15

The Yoneda Lemma

Most constructions in category theory are generalizations of results from other more specific areas of mathematics. Things like products, coproducts, monoids, exponentials, etc., have been known long before category theory. They might have been known under different names in different branches of mathematics. A Cartesian product in set theory, a meet in order theory, a conjunction in logic — they are all specific examples of the abstract idea of a categorical product.

The Yoneda lemma stands out in this respect as a sweeping statement about categories in general with little or no precedent in other branches of mathematics. Some say that its closest analog is Cayley's theorem in group theory (every group is isomorphic to a permutation group of some set).

The setting for the Yoneda lemma is an arbitrary category $\mathbf{C}$ together with a functor F from $\mathbf{C}$ to **Set**. We've seen in the previous section that some **Set**-valued functors are representable, that is isomorphic to a hom-

functor. The Yoneda lemma tells us that all **Set**-valued functors can be obtained from hom-functors through natural transformations, and it explicitly enumerates all such transformations.

When I talked about natural transformations, I mentioned that the naturality condition can be quite restrictive. When you define a component of a natural transformation at one object, naturality may be strong enough to "transport" this component to another object that is connected to it through a morphism. The more arrows between objects in the source and the target categories there are, the more constraints you have for transporting the components of natural transformations. **Set** happens to be a very arrow-rich category.

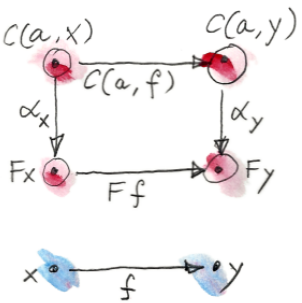
The Yoneda lemma tells us that a natural transformation between a hom-functor and any other functor F is completely determined by specifying the value of its single component at just one point! The rest of the natural transformation just follows from naturality conditions.

So let's review the naturality condition between the two functors involved in the Yoneda lemma. The first functor is the hom-functor. It maps any object x in $\mathbf{C}$ to the set of morphisms $\mathbf{C}(a, x)$ — for a a fixed object in $\mathbf{C}$. We've also seen that it maps any morphism f from $x \to y$ to $\mathbf{C}(a, f)$.

The second functor is an arbitrary **Set**-valued functor F.

Let's call the natural transformation between these two functors α. Because we are operating in **Set**, the components of the natural transformation, like α_x or α_y, are just regular functions between sets:

$$\alpha_x :: \mathbf{C}(a, x) \to Fx$$
$$\alpha_y :: \mathbf{C}(a, y) \to Fy$$



And because these are just functions, we can look at their values at specific points. But what's a point in the set $\mathbf{C}(a, x)$? Here's the key observation: Every point in the set $\mathbf{C}(a, x)$ is also a morphism h from a to x.

So the naturality square for α:

$$\alpha_y \circ \mathbf{C}(a, f) = Ff \circ \alpha_x$$

becomes, point-wise, when acting on h:

$$\alpha_y(\mathbf{C}(a, f)h) = (Ff)(\alpha_x h)$$

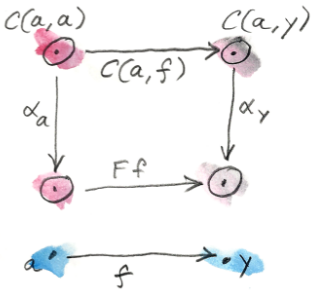
You might recall from the previous section that the action of the hom-functor $\mathbf{C}(a, -)$ on a morphism f was defined as precomposition:

$$\mathbf{C}(a, f)h = f \circ h$$

which leads to:

$$\alpha_y(f \circ h) = (Ff)(\alpha_x h)$$

Just how strong this condition is can be seen by specializing it to the case of $x = a$.



In that case h becomes a morphism from a to a. We know that there is at least one such morphism, $h = \mathbf{id}_a$. Let's plug it in:

$$\alpha_y f = (Ff)(\alpha_a \mathbf{id}_a)$$

Notice what has just happened: The left hand side is the action of α_y on an arbitrary element f of $\mathbf{C}(a, y)$. And it is totally determined by the single value of α_a at $\mathbf{id}_a$. We can pick any such value and it will generate a natural transformation. Since the values of α_a are in the set Fa, any point in Fa will define some α.

Conversely, given any natural transformation α from $\mathbf{C}(a, -)$ to F, you can evaluate it at $\mathbf{id}_a$ to get a point in Fa.

