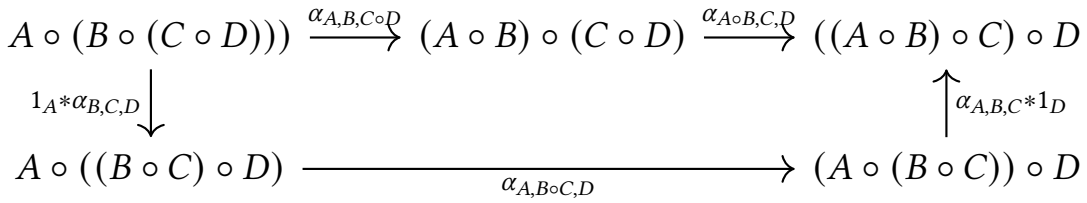
```haskell
main :: IO ()
main = do
  print $ concat' $ (prod' emptyString id) $ ((), "abc") -- "abc"
  print $ leftId ((), "abc")                             -- "abc"
  print $ concat' $ prod' id emptyString $ ("def", ())   -- "def"
  print $ rightId ("def", ())                            -- "def"
```

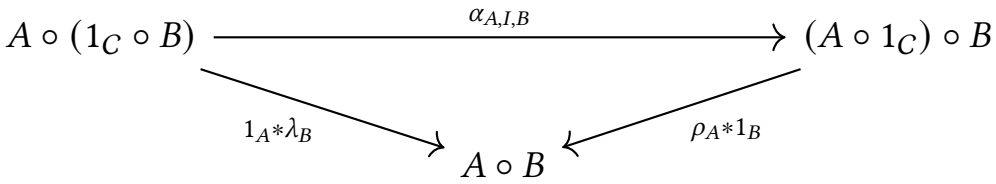
A consequence of (`String`, `concat'`, `emptyString`) being a monoid is that folding left or right on a list of strings using `++` and the empty string as the identity element gives the same result:

```haskell
print $ foldl (++) "" ["a", "b", "c"] -- "abc"
print $ foldr (++) "" ["a", "b", "c"] -- "abc"
```

## 7 MONADS

Monads are incredibly important in functional programming; if you have come this far, this must be the section you've been wanting to read. First, let us recall that, given category $C$, we can obtain the category of endofunctors of $C$, denoted $C^C$. $(C^C, \circ, 1_C)$ is a monoidal category ($\circ$ here represents functor composition). We know that functor composition is associative (i.e. $(F \circ (G \circ H)) = ((F \circ G) \circ H)$) and unital (i.e. $F \circ 1_C = 1_C \circ F = F$), and thus we have natural isomorphisms $\alpha_{A,B,C}, \lambda_A$ and $\rho_A$ that are *equalities*, i.e. $\alpha_{A,B,C} : (A \circ (B \circ C)) = ((A \circ B) \circ C)$, $\lambda_A : 1_C \circ A = A$ and $\rho_A : A \circ 1_C = A$ with components $(\alpha_{A,B,C})_X = 1_{A(B(C(X)))}$, $(\lambda_A)_X = (\rho_A)_X = 1_{A(X)}$. Thus, the commutativity of the pentagon and triangle diagrams for monoidal categories follows immediately:

