# My Notes on the Category Theory Lecture by Dr. Bartosz Milewski on YouTube*

Shou Ya

March 30, 2018

*https://www.youtube.com/user/DrBartosz/videos

# Contents


6.1 Functors . . . 3
6.1.1 Functor properties . . . 3
6.1.2 Interesting functors . . . 4
7.1 Bifunctors . . . 5
7.2 Monoidal Categories, Functoriality of ADTs, Profunctors . . . 7
7.2.1 Monoidal Categories . . . 7
7.2.2 Functoriality of ADTs . . . 7
7.2.3 Profunctors . . . 9
8.1 Function objects, Exponentials . . . 11
8.1.1 Function objects . . . 11
8.1.2 Exponential . . . 12
8.2 Type algebra, Curry-Howard-Lambek isomorphism . . . 14
8.2.1 Type algebra . . . 14
8.2.2 Curry-Howard-Lambek isomorphism . . . 15
9.1 Natural Transformations . . . 17

## 6.1 Functors

Universal contruction is about to pick the "best" embodiment of an idea. *e.g.* there is a lot of product in the universe, so we pick the "best" one, the one that every product gets factorized into. Thus comes the "universal" construction.

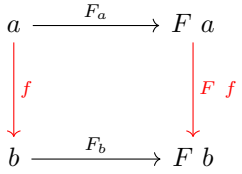
Functors are just mapping between categories.

Why so important? We look for structures/patterns in categories and try to emphasize/extract the pattern from them. Functor is used to find these patterns and map them into a category when we can recognize these patterns.

"Category is the definition of structure". In orther words, "to recognize a certain structure in a category" is the same as defining the pattern as a category. Being able to recognize a category inside another category is just doing pattern recognition.

The mappings we are interested in are those which preserve structures. A function is a mapping from set to set, which does not have a structure. A single set represented in a category is just a category of bunches of objects and with no arrows except the identities. No structure. (Discrete category)

**Definition 6.1.1.** Functor $F$ is a map between categories that preserves the structure. *i.e.* it maps arrow to arrow. For categories $C$, $D$, a functor is that for every $f \in C(a, b)$, $F$ maps it into $F\ f \in D(F\ a, F\ b)$ that preserves the structure.




Since a hom-set is a set, a functor just defines this "huge" function.

Preserving the structure means for $g \circ f \in C$, $F(g \circ f) = (Fg) \circ (Ff) \in D$.

Naturally one have to also make sure $\forall a \in C, F(id_a) = id_{Fa} \in D$.

### 6.1.1 Functor properties

**Definition 6.1.2.** faithful/full Functors that don't collapse structure is called "faithful". A faithful functor is injective on *hom-sets*. Correspondingly, A functor is "full" when it is surjective. Functors that are "full" or "faithful" are only about injective/surjective on the arrows, not about objects. Functor could map two distinct objects $a, b \in C$ into some $c \in D$, as long as the arrows

between $a$ and $b$ doesn't map to the same arrow in $D$, this functor could still be faithful.

### 6.1.2 Interesting functors

**Definition 6.1.3.** Selecting functor The possible functor mapping from category 1 (singleton category) to another category $C$ is unique (up to iso), that is, to map id $\in$ Arr(1) arrow to the id arrow on some object in $D$. This process is like "selecting" an element in $D$.

**Definition 6.1.4.** Constant functor Another important functor mapping from $C$ to $D$ is called constant functor, which maps all arrows in $C$ into the id arrow of an object $c \in D$. Constant functor on object $c$ is denoted as $\Delta_c$.

**Definition 6.1.5.** Endofunctor An endofunctor is a functor that maps from $C$ to itself, $F : C \to C$. Haskell functors are all endofunctors mapping from and to the Hask category. Haskell functors consists of two parts:

1. Mapping between types (types are objects in Hask), *i.e.* Type constructors.
2. Mapping between functions, *i.e.* `fmap`.

