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

Shou Ya

April 19, 2017

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

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.

$$\begin{tikzcd} a \arrow[r, "F_a"] \arrow[d, "f"] & F\ a \arrow[d, "F\ f"] \\ b \arrow[r, "F_b"] & F\ b \end{tikzcd}$$

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 $\mathrm{id} \in \mathrm{Arr}(1)$ 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 are 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)`.

$$\begin{tikzcd} \texttt{(e,x)} \arrow[r, "\texttt{fmap f}"] & \texttt{(e,y)} \\ \texttt{x} \arrow[u, "\texttt{(e,)}"] \arrow[r, "\texttt{f}"] & \texttt{y} \arrow[u, "\texttt{(e,)}"'] \end{tikzcd}$$

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:

```haskell
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:

```haskell
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:

$$
\begin{tikzcd}
& a \times b \arrow[dl, "\pi_1"'] \arrow[dr, "\pi_2"] & \\
a \arrow[dd, "f"] & & b \arrow[dd, "g"] \\
& a' \times b' \arrow[dl, "\pi_1'"'] \arrow[dr, "\pi_2'"] & \\
a' & & b'
\end{tikzcd}
$$

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.

$$
\begin{tikzcd}
& a \times b \arrow[dl, "\pi_1"'] \arrow[dr, "\pi_2"] \arrow[dd, dashed, "f \times g"] \arrow[dddl, red, "\pi_1 \circ f"'] \arrow[dddr, red, "\pi_2 \circ g"] & \\
a \arrow[dd, "f"] & & b \arrow[dd, "g"] \\
& a' \times b' \arrow[dl, "\pi_1'"'] \arrow[dr, "\pi_2'"] & \\
a' & & b'
\end{tikzcd}
$$

Here's how we show that. Since we have π_1 and f, they will compose into $\pi_1 \circ f$, 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.