We have just proven the Yoneda lemma:

> There is a one-to-one correspondence between natural transformations from $\mathbf{C}(a, -)$ to F and elements of Fa.

in other words,

$$\mathbf{Nat}(\mathbf{C}(a, -), F) \cong Fa$$

Or, if we use the notation $[\mathbf{C}, \mathbf{Set}]$ for the functor category between $\mathbf{C}$ and $\mathbf{Set}$, the set of natural transformation is just a hom-set in that category,

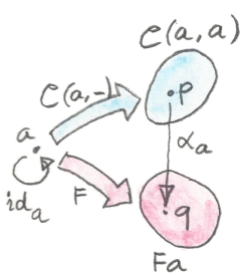
and we can write:

$$[\mathbf{C}, \mathbf{Set}](\mathbf{C}(a, -), F) \cong Fa$$

I'll explain later how this correspondence is in fact a natural isomorphism.

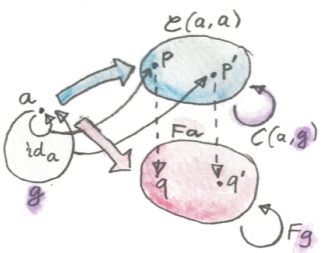
Now let's try to get some intuition about this result. The most amazing thing is that the whole natural transformation crystallizes from just one nucleation site: the value we assign to it at $\mathbf{id}_a$. It spreads from that point following the naturality condition. It floods the image of $\mathbf{C}$ in $\mathbf{Set}$. So let's first consider what the image of $\mathbf{C}$ is under $\mathbf{C}(a, -)$.

Let's start with the image of a itself. Under the hom-functor $\mathbf{C}(a, -)$, a is mapped to the set $\mathbf{C}(a, a)$. Under the functor F, on the other hand, it is mapped to the set Fa. The component of the natural transformation α_a is some function from $\mathbf{C}(a, a)$ to Fa. Let's focus on just one point in the set $\mathbf{C}(a, a)$, the point corresponding to the morphism $\mathbf{id}_a$. To emphasize the fact that it's just a point in a set, let's call it p. The component α_a should map p to some point q in Fa. I'll show you that any choice of q leads to a unique natural transformation.



The first claim is that the choice of one point q uniquely determines the rest of the function α_a. Indeed, let's pick any other point, p' in $\mathbf{C}(a, a)$, corresponding to some morphism g from a to a. And here's where the

magic of the Yoneda lemma happens: g can be viewed as a point p' in the set $\mathbf{C}(a, a)$. At the same time, it selects two *functions* between sets. Indeed, under the hom-functor, the morphism g is mapped to a function $\mathbf{C}(a, g)$; and under F it's mapped to Fg.



Now let's consider the action of $\mathbf{C}(a, g)$ on our original p which, as you remember, corresponds to $\mathbf{id}_a$. It is defined as precomposition, $g \circ \mathbf{id}_a$, which is equal to g, which corresponds to our point p'. So the morphism g is mapped to a function that, when acting on p produces p', which is g. We have come full circle!

Now consider the action of Fg on q. It is some q', a point in Fa. To complete the naturality square, p' must be mapped to q' under α_a. We picked an arbitrary p' (an arbitrary g) and derived its mapping under α_a. The function α_a is thus completely determined.

The second claim is that α_x is uniquely determined for any object x in $\mathbf{C}$ that is connected to a. The reasoning is analogous, except that now we have two more sets, $\mathbf{C}(a, x)$ and Fx, and the morphism g from a to x is mapped, under the hom-functor, to:

$$\mathbf{C}(a, g) :: \mathbf{C}(a, a) \to \mathbf{C}(a, x)$$

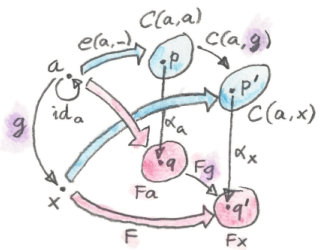
and under F to:

$$Fg :: Fa \to Fx$$

Again, $\mathbf{C}(a, g)$ acting on our p is given by the precomposition: $g \circ \mathbf{id}_a$, which corresponds to a point p' in $\mathbf{C}(a, x)$. Naturality determines the value of α_x acting on p' to be:

$$q' = (Fg)q$$

Since p' was arbitrary, the whole function α_x is thus determined.



What if there are objects in $\mathbf{C}$ that have no connection to a? They are all mapped under $\mathbf{C}(a, -)$ to a single set — the empty set. Recall that the empty set is the initial object in the category of sets. It means that there is a unique function from this set to any other set. We called this function `absurd`. So here, again, we have no choice for the component of the natural transformation: it can only be `absurd`.