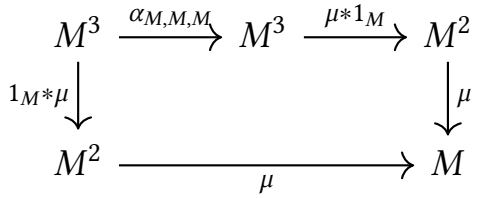


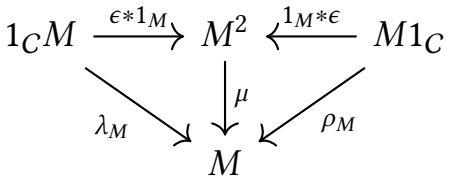




We make special note of the horizontal composition of natural transformations in the diagrams. For the pentagon diagram, recall that $1_A * \alpha_{B,C,D} = A\alpha_{B,C,D}$. As such, the components $(1_A * \alpha_{B,C,D})_X = (A\alpha_{B,C,D})_X = A((\alpha_{B,C,D})_X) = A(1_{B(C(D(X)))})$. By functoriality of $A$, $A(1_{B(C(D(X)))}) = 1_{A(B(C(D(X))))}$. Similarly, since $\alpha_{A,B,C} * 1_D = \alpha_{A,B,C}D$, $(\alpha_{A,B,C} * 1_D)_X = (\alpha_{A,B,C}D)_X = (\alpha_{A,B,C})_{D(X)} = 1_{A(B(C(D(X))))}$. For the triangle, $(1_A * \lambda_B)_X = (A\lambda_B)_X = A((\lambda_B)_X) = A(1_{B(X)}) = 1_{A(B(X))} = (\rho_A)_{B(X)} = (\rho_A B)_X = (\rho_A * 1_B)_X$.

When $\alpha$, $\lambda$ and $\rho$ represent equalities, we have what is known as a *strict monoidal category*. Thus, $C^C$ is a strict monoidal category. As such, we shall do away with the symbol for functor composition (like before) since any interpretation of $ABCD$ for functors $A, B, C$ and $D$ is the same functor.

Now let us determine what a monoid in $C^C$ will look like. Such a monoid $(M, \mu, \epsilon)$ will have natural transformations $\mu : M^2 \Rightarrow M$ and $\epsilon : 1_C \Rightarrow M$ where the following diagrams commute:

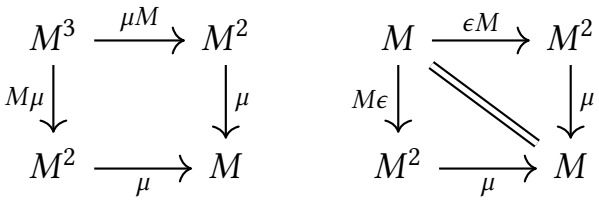



Observe:

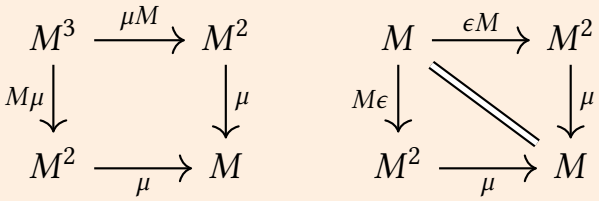
- In the pentagon diagram, $\alpha$ represents an equality, $\mu * 1_M = \mu M$ and $1_M * \mu = M\mu$.
- In the triangle, $1_C M = M 1_C = M$, $\epsilon * 1_M = \epsilon M$, $1_M * \epsilon = M\epsilon$, and $\lambda_M$ and $\rho_M$ represent equalities.

As such, we can collapse each of the two diagrams into a square:




You might be surprised to know that this is the definition of a *monad* on $C$. As such, *a monad on $C$ is a monoid in the category of endofunctors of $C$*.

**Definition 7.1** (Monad)**.** A monad $(M, \mu, \epsilon)$ on $C$ is an endofunctor $M : C \to C$ equipped with two natural transformations $\mu : M^2 \Rightarrow M$ and $\epsilon : 1_C \Rightarrow M$ such that the following diagrams commute:




*Example* 7.1. Recall our list functor $L$ that maps types to a list of that type, and lifts functions on types to functions on lists of those types. $L$ is clearly an endofunctor of $\mathcal{T}$, because the list type(s) are also types. As such, let us define the natural transformation `concatAll` that takes a list of list of types and concatenates its elements together:

```
concatAll :: [[a]] -> [a]
concatAll [] = []
concatAll (x : xs) = x ++ concatAll xs
```

Naturality of `concatAll` should be intuitive. Then, let us define the `singleton` function that puts an object by itself in a list:

```
singleton :: a -> [a]
singleton a = [a]
```

Again, naturality of `singleton` should be intuitive. With these functions, ($L$, `concatAll`, `singleton`) is a monad. The consequence of this is that `concatAll . concatAll` and `concatAll . fmap concatAll` are the same polymorphic function:

```
print $ concatAll . concatAll $ [["a", "b", "c"], ["d", "e"]] -- "abcde"
print $ concatAll . fmap concatAll $ [["a", "b", "c"], ["d", "e"]] -- "abcde"
```