**Remark.** Take this example,

```
1 fmap :: (a → b) → (Maybe a → Maybe b)
2 fmap f Nothing = Nothing
3 fmap f (Just x) = Just (f x)
```

To haskell, which uses parametrically polymorphism, line 2 is the only possible implementation for `fmap f` **Nothing**. However, it's not true to math. For languages which doesn't use parametrically polymorphism, it might be possible to return other values. *e.g.* Return **Just** `0` for if type `a` is **Int**. Haskell is imposing a stronger condition to be a functor.

Let's check if **Maybe** really is a functor. Questions to ask:

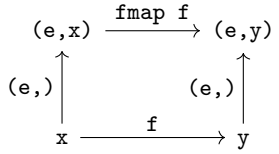
1. Does `fmap` preserves identity?
2. Does `fmap` preserves composition?

Surely it does.

## 7.1 Bifunctors

We start by studying if ADTs form functors. Product type in haskell is actually a type constructor of two argument. *i.e.* `(a,b)` can be written as `(,) a b`.

It's not hard to see that `(,) a`, *i.e.* fix on one parameter `a`, is a functor from `b` to `(a, b)`.




Now think, why don't we make a functor type that has two arguments, so we can make `(,)` a functor on both arguments. In math, we could try to define a category that formalize product of two categories instead of inventing a something new, like a new kind of functor, for that purpose. But what will this category look like? Well it's basically just product of the hom sets. For categories $C$ and $D$, we define $C \times D$ as follows:

- For each pair of objects $c \in C$ and $d \in D$, there is an object $(c, d) \in C \times D$;
- For each pair of arrows $f = c \mapsto c'$ in $C$ and $g = d \mapsto d'$ in $D$, there is an arrow $(f, g) = (c, d) \mapsto (c', d')$ in $C \times D$;
- Composition: $(f', g') \circ (f, g) = (f' \circ f, g' \circ g)$;
- Identity: $\mathrm{id}_{(a,b)} = (\mathrm{id}_a, \mathrm{id}_b)$.

So the functor that takes two arguments is just a functor $C \times D \to E$. This kind of functors is called a bifunctor.

In Haskell, bifunctor is defined as:

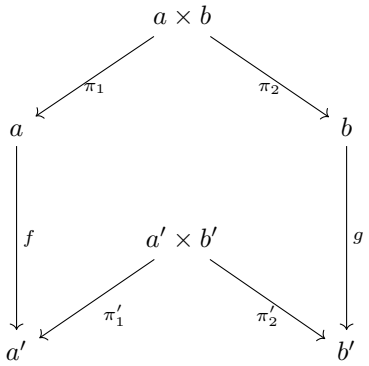
```
class Bifunctor b where
  bimap :: (a → a') → (b → b') → f a b → f a' b'
```

And this is how `(,)` (product) and **Either** (sum) implemented as bifunctors:

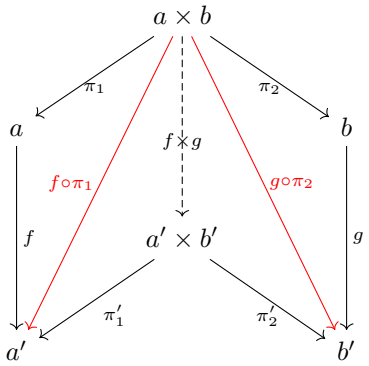
```
instance Bifunctor (,) where
  bimap f g (a,b) = (f a, g b)

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

In a cartesian category $C$, in which there are products for every pair of objects, then this "product" is a bifunctor $C \times C \to C$. This also applies to "co-product" categories. Here's how it works:




So we have this diagram. For each $a, b \in C$, by definition, we have its product $a \times b$ which projects $(\pi_1, \pi_2)$ to $a$ and $b$. And for the morphisms, we have $a \xrightarrow{f} a'$ and $b \xrightarrow{g} b'$. In order for the product $(\times : C \times C \to C)$, to be a bifunctor, we need to show it is possible to lift the pair $f, g$ into something that $a \times b \xrightarrow{f \times g} a' \times b'$. *i.e*there exists a morphism from $a \times b$ to $a' \times b'$ on the diagram above.




Here's how we show that. Since we have $\pi_1$ and $f$, they will compose into $f \circ \pi_1$, same for $\pi_2$ and $g$. By definition of the univeral construction of products, we can see that $a' \times b'$ is the product on $a'$ and $b'$. Therefore, there must exist a unique factorialization from any pair $a \times b$ to $a' \times b'$ (shown as the dashed arrow). This morphism is what we are looking for and therefore we have shown the existence of $\times$ bifunctor.

In haskell, this is not an issue. As the implementation above, we just apply $f$ to $a$ and $g$ to $b$, then we will form $(f\ a, g\ b)$. But this is not enough general for Math, we need to think out of just the Hask category. So the above diagram shows how it work generally in any category.

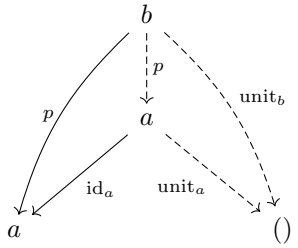
## 7.2 Monoidal Categories, Functoriality of ADTs, Profunctors

### 7.2.1 Monoidal Categories

We start out by seeing how we can generalize monoid in a category. We know pretty well how monoid based on set theory, we got a unit $\mathbf{1} \in \mathbf{Set}$, a binary function $f : \mathbf{Set} \times \mathbf{Set} \to \mathbf{Set}$, few rules and so on.

In monoid categories, we need to think about what it means to multiply two object. This gives rise to product category. The monoidal unit in this category is the unit object, the singleton set in **Set**, or the terminal object to be more general.

Here's a how we the terminal object is a good candidate for the unit. This is to say, in a product category, $(a, ()) \cong a$.




As we can see, the projections $\pi_1 = \text{id}$ and $\pi_2 = \text{unit}$. Given any $b \xrightarrow{p} a$, we have $\text{unit}_b$ which maps $b$ to $()$, we always have a unique factorization $b \overset{p}{\dashrightarrow} a$. This means the projections just worked and we indeed have $(a, ()) \cong a$.

So in a monoidal category (at least product category), monoid unit is the terminal object. This also apply to other categories. Say the coproduct category. In this case the terminal object is the empty set (initial object of the coproduct category). And they both are specialized bifunctor category. All these things as product, coproduct, any bifunctors are called *tensor product* (denoted $C \otimes C \to C$).

What we really need is to have a bifuntor, and a unit for the bifunctor. Then we can get a monoidal category, assuming monoidal properties hold.

### 7.2.2 Functoriality of ADTs

Dr. Milewski has mentioned that ADTs has functoriality by their nature many times in his lecture.

From the previous section, we see how these monoidal categories are related to their corresponding bifunctors. We see, product types are functorial, sum types are functorial, what about other types, **Bool**, **Int**?

So we still have some types that do not depend on type variables, how can we make them instance of **Functor**? We can say they dependent on type indeed, but in a *trivial* way. So here's the place for the constant functor trick.

```
data Const c a = Const c

instance Functor (Const c) where
  fmap :: (a → b) → Const c a → Const c b
  fmap f (Const c) = Const c
```

In category theory, constant functor $\Delta_c$ maps every object in a category to the object $c$ and every morphisms to $\mathrm{id}_c$. Taking it to the Haskell world where every thing is in Hask, the constant functor is used to map everything into the given value. Think of functors working as containers, then the constant functor is like an empty container. It eats up all information and collapse into something that does not contain any inforamtion.

Another trivial functor is the identity functor:

```
data Identity a = Identity a

instance Functor (Identity a) where
  fmap :: (a → b) → Identity a → Identity a
  fmap f (Identity a) = Identity (f a)
```

This functor maps everything back to itself. So this functor is like a container with one thing in it. So this is how we convert any ADTs into our standard notion.

