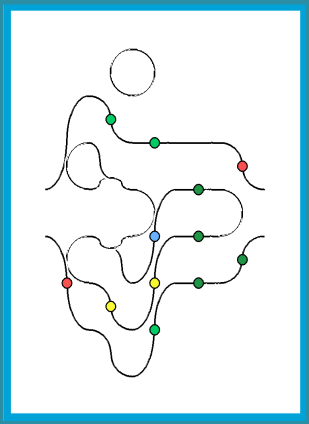
.py file
.tex file
.pdf file
.lean file

CATEGORIES

String diagrams in Lean 4



Elliot Young

1. Introduction

The main goals of this text are to prove:

Main Goals
Proving Fox's theorem
Proving the Yoneda lemma
Proving the adjoint functor theorem
Proving the Isbell duality isomorphism

Special effort has been made to make it approachable and self-contained. All of the theorems and proofs will be written in Lean 4. It also contains graphics called string diagrams. See the Lean 4 Github for installation instructions detailing how to get started with Lean 4, or try this example.

Mathematics is one of the oldest subjects, and as it concerns computer proof assistants, this entails a kind of change and development which is slow. Being in a subject where we are so commonly reminded of the brilliant minds who know things we do not, we have learned to defer to those who seem to have developed better judgement. At times this breeds specialization, at others, deference to talent, wisdom, or maturity.

This approach comes with many flaws, and we may have had higher hopes for decentralization in the Queen of the Sciences. Possibly computers will allow for a flexible higher mathematics which is more provisional where it should be, perminant in areas where an approach is insusceptible of improvement.

This much is not to mention the seemingly inevitable advent of an artificial intelligence which can outperform mathematicians at mathematics. Since this advent lies indefinitely far into the future, we might instead consider nearer benchmarks, each more tractible in one or another way. We may look forward to high quality search engines for mathematics, for instance.

Conglomerate interest itself is a meaningful factor in the development of computer proof assistants, and we should be wary of those who align themselves against its long term benefits. In any case, being part of these critical moments in the history of one of the oldest subjects makes mathematics all the more exciting.

2. Contents

Chapter VII: limits and colimits	
`lim I C:[I,C]⇆C`	the definition of limit
`colim I C:[I,C]⇄C`	the definition of colimit
`lim (Dis X) : [(Dis X),Set]⇄Set`	the limit with a set as a diagram in `Set`
`colim (Dis X) : [(Dis X),Cat]⇄Cat` / height⇉	the limit with a set a diagram in `Cat` The category with two parallel morphisms ⇉
`lim ⇉ Set`	limits over the category ⇉
`colim ⇉ Set`	colimits over the category ⇉
`lim ⇉ Cat`	limits over the category ⇉ in `Cat`
`colim ⇉ Cat`	limits over the category ⇉ in `Cat`
`lim C U`	limits are equilizers of particular products
`colim C U`	colimits are coequilizers of particular coproducts

Chapter VIII: the adjoint functor theorem	
`el F`	the category of elements of a functor F : C ⟶ D
`colim G • (el F) = G × F`	- everything is a colimit of representables
`lim G • (el F) = Hom G F`	-

3. Lean 4

Before we get started defining what a category is, we will cover the basic features of types in Lean 4. The main way we tell Lean 4 what something means is with `def`, which defines a term in dependent type theory. Much in the same way as other computer languages, we then supply the type of the term:

Lean 1

```
def n : Int := 1
```

here we have introduced an integer `n` using the type `Int` that comes with Lean 4. The main feature of a type besides a fascility with dependent product and hom is equality. This satisfies the three properties of an equivalence relation:

Lean 2

```
def reflexivity {X : Type} {x : X} (p : x = x) := Eq.refl p
def symmetry {X : Type} {x : X} {y : X}  (p : x = y) :=
↪  Eq.symm p
def transitivity {X : Type} {x : X} {y : X} {z : X} (p : x =
↪  y) (q : y = z) := Eq.trans p q
notation p "|" q => transitivity p q
```

We must also require that functiosn satisfy extensionality:

Lean 3

```
def extensionality (f g : X → Y) (p : (x:X) → f x = g x) : f =
↪  g := funext p
```

Extensionality says that functions which are equal on all values are themselves equal, and it is featured in what is perhaps the most famous mathematical foundations, ZFC.