Both `concatAll . singleton` and `concatAll . fmap singleton` are the identity function on lists:

```
print $ concatAll . singleton $ "abcde" -- "abcde"
print $ concatAll . fmap singleton $ "abcde" -- "abcde"
```

### 7.1 Why Monads?

Monads give rise to a consequence that is incredibly powerful in programming. Recall from **Definition 7.1** that $(M, \mu, \epsilon)$ is a monad if and only if for all $X$, (M1) $\mu_X \circ \mu_{M(X)} = \mu_X \circ M(\mu_X)$, and (M2) $\mu_X \circ \epsilon_{M(X)} = \mu_X \circ M(\epsilon_X) = 1_{M(X)}$. Further recall what it means for $\mu$ and $\epsilon$ to be natural, i.e. for all objects $A$, $B$ and morphisms $f : A \to B$, $\mu_B \circ M(M(f)) = M(f) \circ \mu_A$ and $M(f) \circ \epsilon_A = \epsilon_B \circ f$.

Now, given monad $(M, \mu, \epsilon)$, and morphisms $f : A \to M(B)$ and $g : B \to M(C)$, let us define a new binary operation $\oplus$ called *Kleisli composition* where $g \oplus f : A \to M(C)$, is given by[6]

$$g \oplus f = \mu_C \circ M(g) \circ f$$

Let us now show a correspondence between our earlier definition of monads and $\oplus$. First, an incredibly elementary lemma:

Lemma 7.1. *Suppose we have parallel morphisms $g, h : A \to B$. $g = h$ if and only if for all morphisms $f : Z \to A$, $g \circ f = h \circ f$.*

*Proof.* From left to right, if $g = h$ then for all $f$, $g \circ f = h \circ f$. This is simple to show by subtituting $g$ with $h$, giving us $h \circ f = h \circ f$. From right to left, since for all $f$ we have $g \circ f = h \circ f$, then we have $g \circ 1_B = h \circ 1_B$. By the property of the identity morphism we get $g = h$.

□

Theorem 7.2. *Fix category $\mathcal{C}$. $(M, \mu, \epsilon)$ is a monad if and only if:*

*A1 For all $f : A \to M(B)$, $g : B \to M(C)$ and $h : C \to M(D)$, $\oplus$ is associative, i.e. $(h \oplus g) \oplus f = h \oplus (g \oplus f)$.*

*A2 For all $f : A \to M(B)$: $\epsilon$ is unital, i.e. $f \oplus \epsilon_A = \epsilon_{M(B)} \oplus f = f$.*

[6] this is also composition in a Kleisli Category.

*Proof.* First we show that conditions A1 and M1 are equivalent.

$$
\begin{aligned}
& (h \oplus g) \oplus f = h \oplus (g \oplus f) && && \text{(A1)} \\
\Leftrightarrow\ & \mu_D \circ M(h \oplus g) \circ f = \mu_D \circ M(h) \circ (g \oplus f) && \triangleright\ \text{expansion on } \oplus \\
\Leftrightarrow\ & \mu_d \circ M(\mu_D \circ M(h) \circ g) \circ f = \mu_D \circ M(h) \circ \mu_C \circ M(g) \circ f && \triangleright\ \text{expansion on } \oplus \\
\Leftrightarrow\ & \mu_D \circ M(\mu_D) \circ M(M(h)) \circ M(g) = \mu_D \circ M(h) \circ \mu_C \circ M(g) && \triangleright\ \text{functoriality of } M \\
\Leftrightarrow\ & \mu_D \circ M(\mu_D) \circ M(M(h)) = \mu_D \circ \mu_{M(D)} \circ M(M(h)) && \triangleright\ \text{naturality of } \mu \\
\Leftrightarrow\ & \mu_D \circ M(\mu_D) = \mu_D \circ \mu_{M(D)} && && \text{(M1)}
\end{aligned}
$$

Now, let us show that conditions A2 and M2 are equivalent.