```
data Maybe a = Nothing | Just a
-- sum types can be decomposed into Either
data Maybe a = Either Nothing (Just a)
-- any constant type can be represented using Const functor
-- and Just is just an Identity functor
data Maybe a = Either (Const () a) (Identity a)
```

Now we have a `Const ()` functor and an `Identity` functor; and **Either** is just a bifunctor. Thus overall **Maybe** is a bifunctor. Since we have the same parameter `a` for `Const ()` and `Identity`, we can add a diagonal functor before the bifunctor of the **Either**/**Maybe**, reducing it into a regular endofunctor.

This process of functorizing any ADT type can be automated, and it's built-in to Haskell as an extension: `DeriveFunctor`. Could there be more than one `fmap` defined for an ADT functor? No, Because of parametricity of polymorphism in Haskell, the canonical one is the only possible one!

**Remark.** A *diagonal functor* maps $C \to C \times C$ by replicating everything including objects and morphisms $a \mapsto (a, a)$.

### 7.2.3 Profunctors

Other than product (`()`)and sum types (**Either**). Another type constructor that takes two types is the function type ($\to$). We can write `(->) a b` to represent `a -> b`. The question is, is $\to$ a bifunctor?

We already knew that `(-> c)` is a functor, called the *Reader functor*.

```
instance Functor (->  c) where
  fmap :: (a -> b) -> (c -> a) -> (c -> b)
  fmap f g = f . g
```

What about we fix on the returning type and see if that's a functor. The `fmap` would have type `fmap :: (a -> b) -> (a -> c) -> (b -> c)`. If we name $f : a \to b$ and $g : a \to c$, we have this diagram: $\begin{array}{ccc} a & \xrightarrow{f} & b \\ \downarrow g & \swarrow & \\ c & & \end{array}$. We need to find the red arrow and obviously it cannot be constructed using $f$ and $g$, so it is not a functor.

Or is it? Why not saying it a functor mapping from the opposite category? This kind of functors is called a *contravariant functor*. Using the container analogy, a contravariant functor is like a container that store negative things: instead of containing something we need to give it something.

**Remark.** The *opposite category* $C^{op}$ of category $C$ is a category where for every objects in $C$ there is an object in $C^{op}$ and for every morphisms $f : a \to b$ in $C$ there is a morphism $f^{op} : b \to a$ in $C^{op}$.

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

data Op c a = Op (a -> c)

instance Contravariant (Co c) where
  contramap :: (b -> a) -> (Op c a) -> (Op c b)
--contramap :: (b -> a) -> (a -> c) -> (b -> c)
  contramap f (Op g) = Op (g . f)
```

So a profunctor is just a bifunctor that is contravariant in the first argument and covariant in the second. What's the problem? [1]

```
class Profunctor p where
-- define dimap
  dimap :: (a' -> a) -> (b -> b') -> p a b -> p a' b'
  dimap f g h = lmap f (rmap g h)
```

[1] https://www.schoolofhaskell.com/school/to-infinity-and-beyond/pick-of-the-week/profunctors#profunctors

```
-- or both lmap and rmap
  lmap :: (a' → a) → p a b → p a' b
  lmap = (`dimap` id)
  rmap :: (b → b') → p a b → p a b'
  rmap = (id `dimap`)
```

---

To function type, let's see how it is implemented. $f : a' \to a$, $g : b \to b'$, and $h : a \to b$. So the diagram is just $a' \xrightarrow{f} a \xrightarrow{h} b \xrightarrow{g} b'$. Bravo, finally we have this

approachable red arrow.

The functions `lmap` and `rmap` correspond to `contramap` and `fmap`, which is a implication of the fact that a profunctor is a bifunctor over a contravariant functor and a covariant functor.

## 8.1 Function objects, Exponentials

### 8.1.1 Function objects

Functions are separated from types so far: we learned in Hask that types are objects and functions are morphisms. For category **Set**, objects are all the sets, and functions are mapping between the objects. Functions from object $a$ to object $b$ forms a set, called hom-set, denoted as $\mathrm{Hom}(a, b)$. And because **Set** contains all sets, this morphisms representing hom-set $\mathrm{Hom}(a, b)$ is also an object in **Set**.

In an arbitrary category, we don't have the hom-sets as internal objects; but we can view them as external things being in other categories. What we want is something we can represent these hom-sets inside the category. It is indeed possible to define these objects in many categories, although not in arbitrary category. Just like we define product and coproduct, the same way we can apply to define this function object. We can use univeral construction to define function object.

