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

Shou Ya

April 19, 2017

Contents

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.

$$
\begin{array}{ccc}
a & \xrightarrow{\ F_a\ } & F\ a \\
{\scriptstyle f}\downarrow & & \downarrow{\scriptstyle F\ f} \\
b & \xrightarrow{\ F_b\ } & F\ b
\end{array}
$$

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 Δ_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)`.

$$
\begin{array}{ccc}
\texttt{(e,x)} & \xrightarrow{\ \texttt{fmap f}\ } & \texttt{(e,y)} \\
\big\uparrow\texttt{(e,)} & & \texttt{(e,)}\big\uparrow \\
\texttt{x} & \xrightarrow{\ \ \texttt{f}\ \ } & \texttt{y}
\end{array}
$$

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:

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

And this is how `(,)` (product) and **Either** (sum) implemented as bifunctors:
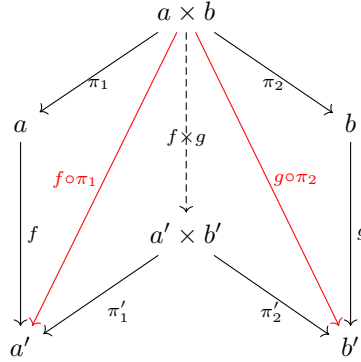
```
1 instance Bifunctor (,) where
2   bimap f g (a,b) = (f a, g b)

4 instance Bifunctor Either where
5   bimap f g (Left a) = Left (f a)
6   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:

$$a \times b$$

$$\pi_1 \qquad \pi_2$$

$$a \qquad\qquad b$$

$$f \qquad a' \times b' \qquad g$$

$$\pi_1' \qquad \pi_2'$$

$$a' \qquad\qquad b'$$
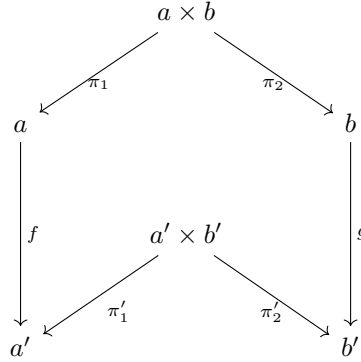
So we have this diagram. For each $a, b \in C$, by definition, we have its product $a \times b$ which projects (π_1, π_2) to a and b. And for the morphisms, we have $a \xrightarrow{f} a'$ and $b \xrightarrow{g} b'$. In order for the product $(\times : C \times C \to C)$, to be a bifunctor, we need to 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.

$$a \times b$$

$$\pi_1 \qquad\qquad \pi_2$$

$$a \qquad\qquad b$$

$$f \circ \pi_1 \qquad f \times g \qquad g \circ \pi_2$$

$$f \qquad a' \times b' \qquad g$$

$$\pi_1' \qquad \pi_2'$$

$$a' \qquad\qquad b'$$

Here's how we show that. Since we have π_1 and f, they will compose into $f \circ \pi_1$, same for π_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 $1 \in \mathbf{Set}$, 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 $\mathbf{Set}$, 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, $(a, ()) \cong a$.
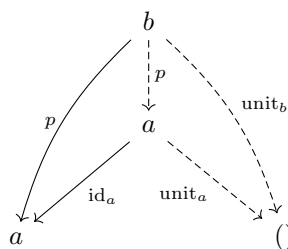
As we can see, the projections $\pi_1 = \mathrm{id}$ and $\pi_2 = \mathrm{unit}$. Given any $b \xrightarrow{p} a$, we have unit_b which maps b to $()$, we always have a unique factorization $b \dashrightarrow{p} a$. This means the projections just worked and we indeed have $(a, ()) \cong a$.

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 $C \otimes C \to C$).

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**? So here's the place for the constant functor trick.

```
1  data Const c a = Const c

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

In category theory, constant functor Δ_c maps every object in a category to the object c and every morphisms to 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:

```
1  data Identity a = Identity a

3  instance Functor Identity a where
4    fmap :: (a → b) → Identity a → Identity a
5    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.

```
1  data Maybe a = Nothing | Just a
2  −− sum types can be decomposed into Either
3  data Maybe a = Either Nothing (Just a)
4  −− any constant type can be represented using Const functor
5  −− and Just is just an Identity functor
6  data Maybe a = Either (Const () a) (Identity a)
```

Now we have a `Const ()` functor and an `Identity` functor; and the **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.

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 ($\to$) `a b` to represent

`a → b`. The question is, is `→` a bifunctor?

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

```
1  instance Functor (→ c) where
2    fmap :: (a → b) → (c → a) → (c → b)
3    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 $f : a → b$ and $g : a → c$, we have this diagram: $\begin{array}{c} a \xrightarrow{f} b \\ \downarrow g \ \ \nearrow \\ c \end{array}$. 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 $f : a → b$ in C there is a morphism $f^{op} : b → a$ in C^{op}.

```
1  class Contravariant f where
2    contramap :: (b → a) → f a → f b

4  data Co c a = Co (a → c)

6  instance Contravariant (Co c) where
7  −−contramap :: (b → a) → (a → c) → (b → c)
8    contramap :: (b → a) → (Co c a) → (Co c b)
9    contramap f (Co g) = Co (f ∘ g)
```

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

```
1  class Profunctor p where
2    dimap :: (a' → a) → (b → b') → p a b → p a' b'
3    dimap f g h = lmap f (rmap g h)

5    lmap :: (a' → a) → p a b → p a' b
6    lmap = (`dimap` id)
7    rmap :: (b → b') → p a b → p a b'
8    rmap = (id `dimap`)
```

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

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.