Type Arithmetic and the Yoneda Lemma

Emily Pillmore

August 29, 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. Hopefully you will be converted to Computational Trinitarianism by the time we're done.

Many of you know these topics already without knowing the details. 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 type 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 () ()
= 2
```

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?

Let's consider a function $Bool \to Bool$. How do we count the number of functions we have? Well, we can list them out:

```
k1 :: Bool -> Bool
k1 b
   | True = True
   | False = False

k2 :: Bool -> Bool
k2 b
   | True = True
   | False = True

-- and so on
```

tl;dr there are exactly 4 inabitants. Now, consider the functions

```
f :: Maybe Bool -> Bool
f ...
```

Upon inspection you might find that there are precisely 8 such functions. Inductively, we see that there are $|b|^{|a|}$-many inhabitants to the type $a \to b$. The cardinality of this type behaves like exponentiation in $\mathbb{N}$!

Considering all we have just discussed, there are caveats to this calculus. For example, how many inhabitants are there in the type $0 \to 1$? How about $1 \to 0$? $0 \to 0$?

The analogy so far:

Types	# of Inhabitants	Sets	Categories
a	$\|A\|$	?	?
(a, b)	$\|A\| \times \|B\|$	?	?
Either a b	$\|A\| + \|B\|$	?	?
$a \to b$	$\|B\|^{\|A\|}$	?	?
()	1	?	?
Void	0	?	?

Claim: we can fill in the rest of the table with direct analogies from Set theory and Category theory.

What does this look like?

```
leftUnitP   :: (1 × a) = a
commuteP    :: (a × b) = (b × a)
associateP  :: ((a × b) × c) = (a × (b × c))
leftUnitC   :: (0 + a) = a
commuteC    :: (a + b) = (b + a)
associateC  :: ((a + b) + c) = (a + (b + c))
currying    :: c^(a×b) = (c^b)^a
leftUnitF   :: a^1 = a
rightUnitF  :: 1^a = 1
```

Why do we care in the first place?

Because proving that a type T has inhabitants is equivalent to proving the theorem correspondnig with T due to the Curry-Howard-Lambek correspondence.

The Leibniz Perspective

When are two things equal? You may have noticed our use of $=$ in the previous slides, but that was a convenient lie.

Are these things equal?

- 1729
- $12^3 + 1^3$
- "The first number expressible by the sum of two cubes in two different ways"

When viewed as expressions, 1729 and $12^3 + 1^3$ are different.

When viewed as integers, they are the same.

Enumeration in this sense is an isomorphism from a type to a finite set.

```haskell
data Iso a b = Iso
    { to   :: a -> b
    , from :: b -> a
    }
    -- fromTo :: (x :: a) -> (from . to) x = x
    -- toFrom :: (x :: b) -> (to . from) x = x
```

We have been suffering from an abuse of notation - what we've really been saying is that two things are equal if we can exhibit an isomorphism between the two.

Proposition (Equality Preservation)

if A *and* B *are isomorphic, and* $a_1, a_2 : A$*, and* $b_1, b_2 : B$*, then* $a_1 = a_2 \iff b_1 = b_2$*.*

Proposition (Gottfried Wilhelm Leibniz, 1646-1716)

"For any x and y, if x is identical to y, then x and y have all the same properties. For any x and y, if x and y have all the same properties, then x is identical to y."

In symbols, $x = y \iff \forall P.Px = Py$ where P is quantified over all properties of x.

Let's see some interesting isomorphisms.

```
leftUnitP :: Iso ((), a) a
leftUnitP = Iso (\((), a) -> a)
                (\a -> ((), a))
commuteP :: Iso (a, b) (b, a)
commuteP = Iso (\(a, b) -> (b, a))
               (\(b, a) -> (a, b))
associateP :: Iso ((a, b), c) (a, (b, c))
associateP = Iso (\((a, b), c) -> (a, (b, c)))
                 (\(a, (b, c)) -> ((a, b), c))
```

What does this remind you of?

```
rightUnitP  :: Iso (a, ()) a
leftUnitP   :: Iso ((), a) a
associateP  :: Iso ((a, b), c) (a, (b, c))
```

```
leftUnitC :: Iso (Either Void a) a
leftUnitC = ...
commuteC :: Iso (Either a b) (Either b a)
commuteC = ...
associateC :: Iso (Either (Either a b) c)
                  (Either a (Either b c))
associateC =
  Iso (\case Left (Left a)  -> Left a;
             Left (Right b) -> Right (Left b);
             Right c        -> Right (Right c))
      (\case Left a          -> Left (Left a);
             Right (Left b)  -> Left (Right b);
             Right (Right c) -> Right c)
```

So what do our arithmetic rules *really* say about our types?