There are two other features of equality with respect to functions which we should be aware of:

Lean 4

```
def equal_arguments {X : Type} {Y : Type} {a : X} {b : X} (f :
↪  X → Y) (p : a = b) : f a = f b := congrArg f p

def equal_functions {X : Type} {Y : Type} {f₁ : X → Y} {f₂ : X
↪  → Y} (p : f₁ = f₂) (x : X) : f₁ x = f₂ x := congrFun p x
```

These are the only features of equality which we will need.

The tutorial here provides a good introduction to using the dependent type theory in Lean.

Chapter I : categories

Section	Description
category	the category structure
Cat	the category of categories
$C \times D$	the product category $C \times D$ for C, D : category
[C, D]	the functor category [C, D] for C, D : category
$C \times - : \text{Cat} \longrightarrow \text{Cat}$	the functor $C \times - : \text{Cat} \longrightarrow \text{Cat}$ for C : category
$[C , -] : \text{Cat} \longrightarrow \text{Cat}$	the functor $[C, -] : \text{Cat} \longrightarrow \text{Cat}$ for C : category
$- \times C \vdash [C, -], \eta, \varepsilon$	the adjunction $- \times C \vdash [C, -], \eta, \varepsilon$ for C : category
$- \times C, \tau, \Delta$	the comonad $- \times C, \tau, \Delta$ for C : category
$[C, -], \iota, \mu$	the monad $[C, -], \iota, \mu$ for C : category
$(- \times C) \bullet (- \times D) \cong (- \times D) \bullet (- \times C)$	commutativity of $- \times C$ and $- \times D$ for C, D : category
$[C,-] \bullet [D,-] \cong [D,-] \bullet [C,-]$	commutativity of [C, -] and [D, -] for C, D : category
$\circledast, \circledast \times - \cong \mathbf{1}\ \text{Cat}, [\circledast, -] \cong \mathbf{1}\ \text{Cat}$	the unit category $\circledast$ and identity laws
Fox's Theorem	a characterization of cartesian closed categories

4. category

Our first goal is to define categories along with some notation for them. Categories were invented my Samuel Eilenberg and Saunders Mac Lane in the 20th century in the course of the work in algebraic topology. A category is a seven tuple consisting of:

1. A class `Obj` of objects
2. For each pair of objects `X,Y:Obj`, a class `Hom` whose elements are called morphisms.
3. For each object `X`, a morphism `Idn X`, also written 1_X.
4. For each triple of objects `X, Y, Z : Obj`, a function `(Hom X Y)` $\times$ `(Hom Y Z)` → `(Hom X Z)`.

such that

1. `∀(X:Obj),∀(Y:Obj),∀(f:Hom X Y),`

   `f∘(Idn X)=f`

2. `∀(X:Obj),∀(Y:Obj),∀(f:Hom X Y),(Idn X)∘f=f`

   `(Idn X)∘f=f`

3. `∀(W:Obj),∀(X:Obj),∀(Y:Obj),∀(Z : Obj),∀(f:Hom W X),∀(g:Hom X Y),∀(h:Hom Y Z),`

   `f ∘ (g ∘ h) = (f ∘ g) ∘ h`

These laws resemble the properties of composition of ordinary functions, ensuring their most basic properties.

The most straightforward definition of a category in Lean 4 is not much different. We record the entries of Eilenberg and Maclane's seven tuple using a Lean `structure`, which is similar to a `class`:

Lean 5

```
-- A category C consists of:
structure category where
  Obj : Type u
  Hom : Obj → Obj → Type v
  Idn : (X:Obj) → Hom X X
  Cmp : (X:Obj) → (Y:Obj) → (Z:Obj) → (_:Hom X Y) → (_:Hom Y
  ↪  Z) → Hom X Z
  Id₁ : (X:Obj) → (Y:Obj) → (f:Hom X Y) →
    Cmp X Y Y f (Idn Y) = f
  Id₂ : (X:Obj) → (Y:Obj) → (f:Hom X Y) →
    Cmp X X Y (Idn X) f = f
  Ass : (W:Obj) → (X:Obj) → (Y:Obj) → (Z:Obj) → (f:Hom W X) →
  ↪  (g:Hom X Y) → (h:Hom Y Z) →
    (Cmp W X Z) f (Cmp X Y Z g h) = Cmp W Y Z (Cmp W X Y f g)
    ↪  h
```