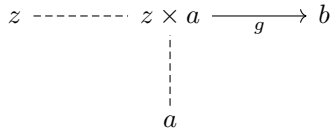
**Remark.** The 3 steps towards universal construction:

1. Define a pattern.
2. Define ranking between matches.
3. Find the best one.

It's clear to see that the pattern will involve three objects, the function $z$, the argument type $a$ and the return type $b$. In a category, we can't directly say $z$ takes argument $a$ returns $b$, so we think how can we work on this. For example, in category **Set**, we can say take $z$ and $a$ to form a pair $(z, a)$ such that $(z, a) \cong b$. This pairing in **Set** represent a cartesian product of $z$ and $a$. This perfectly captures the idea of function application, and this pattern can be generalize to any category.

This pattern implies that in order to define this construction in a category, the category must contain products. This makes more sense when later we discuss function objects as exponentials.

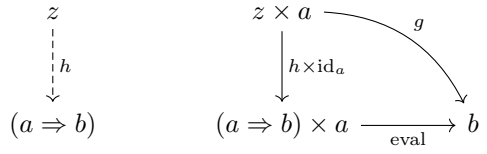
The pattern, drawn in communitive diagram, looks like:




Note that the morphism $g$ is called a *eval* morphism. Next we need to define a ranking to select the best function $z$ from $a$ to $b$.

With the pattern above, we say $(a \Rightarrow b)$ is the function object from $a$ to $b$ if for other patterns defined on $z$ and $a$, we have a unique morphism $h : z \dashrightarrow (a \Rightarrow b)$,

that means we have to be able to factorize $g : z \times a \longrightarrow b$ into eval arrow and a unique arrow $z \times a \dashrightarrow (a \Rightarrow b) \times a$. eval arrow is given by the pattern, so we need to find the other arrow. By previously learned functoriality of ADTs, we know that product is a bifunctor, thus it not only maps objects, it also maps morphisms. By which we can say the unique arrow we are looking for is $h \times \mathrm{id}_a$. The whole construct can be shown as the diagram below:




$a$

(arrows representing $\pi$ maps in products are omitted for clarity)

In short, the function object $(a \Rightarrow b)$ only exists if for every possible $z$ and $g$ in that pattern, there is a unique $h$ from $z$ to $(a \Rightarrow b)$ such that:

$$g = \mathrm{eval} \circ (h \times \mathrm{id})$$

In most languages, functions with two arguments are basically functions with one argument which is a pair. Think of $g$ as such a function, that takes $z$ and $a$ as arguments. Our definition of function objects implies given $z$ and $g$ we actually have a unique arrow $h$, that takes a $z$ and returns a function object $(a \Rightarrow b)$.

This captures the idea of partially applying function is equivalent to the applying function with its arguments given in a pair; the idea can be generalized to functions with multiple arguments. This gives rise to the concept of currying.

```
h :: z → (a → b)
g :: (z, a) → b

curry :: ((a,b)→ c) → (a→ b→ c)
curry f = λa→ (λb→ f (a,b))

uncurry ::(a→ b→ c) → ((a,b)→ c)
uncurry f = λ(a,b)→ (f a) b
```

### 8.1.2 Exponential

Function objects in category theory are actually called exponential. A function object $(a \Rightarrow b)$ will be called $b^a$. This makes sense if you think of the cardinality

of function types. In Haskell, `Bool` $\to$ `Int` is basically (`Int`, `Int`), or `Int` $\times$ `Int` in category notation, or simpler $\texttt{Int}^2$. Remember we learned the correspondence between nature numbers and Hask types: 1 corresponds to unit (), 2 corresponds to `Bool`, etc. Then $\texttt{Int}^2$ is basically $\texttt{Int}^{\texttt{Bool}}$.

The idea shows the connection between products and exponentials (*i.e.* functions). A special kind of categories called **Cartesian Closed Category** (CCC) is useful in programming. Cartesian means there is a product for every pair of objects. Closed means the products are *inside* the category, which means it has all the exponentials objects as well. Also, it must have the terminal object. Why? Terminal object is like the 0-th power of any objects. Sometimes we also want co-products in the category, this kind of categories is called Bi-Cartesian Closed Categories (BCCC), which also has the initial object.