One way of understanding the Yoneda lemma is to realize that natural transformations between **Set**-valued functors are just families of functions, and functions are in general lossy. A function may collapse information and it may cover only parts of its codomain. The only functions that are not lossy are the ones that are invertible — the isomorphisms. It follows then that the best structure-preserving **Set**-valued functors are the representable ones. They are either the hom-functors or the functors that are naturally isomorphic to hom-functors. Any other

functor F is obtained from a hom-functor through a lossy transformation. Such a transformation may not only lose information, but it may also cover only a small part of the image of the functor F in **Set**.

15.1 Yoneda in Haskell

We have already encountered the hom-functor in Haskell under the guise of the reader functor:

```
type Reader a x = a -> x
```

The reader maps morphisms (here, functions) by precomposition:

```
instance Functor (Reader a) where
    fmap f h = f . h
```

The Yoneda lemma tells us that the reader functor can be naturally mapped to any other functor.

A natural transformation is a polymorphic function. So given a functor `F`, we have a mapping to it from the reader functor:

```
alpha :: forall x . (a -> x) -> F x
```

As usual, `forall` is optional, but I like to write it explicitly to emphasize parametric polymorphism of natural transformations.

The Yoneda lemma tells us that these natural transformations are in one-to-one correspondence with the elements of `F a`:

```
forall x . (a -> x) -> F x ≅ F a
```

The right hand side of this identity is what we would normally consider a data structure. Remember the interpretation of functors as generalized containers? `F a` is a container of `a`. But the left hand side is a polymorphic function that takes a function as an argument. The Yoneda lemma tells us that the two representations are equivalent — they contain the same information.

Another way of saying this is: Give me a polymorphic function of the type:

```
alpha :: forall x . (a -> x) -> F x
```

and I'll produce a container of `a`. The trick is the one we used in the proof of the Yoneda lemma: we call this function with `id` to get an element of `F a`:

```
alpha id :: F a
```

The converse is also true: Given a value of the type `F a`:

```
fa :: F a
```

one can define a polymorphic function:

```
alpha h = fmap h fa
```

of the correct type. You can easily go back and forth between the two representations.

The advantage of having multiple representations is that one might be easier to compose than the other, or that one might be more efficient in some applications than the other.

The simplest illustration of this principle is the code transformation that is often used in compiler construction: the continuation passing style or CPS. It's the simplest application of the Yoneda lemma to the identity functor. Replacing `F` with identity produces:

```
forall r . (a -> r) -> r ≅ a
```

The interpretation of this formula is that any type `a` can be replaced by a function that takes a "handler" for `a`. A handler is a function accepting `a` and performing the rest of the computation — the continuation. (The type `r` usually encapsulates some kind of status code.)

This style of programming is very common in UIs, in asynchronous systems, and in concurrent programming. The drawback of CPS is that it involves inversion of control. The code is split between producers and consumers (handlers), and is not easily composable. Anybody who's done any amount of nontrivial web programming is familiar with the nightmare of spaghetti code from interacting stateful handlers. As we'll see later, judicious use of functors and monads can restore some compositional properties of CPS.

15.2 Co-Yoneda

As usual, we get a bonus construction by inverting the direction of arrows. The Yoneda lemma can be applied to the opposite category $\mathbf{C}^{op}$ to give us a mapping between contravariant functors.

Equivalently, we can derive the co-Yoneda lemma by fixing the target object of our hom-functors instead of the source. We get the contravariant hom-functor from $\mathbf{C}$ to **Set**: $\mathbf{C}(-, a)$. The contravariant version of the Yoneda lemma establishes one-to-one correspondence

between natural transformations from this functor to any other contravariant functor F and the elements of the set Fa:

$$\mathbf{Nat}(\mathbf{C}(-, a), F) \cong Fa$$

Here's the Haskell version of the co-Yoneda lemma:

```
forall x . (x -> a) -> F x ≅ F a
```

Notice that in some literature it's the contravariant version that's called the Yoneda lemma.

15.3 Challenges

1. Show that the two functions `phi` and `psi` that form the Yoneda isomorphism in Haskell are inverses of each other.

```
phi :: (forall x . (a -> x) -> F x) -> F a
phi alpha = alpha id

psi :: F a -> (forall x . (a -> x) -> F x)
psi fa h = fmap h fa
```

2. A discrete category is one that has objects but no morphisms other than identity morphisms. How does the Yoneda lemma work for functors from such a category?
3. A list of units `[()]` contains no other information but its length. So, as a data type, it can be considered an encoding of integers. An empty list encodes zero, a singleton `[()]` (a value, not a type) encodes one, and so on. Construct another representation of this data type using the Yoneda lemma for the list functor.

15.4 Bibliography

1. Catsters[1] video.

[1] https://www.youtube.com/watch?v=TLMxHB19khE