Translating the Yoneda Lemma

Alex Hubers

1 The Yoneda Lemma, as stated

The Yoneda Lemma, arguably the "most important result in category theory" Riehl (2017), has for many (me) been a consistent drop off point on the summit of mount category theory. I write this note to argue that, to the functional programmer, the Yoneda Lemma can be put a bit more clearly if one simply relaxes a handful of notational (and, cough, foundational) conventions.

Lemma (Yoneda (covariant))**.** *Let* $\mathcal{C}$ *be a locally small category and* $F : \mathcal{C} \to \mathrm{Set}$ *be a covariant functor. Then*

$$\mathrm{Hom}(\mathrm{Hom}(A, -), F) \simeq F(A)$$

for all objects A *in* $\mathcal{C}$*.*

In English: the set of natural transformations from the covariant hom-functor $\mathrm{Hom}(A, -)$ to F are in bijection with the set $F(A)$. When F is contravariant, the Yoneda lemma relates F to set of natural transformations between F and the contravariant Hom-functor $\mathrm{Hom}(-, A)$.

Lemma (Yoneda (contravariant))**.** *Let* $\mathcal{C}$ *be a locally small category and* $F : \mathcal{C}^{\mathrm{op}} \to \mathrm{Set}$ *be a contravariant functor. Then*

$$\mathrm{Hom}(\mathrm{Hom}(-, A), F) \simeq F(A)$$

for all objects A in $\mathcal{C}$.

1.1 The Yoneda Lemma for ~~dummies~~ the functional programmar

Let us now take a dollop of notational (ahem, foundational) liberties.

To the functional programmer, a natural transformation "is just"[1] a parametrically polymorphic function of type

```
forall x. f x -> g x
```

for functors F and G. Abusing further liberties, the "hom set" of arrows between A and B—written $\mathrm{Hom}(A, B)$—is "just" the type $A \to B$. Likewise, the covariant hom-functor $\mathrm{Hom}(A, -)$ can be written as the type level functor:

```
type Hom x = a -> x
```

That is: the covariant hom-functor sends types X to the set of functions into X from A. Putting one and one together, the set of natural transformations between $\mathrm{Hom}(A, -)$ and F, or $\mathrm{Hom}(\mathrm{Hom}(A, -), F)$, is just the type:

```
forall x. (a -> x) -> f x
```

and so the Yoneda lemma asserts that this type is in bijection with the type `f a`.

```
  forall x. (a -> x) -> f x
≃
  f a
```

That this bijection holds can be witnessed in a single line of Haskell. Look:

[1]This verbiage may be attributed to Kartik.

```haskell
newtype Yo f a = Yo { unYo :: forall x. (a -> x) -> f x }
```

Proof. The bijection can be witnessed easily. Let

```haskell
f :: (forall x. (a -> x) -> f x) -> f a
f φ = φ a id

g :: Functor f => f a -> (forall x. (a -> x) -> f x)
g d r = fmap r d
```

You may check yourself that the two functions are in fact inverse. □

As a further exercise, prove the Yoneda lemma when F is covariant. (Hint: replace the hom functor `type Hom x = a -> x` with the contravariant hom functor `type HomC x = x -> a` and see that the proof is identical. The definition of `g` pans out to be the same: `f` is contravariant and hence sends the arrow `r :: x -> a` to `fmap r :: f a -> f x`).

1.2 Mendler-Algebras and Yoneda

Let `f` be a convariant endofunctor, fix a type `a` and define the contravariant endofunctor `g` as

```haskell
type g x = f x -> a
```

By the contravariant Yoneda lemma, we should expect a bijection between `forall x. (x -> a) -> g x` and `g a`. That is:

```haskell
  forall x. (x -> a) -> g x
= forall x. (x -> a) -> f x -> a
≃ g a
= f a -> a
```

In other words, mendler F-algebras and regular-ass F-algebras are in bijection.

References

E. Riehl. Category theory in context. Aurora: Dover modern math originals. Dover Publications, 2017. ISBN 978-0-486-82080-4.