Type Arithmetic and the Yoneda Lemma

Emily Pillmore

September 1, 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.

If you ever want to talk category theory or programming, I'm around to talk, help, or mentor (within reason).

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!

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

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 numeral $\mathbb{1}$.
- the type with no inhabitants, Nothing or undefined will be denoted $\mathbb{0}$.
- Function types will be denoted by $a \to b$ or by the notation b^a.
- Universal quantification is given by forall or $\forall$.
- The notation $|\cdot| : \mathbf{Type} \to \mathbb{N}$ will denote the function taking a type to its number of type inhabitants
- The notation $[\cdot] : \mathbb{N} \to \mathbf{Type}$ will denote the function taking a natural number n and constructing a type with precisely n inhabitants.

- 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.

We also assume our types are sufficiently set-like as to be usefully enumerable.

Definition (Isomorphism)

An *isomorphism* is a pair of functions $to : a \to b$ and $from : b \to a$ such that $to \cdot from = from \cdot to = \mathbf{id}$.

The existence of an isomorphism between types will be denoted $a \sim b$.

There might be some confusion regarding what exactly we mean by "inhabitants", e.g. are $1 + 1$ and 2 different inhabitants of type **Int**? For now we will treat two expressions of type a to be separate inhabitants of a if and only if there is some predicate $f : a \to \textbf{Bool}$ that distinguishes between then.

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, $\mathbb{1}$ is the type with 1 inhabitant, so

```
𝟙 + 𝟙
~ () + ()
~ Either () ()
-- [/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! Note also that this is isomorphic to **Bool**, since **Bool** has precisely 2 inhabitants.

Instead of referencing these types by their names, we'll just represent them as numerals $\mathbb{1}, \mathbb{2}, \mathbb{3}, \ldots$.

How about **Maybe** containing values of type $\mathbb{2}$?

Again, we can apply some natural deduction, nothing that 2 has precisely two inhabitants

```
Maybe  2
~ Nothing + Just  2
~ 1 +  2
~ 3
```

Thus, the number of inhabitants of **Maybe** can be deduced to be $[1 + |a|]$ 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 → 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 of the type $a \to b$. The cardinality of this type behaves like exponentiation in $\mathbb{N}$!

As we will see later, $a \to b$ is a little bit different from coproducts and products in the way it interacts with forall a when we have (parametric) type polymorphism. The above formula can be used only if a and b are free from any universally quantified variables.

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

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 corresponding 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 association of a representative of an equivalence class of **isomorphisms** of a type with 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* $a_1 = a_2 \Leftrightarrow b_1 = b_2$*.*

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, $x = y \Leftrightarrow \forall P.Px = Py$ 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 are the sets with cardinality $a \times b$, $a + b$, 1, 0, and b^a-many inhabitants?

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	?

Some Category Theory

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 $f : x \to y$, $f 1_x = f = 1_y f$.
- 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

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  -- 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.

Example: what do we mean by cardinality?

Recall that I mentioned this was an association of an isomorphism class of types with a finite set.

Enumeration defines a cardinality functor $|\cdot| : \mathbf{Fin}_{\mathbf{Iso}} \to \mathbf{Set}$.

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

As an important example, for any object c of $\mathbf{C}$, we have the **covariant represented functor of** c, $\mathbf{C}(c, -)$, which is a functor $\mathbf{C} \to \mathbf{Set}$.

LIkewise, we have the **contravariant functor represented by** c: $\mathbf{C}(-, c) : \mathbf{C}^{\mathrm{op}} \to \mathbf{Set}$.

We know these well! They're our old friends **Reader** and **Coreader**.

```haskell
newtype Reader t a = Reader (t -> a)
instance Functor (Reader t) where
  -- fmap :: (a -> b) -> Reader t a -> Reader t b
  fmap f (Reader t) = Reader (f . t)