$$
\begin{aligned}
& f \oplus \epsilon_A = \epsilon_{M(B)} \oplus f && = f && && \text{(A2)} \\
\Leftrightarrow\ & f \oplus \epsilon_A = \epsilon_{M(B)} \oplus f && = 1_{M(B)} \circ f && \triangleright\ \text{identity morphism} \\
\Leftrightarrow\ & \mu_B \circ M(f) \circ \epsilon_A = \mu_B \circ M(\epsilon_B) \circ f && = 1_{M(B)} \circ f && \triangleright\ \text{expansion on } \oplus \\
\Leftrightarrow\ & \mu_B \circ \epsilon_{M(B)} \circ f = \mu_B \circ M(\epsilon_B) \circ f && = 1_{M(B)} \circ f && \triangleright\ \text{naturality of } \epsilon \\
\Leftrightarrow\ & \mu_B \circ \epsilon_{M(B)} = \mu_B \circ M(\epsilon_B) && = 1_{M(B)} && && \text{(M2)}
\end{aligned}
$$

□

Let us call morphisms $f : A \to M(B)$ as monadic morphisms. Theorem 7.2 shows us that a monad allows us to compose monadic morphisms via Kleisli composition associatively and unitally, and conversely, any definition of natural transformations $\mu$ and $\epsilon$ together with functor $M$ that gives associativity and unity of Kleisli composition of monadic morphisms is a monad. This is precisely the motivation behind the `Monad` typeclass in Haskell.

Let us attempt to define our own monad typeclass in Haskell, where `return'` is $\epsilon$ and `join'` is $\mu$:

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

Then, let us define Kleisli composition for all monads in Haskell:

```haskell
(<=<) :: Monad' m => (b -> m c) -> (a -> m b) -> a -> m c
g <=< f = join' . fmap g . f
```

As an example, let us create the list monad (where `return'` is the same as `singleton` and `join'` is the same as from *Example* 7.1), and two list-producing functions

```haskell
-- List monad
instance Monad' [] where
    return' a = [a]
    join' [] = []
    join' (x : xs) = x ++ join' xs
-- List-producing functions
f :: String -> [Int]
f x = [length x * 2]
g :: Num a => a -> [a]
g x = [x + 1, x + 2, x + 3]

main :: IO ()
main = do
    print $ g <=< f $ "abc" -- [7,8,9]
```

### 7.2 Connections to Monads in Programming

However, in programming, our example earlier may be somewhat awkward. Let us look at another example. In many other languages, the `Option` type constructor represents an optional value, i.e. `Option a` is either `Some a` or it is nothing, i.e. `None`. Clearly, this `Option` type is also a functor. Let us also make `Option` a monad, so that we can compose functions that return optional values:

```haskell
-- The Option monad
data Option a = Some a | None deriving Show

instance Functor Option where
    fmap f (Some x) = Some $ f x
    fmap f None = None

instance Monad' Option where
    return' a = Some a
    join' None = None
    join' (Some (Some a)) = Some a
    join' (Some None) = None

-- A divide function that does not divide by 0
divideBy :: Int -> Int -> Option Int
divideBy a b = case a of
    0 -> None
    x -> Some $ b `div` x

main :: IO ()
main = do
    print $ divideBy 3 <=< divideBy 4 $ 24 -- Some 2
    print $ divideBy 3 <=< divideBy 0 $ 24 -- None
    print $ divideBy 0 <=< divideBy 4 $ 24 -- None
```

What we would really like to have is a way to express sequential Kleisli composition, i.e. instead of `h <=< g <=< f $ x`, we could write something like `f x >>=>> g >>=>> h` which means 'first do `f x`, then monadically apply `g` to it, finally monadically apply `h` to that result'. It is relatively simple to define `>>=>>`:

```haskell
(>>=>>) :: Monad' m => m a -> (a -> m b) -> m b
x >>=>> f = join' . fmap f $ x
```

We can see from this definition that `g <=< f` is `\x -> f x >>=>> g`. This rather miniscule addition makes it syntactically convenient to compose monadic results via Kleisli composition:

```haskell
f = divideBy 2
g = divideBy 3
h = divideBy 4
z = divideBy 0
main :: IO ()
main = do
    print $ f 48 >>=>> g >>=>> h -- Some 2
    print $ f 48 >>=>> z >>=>> h -- None
    print $ f 48 >>=>> g >>=>> z -- None
    print $ z 48 >>=>> g >>=>> h -- None
```

In fact, if we provide a definition of >>=>> for each monad, they do not need to also define `join'` since we can define `join'` based on >>=>> for any monad:

```
join' :: Monad' m => m (m a) -> m a
join' x = x >>=>> id
```