With products and co-products both being monoidal, we can combine them together to form a semi-ring, and do algebra on them. Now we add exponential to the system. With exponentials we can do more algebra, for example, $a^0 = 1$.

We can interpret $a^0 = 1$ as `Void` $\to a \sim ()$, *i.e.* a function from void type to $a$ is equivalent to the unit type. We need to show the function on the left hand side exists and is unique. Turns out there is this function, called **absurd**.

What about $1^a = 1$? It is a function takes $a$ and returns unit (); we say it is equivalent to just (). There is only one such function indeed, called `const` ().

Let's try another one, $a^1 = a$, This is saying a function from () to $a$ is equivalent to $a$. These functions are just the selecting functions; there are as many as the number of elements in $a$.

## 8.2 Type algebra, Curry-Howard-Lambek isomorphism

### 8.2.1 Type algebra

We have learned the correspondence between some categorical structures and algebra.

| Algebra | Categorical Structure |
|---|---|
| $a + b$ | Coproduct |
| $a \times b$ | Product |
| $a^b$ | Exponential (Function Object) |

Figure 1: Categorical representation of algebra operations

Now we figure out how to do algebra in the category of types. Look at this algebra expression,

$$a^{b+c} = a^b \times a^c$$

This is essentially saying that **Either** `b c` $\rightarrow$ `a` is equivalent to a pair of functions `(b` $\rightarrow$ `a, c` $\rightarrow$ `a)`. We can do case analysis: Either type gives one of its params, if given parameter is `b`, it should call function `b` $\rightarrow$ `a`; if given `c`, it should call `c` $\rightarrow$ `a`. It works. It's easy to show the other way around from `(b` $\rightarrow$ `a, c` $\rightarrow$ `a)` to **Either** `b c` $\rightarrow$ `a`.

Let's see the next one,

$$(a^b)^c = a^{b \times c}$$

It should be easy to recognize this pattern is exactly what currying does, that is, `c` $\rightarrow$ `(b` $\rightarrow$ `a)` is equivalent to `(b, c)` $\rightarrow$ `a`.

Next one,

$$(a \times b)^c = a^c \times b^c$$

This is saying `c`$\rightarrow$ `(a,b)` is equivalent to `(c` $\rightarrow$ `a, c` $\rightarrow$ `b)`, which makes sense since given a function that takes a `c` and spit out `a` and `b` is the same as given two function, both take a `c`, one spit out `a` and another one spit out `b`.

All above algebra holds in Haskell and other BCCC languages.

### 8.2.2 Curry-Howard-Lambek isomorphism

We started to realize that the algebra we learned in high school seems to apply to a broader and a broader range of things. Starting from numbers, then to variables, then to functions, then to types and categories. All these things are so similar and the similarity extends further.

The other thing fall out from this is that the same structures here we see in types and categories appear in logics as well. This is the famous **Curry-Howard isomorphism**, or **Proposition as Types**. So all these we talked about in type theory and categories today actually have one-to-one correspondence in logic.

The isomorphism between type theory and logic starts with identifying what it means to be a proposition. A proposition is a statement that can either be true or false. In type theory, a type can be either inhabited or not. The truth of a proposition corresponds to that the type has elements, that is, they are inhabited.

Most of the types we learned are inhabited, while there are some that are not, so they corresponds to false propositions. A classic example we know is the **Void** type, which has no members.

Another thing we have in logic is proof. If we want to prove a proposition we can construct an element of the corresponding type. In logic there are two basic propositions true and false, true is always true, and false is always false. The corresponding things in type is: **Void** type corresponds to false, `Unit` type corresponds to true.

Then we have conjunction ($\wedge$). The conjunction of $a$ and $b$ ($a \wedge b$) is corresponds to the pair `(a,b)` in type theory. We can only form the pair type if only we have both an element from $a$ and an element from $b$.

Disjunction ($\vee$) works similarly, in the way that $a \vee b$ corresponds to **Either** `a b`. If we have either an element of $a$ or an element of $b$, we can construct a **Either** `a b`.

