PEANO ARITHMETIC USING LAMBDA CALCULUS

WHAT IS $\mathbb{N}$?

According to Lean:

```
inductive Nat where
    | zero : Nat
    | succ : Nat → Nat
```

According to Lean:

```
inductive Nat where
    | zero : Nat
    | succ : Nat → Nat
```

According to Haskell:

```
data Nat where
    Zero :: Nat
    Succ :: Nat -> Nat
```

According to me:

According to me:

- We have an endofunctor $F : \mathrm{Sets} \to \mathrm{Sets}, X \mapsto \mathrm{pt} \sqcup X$.

According to me:

- We have an endofunctor $F : \mathrm{Sets} \to \mathrm{Sets}, X \mapsto \mathrm{pt} \sqcup X$.
- We take the category $\mathrm{Mod}_F := \mathrm{Mod}_F(\mathrm{Sets})$:

According to me:

- We have an endofunctor $F : \mathrm{Sets} \to \mathrm{Sets}, X \mapsto \mathrm{pt} \sqcup X$.
- We take the category $\mathrm{Mod}_F := \mathrm{Mod}_F(\mathrm{Sets})$:
  - Objects are morphisms $\alpha : F(X) \to X$ in Sets.

According to me:

- We have an endofunctor $F : \mathrm{Sets} \to \mathrm{Sets}, X \mapsto \mathrm{pt} \sqcup X$.
- We take the category $\mathrm{Mod}_F := \mathrm{Mod}_F(\mathrm{Sets})$:
  - Objects are morphisms $\alpha : F(X) \to X$ in Sets.
  - Morphisms are commutative diagrams

$$\begin{array}{ccc} F(X) & \longrightarrow & X \\ \downarrow & & \downarrow \\ F(Y) & \longrightarrow & Y. \end{array}$$

According to me:

- We have an endofunctor $F : \mathrm{Sets} \to \mathrm{Sets}, X \mapsto \mathrm{pt} \sqcup X$.
- We take the category $\mathrm{Mod}_F := \mathrm{Mod}_F(\mathrm{Sets})$:
  - Objects are morphisms $\alpha : F(X) \to X$ in Sets.
  - Morphisms are commutative diagrams

$$\begin{array}{ccc} F(X) & \longrightarrow & X \\ \downarrow & & \downarrow \\ F(Y) & \longrightarrow & Y. \end{array}$$

- $\mathbb{N}$ is the initial object of Mod_F.

An object of Mod_F encodes two pieces of data:

$$\alpha : \mathrm{pt} \sqcup X \longrightarrow X$$

An object of Mod_F encodes two pieces of data:

$$\alpha : \mathrm{pt} \sqcup X \longrightarrow X$$

- $z := \alpha|_{\mathrm{pt}} : \mathrm{pt} \to X$ picks out an element of X,

An object of Mod_F encodes two pieces of data:

$$\alpha : \mathrm{pt} \sqcup X \longrightarrow X$$

- $z := \alpha|_{\mathrm{pt}} : \mathrm{pt} \to X$ picks out an element of X,
- $s := \alpha|_X : X \to X$ is a specified endomorphism.

An object of Mod_F encodes two pieces of data:

$$\alpha : \mathrm{pt} \sqcup X \longrightarrow X$$

- $z \coloneqq \alpha|_{\mathrm{pt}} : \mathrm{pt} \to X$ picks out an element of X,
- $s \coloneqq \alpha|_X : X \to X$ is a specified endomorphism.

For $\mathbb{N}$:

An object of Mod_F encodes two pieces of data:

$$\alpha : \mathrm{pt} \sqcup X \longrightarrow X$$

- $z \coloneqq \alpha|_{\mathrm{pt}} : \mathrm{pt} \to X$ picks out an element of X,
- $s \coloneqq \alpha|_X : X \to X$ is a specified endomorphism.

For $\mathbb{N}$:

- $0 : \mathrm{pt} \to \mathbb{N}$ picks out 0,

An object of Mod_F encodes two pieces of data:

$$\alpha : \mathrm{pt} \sqcup X \longrightarrow X$$

- $z := \alpha|_{\mathrm{pt}} : \mathrm{pt} \to X$ picks out an element of X,
- $s := \alpha|_X : X \to X$ is a specified endomorphism.

For $\mathbb{N}$:

- $0 : \mathrm{pt} \to \mathbb{N}$ picks out 0,
- $+1 : \mathbb{N} \to \mathbb{N}$ is the successor function.

Observe that, in **Set**, the initial object of the endofunctor $f : X \to 1 + X$ is $\langle 0, ++ \rangle : 1 + \mathbb{N} \to \mathbb{N}$, where $\mathbb{N}$ contains 0 the smallest natural and $++$ is the successor operator. Because monomorphisms to initial objects are isomorphisms, we can conclude that the only subalgebra of $\mathbb{N}$ is $\mathbb{N}$ itself.
This is the principle of induction.

CRASH COURSE
ON
LAMBDA CALCULUS

Untyped lambda calculus consists only of functions that can be applied to other functions.

Untyped lambda calculus consists only of functions that can be applied to other functions.

A lambda expression looks like "$\lambda x.\, E$".

Untyped lambda calculus consists only of functions that can be applied to other functions.

A lambda expression looks like "$\lambda x.\, E$".

Function application is done by (left associative) juxtaposition:

- "$f\ x$" is "$f(x)$",
- "$(\lambda a.\, a\ b)\ g$" is "$g\ b$" is "$g(b)$".

EXAMPLE

EXAMPLE

```
1 if    ≔ λc. λt. λf. c t f
2 true  ≔ λt. λf. t
3 false ≔ λt. λf. f
```

EXAMPLE

```
1 if    ≔ λc. λt. λf. c t f
2 true  ≔ λt. λf. t
3 false ≔ λt. λf. f
```

```
1 if true A B
2
3
4
5
```

EXAMPLE

```
1 if    ≔ λc. λt. λf. c t f
2 true  ≔ λt. λf. t
3 false ≔ λt. λf. f
```

```
1 if true A B
2 = (λc. λt. λf. c t f) true A B
3
4
5
```

EXAMPLE

```
1 if    ≔ λc. λt. λf. c t f
2 true  ≔ λt. λf. t
3 false ≔ λt. λf. f
```

```
1 if true A B
2 = (λc. λt. λf. c t f) true A B
3 = true A B
4
5
```

EXAMPLE

```
1 if    ≔ λc. λt. λf. c t f
2 true  ≔ λt. λf. t
3 false ≔ λt. λf. f
```

```
1 if true A B
2 = (λc. λt. λf. c t f) true A B
3 = true A B
4 = (λt. λf. t) A B
5
```

EXAMPLE

```
1 if    ≔ λc. λt. λf. c t f
2 true  ≔ λt. λf. t
3 false ≔ λt. λf. f
```

```
1 if true A B
2 = (λc. λt. λf. c t f) true A B
3 = true A B
4 = (λt. λf. t) A B
5 = A
```

EXAMPLE

```
1 if    ≔ λc. λt. λf. c t f
2 true  ≔ λt. λf. t
3 false ≔ λt. λf. f
```

```
1 if false A B
2
3
4
5
```

EXAMPLE

```
1 if    ≔ λc. λt. λf. c t f
2 true  ≔ λt. λf. t
3 false ≔ λt. λf. f
```

```
1 if false A B
2 = (λc. λt. λf. c t f) false A B
3
4
5
```

EXAMPLE

```
1 if    ≔ λc. λt. λf. c t f
2 true  ≔ λt. λf. t
3 false ≔ λt. λf. f
```

```
1 if false A B
2 = (λc. λt. λf. c t f) false A B
3 = false A B
4
5
```

EXAMPLE

```
1 if    ≔ λc. λt. λf. c t f
2 true  ≔ λt. λf. t
3 false ≔ λt. λf. f
```

```
1 if false A B
2 = (λc. λt. λf. c t f) false A B
3 = false A B
4 = (λt. λf. f) A B
5
```

EXAMPLE

```
1 if    ≔ λc. λt. λf. c t f
2 true  ≔ λt. λf. t
3 false ≔ λt. λf. f
```

```
1 if false A B
2 = (λc. λt. λf. c t f) false A B
3 = false A B
4 = (λt. λf. f) A B
5 = B
```

HOW DO WE COUNT IN LAMBDA CALCULUS?

CHURCH ENCODING

Big idea:

CHURCH ENCODING

Big idea:

> *A natural number $n \in \mathbb{N}$ tells you **how many** of something you have.* 
>

CHURCH ENCODING

Big idea:

> *A natural number $n \in \mathbb{N}$ tells you* ***how many*** *of something you have.* 

In lambda calculus:

$$n\ f\ x = \underbrace{f\ (f\ (\cdots (f}_{n \text{ times}}\ x))).$$

ARITHMETIC

ARITHMETIC

```
1 succ ≔ λn. λf. λx. f (n f x)
2
3
4
```

ARITHMETIC

```
succ ≔ λn. λf. λx. f (n f x)
plus ≔ λm. λn. λf. λx. m f (n f x)
```

ARITHMETIC

```
succ ≔ λn. λf. λx. f (n f x)
plus ≔ λm. λn. λf. λx. m f (n f x)
mult ≔ λm. λn. λf. λx. m (n f) x
```

ARITHMETIC

```
succ ≔ λn. λf. λx. f (n f x)
plus ≔ λm. λn. λf. λx. m f (n f x)
mult ≔ λm. λn. λf. λx. m (n f) x
pred ≔
```

ARITHMETIC

```
succ ≔ λn. λf. λx. f (n f x)
plus ≔ λm. λn. λf. λx. m f (n f x)
mult ≔ λm. λn. λf. λx. m (n f) x
pred ≔ ???
```

How do we subtract 1??!?

How do we subtract 1??!?

Wikipedia

SCOTT ENCODING

THE OTHER SCHOOL OF THOUGHT

Inductive types have this great feature called

✨*pattern matching*✨

For example, to describe a binary tree in Haskell:

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)
```

For example, to describe a binary tree in Haskell:

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)
```

Then to use a tree:

```
compute :: Tree a -> SomeType
compute t = case t of
    Leaf       -> ...
    Node l a r -> ...
```

For example, to describe a binary tree in Haskell:

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)
```

Then to use a tree:

```
compute :: Tree a -> SomeType
compute t = case t of
    Leaf       -> ...
    Node l a r -> ...
```

We can also do pattern-matching for `Nat`!

CONTINUATION-PASSING STYLE

CONTINUATION-PASSING STYLE

The functional programmers of the last century came up with some pretty wacky ideas.

CONTINUATION-PASSING STYLE

The functional programmers of the last century came up with some pretty wacky ideas.

If I have a value $a \in A$:

CONTINUATION-PASSING STYLE

The functional programmers of the last century came up with some pretty wacky ideas.

If I have a value $a \in A$:

- instead of giving you a, I can give you a function

$$\mathrm{ev}_a : \forall R.\, (A \to R) \to R, \quad \mathrm{ev}_a(f) = f(a).$$

CONTINUATION-PASSING STYLE

The functional programmers of the last century came up with some pretty wacky ideas.

If I have a value $a \in A$:

- instead of giving you a, I can give you a function

$$\mathrm{ev}_a : \forall R.\, (A \to R) \to R, \quad \mathrm{ev}_a(f) = f(a).$$

- the input f is a “continuation” of the computation.

Actually, this idea is not so unfamiliar.

Actually, this idea is not so unfamiliar.

For example:

Actually, this idea is not so unfamiliar.

For example:

- the canonical isomorphism $V \to V^{vv}$ for f.d. V;

Actually, this idea is not so unfamiliar.

For example:

- the canonical isomorphism $V \to V^{\vee\vee}$ for f.d. V;
- the R-points $X(R) = \mathrm{CRings}(A, R)$ for affine $X = \operatorname{Spec} A$;

Actually, this idea is not so unfamiliar.

For example:

- the canonical isomorphism $V \to V^{\vee\vee}$ for f.d. V;
- the R-points $X(R) = \mathrm{CRings}(A, R)$ for affine $X = \operatorname{Spec} A$;
- the Yoneda embedding $A \mapsto h_A = \mathrm{Hom}(A, -)$…

Actually, this idea is not so unfamiliar.

For example:

- the canonical isomorphism $V \to V^{\vee\vee}$ for f.d. V;
- the R-points $X(R) = \mathrm{CRings}(A, R)$ for affine $X = \operatorname{Spec} A$;
- the Yoneda embedding $A \mapsto h_A = \mathrm{Hom}(A, -)$…

Wait 🤔, maybe all of these examples are the last example…


Always has been
Wait, it's all Yoneda?

Taking a closer look 🤔

$$\mathrm{ev}_a : \forall R.\,(A \to R) \to R, \quad \mathrm{ev}_a(f) = f(a)$$

ev_a is just a natural transformation $\mathrm{Hom}(A, -) \Rightarrow \mathrm{Id}$!!!

By Yoneda, $A \cong \mathrm{Nat}(\mathrm{Hom}(A, -), \mathrm{Id})$.

SLOGAN

PATTERN MATCHING + YONEDA = SCOTT ENCODING

Let’s go back to the tree example:

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)
```

Let’s go back to the tree example:

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)
```

We can write this as $T_A = \mathrm{pt} \sqcup (T_A \times A \times T_A)$.

Let's go back to the tree example:

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)
```

We can write this as $T_A = \mathrm{pt} \sqcup (T_A \times A \times T_A)$.

So to map out of T_A (into R), we just need two things:

Let's go back to the tree example:

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)
```

We can write this as $T_A = \text{pt} \sqcup (T_A \times A \times T_A)$.

So to map out of T_A (into R), we just need two things:

1. A function $\text{pt} \to R$.
2. A function $T_A \times A \times T_A \to R$.

Let's go back to the tree example:

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)
```

We can write this as $T_A = \mathrm{pt} \sqcup (T_A \times A \times T_A)$.

So to map out of T_A (into R), we just need two things:

1. A function $\mathrm{pt} \to R$.
2. A function $T_A \times A \times T_A \to R$.

So for a tree $t \in T_A$, evaluation at t looks like

$$\mathrm{ev}_t : \forall R.\, R \to (T_A \times A \times T_A \to R) \to R.$$

THE SCOTT TREE

```
type STree a
    =  forall r.
       r                                    -- Leaf                        -> r
    -> (STree a -> a -> STree a -> r) -- Node (Tree a) a (Tree) -> r
    -> r
```

A tree is not so dissimilar from a natural number…

```
data Nat = Zero
         | Succ Nat
```

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)
```

A tree is not so dissimilar from a natural number…

```
data Nat = Zero
         | Succ Nat
```

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)
```

$$\forall R.\, R \to (\mathbb{N} \to R) \to R \qquad \forall R.\, R \to (T_A \times A \times T_A \to R) \to R$$

$$\forall R.\, R \to (\mathbb{N} \to R) \to R$$

$$\forall R.\, R \to (\mathbb{N} \to R) \to R$$

Looking at this function signature, we see exactly the two cases that make up a natural number:

$$\forall R.\, R \to (\mathbb{N} \to R) \to R$$

Looking at this function signature, we see exactly the two cases that make up a natural number:

- $n = 0$: the zero case,

$$\forall R.\, R \to (\mathbb{N} \to R) \to R$$

Looking at this function signature, we see exactly the two cases that make up a natural number:

- $n = 0$: the zero case,
- $n = n' + 1$: the successor case.

$$\forall R.\, R \to (\mathbb{N} \to R) \to R$$

Looking at this function signature, we see exactly the two cases that make up a natural number:

- $n = 0$: the zero case,
- $n = n' + 1$: the successor case.

The $\mathbb{N} \to R$ "deconstructs" the successor case into the predecessor:

```
pred :: Nat -> Nat
pred n = case n of
    Zero    -> undefined
    Succ n' -> n' -- here
```

$$\forall R.\, R \to (\mathbb{N} \to R) \to R$$

So in Scott encoding, the predecessor is simply handed to us!

$$\forall R.\, R \to (\mathbb{N} \to R) \to R$$

So in Scott encoding, the predecessor is simply handed to us!

```
type SNat = forall r. r -> (SNat -> r) -> r

pred :: SNat -> SNat
pred n = n undefined (\n' -> n')
```

Dead simple!

Okay, but we still need to solve the Church version.

$$\forall R.\, R \to (\mathbb{N} \to R) \to R$$

$$\forall R.\, R \to (\mathbb{N} \to R) \to R$$

What was Church encoding again?

$$\forall R.\, R \to (\mathbb{N} \to R) \to R$$

What was Church encoding again?

$$n\ f\ x = \underbrace{f\ (f\ (\cdots (f}_{n \text{ times}}\ x)))$$

$$\forall R.\, R \to (\mathbb{N} \to R) \to R$$

What was Church encoding again?

$$n\ f\ x = \underbrace{f\ (f\ (\cdots (f}_{n \text{ times}}\ x)))$$

$$\forall R.\, R \to (R \to R) \to R$$

SO WHAT'S GOING ON?

REVISITING *F*-MODULES

REVISITING F-MODULES

Given an endofunctor $F : \mathrm{Sets} \to \mathrm{Sets}$, we can talk about its module category $\mathrm{Mod}_F := \mathrm{Mod}_F(\mathrm{Sets})$ defined as before.

$$(\alpha : F(X) \longrightarrow X)$$

REVISITING F-MODULES

Given an endofunctor $F : \mathrm{Sets} \to \mathrm{Sets}$, we can talk about its module category $\mathrm{Mod}_F := \mathrm{Mod}_F(\mathrm{Sets})$ defined as before.

$$(\alpha : F(X) \longrightarrow X)$$

If Mod_F has an initial object $(\mathbb{M}, i)$, the unique map

$$! : (\mathbb{M}, i) \to (X, \alpha)$$

is called the *fold* map or *catamorphism*.

For example, for a fixed set A, the endofunctor

$$F(X) = \mathrm{pt} \sqcup (A \times X)$$

has as the initial object of its module category List(A):

```
data List a = [] | a : (List a)
```

For example, for a fixed set A, the endofunctor

$$F(X) = \mathrm{pt} \sqcup (A \times X)$$

has as the initial object of its module category List(A):

```
data List a = [] | a : (List a)
```

Aside: The colon ":" is used to denote concatenation onto a list. So for example

$$a_0{:}a_1{:}a_2{:}[] = a_0a_1{:}[a_2] = a_0{:}[a_1, a_2] = [a_0, a_1, a_2].$$

$$F(X) = \mathrm{pt} \sqcup (A \times X)$$

The map $\alpha : F(X) \to X$ of an F-module consists of:

- a map $e : \mathrm{pt} \to X$, or just $e \in X$; and
- a map $c : A \times X \to X$.

The *fold* map out of List(A) is so called because the unique morphism $! : \text{List}(A) \to (X, \alpha)$ must satisfy (proof)

- $!([]) = e$,
- $!(a{:}as) = c(a, !(as))$.

The *fold* map out of List(A) is so called because the unique morphism $! : \text{List}(A) \to (X, \alpha)$ must satisfy (proof)

- $!([]) = e$,
- $!(a{:}as) = c(a, !(as))$.

So the fold of a list $[a_0, a_1, a_2, \cdots, a_n]$ is forced to be

$$!([a_0, a_1, a_2, \ldots, a_n]) = c(a_0, c(a_1, c(a_2, c(\ldots, c(a_n, e))))).$$

$!$ recursively "folds" the list into a single element of X.

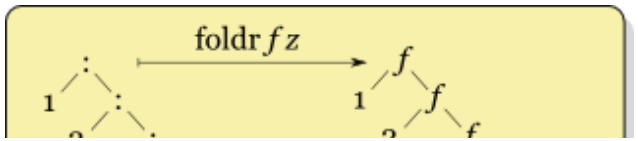
Fold (higher-order function)

In functional programming, a **fold** is a higher-order function that analyzes a recursive data structure and, through use of a given combining operation, recombines the results of recursively processing its constituent parts, building up a return value. Fold is also termed as **reduce**, **accumulate**, **aggregate**, **compress**, or **inject**. Typically, a fold is presented with a combining function, a top node of a data structure, and possibly some default values to be used under certain conditions. The fold then proceeds to combine elements of the data structure's hierarchy, using the function in a systematic way.

Folds are in a sense dual to unfolds, which take a *seed* value and apply a function corecursively to decide how to progressively construct a corecursive data structure, whereas a fold recursively breaks that structure down, replacing it with the results of applying a combining function at each node on its terminal values and the recursive results (catamorphism, versus anamorphism of unfolds).

As structural transformations

Folds can be regarded as consistently replacing the structural components of a data structure with functions and values. Lists, for example, are built up in many functional languages from two primitives: any list is either an empty list, commonly called *nil* (`[]`), or is constructed by prefixing an element in front of another list, creating what is called a *cons* node (`Cons(X1,Cons(X2,Cons(...(Cons(Xn,nil)))))`), resulting from application of a `cons` function (written down as a colon `(:)` in Haskell). One can view a fold on lists as *replacing* the *nil* at the end of the list with a specific value, and *replacing* each *cons* with a specific function. These replacements can be viewed as a diagram:



For natural numbers, the endofunctor is

$$F(X) = \mathrm{pt} \sqcup X.$$

(A special case of lists $F(X) = \mathrm{pt} \sqcup (A \times X)$ with $A = \mathrm{pt}$.)

For natural numbers, the endofunctor is

$$F(X) = \mathrm{pt} \sqcup X.$$

(A special case of lists $F(X) = \mathrm{pt} \sqcup (A \times X)$ with $A = \mathrm{pt}$.)

The map $\alpha : F(X) \to X$ of an F-module consists of:

- a map $z : \mathrm{pt} \to X$, or just $z \in X$; and
- a map $s : X \to X$.

For natural numbers, the endofunctor is

$$F(X) = \mathrm{pt} \sqcup X.$$

(A special case of lists $F(X) = \mathrm{pt} \sqcup (A \times X)$ with $A = \mathrm{pt}$.)

The map $\alpha : F(X) \to X$ of an F-module consists of:

- a map $z : \mathrm{pt} \to X$, or just $z \in X$; and
- a map $s : X \to X$.

The *fold* map $! : \mathbb{N} \to (X, \alpha)$ is thus $!(n) = s^n(z)$: precisely the Church encoding (proof).

To see the Church encoding more clearly, we just need to apply the Yoneda lemma.

To see the Church encoding more clearly, we just need to apply the Yoneda lemma.

Letting $U : \mathrm{Mod}_F \to \mathrm{Sets}$ be the forgetful functor,

To see the Church encoding more clearly, we just need to apply the Yoneda lemma.

Letting $U : \mathrm{Mod}_F \to \mathrm{Sets}$ be the forgetful functor,

$$\mathbb{N} \cong \mathrm{Nat}(\mathrm{Mod}_F((\mathbb{N}, i), -), U)$$

To see the Church encoding more clearly, we just need to apply the Yoneda lemma.

Letting $U : \mathrm{Mod}_F \to \mathrm{Sets}$ be the forgetful functor,

$$\begin{aligned}\mathbb{N} &\cong \mathbf{Nat}(\mathrm{Mod}_F((\mathbb{N}, i), -), U) \\ &\cong \int_{(X,\alpha)\in \mathrm{Mod}_F} \mathrm{Sets}(\mathrm{Mod}_F((N, i), (X, \alpha)), U(X, \alpha))\end{aligned}$$

To see the Church encoding more clearly, we just need to apply the Yoneda lemma.

Letting $U : \mathrm{Mod}_F \to \mathrm{Sets}$ be the forgetful functor,

$$\begin{aligned}
\mathbb{N} &\cong \mathbf{Nat}(\mathrm{Mod}_F((\mathbb{N}, i), -), U) \\
&\cong \int_{(X,\alpha)\in \mathrm{Mod}_F} \mathrm{Sets}(\mathrm{Mod}_F((N, i), (X, \alpha)), U(X, \alpha)) \\
&\cong \int_{(X,\alpha)\in \mathrm{Mod}_F} \mathrm{Sets}(\mathrm{pt}, X)
\end{aligned}$$

To see the Church encoding more clearly, we just need to apply the Yoneda lemma.

Letting $U : \mathrm{Mod}_F \to \mathrm{Sets}$ be the forgetful functor,

$$\begin{aligned}
\mathbb{N} &\cong \mathbf{Nat}(\mathrm{Mod}_F((\mathbb{N}, i), -), U) \\
&\cong \int_{(X,\alpha)\in \mathrm{Mod}_F} \mathrm{Sets}(\mathrm{Mod}_F((N, i), (X, \alpha)), U(X, \alpha)) \\
&\cong \int_{(X,\alpha)\in \mathrm{Mod}_F} \mathrm{Sets}(\mathrm{pt}, X) \\
&\cong \int_{(X,\alpha)\in \mathrm{Mod}_F} X.
\end{aligned}$$

$$\mathbb{N} \cong \int_{(X,\alpha)\in \mathrm{Mod}_F} X.$$

But fixing $X \in \mathrm{Sets}$, F-module structures on X are simply elements of $\mathrm{Sets}(F(X), X) = \mathrm{Sets}(\mathrm{pt} \sqcup X, X) = X \times \mathrm{Sets}(X, X)$.

But fixing $X \in \mathrm{Sets}$, F-module structures on X are simply elements of $\mathrm{Sets}(F(X), X) = \mathrm{Sets}(\mathrm{pt} \sqcup X, X) = X \times \mathrm{Sets}(X, X)$.

Therefore

$$\begin{aligned} \mathbb{N} &\cong \int_{(X,\alpha) \in \mathrm{Mod}_F} X \\ &\cong \int_{X \in \mathrm{Sets}} \mathrm{Sets}(\mathrm{Sets}(F(X), X), X) \end{aligned}$$

But fixing $X \in \mathrm{Sets}$, F-module structures on X are simply elements of $\mathrm{Sets}(F(X), X) = \mathrm{Sets}(\mathrm{pt} \sqcup X, X) = X \times \mathrm{Sets}(X, X)$.

Therefore

$$\begin{aligned}
\mathbb{N} &\cong \int_{(X,\alpha) \in \mathrm{Mod}_F} X \\
&\cong \int_{X \in \mathrm{Sets}} \mathrm{Sets}(\mathrm{Sets}(F(X), X), X) \\
&\cong \int_{X \in \mathrm{Sets}} \mathrm{Sets}(\mathrm{Sets}(X \times \mathrm{Sets}(X, X), X), X).
\end{aligned}$$

$$\mathbb{N} \cong \int_{X \in \mathsf{Sets}} \mathsf{Sets}(X \times \mathsf{Sets}(X, X), X).$$

What this says is that natural numbers are exactly (natural) families of functions

$$X \to (X \to X) \to X.$$

$$\mathbb{N} \cong \int_{X \in \mathrm{Sets}} \mathrm{Sets}(X \times \mathrm{Sets}(X, X), X).$$

What this says is that natural numbers are exactly (natural) families of functions

$$X \to (X \to X) \to X.$$

$$n\ f\ x = \underbrace{f\ (f\ (\cdots (f}_{n \text{ times}}\ x)))$$

$$\forall R.\, R \to (R \to R) \to R$$

ANOTHER PERSPECTIVE

ANOTHER PERSPECTIVE

Notice that $\mathbb{N}$ is a fixed point of F ($\mathbb{N} \cong \mathrm{pt} \sqcup \mathbb{N}$).

ANOTHER PERSPECTIVE

Notice that $\mathbb{N}$ is a fixed point of F ($\mathbb{N} \cong \mathrm{pt} \sqcup \mathbb{N}$).

So we can also apply the Yoneda lemma in the following way:

$$\mathbb{N} \cong F(\mathbb{N}) \cong \mathrm{Nat}(\mathrm{Sets}(F(\mathbb{N}), -), \mathrm{Id})$$
$$\cong \int_{X \in \mathrm{Sets}} \mathrm{Sets}(\mathrm{Sets}(F(\mathbb{N}), X), X)$$

ANOTHER PERSPECTIVE

Notice that $\mathbb{N}$ is a fixed point of F ($\mathbb{N} \cong \mathrm{pt} \sqcup \mathbb{N}$).

So we can also apply the Yoneda lemma in the following way:

$$\begin{aligned} \mathbb{N} \cong F(\mathbb{N}) &\cong \mathrm{Nat}(\mathrm{Sets}(F(\mathbb{N}), -), \mathrm{Id}) \\ &\cong \int_{X \in \mathrm{Sets}} \mathrm{Sets}(\mathrm{Sets}(F(\mathbb{N}), X), X) \end{aligned}$$

This now tells us that natural numbers are also (natural) families of functions

$$X \to (\mathbb{N} \to X) \to X.$$

WHAT IS $\mathbb{N}$?

WHAT IS $\mathbb{N}$?

Church encoding:

$$\mathbb{N} \cong \int_{X \in \mathrm{Sets}} \mathrm{Sets}(\mathrm{Sets}(F(X), X), X)$$

Scott encoding:

$$\mathbb{N} \cong \int_{X \in \mathrm{Sets}} \mathrm{Sets}(\mathrm{Sets}(F(\mathbb{N}), X), X)$$

WHAT IS $\mathbb{N}$?

Church encoding:

$$\mathbb{N} \cong \int_{X \in \mathrm{Sets}} \mathrm{Sets}(\mathrm{Sets}(F(X), X), X)$$

```
type CNat = forall r. r -> (r -> r) -> r
```

Scott encoding:

$$\mathbb{N} \cong \int_{X \in \mathrm{Sets}} \mathrm{Sets}(\mathrm{Sets}(F(\mathbb{N}), X), X)$$

```
type SNat = forall r. r -> (SNat -> r) -> r
```

WHAT IS $\mathbb{N}$?

Church encoding:

$$\mathbb{N} \cong \int_{X \in \mathrm{Sets}} \mathrm{Sets}(\mathrm{Sets}(F(X), X), X)$$

```
type CNat = forall r. r -> (r -> r) -> r
```

- Receives result of recursion.

Scott encoding:

$$\mathbb{N} \cong \int_{X \in \mathrm{Sets}} \mathrm{Sets}(\mathrm{Sets}(F(\mathbb{N}), X), X)$$

```
type SNat = forall r. r -> (SNat -> r) -> r
```

- Does one level of pattern matching.

WHAT DOES LEAN DO?

WHAT DOES LEAN DO?

It turns out Lean is a smart cookie and does both.

WHAT DOES LEAN DO?

It turns out Lean is a smart cookie and does both.

One can define the category of *F-paramorphisms* when the initial object $(\mathbb{M}, i) \in \mathsf{Mod}_F$ exists:

WHAT DOES LEAN DO?

It turns out Lean is a smart cookie and does both.

One can define the category of *F-paramorphisms* when the initial object $(\mathbb{M}, i) \in \mathrm{Mod}_F$ exists:

- Objects are morphisms $\alpha : F(\mathbb{M} \times X) \to X$ in Sets.

WHAT DOES LEAN DO?

It turns out Lean is a smart cookie and does both.

One can define the category of *F-paramorphisms* when the initial object $(\mathbb{M}, i) \in \mathrm{Mod}_F$ exists:

- Objects are morphisms $\alpha : F(\mathbb{M} \times X) \to X$ in Sets.
- Morphisms are the obvious commutative diagrams.

WHAT DOES LEAN DO?

It turns out Lean is a smart cookie and does both.

One can define the category of *F-paramorphisms* when the initial object $(\mathbb{M}, i) \in \mathrm{Mod}_F$ exists:

- Objects are morphisms $\alpha : F(\mathbb{M} \times X) \to X$ in Sets.
- Morphisms are the obvious commutative diagrams.
- Repeating the Yoneda argument,

$$\mathbb{N} \cong \int_{X \in \mathrm{Sets}} \mathrm{Sets}(\mathrm{Sets}(F(\mathbb{N} \times X), X), X).$$

We can see this in the recursor for the natural numbers here:

```
Nat.rec.{u} {motive : Nat → Sort u}
  (zero : motive zero)
  (succ : (n : Nat) → motive n → motive n.succ)
  (t : Nat) :
  motive t
```

We can see this in the recursor for the natural numbers here:

```
Nat.rec.{u} {motive : Nat → Sort u}
  (zero : motive zero)
  (succ : (n : Nat) → motive n → motive n.succ)
  (t : Nat) :
  motive t
```

Let's reduce to the non-dependent setting by choosing a constant motive motive(n) = X.

We can see this in the recursor for the natural numbers here:

```
Nat.rec.{u} {motive : Nat → Sort u}
  (zero : motive zero)
  (succ : (n : Nat) → motive n → motive n.succ)
  (t : Nat) :
  motive t
```

Let's reduce to the non-dependent setting by choosing a constant motive motive(n) = X.

After providing `t : Nat`, the signature reduces to

$$X \to (\mathbb{N} \to X \to X) \to X.$$

`pred`

Let’s solve our original problem now.

IDEA

$$n\ f\ x = \underbrace{f\ (f\ (\cdots\ (f}_{n \text{ times}}\ x)))$$

IDEA

$$\text{pred}\ n\ f\ x = n\ f'\ x' = \underbrace{f'\ (f'\ (\cdots (f'}_{n \text{ times}}\ x')))$$

$$\text{pred}\ n\ f\ x = n\ f'\ x' = \underbrace{f'\ (f'\ (\cdots (f'}_{n \text{ times}}\ x')))$$

$$x' = (\text{true},\ x)$$

$$f'\ (\text{true},\ x) = (\text{false},\ x)$$
$$f'\ (\text{false}, x) = (\text{false}, f\ x)$$

$$\text{pred}\, n\, f\, x = n\, f'\, x' = \text{pr}_2\ \underbrace{f'\,(f'\,(\cdots(f'}_{n \text{ times}}\ x')))$$

$$x' = (\text{true},\ x)$$

$$f'\,(\text{true},\ x) = (\text{false},\ x)$$
$$f'\,(\text{false}, x) = (\text{false}, f\ x)$$

$$\text{pred}\ n\ f\ x = n\ f'\ x' = \text{pr}_2\ \underbrace{f'\ (f'\ (\cdots (f'}_{n \text{ times}}\ x')))$$

$$x' = (\text{true},\ x)$$

$$f'\ (\text{true},\ x) = (\text{false},\ x)$$
$$f'\ (\text{false}, x) = (\text{false}, f\ x)$$

…but in lambda calculus.

$$\text{pred}\ n\ f\ x = n\ f'\ x' = \text{pr}_2\ \underbrace{f'\ (f'\ (\cdots (f'}_{n \text{ times}}\ x')))$$

$$x' = (\text{true},\ x)$$

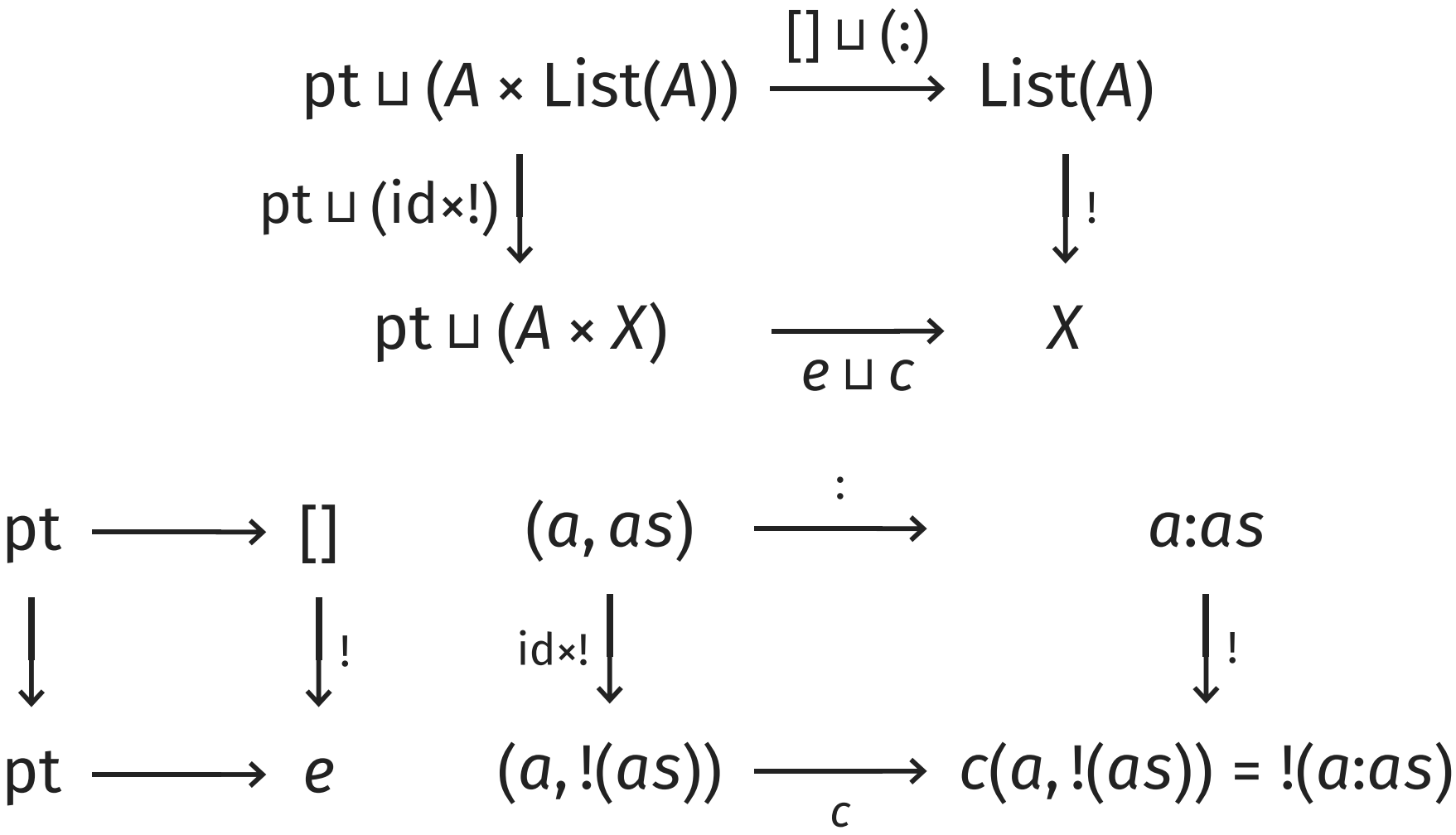
$$f'\ (\text{true},\ x) = (\text{false},\ x)$$
$$f'\ (\text{false}, x) = (\text{false}, f\ x)$$

…but in lambda calculus.

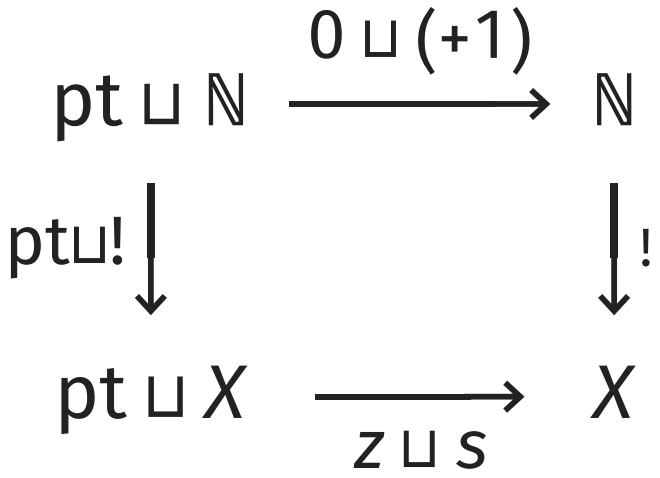
We use Church/Scott encoding!

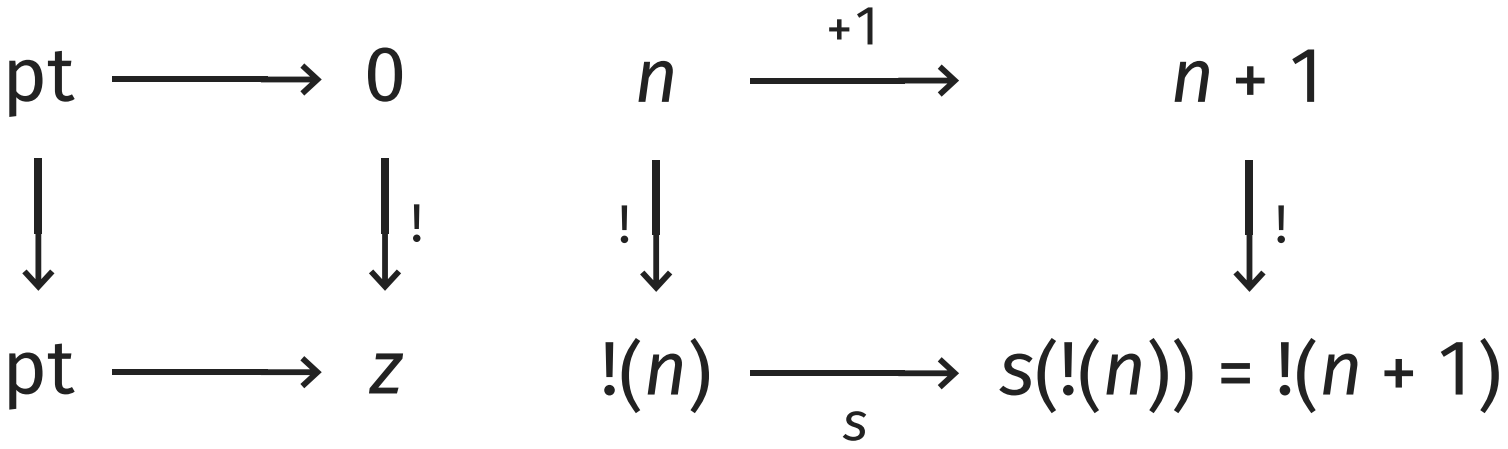
$$|-| : \mathrm{Fin} \to \mathbb{N}$$

APPENDIX



Go back


0 ⊔ (+1)
pt ⊔ ℕ
ℕ
pt⊔!
!
pt ⊔ X
X
z ⊔ s


+1
pt
0
n
n + 1
!
!
!
pt
z
!(n)
s(!(n)) = !(n + 1)
s