newtype Coreader t a = Coreader (a -> t)
instance Contravariant (Coreader t) where
  contramap f (Coreader k) = Coreader (k . f)
```

To each functor $F : \mathbf{C} \to \mathbf{D}$ is associated data - a family of maps from each object c of $\mathbf{C}$ to objects Fc of $\mathbf{D}$ and likewise between morphisms. But this begs the question - how can we translate the data of one functor into the data of another?

For one, this entails we have two functors $F, G : \mathbf{C} \to \mathbf{D}$ with the same domain and codomain. Second, we need additional data such that to each c of $\mathbf{C}$ and Fc of $\mathbf{D}$, we can translate Fc to Gc in a natural way.

Definition (Natural Transformation)

Let $F, G : \mathbf{C} \to \mathbf{D}$ be functors. A **natural transformation** α consists of the following data: to each c of $\mathbf{C}$, a **component** map $\alpha_c : Fc \Rightarrow Gc$ such that the following square commutes for any $f : c \to d$ of $\mathbf{C}$:

$$\begin{array}{ccc} Fc & \xrightarrow{\alpha_c} & Gc \\ {\scriptstyle Ff}\downarrow & & \downarrow{\scriptstyle Gf} \\ Fd & \xrightarrow[\alpha_d]{} & Gd \end{array}$$

I.e., $\alpha_d \cdot Ff = Gf \cdot \alpha_c$.

In haskell, this is rather simple to implement:

```haskell
type f ~> g = forall a. f a -> g a
-- given f : a -> b, natural transf. t,
-- t . fmap f = fmap f . t
```

A **natural isomorphism** is a natural transformation where each component map is an isomorphism. We will denote this $\alpha : F \cong G$.

One form of category that we will look at later is that of a **functor category**, denoted $[\mathbf{C}, \mathbf{D}]$, whose objects are functors $\mathbf{C} \to \mathbf{D}$, and whose morphisms are natural transformations between these functors.

"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 *Void*

Indeed for every type a, the types $a \to ()$ and $\mathit{Void} \to a$ has precisely one inhabitant .

Namely, we have the following functions:

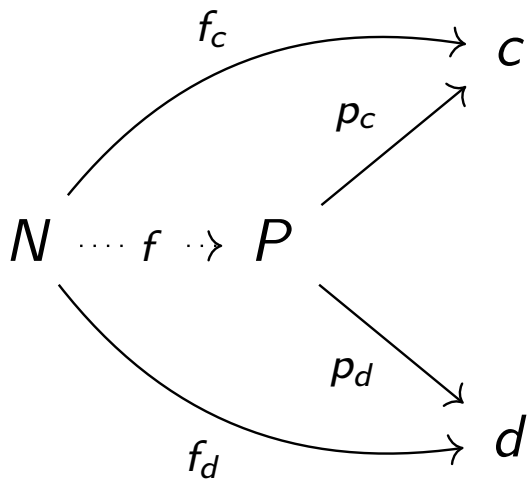
```haskell
-- a -> ()
k :: a -> ()
k = const ()

-- |⊥| -> 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:



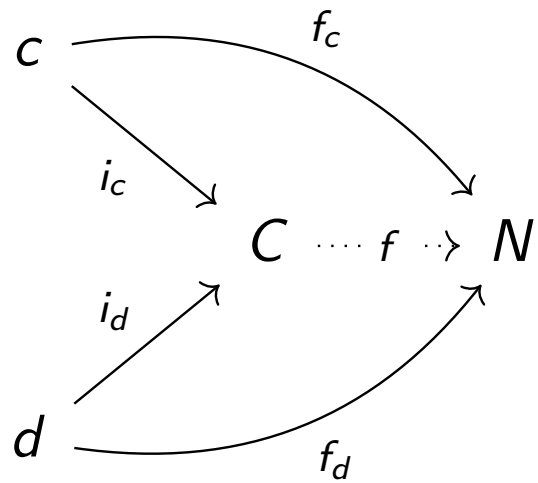
We are all familar with these: in Haskell and Scala, we have the type (a, b). Its defined by having two operators:

```
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
```

The diagrams for products and coproducts define their **universal properties**. In an imprecise sense, this means that these constructions are initial themselves in the appropriate category.

Let's now consider morphisms to and from products and coproducts, beginning with products first.

Consider $f \in \mathbf{C}(a \times b, c)$. What data is necessary to produce c?

f specifies an element $f(a, b)$ of $\mathbf{C}$. Therefore, we require that f have the additional data that when partially applied to b, $f(-, b)$ specifies a function $\phi : a \to c$.

Thus, to specify a function $\mathbf{C}(a \times b, c)$ is to specify an equivalent function $\mathbf{C}(b, c^a)$ - hence, an isomorphism.

We are familiar with this function!

```
curry :: ((a,b) -> c) -> a -> b -> c
curry k a b = k (a,b)

uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry k (a,b) = k a b
```

The same deduction can be applied to coproducts. What data can we glean from $f \in \mathbf{C}(a + b, c)$?

When $a + b$ is introduced, we have either the left or the right hand side.

Therefore, the data equivalent to f consists of a pair of function $a \to c$ and $b \to c$ - one for each side.

In symbols, $\mathbf{C}(a + b, c) \cong \mathbf{C}(a, c) \times \mathbf{C}(b, c)$

```haskell
either
  :: Either a b -> (a -> c) -> (b -> c) -> c
either (Left a) f _ = f a
either (Right b) _ g = g b

pair :: (a -> c, b -> c) -> Either a b -> c
pair (f,g) (Left a) = f a
pair (f,g) (Right b) = g b
```

Similarly, how would you deduce that $(a \times b)^c \cong a^c \times b^c$?

The Yoneda Perspective

We've filled in the rest of our table!

Types	Sets	# of Inhabitants	Categories
a	A	$\lvert A\rvert$	objects of $\mathbf{C}$
(a, b)	$A \times B$	$\lvert A\rvert \times \lvert B\rvert$	$c \times d$
Either a b	$A \sqcup B$	$\lvert A\rvert + \lvert B\rvert$	$c + d$
$a \to b$	set functions	$\lvert B\rvert^{\lvert A\rvert}$	morphisms in $\mathbf{C}$
()	$\{*\}$	1	terminal objects
Void	$\emptyset$	0	initial objects

Let's start from code and build some intuition, seeing if we can derive a category theoretic Leibniz perspective.

What would such a perspective look like?

To remind everyone, Leibniz' rule is $a = b \Leftrightarrow \forall P.Pa = Pb$.

Propositions, as we have seen, correspond with morphisms to or from an object in a category.

Likewise, equality (in this context) is represented as an isomorphism.

Candidate: $a \cong b \Leftrightarrow \forall x.\mathbf{C}(a, x) \cong \mathbf{C}(b, x)$

This is a provable statement.

Definition (Yoneda Embedding (corollary))

a is isomorphic to b if and only if $\mathbf{C}(a, -)$ is naturally isomorphic to $\mathbf{C}(b, -)$

Proof of ($\Leftarrow$).

Suppose $\mathbf{C}(x, a) \cong \mathbf{C}(x, b)$ for all objects x. Taking x to be a and b successively, then, we have bijections:

$$\mathbf{C}(a, a) \quad \cong \quad \mathbf{C}(b, a) \qquad\qquad \mathbf{C}(a, b) \quad \cong \quad \mathbf{C}(b, b)$$

□

Proof of ($\Leftarrow$).

Suppose $\mathbf{C}(x, a) \cong \mathbf{C}(x, b)$ for all objects x. Taking x to be a and b successively, then, we have bijections:

$$\begin{array}{ccc} \mathbf{C}(a,a) & \cong & \mathbf{C}(b,a) \\ \cup & & \\ 1_a & & \end{array} \qquad \begin{array}{ccc} \mathbf{C}(a,b) & \cong & \mathbf{C}(b,b) \\ & & \cup \\ & & 1_b \end{array}$$

□

Proof of ($\Leftarrow$).

Suppose $\mathbf{C}(x, a) \cong \mathbf{C}(x, b)$ for all objects x. Taking x to be a and b successively, then, we use bijections:

$$\begin{array}{ccc} \mathbf{C}(a,a) & \cong & \mathbf{C}(b,a) \\ \cup & & \cup \\ 1_a & \longrightarrow & \psi \end{array} \qquad \begin{array}{ccc} \mathbf{C}(a,b) & \cong & \mathbf{C}(b,b) \\ \cup & & \cup \\ \phi & \longleftarrow & 1_b \end{array}$$

To define functions ϕ and ψ. In fact, these functions are inverses! $\square$

Proof of ($\Rightarrow$).

Functors preserve isomorphisms (this is a simple check). $\square$

In previous slides we mentioned a functor "represented" by an object.

What does this mean?

Indeed, we have a notion of a functor which relates an object to perspectives of that object i.e. morphisms to or from the object to all others.

"Objects exist in relation to all other objects"

"Everything in category theory can be proven with a combination of the Yoneda Lemma and some facts about the category of Sets"

This is the Yoneda perspective. Two objects are isomorphic if and only if the functors represented by them are naturally isomorphic.

Definition (Representable Functor)

A **representable** functor is a functor $F : \mathbf{C} \to \mathbf{Set}$ along with the following data:

- an object c that is said to **represent** F
- a natural isomorphism $\Phi : F \cong \mathbf{C}(c, -)$

This data together is called a **representation** of F.

```haskell
class Representable f c | f -> c where
  tabulate :: f x -> (c -> x)
  index :: (c -> x) -> f x
```

Definition (Representable Yoneda)

Let $F : \mathbf{C} \to \mathbf{Set}$ be a functor where $\mathbf{C}$ is a locally small category (every collection of morphisms $\mathbf{C}(c, d)$ between any two objects forms a set). Then, $Fc \cong [\mathbf{C}, \mathbf{Set}](\mathbf{C}(c, -), F)$. This bijection associates a natural transformation $\alpha : \mathbf{C}(c, -) \Rightarrow F$ with the element $\alpha_c(1_c)$ in Fc.

Furthermore, this correspondence is natural in c and in F.

Now, we won't prove this lemma here (it's a bit long for a short talk like this), but we can use this fact to show the operational intuition behind many statements.

Different Yoneda lemmas

Covariant functor: $(\forall x.(a \to x) \to f\ x) \cong f\ a.$

Contravariant functor: $(\forall x.(x \to a) \to f\ x) \cong f\ a.$

Expressed in Haskell these look like this:

```
covariant :: Functor f =>
             Iso (forall x. (a -> x) -> f x) (f a)
covariant = Iso (\f -> f identity)
                (\fa -> \f -> fmap f fa)
contravariant :: Contravariant f =>
                 Iso (forall x. (x -> a) -> f x) (f a)
contravariant = Iso (\f -> f identity)
                    (\fa -> \f -> contramap f fa)
```

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

from :: f a -> (forall x. (a -> x) -> f x)
from fa = \f -> fmap f fa
```

```haskell
fromTo :: Functor f => (forall x. (a -> x) -> f x) ->
                       (forall x. (a -> x) -> f x)
fromTo = from . to
fromTo = (\fa -> \g -> fmap g fa) .
         (\f -> f id)
fromTo = \f -> \g -> fmap g (f id)
-- Using parametricity
fromTo = \f -> \g -> f (g . id)
-- Function composition
fromTo = \f -> \g -> f g
-- Eta reduction
fromTo = \f -> f
fromTo = id
```

```
toFrom :: Functor f => f a -> f a
toFrom = to . from
toFrom = (\f -> f id) . (\fa -> \g -> fmap g fa)
toFrom = \fa -> (\g -> fmap g fa) id
toFrom = \fa -> fa
toFrom = id
```

Bringing it all back home

How many inhabitants does $\forall a.a \to a$ have?

```
∀ a. a -> a
  ≅ ∀ a. (() -> a) -> a
  -- newtype Id a = a
  ≅ ∀ a. (() -> a) -> Id a
  ≅ Id ()
  ≅ ()
```

Since () has only one inhabitant, we conclude that $\forall a.a \to a$ has only one inhabitant as well.

```
∀ a x. (a, x) -> a
    -- uncurrying
    ≅ ∀ a x. x -> a -> a
    -- () -> x ≅ x
    ≅ ∀ a x. (() -> x) -> a -> a
    -- Yoneda with newtype F a x = a -> a
    ≅ ∀ a. F a ()
    ≅ ∀ a. a -> a
    -- reusing the proof above
    ≅ ()
```

```
∀ a. (a, a) -> a
    -- (a, a) ≅ Bool -> a
    ≅ ∀ a. (Bool -> a) -> a
    -- Yoneda with newtype F x = x
    ≅ Bool
```

```
∀ a b c. (a -> c) -> b -> a -> c
    -- newtype F a b x = b -> a -> x
    ≅ ∀ a b. F a b a
    ≅ ∀ a b. b -> a -> a
    -- reusing one of the above theorems
    ≅ ()
```

```
∀ a b c. (a -> b) -> (b -> c) -> (a -> c)
    -- reorder the arguments
    ≅ ∀ a b c. (b -> c) -> (a -> b) -> a -> c
    -- Yoneda with newtype F a b x = (a -> b) -> a -> x
    ≅ ∀ a b. F a b b
    ≅ ∀ a b. (a -> b) -> a -> b
    -- Yoneda with newtype G a x = a -> x
    ≅ ∀ a. G a a
    ≅ ∀ a. a -> a
    -- reusing the above theorems
    ≅ ()
```

```
∀ a. (a -> 0, a -> 0) -> (a -> 0)
    ≅ ∀ a. a -> (a -> 0, a -> 0) -> 0
    ≅ ∀ a. (1 -> a) -> (a -> 0, a -> 0) -> 0
    ≅ (1 -> 0, 1 -> 0) -> 0
    ≅ (0, 0) -> 0
    ≅ 0 -> 0
    ≅ 1
```

For more exercises, see a joint talk I did with @alexknvl on Isomorphic reasoning, which is the 6 hour version of this talk (slides at cohomolo.gy).