Notes on Bartosz Milewski - Truth about Types (Lambda Days 2016)

Rakhim Davletkaliyev

February 15, 2019

Contents

1 Intro

Category theory abstracts many branches of math. You can describe topology, logic, Banach spaces and other concepts in terms of category theory. Why is it so? Does category theory describe the nature, the intrinsic reality, and we happened to discover it like physicists discover the laws of the universe?

The main idea of category theory is composition. This is why it is so relevant in programming. Composition is the essence of category theory.

Is composability intrinsic in nature? It seems like that on a larger scale, and classical physics shows how everything is neatly composable. The ancient idea of atoms is the idea of perfect composability: things are composed with indivisible elementary particles, and the higher you go, the more composable everything is. Atoms make molecules, molecules make objects, object make bigger objects and interact using laws that are themselves composable. Newtonian mechanics describes objects in the universe that follow the rules of interaction that are clearly affecting each other, acting like independent agents.

But the deeper you go, the less composable it becomes. Two quantum particles don't interact in a simply composable fashion, their wave functions do not just affect each other in a stable manner. Two quantum particles near one another start to behave like a special two-particle state, not two wave functions. This is why quantum theory is so hard to describe and this is probably why physicists struggle with these theories. They go against our intuition of a composable universe.

So, maybe composability is not intrinsic in nature, maybe it is intrinsic for our limited monkey brains. We need composability in order to deal with the complex reality. Maybe, higher super-beings wouldn't invent category theory at all.

2 Truth

In logic, there is an axiom that basically says "truth is true", and you don't need to prove it.

$$\frac{}{\top true}\top_I \tag{1}$$

In type theory, it corresponds to unit type:

$$() : () \tag{2}$$

And in category theory it corresponds to the terminal object, which we will cover later.

Three things look different, but they are actually just different notations for the same concept.

As for proofs, in logic you'd prove a proposition A by providing some statements:

$$\frac{[...]}{A} \tag{3}$$

In type theory, you'd prove that type **A** is inhabited (has members):

$$\Gamma \vdash x : A \tag{4}$$

In the context of categories, a proof would be a morphism from terminal object:

$$1 \to A \tag{5}$$

3 Category theory

A category is a collection of objects and arrows.

Similar to a directed graph. Nodes are called **objects** (a, b, c,...). Arrows between objects are called **morphisms**: `f :: a -> b`.

3.1 Composability

Arrows are composable:

$$f :: a \to b \tag{6}$$

$$g :: b \to c \tag{7}$$

$$g \circ f :: a \to c \tag{8}$$

The notation $g \circ f$ means "apply `g` after `f`".

So, composition is associative.

3.2 Identity arrows

There always exist identity arrows that point to the same object:

$$id_a :: a \to a \tag{9}$$

$$id \circ f = f \tag{10}$$

$$g \circ id = g \tag{11}$$

There could be multiple arrows from `a` to `a`, but at least one is identity.

3.3 Set

Set is an example of a category. Set is a category in which:

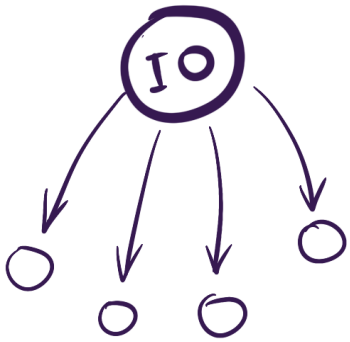
- Objects are sets
- Arrows are functions

In category theory, you cannot "look" inside objects, you only care about their connections. So, two sets are considered to be the same or different based on the arrows, but not the contents.

4 Universal constructions in category theory

4.1 Initial object

An initial object is an object that has a unique arrow to all other objects. Only one arrow per object.



In Set, the equivalent would be an empty set $\varnothing$. A function $varnothing \to a$ is a function that maps an element of empty set to any other element of other set. This doesn't make sense: an element of empty set? But this is just an explanation in terms of set theory, and while it doesn't make sense, we can think about this nevertheless, as of "objects defined by their arrow property".

You can always create a function from an empty set, but you'll never be able to call this function. In the abstract, in vacuum this is true.