What we have done was the re-construct the **Monad** typeclass and **Maybe** and **[]** monads in Haskell.

```
-- Monad typeclass (built-in in Haskell)
class Functor m => Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b -- >>=>>

-- List monad (built-in in Haskell)
instance Monad [] where
    return a = [a]
    [] >>= f = []
    (x : xs) >>= f = f x ++ (xs >>= f)

-- Maybe (Option) monad (built-in in Haskell)
data Maybe a = Just a -- Some a
             | Nothing -- None
instance Monad Maybe where
    return a = Just a
    Nothing >>= f = Nothing
    Some x >>= f = f x

-- join function for all monads
join :: Monad m => m (m a) -> m a
join x = x >>= id

-- kleisli composition
(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> a -> m c
g <=< f = join . fmap g . f

-- A divide function that does not divide by 0
divideBy :: Int -> Int -> Maybe Int
divideBy a b = case a of
    0 -> Nothing
    x -> Just $ b `div` x

f = divideBy 2
g = divideBy 3
h = divideBy 4
z = divideBy 0
main :: IO ()
main = do
    print $ f 48 >>= g >>= h -- Just 2
    print $ f 48 >>= z >>= h -- Nothing
    print $ f 48 >>= g >>= z -- Nothing
    print $ z 48 >>= g >>= h -- Nothing
```

Finally, a natural question to ask would be, how do we know that our list and maybe monads are actually monads? As per Theorem 7.2, we can show that these are monads by showing associativity and unity of Kleisli composition. However, other programming texts usually give a different set of laws expressed in terms of `>>=`. These laws are typically written as the *monad laws* for all `x`:

H1. `return x >>= f ==== f x`

H2. `f x >>= return ==== f x`

H3. `f x >>= (\y -> (g y >>= h)) ==== (f x >>= g) >>= h`

We shall show this to be equivalent to conditions A1 and A2 (and by extension, M1 and M2) shown in Theorem 7.2.

Corollary 7.3. *An endofunctor (along with natural transformations $\epsilon$ and $\mu$ defined in the obvious way) has associative and unital Kleisli composition if and only if it satisfies the monad laws.*

*Proof.* Let us first show that condition A1 in Theorem 7.2 is met if and only if H3 is met:

```
                (h <=< g) <=< f ==== h <=< (g <=< f)
<=>         ((h <=< g) <=< f) x ==== (h <=< (g <=< f)) x
<=>           f x >>= (h <=< g) ==== (g <=< f) x >>= h
<=>  f x >>= (\y -> g y >>= h) ==== (f x >>= g) >>= h
```

Finally we show that condition A2 in Theorem 7.2 is met if and only both H1 and H2 are met.

```
      (f <=< return) ==== (return <=< f) ==== f
<=> (f <=< return) x ==== (return <=< f) x ==== f x
<=>   return x >>= f ==== f x >>= return ==== f x
```

□

## 8 CONCLUSION

We have shown, through immense suffering, that we can construct a category of types $\mathcal{T}$ with morphisms and functions between these types. From this, we have also shown.

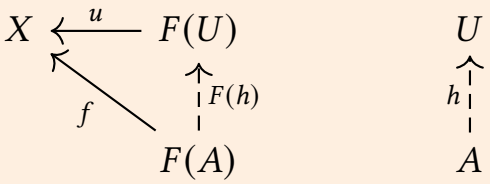
1. a functor (in the programming sense) is precisely an endofunctor on $\mathcal{T}$;
2. product and function types (in the programming sense) are precisely product and exponential objects in $\mathcal{T}$;
3. a polymorphic function (in the programming sense) is precisely a natural transformation between two parallel endofunctors on $\mathcal{T}$;
4. a monoid (in the programming sense) is precisely a monoid in the monoidal category $\mathcal{T}$ induced by the cartesian product and the unit type;
5. a monad (in the programming sense) is precisely a monad on $\mathcal{T}$, which is a monoid in the category of endofunctors of $\mathcal{T}$, which we know is a strict monoidal category, induced by functor composition and the identity functor;
6. if in defining a monad (in the programming sense) we satisfy the three monad laws (in the programming sense), what we have is actually a monad on $\mathcal{T}$;

Your reward for finishing this document? Bragging rights.

