Type Arithmetic and the Yoneda Lemma

Emily Pillmore

August 28, 2019

Introduction

My name is Emily Pillmore.

I am a hopeful mathematician, and a moonlight programmer.

- Twitter (@emi1ypi)
- Meetups in NYC: NY Homotopy Type Theory, NY Category Theory, and the NY Haskell User Group.
- Discord: Haskell $\cap$ Dank Memes: https://discord.gg/2x2fYSK.
- Personal: All of my slides and meetup content are hosted at cohomolo.gy.

I got my start in Scala, with Runar's *Functional Programming in Scala* (the red one - there is now a blue one)

I now work at a company called **Kadena**, designing a language and a couple of cool blockchains

Today, we're going to learn to **count**. Then we will learn to **add**, **multiple**, and even **take powers**.

...and we will do this with the language of Type Theory, Category Theory, and Constructive Logic.

Many of you know these topics already. Much of it is folklore within the community. But there are new ways of thinking about it that I gaurantee you haven't seen before, and some of it even translates into our work on Profunctor Optics!

If Bartosz is in this audience, everyone stare at him right now.

We'll see if we have time for that last one. If we don't get to it, it will be included as an optional set of slides.

First, we'll start off with some exercises to illustrate the point, proving free theorems and basic facts about types (e.g. reasoning through why $\forall a.a \to a$ has precisely 1 inhabitant), and then draw direct lines with the Lambek correspondence.

Second, we will try to understand the Leibniz perspective and how it informs the Curry-Howard correspondence.

Third, we will begin to build the requisite knowledge for understanding the Yoneda perspective, and build some categorical constructs, and an understanding of representation.

Then, we will tie it all together.

Special thanks to Jon Pretty (@propensive)

Some Type Arithmetic

Let's do a quick warm up

We will use the following conventions:

- the type () with precisely 1 inhabitant (called Unit) will be denoted by the number 1.
- the type with no inhabitants, Nothing or undefined will be denoted 0.
- Function types will be denoted by $a \to b$ or by the notation b^a.
- Universal quantification is given by forall or $\forall$.

- Our calculus assumes that we have all finite coproducts and will denote them by the plus sign $+$. In Scala and Haskell, the binary coproduct is Either.
- Our calculus assumes that we have all finite products, and we will denote they by the times sign *. In Scala and Haskell, the binary product is (,) or ((,) is an alias for Tuple2 in Scala.

Proposition: Suppose we were to treat a polymorphic like your standard high school arithmetic. What would this arithmetic look like?

Does $1 + 1 = 2$? How can we tell?

By simple counting. Axiomatically, 1 is the type with 1 inhabitant, so

```
1 + 1
= () + ()
= Either () ()
```

By inspection, we see that Either has precisely two cases - Left and Right containing a single type. Thus, two cases, with one inhabitant in each case. Therefore we have 2 inhabitants!

How about *Maybe* containing values of type
Bool $= \{\top, \bot\}$?

Again, we can apply some natural deduction, nothing that ***Bool*** has precisely two inhabitants

```
Maybe Bool
= Nothing + Just (True + False)
= 1 + 1 + 1
= 3
```

Simple right? Lets move on.

Multiplication operates in the similar way. Using products, we can count $\mathbf{Bool} * \mathbf{Bool}$. We have precisely four cases:

- $(True, True)$
- $(True, False)$
- $(False, True)$
- $(False, False)$

This works out as well! How about exponents?