In Haskell there's Absurd function:

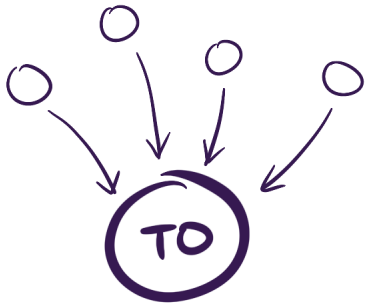
```
Absurd :: Void -> a
```

Which is a reference to "ad absurdum": from falsehood, you can derive anythings. "If pigs can fly, then I am the president".

`Void` here represents a type `Void`, an uninhabited type. This is not the same void-type we use in C/C++.

4.2 Terminal object

The inverse of initial object is the terminal object. An object with a unique arrow from all objects.



In set, it's a singleton set. Unique function: for every element of set **a** return the single element of the singleton set. In Haskell, the unit function ignores the argument and returns the one element from the set. The unit type `()` has one element `()`.

Type:

```
unit :: a -> ()
```

Element of this type:

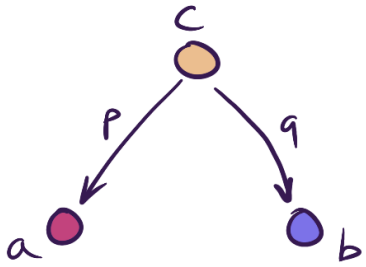
```
unit _ -> ()
```

If there are many singleton sets, they are isomorphic (i.e., a set of one apple is the same as a set of one orange).

5 Product

We need these universal constructions because they provide a way of defining things (since you cannot look inside the objects). One of the things that can be defined is the product.

- `c` is a product of `a` and `b`
- two arrows `p` and `q` (projections)
- in Set: Cartesian product, pairs of elements
- in logic: logical `AND` (conjuction elimination)



The only thing we know is that there are two morphisms. If we translate this into logic, it corresponds exactly to the elimination (AND). If `a AND b` is true, then `a` is true:

$$\frac{a \wedge b}{a} \tag{12}$$

If `a AND b` is true, `b` is true:

$$\frac{a \wedge b}{b} \tag{13}$$

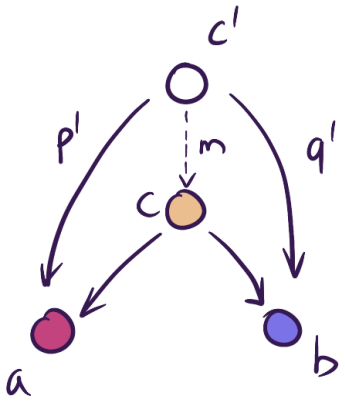
This corresponds to morphisms like so:

```
c :: (a, b)
p(a, b) = a
q(a, b) = b
```

There can be many objects like `c` with such two projections. Only one could the product. Which one? We need to pinpoint the correct `c` that corresponds to the Cartesian product. We need a way to evaluate the instance of this pattern of projections.

If we can get all similar patterns in the category, we need a way to distinguish the "top" one, the one that matches the the best. That one would be the product.

Here we have two examples of such pattern:



`c'` is a candidate product and it has `p'` and `q'` projections. `c` is better if there is a unique arrow (morphism) `m` from `c'` to `c` that fulfils these conditions:

$$\begin{aligned} m &:: c' \to c \\ p' &= p \circ m \\ q' &= q \circ m \end{aligned} \tag{14}$$

Both of these have the common factor `m` (but instead of multiplication you have composition). So they are not really elementary, since we can factor out a common `m` out of them. This makes `c` better ("more pure").

This forms ranking among candidates. In some cases you can indeed use ranking to get the unique best option and that will be the product. Not all categories would have this.

Again, there is a connection to logic:

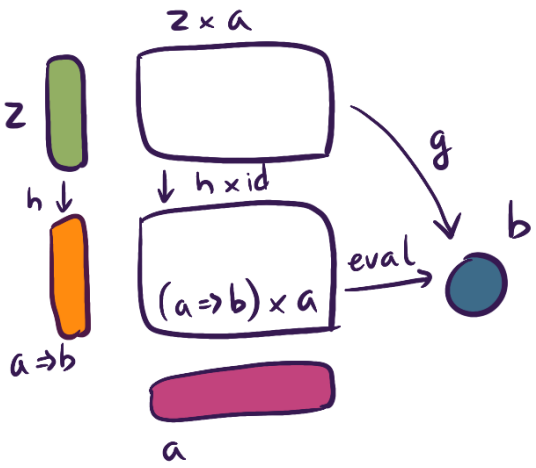
```
if a follows from c'        (p')
and b follows from c'       (q')
a ^ b (c) follows from c'   (m)
```

6 Function object

Another universal construction.

What is type function in Haskell? It's not really a morphism. Morphism is not an object. A set of morphisms between two objects corresponds to type `a->b`. Is there an object in the category that corresponds to the set of morphisms? If so, it's called the Function Object.

In order to define the function object (in Haskell, it's `a->b`, which is a type), you have to have products.



If you take a function `a->b` and apply argument `a`, you get `b`. Similar to product, if there is another candidate object `z`, that evaluates to the same `b`, and you can find unique morphism `h` then you can factor it out and prove that `a->b` candidate is the best.

This corresponds to modus ponens in proof theory.

$$\frac{(a => b) \wedge a}{b} \tag{15}$$

If you have a function from `a` to `b` **and** `a` is true, then `b` is also true.

7 Negation

Another thing you can construct is negation. `Not A` corresponds to an arrow (morphism) `A -> Void`. Why is this a negation?

In the context of type theory:

- If `A` is inhabited (has elements), then `A->Void` is not inhabited. Because otherwise you'd be able to create an element of `Void`, which suppose to have no elements.
- If `A` is not inhabited (is Void), then `Void->Void` is id_{void}.

8 Cartesian Closed Category

Combining these three things, we get a Cartesian Closed Category (CCC):

- Has all products (Cartesian)
- Has all function objects (exponentials) (this makes it closed)
- Has terminal object (nullary product)

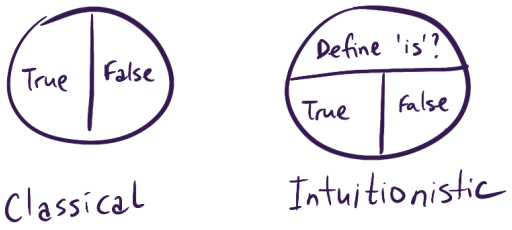
9 Curry-Howard-Lambek

This is an extension of Curry-Howard isomorphism. CCC is a model for simply typed lambda calculus. In addition to the fact that simply typed lambda calculus can be a model for logic.

- Objects are types
- Morphisms are terms (in logic/programming)
- Environment Γ is a prodict of judgements `a:A`
- Empty environment is `()` : `()`

10 Logical universes

There are two major kinds of logic: Classical and Intuitionistic.



In classical logic, any proposition is either true or false. But then Kurt Gödel comes along and says "no, there are some statements that are neither false nor true", which leads to intuitionistic logic where a statement can be true, can be false, but you can also have some gray area where you ask what "is" means.

10.1 Intuitionistic logic

- No LEM: law of excluded middle (either A or not A). `A | (A->Void)` is not provable.
- We cannot eliminate double negation: `Not Not A` is not the same as `A` (`((A->Void)->Void` is not the same as `A`)

Curry-Howard equivalence says that simple typed lambda calculus is equivalent to intuitionistic logic. You cannot prove LEM or double negation using lambda calculus.

So, we cannot do classical logic? This means it's not relevant to programming then?

10.2 Goedel Gentzen

Turns out, classical logic can be embedded in intuitionistic logic. Classical logic = Intuitionistic + double negation elimination (or LEM).

You can take a proposition in classical logic that is provable, you can translate it into some other proposition in intuitionistic logic, and then translate the proof also that does not use double negation elimination or LEM, and you will get something. There is a correspondence between propositions in two types of logic.

The transformation of proposition is simple: invert everything by applying double negation to all assumptions. Get a new theorem, prove it in intuitionistic logic, apply double negation to it, you get back the result of that first theorem.

Double twist, prove, double twist!

What does it mean? It means **continuations**. Double negation is `(a->Void)->Void`, or more general:

$$(a \to r) \to r \tag{16}$$

This $(a \to r) \to r$ is a continuation. If you extend lambda calculus with continuation (or do CPS transformation), you have a way to embed classical logic into programming.

10.3 Yoneda's Lemma

This is a deeper topic in category theory, and Bartosz mentions it without explanation just to show the importance of continuation in category theory. Read more about Yoneda's lemma on Wikipedia.

11 Conclusions

Type theory (typed lambda calculus), category theory (cartesian closed) and logic are all the same, just different notations. There is a lot of cross-polination between these areas. Is the Grand Unified Theory coming soon? Homotopy Type Theory?

Maybe.