# A UNIVERSAL PROPERTIES, FORMALLY

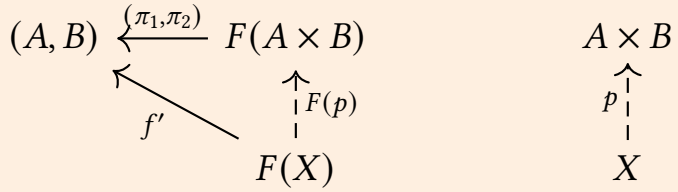
**Definition A.1** (Universal Morphism)**.** Let $F : C \to \mathcal{D}$ be a functor between categories $C$ and $\mathcal{D}$. Let $A$ and $U$ be objects of $C$, and $X$ be an object of $\mathcal{D}$.
Then, a *universal morphism* from $F$ to $X$ is a unique pair $(U, u : F(U) \to X)$ that satisfies the following *universal property*:

For any morphism $f : F(A) \to X$ in $\mathcal{D}$, there exists a unique morphism $h : A \to U$ in $C$ such that the following diagram commutes:

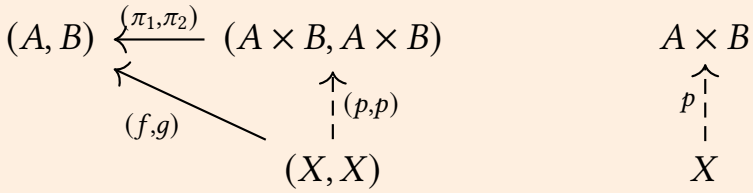



We shall now re-define our characterization of the categorical product in **Definition** 4.1 as a universal property.

**Definition A.2** (Product)**.** Let the functor $F : C \to C \times C$ be a functor from the category $C$ to its product category (defined in **Definition** 5.1), given as $F(X) = (X, X)$ on all objects $X$ and $F(f : A \to B) = (f, f)$ on all morphisms $f$. Then, $(A \times B, (\pi_1, \pi_2) : F(A \times B) \to (A, B))$ is a universal morphism from $F$ to $(A, B)$ which characterizes the product $A \times B$. This means that for all objects $X$ in $C$ and morphisms $f' : F(X) \to (A, B)$ in $C \times C$, there exists a unique morphism $p : X \to A \times B$ which makes the following diagram commute:




Replacing $F(x)$ with $(x, x)$ everywhere and the morphism $f' : F(X) \to (A, B)$ with a pair of morphisms $(f : X \to A, g : X \to B)$ in the commutative diagram above gives us the following commutative diagram:

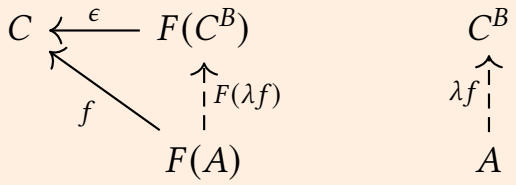



Destructuring the pairs in the triangle on the left we recover the commutative diagram in $C$ given in **Definition** 4.1.

Now we can also re-define our characterization of the exponential object in **Definition** 4.4 as a universal property.

**Definition A.3** (Exponential Object)**.** Suppose we have a category $C$ with objects $B$ and $C$, and the category contains all binary products with $B$, i.e. for all objects $A$ in $C$ then $A \times B$ is also in $C$.

We define the functor $F : \mathcal{C} \to \mathcal{C}$ given by $F(A) = A \times B$ for all objects $A$ in $\mathcal{C}$ and $F(f) = f \times 1_B$ for all morphisms $f$ in $\mathcal{C}$. Then, $(C^B, \epsilon : C^B \times B \to C)$ is a universal morphism from $F$ to $C$ which characterizes the exponential object $C^B$. This means that for all morphisms $f : F(A) \to C$ in $\mathcal{C}$, there exists a unique morphism $\lambda f : A \to C^B$ such that the following diagram commutes:




Replacing $F(A)$ with $A \times B$ for all objects $A$ and $F(f)$ with $f \times 1_B$ for all morphisms $f$ allows us to recover the original commutative diagram shown in **Definition** 4.4: