Isomorphic Reasoning: Counting Polymorphic Type Inhabitants

Emily Pillmore, Alexander Konovalov

May 2019

Who are we?

But who are we and what do we do?

Why this workshop?

- Learn you a **good deal of Category Theory**, λ-calculus, and Type Theory.
- Learn powerful techniques to think about types and programs.
- Learn how to rigorously show that $\forall a.a \to a$ has only one inhabitant.
- Learn a different way of thinking about *free*-theorems and properties.
- Learn how to rigorously show that **ap** is the same as **zip**.

Why isomorphic reasoning?

We often get overly hyped about a new functional technique not realizing that it is the same old idea, just with a different flavor. Seeing such connections between ideas will make you a better developer.

Why count type inhabitants?

For compiler writers: Knowing that a type T has only one inhabitant **simple** : T allows us to simplify any complicated **complex** : T and replace it with **simple** : T.

Why count type inhabitants?

For IDE writers:

- If we can find the number of inhabitants, we can (likely) also enumerate them. This allows for type-based auto-completion up to **observational equivalence**.
- If we know that there are no inhabitants, we (likely) have a constructive proof of that, which means we can either auto-complete functions $0 \to a$, or we can outright allow the user to say "this is impossible", like Idris does.

Why count type inhabitants?

Proving that a type P has inhabitants is equivalent to proving the theorem corresponding with P due to Curry-Howard-Lambek correspondence.

Why count type inhabitants?

So what do we mean by counting?

Why count type inhabitants?

When are two things equal?

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

Expressions

If 1729 and $12^3 + 1$ are expressions, they are different.

Expressions

```haskell
data Expr = Lit Int | Add Expr Expr | Pow Expr Expr

first  = Lit 1729
second = Add (Pow (Lit 12) (Lit 3)) (Lit 1)
```

If 1729 and $12^3 + 1$ are integers, they are the same.

Expressions

```haskell
first :: Int
first  = 1729
second :: Int
second = 12^3 + 1
```

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.

$x = y \Leftrightarrow \forall P.Px \leftrightarrow Py$ where P is quantified over all properties of an object.

- We can't see the difference of them as values within a pure functional language.
- We can see the difference between the actual expression trees.
- We don't really care about the latter as since it does not affect the semantics.

But How?

How does this apply to functional programming? category theory?

But How?

Lets look at an example

Examples of enumeration

- Bool $= T \vee F$ has precisely 2 values - true of false.
- tuples
- either
- maybe
- $\rightarrow$

Isomorphisms

- So far we've been enumerating things.
- But enumeration is essentially an isomorphism to a finite set.

Isomorphisms

For instance enumerating the elements of **Bool** gives an bijection (an isomorphism of sets) to a set with 2 elements $\{\top, \bot\}$.

Isomorphisms

An isomorphism is a pair of two functions, $\mathbf{to} : A \to B$ and $\mathbf{from} : B \to A$, such that $\mathbf{to} \cdot \mathbf{from} = id_B$ and $\mathbf{from} \cdot \mathbf{to} = id_A$.

```
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
}
```

Isomorphisms preserve equality

So why are we interested in isomorphisms?

Equality Preservation

If A and B are isomorphic, and $a_1 : A, a_2 : A$ map to $b_1 : B, b_2 : B$ respectively, then $a_1 = a_2 \iff b_1 = b_2$.

We can prove it very easily:

$$
\begin{aligned}
a_1 &= a_2 \\
\text{to } a_1 &= \text{to } a_2 \\
b_1 &= b_2 \\
\text{from } b_1 &= \text{from } b_2 \\
a_1 &= a_2
\end{aligned}
$$

Note that this proof will work regardless of what we mean by $=$, as long as it is preserved by function application: $a = b \Rightarrow f\ a = f\ b$.

But for observational equivalence, equality preservation is almost literally built into the definition!

$$a = b \iff \forall P.(P\ a \leftrightarrow P\ b)$$

Note that this does not hold for **Eq** as found in Haskell, because equality is not preserved for floating point numbers:

```
woops :: Float -> Bool
woops x = (1 / x) > 0
comparison = woops +0.0 == woops -0.0
```

Since equality is preserved by isomorphisms, so is the number of inhabitants:

Inhabitant Count Preservation

If A and B are isomorphic, then they have the same number of inhabitants.

More isomorphism examples

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))
```

```haskell
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)
```

```haskell
currying :: Iso ((a, b) -> c) (a -> b -> c)
currying = Iso (\f -> \a -> \b -> f (a, b))
               (\f -> \case (a, b) -> f a b)

leftUnitF :: Iso (() -> a) a
leftUnitF = Iso (\f -> f ()) (\a -> \_ -> a)

rightUnitF :: Iso (a -> ()) ()
rightUnitF = Iso (\_ -> ()) (\_ -> \_ -> ())
```

Let's take a leap of faith here, and replace (a, b) with $a \times b$, Either a b with $a + b$, $a \to b$ with b^a, () with 1, Void with 0, and Iso a b with $a \cong b$, for example:

```
foo :: Iso (Either Void ((), a)) a
foo :: (0 + (1 × a)) ≅ a
```

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
```

Let's see if our intuition is right. Convince yourself that you can construct these:

```
distribute :: Iso (a, Either b c) (Either (a, b) (a, c))
```

$$\text{distribute} :: (a, b + c) \cong ((a \times b) + (a \times c))$$

```
-- Hint: use identity of indiscernibles
introVoid :: Iso (Void -> a) ()
```

$$\text{introVoid} :: a^0 \cong 1$$

```
-- This will turn out to be a very useful isomorphism
pairing :: Iso (Either a b -> c) (a -> c, b -> c)
```

$$\text{pairing} :: c^{a+b} \cong c^a \times c^b$$

The type algebra

There is a correspondence between categories, types, sets, and sets of natural numbers.

Types	Sets	Naturals	Categories
a	A	$\lvert A\rvert$	$A \in \mathbf{C}$
(a, b)	$A \times B$	$\lvert A\rvert \times \lvert B\rvert$	$A \otimes B$
Either a b	$A \sqcup B$	$\lvert A\rvert + \lvert B\rvert$	$A \oplus B$
$a \to b$	set functions	$\lvert B\rvert^{\lvert A\rvert}$	$f \in \mathbf{C}(A, B)$
()	$\{*\}$	1	terminal objects
Void	$\emptyset$	0	initial objects

Goals

In functional programming we work with a subtle form of arithmetic every day, but it's rare that we actually *understand* what it is we're doing. It turns out to be a very deep process with a very intuitive facade!

Steps

What does it mean to have this correspondence? We will the language of category theory to build the formalism using the following steps:

- Step 0: Preliminary definitions
- Step 1: (De-)categorification
- Step 2: The Yoneda Lemma
- Step 3: Representability
- Step 4: Proofs of examples

And optionally if we have time,

- Step 5: Adjunctions, Limits, Colimits
- Step 6: A Proof Using LAPC, RAPL

Steps

Then, we will count! Lets get started with the preliminaries.

Definitions

Definition (Category)

A **Category** consists of the following data:

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

such that

- each morphism has specified **domain** and **codomain** objects. The notation $f : x \to y$ signifies that the morphism f has domain x and codomain y
- each object has an associated **identity morphism** $1_x : x \to y$
- For any pair of morphisms $f : x \to y, g : y \to z$ there exists a morphism $gf : x \to z$

Definitions

This data is subject to the following axioms:

- For any $f : x \to y$, the composites $1_y f$ and $f 1_x$ are both equal to f
- For any composable triple f, g, h, the composites $h(gf)$ and $(hg)f$ are equal, and we will simply denote them hgf.
- Between any two objects x, y in the category $\mathbf{C}$, we may speak of the collection of morphisms with domain x and codomain y called $Hom_{\mathbf{C}}(x, y)$. Sometimes, we will denote this object $\mathbf{C}(X, Y)$ for ease of use.

This is to say that the law of composition is *associative* and *unital* with the morphisms $1_{(-)}$ serve as two-sided identities

Definitions

i. **Set** has sets as its objects, and functions with specified domain and codomain as morphisms
ii. **Pos** has partially-ordered sets as objects and order-preserving functions as morphisms
iii. **Top** has topological spaces as objects and continuous functions as morphisms
iv. **Mon** has the single point set 1 as its only object, and its morphisms are the elements of the monoid with composition given by the monoid operation.
v. **Hask**... (just kidding)

Definitions

All of these are examples of concrete categories — categories in which the collection of objects and collection of morphisms are both *sets*. However, there are categories with in which one or more of these collections is larger than sets (like the category of categories), so we must introduce some subtlety to the theory in order to avoid category-theoretic versions of Russel's paradox in inopportune places.

Definitions

Lets introduce a few definitions that we'll use sparingly when in need:

Definition

A category $\mathbf{C}$ is...

- **concrete** if its collection of objects and morphisms are both sets.
- **small** if only its collection of morphisms is a set.
- **locally small** if for any two objects x and y, $\mathbf{C}(x, y)$ is a set.

For our purposes, when working with types, we are usually working in a concrete category, if not $\mathbf{C}$ directly.

Definitions

Definition

For every category $\mathbf{C}$ we may speak about its **dual** notion $\mathbf{C}^{op}$, the **opposite category** of $\mathbf{C}$, which consists of the following data:

- the same objects as $\mathbf{C}$
- a morphism f^{op} in $\mathbf{C}^{op}$ for each morphism f in $\mathbf{C}$ with the codomain and domain reversed. i.e. when $f : x \to y$, $f^{op} : y \to x$.

Definitions

The data described in the previous slide defines a category in relation to itself. The process of "turning around the arrows" or "swapping domains and codomains" exhibits a syntactic self-duality for every category, retaining the precisely the same information as $\mathbf{C}$. In this way, we have a unique perspective in Category Theory: For every theorem proven in general for a category, we immediately prove its dual, since the opposite category is a valid category in its own right. More generally, if one proves theorems "for all categories $\mathbf{C_1}, \mathbf{C_2}, \ldots, \mathbf{C_n}$", then this leads to 2^n dual theorems. In practice though, these dual theorems do not differ meaningfully from the original.

Definitions

We will now introduce the types of morphisms you will encounter in the wild. Much like the injective/bijective/surjective functions on sets that we are used to, we have abstract version notions of these concepts.

Definitions

Definition (Isomorphism)

An **isomorphism** in a category is a morphism $f : x \to y$ for which there exists a morphism $g : y \to x$ such that $gf = 1_x$ and $fg = 1_y$ (g is a two-sided inverse). The objects x and y are said to be **isomorphic**, and we denote this $x \cong y$.

Definitions

Definition (Functor)

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

- An object $Fc \in \mathbf{D}$ for each object $c \in \mathbf{C}$
- A morphism $Ff : Fc \to Fc'$ in $\mathbf{D}$ for each morphism $f \in \mathbf{C}$

Additionally, the following must be satisfied:

- For any composable pair f, g in $\mathbf{C}$, $Fg \cdot Ff = F(g \cdot f)$
- For each object $c \in \mathbf{C}$, $F(1_c) = 1_{Fc}$

Definitions

So far, we have defined a functor which acts uniformly on arrows. However, there exist functors which have a tendency to take arrows in the source category and flip the direction of the arrows such that they point the opposite way in the target category i.e. for any such functor $F : \mathbf{C} \to \mathbf{D}$ and any objects c, c' and morphism $f : c \to c'$ in $\mathbf{C}$, we have $Ff : Fc' \to Fc$. Such functors are called **contravariant** functors, in contrast to their siblings which are called **covariant**.

Definitions

Instead of differentiating each functor by variance, note that every contravariant functor is a covariant functor $F : \mathbf{C}^{op} \to \mathbf{D}$. Instead of juggling variance, we will simply be sure to specify the variance of a functor by whether or not it maps from an opposite category.

Definitions

- The power set functor $P : \mathbf{Set} \to \mathbf{Set}$ that sends a set A to its power set $PA = \{U : U \subset A\}$ and a function $f : A \to B$ to the direct image function $f_* : PA \to PB$ which sends $U \subset A$ to $f_*(U) \subset B$
- The contravariant power set functor $P : \mathbf{Set}^{op} \to \mathbf{Set}$ sends a set A to its power set PA, and every function $f : A \to B$ to the inverse-image function $f^{-1} : PB \to PA$ such that $V \subset B \mapsto f^{-1}(V) \subset A$.

Definitions (cont'd)

- For any $c \in \mathbf{C}$ the covariant Hom-functor $\mathbf{C}(c, -) : \mathbf{C} \to \mathbf{C}$ taking objects in $\mathbf{C}$ to their corresponding hom-sets in **Set**, and morphisms to post-composition.
- For any $c \in \mathbf{C}^{op}$ the contravariant Hom-functor $\mathbf{C}(-, c) : \mathbf{C}^{op} \to \mathbf{C}$ taking objects in $\mathbf{C}$ to their corresponding hom-sets in **Set** and morphisms to pre-composition.

Definitions (cont'd)

Definition (Initial and Terminal objects)

An object $c \in \mathbf{C}$ is called **terminal**, if for all other objects in $c' \in \mathbf{C}$, there exists a single unique (up to isomorphism) morphism $c' \to c$. Dually, an object is called **initial** if there exists a single unique (up to isomorphism) morphism from c to any other $c' \in \mathbf{C}$.

Definitions (cont'd)

- In **Set**, the singleton set $\{*\}$ is terminal and the empty set $\emptyset$ is initial.
- In the categories **Grp** of groups and group homomorphisms and **Mon** of monoids and monoid homomorphisms, the trivial group/monoid 1 is both initial and terminal
-

Definition (Natural Transformation)

A **natural transformation** between two functors $F, G : \mathbf{C} \to \mathbf{D}$ consists of the following data:

- To each $c \in C$, a component morphism $\alpha_c : Fc \to Gc$ exists such that the following diagram commutes or any $c, c' \in \mathbf{C}$ and $f : c \to c'$:

$$\begin{tikzcd} Fc \arrow[r, "\alpha_c"] \arrow[d, "Ff"] & Gc \arrow[d, "Gf"] \\ Fc' \arrow[r, "\alpha_{c'}"] & Gc' \end{tikzcd}$$

There is such a thing as a **natural isomorphism** as well. How might you define one?

Definition (Natural Transformation)

A **natural transformation** between two functors $F, G : \mathbf{C} \to \mathbf{D}$ consists of the following data:

- To each $c \in C$, a component morphism $\alpha_c : Fc \to Gc$ exists such that the following diagram commutes or any $c, c' \in \mathbf{C}$ and $f : c \to c'$:

$$\begin{tikzcd} Fc \arrow[r, "\alpha_c"] \arrow[d, "Ff"] & Gc \arrow[d, "Gf"] \\ Fc' \arrow[r, "\alpha_{c'}"] & Gc' \end{tikzcd}$$

Definition (natural isomorphism)

A **natural isomorphism** is a natural transformation where each component is an isomorphism. We'll denote them as $\alpha : F \cong G$.

Definitions

In effect, this means that the following naturality condition holds for all $c, c' \in \mathbf{C}$ for a given transformation $\alpha : F \Rightarrow G$:

$$\alpha_{c'} F = G \alpha_c$$

Definitions

This is called a *naturality* condition. To say that α is *natural* in c is to say there exists a component morphism α_c providing a relationship between Fc and Gc.

Definitions

In other words, if you think of functors as containers of values, natural transformations modify the shape of the container, but does not modify its contents. In some sense, the components of a natural transformation are orthogonal to functor maps. If you imagine $F \Rightarrow F$, then the components of such a transformation correspond with permutations of F.

Definitions

Natural transformations are ubiquitous. In a sense, they form "analogies between relationships". If one takes the perspective that functors define an intimate relationship between structures, then natural transformations provide a way of relating those relationships.

Definitions

Examples of natural transformations appear everywhere in the wild. Here are a few:

- There is a natural transformation $\eta_A : 1_{\mathbf{Set}} \Rightarrow P$ from the identity to the covariant power set functor whose components $\eta_A : A \to PA$ are functions that carry $a \in A$ to $a \in PA$.
- The open and closed subset functors are naturally isomorphic when regarded as functors $\mathcal{O}, \mathcal{C} : \mathbf{Top}^{op} \to \mathbf{Set}$. The components are defined by taking an open subset of X to its complement, which is closed. Thus, the "naturality" condition asserts that forming complements commutes with taking preimages (i.e. $f^{-1}(V)^c \cong f^{-1}(V^c)$).

Categorification

What is **(de-)categorification**?

Categorification

A process by which set-theoretic concepts are expressed in terms of category theory, or concepts in category theory to higher category theory.

Categorification

Lets begin by looking at the relationship between sets, categories, and arithmetic in the natural numbers.

Categorification

Let's revisit our table.

Types	Sets	Naturals	Categories
a	A	$\lvert A\rvert$	$A \in \mathbf{C}$
(a, b)	$A \times B$	$\lvert A\rvert \times \lvert B\rvert$	$A \otimes B$
Either a b	$A \sqcup B$	$\lvert A\rvert + \lvert B\rvert$	$A \oplus B$
$a \to b$	set functions	$\lvert B\rvert^{\lvert A\rvert}$	$f \in \mathbf{C}(A, B)$
()	$\{*\}$	1	terminal objects
Void	$\emptyset$	0	initial objects

Categorification

What is a natural number?

Categorification

One way to represent the natural numbers is as cardinalities of finite sets.

$$a :\equiv |A| \qquad b :\equiv |B| \qquad c :\equiv |C|$$

Categorification

What is true when $a = b$?

Categorification

If we use the representation that natural numbers correspond with the cardinality of finite sets, then there is a correspondence between equality of natural numbers, and equivalence classes of isomorphisms of finite sets.

Categorification

In fact, isomorphism classes of finite sets form a subcategory $\mathbf{Fin_{Iso}}$ of **Set** with objects as isomorphism classes and morphisms as isomorphisms.

Categorification

The taking of cardinalities constructs a functor $|-|\colon \mathbf{Fin}_{\mathbf{Iso}} \to \mathbf{Set}$ taking isomorphism classes to a set whose elements are in bijection with a subset of the natural numbers.

Categorification

It's natural now, to also ask about multiplication and addition of natural numbers.

Categorification

Lets apply the same sort of categorification.

Categorification

What sets have size $a \times b$, $b + c$ and c^d?

Categorification

By counting,

- $|A \times B| \cong a \times b$
- $|B \sqcup C| \cong b + c$.
- $|C^D| \cong c^d$

Categorification

So what is the nature of (de-)categorification?

Categorification

It establishes a correspondence between objects in a category with objects in **Set** (or vice versa).

Categorification

Summary:

$$A \times B \Rightarrow a \times b \leftrightsquigarrow |A \times B| \cong a \times b$$
$$B \sqcup C \Rightarrow b + c \leftrightsquigarrow |B \sqcup C| \cong b + c$$
$$C^D \Rightarrow c^d \leftrightsquigarrow |C|^{|D|} \cong c^d$$

Categorification

So what does this correspond to?

$$a \times (b + c) = (a \times b) + (a \times c)$$

Categorification

So what does correspond to?

$$a \times (b + c) = (a \times b) + (a \times c)$$

Answer:

$$A \times (B + C) \cong (A \times B) \sqcup (A \times C)$$

Yoneda

A perspective:

An objects is fully determined by its relationship to other objects.

Yoneda

Translation:

An object is fully determined by the morphisms and to and from the object.

Yoneda

Q: But why is this important? Why do we care?

To quote from a friend:

"Most statements in elementary category theory can be proved using some variant statement of yoneda together with information about the category of sets"

The Yoneda Lemma

A and B are isomorphic if and only if

- For all X, $\mathbf{C}(A, X) \cong \mathbf{C}(B, X)$
- the isomorphisms $\mathbf{C}(A, X) \cong \mathbf{C}(B, X)$ are *natural* with respect to any function $f : X \to Y$.

Yoneda

Q: But what does this mean?

Yoneda

The Yoneda Lemma

A and B are isomorphic if and only if the functors $\mathbf{C}(A, -)$ and $\mathbf{C}(B, -)$ are naturally isomorphic.

Yoneda

The Yoneda Lemma

A and B are isomorphic if and only if the functors $\mathbf{C}(A, -)$ and $\mathbf{C}(B, -)$ are naturally isomorphic.

Proof ($\Leftarrow$).

Let $\mathbf{C}(A, X) \cong \mathbf{C}(B, X)$ for every X. Let $X = A$, and $X = B$. Then, we prove this using the following bijections and a diagram chase (see board):

$$\mathbf{C}(A, A) \cong \mathbf{C}(B, A) \cong \mathbf{C}(B, B)$$

$\square$

The Yoneda Lemma

A and B are isomorphic if and only if the sets $\mathbf{C}(A, X)$ and $\mathbf{C}(B, X)$ are naturally isomorphic.

Proof ($\Rightarrow$).

Note that $\mathbf{C}(A, -)$ is a functor $\mathbf{C} \to \mathbf{C}$. Functors preserve isomorphisms. Hence, $\mathbf{C}(A, X) \cong \mathbf{C}(B, X)$ for all X.

$\square$

Representable Yoneda

Consider the previous statement, that
$A \cong B \iff \mathbf{C}(A, X) \cong \mathbf{C}(B, X)$ for all X. Lets consider just $\mathbf{C}(A, X)$.

Representable Yoneda

In some sense the functor $\mathbf{C}(A, -)$ is determined by both an object $A \in \mathbf{C}$, as well as the functor $\mathbf{C}(A, -)$.

Representable Yoneda

Suppose I wanted to generalize the previous Yoneda lemma to any functor from $F \in [\mathbf{C}, \mathbf{Set}]$. What would be needed?

Representable Yoneda

We would need a the same representing object, of course, but what properties would we need F to have?

Representable Yoneda

We have already seen a mechanism that can transform one functor into another: a natural transformation! In fact, we need something slightly stronger than a transformation - we must require that this be a natural isomorphism!

Representable Yoneda

Definition (Representable Functors)

A functor $F \in [\mathbf{C}, \mathbf{Set}]$ is called **representable** if it is naturally isomorphic to $\mathbf{C}(A, -)$ for some object $A \in \mathbf{C}$. A **representation** of F is a pair (A, Φ) of its **representing object**, A, and a natural isomorphism $\Phi : \mathbf{C}(A, -) \cong F$.

Representable Yoneda

Indeed, this gives us a new, and characterization of the Yoneda lemma in terms of representable functors.

Representable Yoneda

Representability says that $FX \cong_{\Phi_X} \mathbf{C}(A, X)$, but what we truly want to prove is that when $A \cong B$, we have $FA \cong FB$.

Representable Yoneda

In fact, we want that if $A \cong B$, then $\mathbf{C}(A, -) \cong FA$ and $\mathbf{C}(B, -) \cong FB$.

Representable Yoneda

In philosophical terms, this would allow us to say that whenever a representation (A, Φ) of F that is isomorphic to some other representation (B, Ψ), then $FA \cong FB$. In order to prove this, we can show that the set of all transformations of $\mathbf{C}(A, -) \Rightarrow F$ is in correspondence with FA.

Yoneda Lemma

Definition (Yoneda Lemma)

Let $F \in [\mathbf{C}, \mathbf{Set}]$ be a representable functor, and let $A \in C$. The set of natural transformations $[\mathbf{C}, \mathbf{Set}](\mathbf{C}(A, -), F)$ is in bijection with FA for all $A \in \mathbf{C}$. This bijection associates a morphism (natural transformation) $\alpha : \mathbf{C}(A, -) \Rightarrow F$ to the identity $\alpha(1_A) \in FA$

Representability

It turns out that our work has been about the correspondence between representable functors and natural isomorphisms all along!

Now that we are familiar with category-theoretic definition of Yoneda Lemma, we will soon see how we can apply it to counting. But first,

Functors preserve isomorphisms

If $a \cong b$, then $f\ a \cong f\ b$ for covariant, contravariant, and invariant functors.

Proof that $a \cong b \Rightarrow f\ a \cong f\ b$ for covariant functors:

```
toF :: Functor f => f a -> f b
toF fa = fmap to fa
fromF :: Functor f => f b -> f a
fromF fb = fmap from fb
fromTo :: Functor f => f a -> f a
fromTo = from . to
fromTo = (\x -> fmap from x) . (\x -> fmap to x)
fromTo = \x -> fmap from (fmap to x)
fromTo = \x -> fmap (from . to) x
fromTo = identity
```

Functors preserve isomorphisms

If $a \cong b$, then $f\ a \cong f\ b$ for covariant, contravariant, and invariant functors.

Here are some simple exercises to drive this home:

1. Prove that $(\text{Bool}, ()) \to \text{Either}\ (\text{Either}\ ()\ ())\ \text{Void}$ is isomorphic to $\text{Bool} \to \text{Bool}$, and then show that this is isomorphic to a type with four inhabitants.
2. Prove that $(a \to a \to z) \to z$ is isomorphic to $((Bool \to a) \to z) \to z$.

Finally, we can do Yoneda lemma in Haskell and λ-calculus:

Different Yoneda lemmas

$$\begin{aligned} \text{Covariant functor:} \quad & (\forall x.(a \to x) \to f\ x) \cong f\ a. \\ \text{Contravariant functor:} \quad & (\forall x.(x \to a) \to f\ x) \cong f\ a. \end{aligned}$$

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)
```

Let's look at the individual components of the first isomorphism:

```
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
```

```haskell
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
```

Let's apply Yoneda lemma ($\forall x.(a \to x) \to f\ x \cong f\ a$) to a simple example:

```
∀ 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. (a, a) -> (a, a)
    -- (a, a) ≅ Bool -> (a, a)
    ≅ ∀ a. (Bool -> a) -> (a, a)
    -- Yoneda with newtype F x = (x, x)
    ≅ (Bool, Bool)
    -- Fin 4 is a type with 4 inhabitants.
    ≅ Fin 4
```

```
∀ 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 b. (a -> b) -> b -> a -> b
    ≅ ∀ a b. (a -> b) -> (1 -> b) -> a -> b
    -- using pairing
    ≅ ∀ a b. ((a + 1) -> b) -> a -> b
    -- Yoneda with newtype F a x = a -> x
    ≅ ∀ a. F a (a + 1)
    ≅ ∀ a. a -> (a + 1)
    -- a ≅ 1 -> a
    ≅ ∀ a. (1 -> a) -> (a + 1)
    -- Yoneda with newtype G x = x + 1
    ≅ G 1
    ≅ 1 + 1
    ≅ 2
```

```
∀ a. (0 -> a) -> a -> 0
    ≅ ∀ a. (0 -> a) -> (1 -> a) -> 0
    ≅ ∀ a. (1 -> a) -> 0
    -- Yoneda with newtype F x = 0
    ≅ F 1
    ≅ 0
```

```
∀ a b. (a -> 0, b) -> 2 × b
    ≅ ∀ a b. b -> (a -> 0) -> 2 × b
    ≅ ∀ a b. (1 -> b) -> (a -> 0) -> 2 × b
    -- Yoneda with newtype F a x = (a -> 0) -> 2 |×| x
    ≅ ∀ a b. F a 1
    ≅ ∀ a b. (a -> 0) -> 2 × 1
    -- Add (0 -> a) ≅ 1
    ≅ ∀ a. (0 -> a) -> (a -> 0) -> 2
    -- Yoneda with newtype G x = (x -> 0) -> 2
    ≅ G 0
    ≅ (0 -> 0) -> 2
    ≅ 1 -> 2
    ≅ 2
```

```
∀ a. (a -> 0) -> 0
    ≅ ∀ a. (0 -> a) -> (a -> 0) -> 0
    -- Yoneda with newtype G x = (x -> 0) -> 0
    ≅ (0 -> 0) -> 0
    ≅ 1 -> 0
    ≅ 0
```

```
∀ a. (a -> 0) -> a -> 0
    ≅ ∀ a. a -> (a -> 0) -> 0
    -- Yoneda with newtype G x = (x -> 0) -> 0
    ≅ (1 -> 0) -> 0
    ≅ 0 -> 0
    ≅ 0
```

```
∀ 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
```

```
∀ a b. (a -> b) -> Maybe a -> Maybe b
    -- Yoneda with newtype F a x = Maybe a -> Maybe x
    ≅ ∀ a. Maybe a -> Maybe a
    -- Maybe's definition
    ≅ ∀ a. (1 + a) -> Maybe a
    -- Pairing
    ≅ ∀ a. (1 -> Maybe a) × (a -> Maybe a)
```

We've hit an impasse. We can't apply Yoneda Lemma directly, since *a* occurs both in positive and negative positions.

We need:

Distributing $\forall$ over $\times$.

$$\forall x.fx \times gx \cong (\forall x.fx) \times (\forall x.gx)$$

and (*non-trivial*)

Distributing $\forall$ over $+$.

$$\forall x.fx + gx \cong (\forall x.fx) + (\forall x.gx)$$

Proof for $\forall x.fx \times gx \cong (\forall x.fx) \times (\forall x.gx)$:

```
-- Dependent types are for clarity...
to :: ((x :: *) -> (f x, g x))
    -> ((x :: *) -> f x, (x :: *) -> g x)
to pair = (\t -> fst (pair t), \t -> snd (pair t))

from :: ((x :: *) -> f x, (x :: *) -> g x)
        -> ((x :: *) -> (f x, g x))
from pair = (\t -> (fst pair t, snd pair t))
```

Proof for $\forall x.fx + gx \cong (\forall x.fx) + (\forall x.gx)$:

```
to :: ((x :: *) -> Either (f x) (g x))
    -> Either ((x :: *) -> f x) ((x :: *) -> g x)
-- Using parametricity here:
to e = case (e ()) of
    Left  fx -> Left  (\t -> (unsafeCoerce fx :: f t))
    Right gx -> Right (\t -> (unsafeCoerce gx :: f t))

from :: Either ((x :: *) -> f x) ((x :: *) -> g x)
        -> ((x :: *) -> Either (f x) (g x))
from (Left fx)  = \t -> Left  (fx t)
from (Right gx) = \t -> Right (gx t)
```

Let's back to Maybe a → Maybe a.

```
≅ ∀ a. (1 -> Maybe a) × (a -> Maybe a)
-- Distributing ∀ over ×
≅ (∀ a. Maybe a) × (∀ a. a -> Maybe a)
-- Maybe's definition
≅ (∀ a. (1 + a)) × (∀ a. a -> Maybe a)
-- 0 -> a introduction (as a unit)
≅ (∀ a. (0 -> a) -> (1 + a)) × (∀ a. a -> Maybe a)
-- Yoneda lemma with newtype F x = 1 + x
≅ (1 + 0) × (∀ a. a -> Maybe a)
-- Simplifying via x + 0 = x and 1 × x = x
≅ ∀ a. a -> Maybe a
-- unit introduction and Maybe's definition
≅ ∀ a. (1 -> a) -> (1 + a)
-- Yoneda lemma with f x = 1 + x
≅ 2
```

Exercises:

1. Explicitly list the two inhabitants of $\forall\ a\ b.(a \to b) \to \text{Maybe}\ a \to \text{Maybe}\ b$.
2. Prove that $\forall\ z.(a \to a \to z) \to z$ is isomorphic to (a, a).
3. Prove: If f is a covariant functor, then $\forall x.f\ x \cong f\ 0$.
4. Prove: If f is a contravariant functor, then $\forall x.f\ x \cong f\ 1$.
5. Prove: If f is a phantom functor, then $\forall x.f\ x \cong f\ 1 \cong f\ 0$.

Exercises:

1. Explicitly list the two inhabitants of $\forall\ a\ b.(a \to b) \to \text{Maybe}\ a \to \text{Maybe}\ b$.
2. Prove that $\forall\ z.(a \to a \to z) \to z$ is isomorphic to (a, a).
3. Prove: If f is a covariant functor, then $\forall x.f\ x \cong f\ 0$.
4. Prove: If f is a contravariant functor, then $\forall x.f\ x \cong f\ 1$.
5. Prove: If f is a phantom functor, then $\forall x.f\ x \cong f\ 1 \cong f\ 0$.

Representable functors

Yoneda Lemma tells us that $(\forall x.(a \to x) \to g\ x) \cong g\ a$, but what if instead of $a \to x$ we have some arbitrary $f\ x$ - a natural transformation between functors $\forall x.f\ x \to g\ x$.

If $f\ x \cong k \to x$ for some k, then we get back to the good old Yoneda. Such functors are called *representable*.

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

Representable functors

That's a nice shortcut, but it is pretty limiting. There are only so many representable functors (exercise: list at least 3).

What if $f = \text{Maybe}$, or say, more generally, $f\ x \cong f_1\ x + f_2\ x$, where $f_1\ x \cong k_1 \to x$ and $f_2\ x \cong k_2 \to x$ (in the case of Maybe, $k_1 \cong 0$ and $k_2 \cong 1$):

```
∀ x. f x -> g x
    ≅ ∀ x. (f1 x + f2 x) -> g x
    -- Pairing
    ≅ ∀ x. (f1 x -> g x) × (f2 x -> g x)
    -- Distributing ∀ over ×.
    ≅ (∀ x. (k1 -> x) -> g x) × (∀ x. (k2 -> x) -> g x)
    -- Applying Yoneda Lemma twice
    ≅ (g k1) * (g k2)
```

In the case of $\forall x.\text{Maybe } x \to \text{Maybe } x$, we get:

```
∀ x. Maybe x -> Maybe x
    ≅ Maybe 0 × Maybe 1
    ≅ 1 × 2
    ≅ 2
```

Generalizing

```
∀ x. f x -> g x
    ≅ ∀ x. (Σ_i f_i x) -> g x
    -- Pairing
    ≅ ∀ x. Π_i f_i x -> g x
    -- Distributing ∀ over Π.
    ≅ Π_i (∀ x. (k_i -> x) -> g x)
    -- Applying Yoneda Lemma a lot.
    ≅ Π_i g k_i
```

Functors with this shape, $fx \cong \Sigma_i k_i \to x$, are called *containers*. Almost every type constructor with a functor instance is a container:

```
Maybe x          ≅ (0 -> x) + (1 -> x)
(Bool, x)        ≅ (1 -> x) + (1 -> x)
Const Bool x     ≅ (0 -> x) + (0 -> x)
Either Bool x ≅ (0 -> x) + (0 -> x) + (1 -> x)
[x] ≅ (0 -> x) + (1 -> x) + (2 -> x) + ... ≅ Σ_i x^i
```

The only exceptions I know of is $(x \to 0) \to 0$ and similar constructions.

Take a closer look at:

`[x] ≅ (0 -> x) + (1 -> x) + (2 -> x) + ...` $\cong \Sigma_i \; x^i$

What does this remind you of?

Consider some function f that has the same signature as *map* on lists:

```
∀ a b. (a -> b) -> [a] -> [b]
    -- Yoneda with f x = [a] -> [x]
    ≅ ∀ a. [a] -> [a]
    -- using the container formula
    ≅ ∀ a. (Σ_i a^i) -> [a]
    ≅ Π_i [i]
    -- expressing it using dependent types and renaming i
    ≅ (n : Nat) -> [Fin n]
```

```
∀ a b. (a -> b) -> [a] -> [b]
    ≅ (n : Nat) -> [Fin n]
```

Compare that with the free theorem for $m : \forall\ a\ b.(a \to b) \to [a] \to [b]$:

m f = m id . map f = map f . m id

Similarly if take a look at filter's signature,

```
∀ a. (a -> Bool) -> [a] -> [a]
    ≅ ∀ a. [a] -> (a -> Bool) -> [a]
    -- using the container formula
    ≅ ∀ a. (Σ_i a^i) -> (a -> Bool) -> [a]
    ≅ Π_i (i -> Bool) -> [i]
    -- expressing it using dependent types
    -- and renaming i to n
    ≅ (n : Nat) -> Set (Fin n) -> [Fin n]
```

Deriving **ap** through **zip**

Let's prove that any function that satisfies **zip**'s signature is isomorphic to some function that satisfies **ap**'s signature assuming that F is a functor.

```
∀ a b. (F a, F b) -> F (a, b)
    -- Reverse Yoneda,
    -- F (a, b) ≅ ∀ z. (a -> b -> z) -> F z
    ≅ ∀ a b z. (F a, F b) -> (a -> b -> z) -> F z
    ≅ ∀ a b z. F a -> F b -> (a -> b -> z) -> F z
    ≅ ∀ a b z. (a -> b -> z) -> F a -> F b -> F z
    ≅ ∀ a b z. (a -> (b -> z)) -> (F a -> F b -> F z)
    -- Contravariant Yoneda.
    ≅ ∀ b z. F (b -> z) -> F b -> F z
    -- Renaming.
    ≅ ∀ a b. F (a -> b) -> F a -> F b
```

aplus and *mplus*

```
∀ a b. (F a, F b) -> F (a + b)
    -- Reverse Yoneda,
    -- F (a + b) ≅ ∀ z. (a + b -> z) -> F z
    ≅ ∀ a b z. (F a, F b) -> (a + b -> z) -> F z
    ≅ ∀ a b z. F a -> F b -> (a -> z) -> (b -> z) -> F z
    ≅ ∀ a b z. (a -> z) -> F a -> F b -> (b -> z) -> F z
    -- Contravariant Yoneda.
    ≅ ∀ b z. F z -> F b -> (b -> z) -> F z
    ≅ ∀ b z. (b -> z) -> F b -> F z -> F z
    -- Contravariant Yoneda.
    ≅ ∀ z. F z -> F z -> F z
```

What is not free about **fmap**?

```
∀ a b. (a -> b) -> F a -> F b
    ≅ ∀ z. F z -> F z
```

join and **bind**

```
∀ a. F (F a) -> F a
    ≅ ∀ a z. (z -> F a) -> F z -> F a
```

Automating counting

First of all, let's take a look at the final result - `http://alexknvl.com:8080/index.html`.

One important thing to understand is that we are not going to be able to solve this problem exhaustively (at least in this workshop). Like integration in calculus.

De Brujin representation of type bindings.

```
data Type = Fin  Integer
          | Prod Type Type
          | Sum  Type Type
          | Arr  Type Type
          | Forall Type
          | Use Int
```

```haskell
data Rule = Rule {
  runRule :: Type -> [Type]
}
...
small :: Type -> Either [Type] Integer
small (Fin x) = Right x
small e = Left $ runRule rule e
```

```haskell
solve :: (Hashable a, Eq a)
      => Int -> a
      -> (a -> Either [a] b)
      -> Either (HS.HashSet a) b
solve max seed small = go max [seed] HS.empty where
  go k l v | k <= 0 = Left v
  go _ [] v         = Left v
  go s (h : t) visited =
    if HS.member h visited then go (s - 1) t visited
    else case small h of
      Right b -> Right b
      Left more -> go (s - 1) (t ++ more)
                      (HS.insert h visited)
```

Double-Yoneda and HKTs

Double Yoneda

$F\ A \cong \forall\ f.(\forall\ x.F\ x \to f\ x) \to f\ A$ with no restrictions on F or f!

```
∀ f. f A -> f B
    -- newtype G x = (x = A)
    -- f A ≅ ∀ x. G x -> f x
    ≅ ∀ f. (∀ x. G x -> f x) -> f B
    -- Double-Yoneda
    ≅ G B
    ≅ A = B
```

Yoneda (Propositional Equality)

Yoneda (Propositional Equality)

$f\ a \cong \forall\ x.(a = x) \to f\ x$ with no restrictions on f!

```
∀ f z. f z -> f z
    -- newtype F z x = (x = z)
    ≅ ∀ z f. (1 -> f z) -> f z
    ≅ ∀ z f. (∀ x. F z x -> f x) -> f z
    ≅ ∀ z. F z z
    ≅ ∀ z. z = z
    ≅ 1
```

Recursive types

What to do about $\forall a.(a \to a) \to (a \to a)$?

```
∀ a. (a -> a) -> a -> a
    -- (a -> a, a) ≅ (1 + a) -> a
    ≅ ∀ a. ((1 + a) -> a) -> a
    -- define f x = 1 + x
    ≅ ∀ a. (f a -> a) -> a
    ≅ μ x. f x
    ≅ μ x. 1 + x
```

```
∀ a b. (a -> b -> a) -> a -> b -> a
    -- move parameters around
    ≅ ∀ a b. b -> (a -> b -> a) -> a -> a
    ≅ ∀ a b. b -> ((a × b + 1) -> a) -> a
    ≅ ∀ b. b -> (μ x. x × b + 1)
    -- Yoneda
    ≅ μ x. x × () + 1
    ≅ μ x. x + 1

∀ a. (a -> a) -> a
    -- newtype F x = x
    ≅ μ x. x
    ≅ 0
```

Checking that $\mu\ x.\ f\ x$ is inhabited

Is there a mechanical way to check if $\mu\ x.\ f\ x$ is inhabited? Note that if $\mu\ x.\ f\ x$ is inhabited, $\mu\ x.\ f\ x > 0$, then

```
μ x. f x > 0
∀ x. (f x -> x) -> x > 0
-- substituting x = 0
(f 0 -> 0) -> 0 > 0
-- double-negation translation preserves > 0
f 0 > 0
```

and we conclude that $\mu\ x.\ f\ x > 0$ implies $f\ 0 > 0$.

Checking that $\mu\ x.\ f\ x$ is inhabited

On the other hand,

```
f 0 > 0
-- assuming f is a covariant functor,
-- we can map it with 0 -> (μ x. f x)
f (μ x. f x) > 0
-- and we conclude
μ x. f x > 0
```

Hence, $\mu\ x.\ f\ x$ is inhabited if and only if $f\ 0$ is inhabited (assuming f is a functor).

Universal Properties

We have learned a great deal about yoneda, and we have shown some surprising results that begin to touch upon how it can be used to "count" type inhabitants.

Universal Properties

So far, we have shown that currying, pairing, and exponentiation have an interesting arithmetic.

Universal Properties

How do we realize these properties categorically?

Universal Properties

In a sense, these properties are *universal* in that they are very "natural" actions to take - they manifest properties of maps to and from products, coproducts, and exponents.

Universal Properties

Because it's often germane to our studies to look at the hom-sets of maps to and from these objects in order to learn more about them, lets take a look at products and coproducts

Universal Properties

Suppose we wanted to talk about $\mathbf{C}(A \times B, X)$.

Universal Properties

What does a function $A \times B \to X$ look like?

Universal Properties

Lets start with the categorical definition of a Product.

Universal Properties

Definition

Product Let $\mathbf{C}$ be a category, and let $X, Y \in \mathbf{C}$. A product P is an object of $\mathbf{C}$ together with functions $\pi_X : P \to X$ and $\pi_Y : P \to Y$ such that the following holds:

- For any other $N \in \mathbf{C}$ and morphisms $f_X : N \to X$ and $f_Y : N \to Y$, we have that a unique, induced morphism $f : N \to P$ such that $f_X = \pi_X f$ and $f_Y = \pi_Y f$.

P is usually denoted $X \times Y$.

More succinctly, the object $X \times Y$ in the following diagram commutes for any given N and morphisms f_Y, f_X:

$$\begin{tikzcd}
 & & X \\
N \arrow[r, dashed, "f"] \arrow[urr, bend left, "f_Y"] \arrow[drr, bend right, "f_X"'] & X \times Y \arrow[ur, "\pi_X"] \arrow[dr, "\pi_Y"] & \\
 & & Y
\end{tikzcd}$$

Universal Properties

A function $f : A \times B \to X$ contains the following data:

Meditations on $\times$

A function $f : A \times B \to X$ contains the following data:

- For $A \times B$, the function f must specify an element $f(a, b) \in X$.
- Additionally, for any $b \in A$, the function $f(-, b)$ must specify a function $A \to X$ sending a to $f(a, b)$

Universal Properties

Thus, all that needed in order to define a function $A \times B \to X$ is the data that sends a $b \in A$ to the partial application of the function f to form a function $B \to \mathbf{C}(A, X) : a \mapsto f(-, b)$ which takes a a and produces a $x \in C$ by completing the partial evaluation $f(a, b)$.

Universal Properties

This should look familiar. *Because it's currying*!. The full statement of currying is the following:

Currying

$$\mathbf{C}(A \times B, C) \cong \mathbf{C}(B, \mathbf{C}(A, X))$$

Currying

Let's abstract that a bit. Suppose we are working exclusively in a category $\mathbf{C}$ with products and exponents.

Currying

There are two functors in play, and their hom sets are isomorphic. The product functor $A \times - : \mathbf{C} \to \mathbf{C}$, and an exponential functor $A^{(-)} : \mathbf{C} \to \mathbf{C}$.

Currying

By taking $\mathbf{C}(B, C) :\equiv B^C$, statement of currying can be refactored into the following: $\mathbf{C}(A \times B, C) \cong \mathbf{C}(B, C^A)$.

Currying

Note that if we remove the parameters of the functor, then we are left with a statement about the natural isomorphism whose components characterize currying:

$$\Phi_{X,Y} : \mathbf{C}(A \times X, Y) \cong \mathbf{C}(A, Y^X)$$

Adjunction

This is what is known as an **adjunction**, which characterizes the logical dual of a statement category theory.

Adjunctions

Definition (Adjunction)

Let $F : \mathbf{C} \to \mathbf{D}$ and $G : \mathbf{D} \to \mathbf{C}$ be functors. We say that F and G are **adjoint** (left and right adjoint resp.), or notationally $F \dashv G$ if there exists a natural isomorphism $\Phi : D(F-, =) \cong C(-, G =)$. The components of Φ are set bijections
$\Phi_{X,Y} : \mathbf{D}(FX, Y) \cong \mathbf{C}(X, GY)$

Adjunctions

In category theory, one of the strongest statements one can make is that two statements are logical "opposites". It is in fact the case that universal properties of our products and coproducts are characterized by relevant statements and their logical opposites. I fact, products and coproducts are dual themselves!

Adjunctions

It is a theorem which we will leave to the optional slide set that states that coproducts are left adjoint to a diagonal functor, and products are right adjoint to it.

Pairing

Lets see what $B + C \to X$ looks like and see if we can do the same thing!

Pairing

First, let us define coproducts.

Pairing

Definition

Pairing Let $\mathbf{C}$ be a category and let $X, Y \in \mathbf{C}$. A *coproduct* C is an object of $\mathbf{C}$ together with functions $i_X : X \to C$ and $i_Y : Y \to C$ such that the following holds:

- For any other $N \in \mathbf{C}$, and morphisms $f_X : X \to N$ and $f_Y : Y \to N$, there is a unique, induced morphism $f : C \to N$ such that $f_X = fi_X$ and $f_Y = fi_Y$.

The coproduct of C is generally denoted $X + Y$

Or, more succinctly, the following diagram commutes for any given N and morphisms f_Y, f_X:

$$\begin{tikzcd}
X \arrow[dr, "i_X"] \arrow[drr, bend left, "f_X"] & & \\
& X + Y \arrow[r, dashed, "f"] & N \\
Y \arrow[ur, "i_Y"] \arrow[urr, bend right, "f_Y"'] & &
\end{tikzcd}$$

Pairing

Note that all that's needed to define a function $f : B + C \to X$ is then to define a pair of functions $g : B \to X$ and $h : C \to X$ associating each $b \in B$ and $c \in C$ with an element of X.

Pairing

Thus, we obtain the following:

Pairing

$$\mathbf{C}(B + C, X) \cong \mathbf{C}(B, X) \times \mathbf{C}(C, X)$$

Pairing

Notice what happens when we "erase" the parameters of the pairing isomorphism just as we did for currying, and consider the natural isomorphism lying underneath.

Pairing

Again, there are two functors belying this statement - the coproduct functor $Y + (-) : \mathbf{C} \to \mathbf{C}$, but other one is a little different. If we note that a the product of hom-sets gives us a pair of morphisms $f \times g$, then we can also note that this product of hom-sets operates on products of elements. In fact, we have $(f \times g) : (B, C) \to (X, X)$. Ergo, we have $f \times g \in [\mathbf{C} \times \mathbf{C}]((B, C), (X, X))$.

Pairing

Therefore our functor on the right is one that takes any $X \in \mathbf{C}$ to its diagonal pair, $\Delta : \mathbf{C} \to \mathbf{C} \times \mathbf{C}$, and the components of our isomorphism take the form of

$$\Phi_{X,Y} : \mathbf{C}(B + X, Y) \to \mathbf{C}(B \times X, \Delta Y) \cong \mathbf{C}(B, Y) \times \mathbf{C}(X, Y)$$

Pairing

And the adjunction is complete.

Representability

Note the duality of it all! The diagrams obtained in the previous slides are dual universal properties of the product and coproduct respectively.

Exercise

Can you show that $(B \times C)^A \cong B^A \times C^A$?

Proof

Lets prove the following:

$$a \times (b + c) = (a \times b) + (a \times c)$$