```
leftUnitP   :: (1 × a) ≅ a
commuteP    :: (a × b) ≅ (b × a)
associateP  :: ((a × b) × c) ≅ (a × (b × c))
leftUnitC   :: (0 + a) ≅ a
commuteC    :: (a + b) ≅ (b + a)
associateC  :: ((a + b) + c) ≅ (a + (b + c))
currying    :: c^(a×b) ≅ (c^b)^a
leftUnitF   :: a^1 ≅ a
rightUnitF  :: 1^a ≅ 1
```

By inspection, we can show that there is a correspondence between sets and types by looking at their cardinalities and inhabitants respectively.

What type has precisely $(a * b)$-many inhabitants? What set? Likewise, what type has precisely $(a + b)$-many inhabitants? What set?

We continue to build the picture.

There is a correspondence between types, sets, natural numbers.

Types	Sets	# of Inhabitants	Categories
a	A	$\lvert A\rvert$	?
(a, b)	$A \times B$	$\lvert A\rvert \times \lvert B\rvert$	?
Either a b	$A \sqcup B$	$\lvert A\rvert + \lvert B\rvert$	?
$a \to b$	set functions	$\lvert B\rvert^{\lvert A\rvert}$	?
()	$\{*\}$	1	?
Void	$\emptyset$	0	?

The Yoneda Perspective

Claim: the same correspondence can be made within the language of a bicartesian-closed category.

Lets elaborate on what that means. But first, some definitions.

Definition (Category)

A **category C** consists of the following data:

- a collection of **objects** $x, y, z, ...$
- a collection of **morphisms** $f, g, h, ...$

Such that

- each morphism has specified **domain** and **codomain** objects so that the notation $f : x \to y$ signifies a morphism with domain x and codomain y.
- each object has an associated **identity** morphism $1_x : x \to x$ which acts as a two-sided unital element for composition.
- to each pair of morphisms $f : x \to y$, $g : y \to z$, there exists a **composite** arrow $h : x \to z$ where $h :\equiv gf$.

For every category we may speak of its dual notion, $\mathbf{C}^{\mathrm{op}}$, the **opposite** category of $\mathbf{C}$, consisting of the same object data, but with the domain and codomain of each morphism reversed.

This data is subject to some axioms.

- For any $f : x \to y$, $f1_x = f = 1_y f$.
- For any composable triple f, g, h, $h(gf) = (hg)f$, and will simply be denoted hgf.
- Between any two objects x, y in a category $\mathbf{C}$, there exists an object $\mathbf{C}(x, y)$ consisting of all morphisms with domain x and codomain y. This is usually called $Hom_{\mathbf{C}}(x, y)$.

Indeed, we are familiar with morphisms, identities, and composition already. In haskell, these translate into types $a \to b$, the* identity function id, and composition $(.)$.

In scala, these are roughly the same, modulo naming conventions - $A \Rightarrow B$, $identity$, $compose$.

Isomorphisms in this context become a pair of morphisms $f : c \to d$ and $g : d \to c$ such that $gf = 1_c$ and $fg = 1_y$.

If x and y are isomorphic, we will denote this by $x \cong y$.

Definition (Functor)

A functor $F : \mathbf{C} \to \mathbf{D}$ between categories consists of the following data:

- an object Fc in $\mathbf{D}$ for every object c of $\mathbf{C}$
- a morphism $Ff : Fc \to Fd$ in $\mathbf{D}$ for each morphism $f : c \to d$ of $\mathbf{C}$.

Additionally, we require that $Fg \cdot Ff = F(gf)$ when g and f are a composable pair, and that for every object c in $\mathbf{C}$, $F1_c = 1_{Fc}$.

In Haskell, we represent this data as a typeclass