Implication ($\Rightarrow$) corresponds to the function type. $a \Rightarrow b$ corresponds to the function type `a` $\rightarrow$ `b`. The function type `a` $\rightarrow$ `b` is essentially meaning, give me an element of `a`, I'll give you an element of `b`. In other words, if I have the function then I can produce `b` from `a`. The same is true for implication. In logic, if we have $(a \Rightarrow b) \wedge a$ we can conclude $b$. Likewise, if we have an inhibited pair `(a` $\rightarrow$ `b, b)`, we can derive an element of `b`.

Curry-Howard proposed the correpondence between logic and type theory, and Lambek contributed the correspondence between them and category theory.

A cartesian closed category is a model for logic and also a model for type theory, where all these three things are the same. Some mathematicians may discover some idea in logic, and then computer scientist can pick it up and

| Category | Logic | Type |
|---|---|---|
| terminal object | true | `()` |
| initial object | false | **Void** |
| product | $a \wedge b$ | `(a,b)` |
| coproduct | $a \vee b$ | **Either** `a b` |
| exponetial | $a \Rightarrow b$ | `a` $\rightarrow$ `b` |

Figure 2: Table showing the correspondance

find it can be some type. Some modern concepts are discovered and found applicable in different fields in this way. An example is Linear Logic, which computer scientists translated into programming languages using Curry-Howard Correspondence, called Linear Type. In a linear type system each object is used exactly once. This system makes up the basis of Rust's ownership and C++'s `unique_ptr`.

## 9.1 Natural Transformations

Natural transformation is the third thing in the most important foundations of category theory. The first is the definition of category, and the second is functor, natural transformation is the third.

Category is about structure. Functors are mappings between categories that preserves the structure. The intuition about functor is that they take a category and embed it into another category. It's like searching a pattern in a category. We would also want to compare two functors, or said differently, images given by functors. So compare things by mapping one into another, that's a natural way of modeling comparsion.

Natural transformations are defined as structure-preserving mappings between functors. Given two categories, $C$ and $D$, and two functors $F$ and $G$ which $F, G : C \to D$, let $a$ be an object in $C$, which $F$ maps $a$ to $Fa$, $G$ maps $a$ to $Ga$. If we want to map one functor to another functor, we want to map $Fa$ to $Ga$. We already have morphisms in $D$ for mapping $Fa$ to $Ga$. To define mapping between functors on $a$ is the same as picking a morphism in $D$ that maps from $Fa$ to $Ga$. So a natural transformation will, for every object $a$ in $C$, pick a morphism in $\mathrm{Hom}_D(Fa, Ga)$. This way a natural transformation is creating a family of morphisms in $D$. These individual morphisms are called *components* of the natural transformation. Denoted as $\alpha_a$ where $\alpha$ is the natural transformation.

So far, we only talked about what a natural transformation does on objects, we also need to take a look on morphisms. Given objects $a, b$ in $C$ and morphism $f : a \to b$, we have the functors $F, G$ which map $a$ to $Fa$, $Ga$, and map $b$ to $Fb$, $Gb$, respectively. Let natural transform $\alpha$ on $a$ and $b$ be $\alpha_a : Fa \to Ga$ and $\alpha_b : Fb \to Gb$.

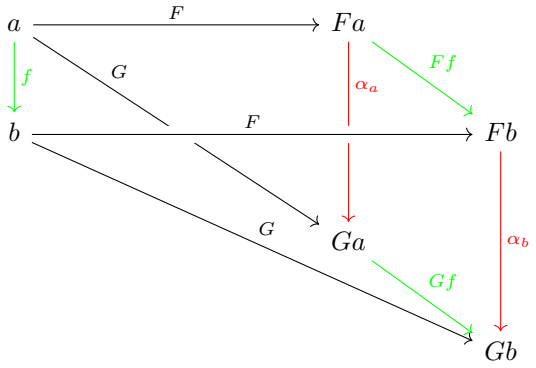



Figure 3: natural transofrm

We need to have the natural transform preserve structures of $F$ and $G$. That means these two paths from $Fa$ to $Gb$ must be equal. In other words, the

diagram must commute.

Naturality condition is very strong

Example in set. Natruality is almost determined.