```
class fmap :: (a -> b) -> f a -> f c
-- fmap id fa = fa
-- fmap g . fmap f = fmap (g . f)
```

Functors that act uniformly on arrows are called **covariant**. However, if a functor is mapping to or from an opposite category to a normal category, then one calls these functors **contravariant**.

Operationally, this can be seen as "flipping the direction of arrows" in the target category.

This is often unnecessary data, as the saying a "F is a contravariant functor from $\mathbf{C}$ to $\mathbf{D}$" is precisely saying one has a covariant functor $F : \mathbf{C}^{\mathrm{op}} \to \mathbf{D}$. When it matters, I'll just tell you which one is opposite.

"Special" objects are determined by how morphisms to or from them act in relation to other objects.

Definition

Terminal and Initial objects An object c of $\mathbf{C}$ is called **terminal** if there is exactly 1 unique morphism from any other object in $\mathbf{C}$ to c. Dually, an object c of $\mathbf{C}$ is **initial** if there is exactly 1 unique morphism to any other object in $\mathbf{C}$.

We have seen two such objects that act like terminal and initial objects already: $()$ and $\bot$

Indeed for every type a, the types $a \to ()$ and $\bot \to a$ has precisely one inhabitant .

Namely, we have the following functions:

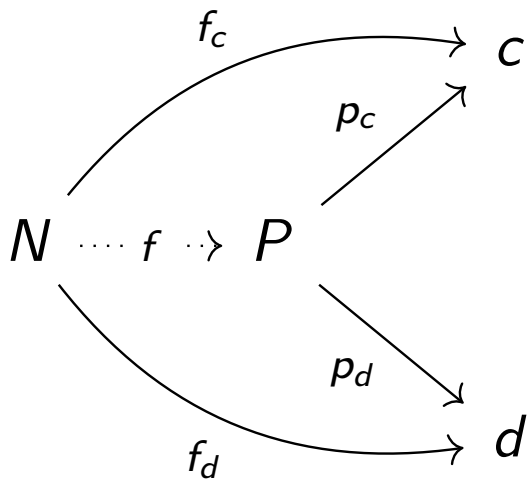
```
-- a -> ()
k :: a -> ()
k = const ()

-- |⊥| -> a
t :: Void -> a
t = absurd
```

In the same way that we have terminal and initial objects, constructions for products and coproducts in category theory follow suit.

Definition (Product)

A **product** in a category $\mathbf{C}$ is an object P together with morphisms $p_c : P \to c$ and $p_d : P \to d$ for c, d in $\mathbf{C}$ such that for any other object N with morphisms $f_c : N \to c$ and $f_d : N \to d$, then each $f_{(-)}$ can be factored through a unique function $f : N \to P$ such that $f_c = p_c f$ and $f_d = p_d f$.

This is diagram gives this data succinctly:



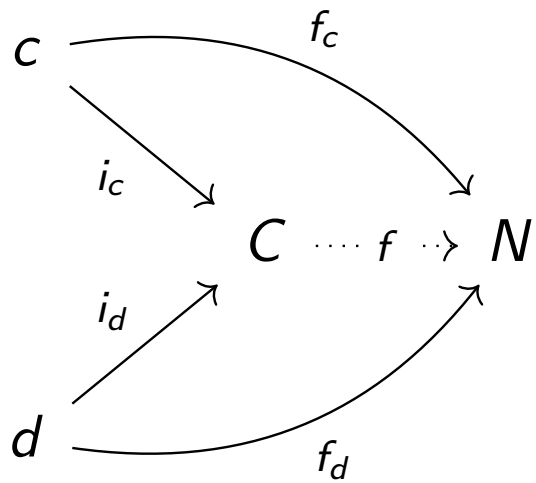
We are all familar with these: in Haskell and Scala, we have the type (a, b). Its defined by having two operators:

```
fst :: (a,b) -> a
fst (a,_) = a

snd :: (a,b) -> b
snd (_,b) = b
```

Definition (Coroduct)

A **coproduct** in a category $\mathbf{C}$ is an object P together with morphisms $i_c : c \to C$ and $i_d : C \to d$ for c, d in $\mathbf{C}$ such that for any other object N with morphisms $f_c : c \to N$ and $f_d : d \to N$, then each $f_{(\_)}$ can be factored through a unique function $f : C \to N$ such that $f_c = fi_c$ and $f_d = fi_d$.

This is diagram gives this data succinctly:



We are all familar with these: in Haskell and Scala, we have the type Either a b. Its defined by having two operators:

```
inl :: a -> Either a b
inl = Left

inr :: b -> Either a b
inr = Right
```

Bringing it all back home