We have here adopted a system which uses three letter combinations such as `Hom` and `Idn` to name the seven entries of the category structure. This is part of a larger precedent we will take to use three letter combinations for the entries of a structure.

We will use the following notation to accompany the category structure:

Lean 6

```
-- Notation for the identity map which infers the category:
def identity_map {C : category} (X : C.Obj) := C.Idn X
notation "𝟙" => identity_map

-- Notation for the domain of a morphism:
def Dom {C : category} {X : C.Obj} {Y : C.Obj} (_ : C.Hom X Y)
↪  := X

-- Notation for the codomain of a morphism:
def Cod {C : category} {X : C.Obj} {Y : C.Obj} (_ : C.Hom X Y)
↪  := X

-- Notation for composition which infers the category and
↪  objects:
def composition {C : category} {X : C.Obj} {Y : C.Obj} {Z :
↪  C.Obj} (f : C.Hom X Y) (g : C.Hom Y Z) : C.Hom X Z :=
↪  C.Cmp X Y Z f g
notation g "∘" f => composition f g
```

We would like for an equation between two objects to produce a morphism. Such a morphism can be produced using Lean's substitute tactic `subst`:

Lean 7

```
-- obtaining a morphism from an equality
def Map {C : category} {X : C.Obj} {Y : C.Obj} (p : X = Y) :
↪  C.Hom X Y := by
subst p
exact C.Idn X
```

The notion of an isomorphism is essential to categories. It consists of two morphisms which are inverse to each other:

Lean 8

```
-- definition of an isomorphism from X to Y
structure isomorphism {C : category} (X : C.Obj) (Y : C.Obj)
↪  where
  Fst : C.Hom X Y
  Snd : C.Hom Y X
  Id₁ : (Fst ∘ Snd) = 𝟙 Y
  Id₂ : (Snd ∘ Fst) = 𝟙 X
```

We use the ≅ symbol to notate it:

Lean 9

```
-- notation for isomorphisms from X to Y (≅)
notation X "≅" Y => isomorphism X Y
```

The inverse of an isomorphism is straightforward to define:

Lean 10

```
-- defining the inverse of an isomorphism between objects X
↪  and Y
def inverse {C : category} {X : C.Obj} {Y : C.Obj} (f : X ≅ Y)
↪  : Y ≅ X := {Fst := f.Snd, Snd := f.Fst, Id₁ := f.Id₂, Id₂
↪  := f.Id₁}
```

Lean 4 uses unicode characters, and this entails an extensive variety of characters to choose from. We can use the usual unicode superscripts as notation for the inverse using Lean's `notation` feature:

Lean 11

```
-- notation for inverse : isos from X to Y to isos from Y to X
notation f "⁻¹" => inverse f
```

5. Set

`Set` is perhaps the simplest example of a category. We define this category first. Since the `category` structure has seven entries, we make seven definitions, one of each constituent, before assembling them into `Set`. We start `Obj`,`Hom`,`Idn`, and `Cmp`:

Lean 12

```
-- defining the objects of the category Set
def SetObj : Type 1 := Type

-- defining the morphisms of the category Set
def SetHom (X : SetObj) (Y : SetObj) : Type := X → Y

-- defining the identity morphism of an object in the category
↪  Set
def SetIdn (X : SetObj) : SetHom X X := λ (x : X) => x

--  defining composition in the category Set
def SetCmp (X : SetObj) (Y : SetObj) (Z : SetObj) (f : SetHom
↪  X Y) (g : SetHom Y Z) : (SetHom X Z) := λ (x : X) => (g (f
↪  x))
```

Next we show the three properties that make `Set` a category:

Lean 13

```
-- proving the first identity law for composition in Set
def SetId₁ (X : SetObj) (Y : SetObj) (f : SetHom X Y) : SetCmp
↪  X Y Y f (SetIdn Y) = f := sorry

-- proving the second identity law for composition in Set
def SetId₂ (X : SetObj) (Y : SetObj) (f : SetHom X Y) : SetCmp
↪  X X Y (SetIdn X) f = f := sorry

-- proving the associativity law for composition in Set
def SetAss (W : SetObj) (X : SetObj) (Y : SetObj) (Z : SetObj)
↪  (f : SetHom W X) (g : SetHom X Y) (h : SetHom Y Z) :
↪  SetCmp W X Z f (SetCmp X Y Z g h) = SetCmp W Y Z (SetCmp W
↪  X Y f g) h := sorry
```

To assemble constituents into a structure, we use the notation `def instance:structure:={}`:

Lean 14

```
-- defining the category Set
def Set : category := {Obj := SetObj, Hom := SetHom, Idn :=
↪  SetIdn, Cmp := SetCmp, Id₁ := SetId₁, Id₂ := SetId₂, Ass :=
↪  SetAss}
```

6. Cat

Our next goal is to define the category of categories, `Cat`. Since the category structure has seven constituents, the construction will have seven steps. We begin with defining `Cat.Hom`, which we call `functor`.

Lean 15

```
-- definition of a functor
structure functor (C : category) (D : category) where
   Obj : ∀(_ : C.Obj),D.Obj
   Hom : ∀(X : C.Obj),∀(Y : C.Obj),∀(_ : C.Hom X Y),D.Hom (Obj
   ↪  X) (Obj Y)
   Idn : ∀(X : C.Obj),Hom X X (C.Idn X) = D.Idn (Obj X)
   Cmp : ∀(X : C.Obj),∀(Y : C.Obj),∀(Z : C.Obj),∀(f : C.Hom X
   ↪  Y),∀(g : C.Hom Y Z),
   D.Cmp (Obj X) (Obj Y) (Obj Z) (Hom X Y f) (Hom Y Z g) = Hom
   ↪  X Z (C.Cmp X Y Z f g)
```

Because of its significance in this text, we use special notation for the functor:

Lean 16

```
-- notation for the type of Hom from a category C to a
↪  category D
notation C "⟶" D => functor C D
```

We also use special notation for the domain and codomain of a functor which is distinct from the `Dom` and `Cod` notation for other categories:

Lean 17

```
-- Notation for the domain of a functor:
def domain {C : category} {X : C.Obj} {Y : C.Obj} (_ : C.Hom X
↪  Y) := X
notation "" => domain
```

Lean 18

```
-- Notation for the domain of a functor:
def codomain {C : category} {X : C.Obj} {Y : C.Obj} (_ : C.Hom
↪  X Y) := Y
notation "" => codomain
```

Next in line for the construction of the category `Cat` is `Cat.Idn`, which gives the identity functor of a given category. Since the functor structure has five constituents, this will take five steps:

Lean 19

```
-- definition of the identity functor on objects
def CatIdnObj (C : category) :=
fun(X : C.Obj)=>
X

-- definition of the identity functor on morphisms
def CatIdnMor (C : category) :=
fun(X : C.Obj)=>
fun(Y : C.Obj)=>
fun(f : C.Hom X Y)=>
f

-- proving the identity law for the identity functor
def CatIdnIdn (C : category) :=
fun(X : C.Obj)=>
Eq.refl (𝟙 X)

-- proving the compositionality law for the identity functor
def CatIdnCmp (C : category) :=
fun(X : C.Obj)=>
fun(Y : C.Obj)=>
fun(Z : C.Obj)=>
fun(f : C.Hom X Y)=>
fun(g : C.Hom Y Z)=>
Eq.refl (g ∘ f)

-- defining the identity functor
def CatIdn (C : category) : functor C C :=
{Obj := CatIdnObj C, Hom := CatIdnMor C, Idn := CatIdnIdn C,
↪  Cmp := CatIdnCmp C}
```

It too gets special notation matching the bold theme:

Lean 20

```
-- notation for the identity functor
notation "1" => CatIdn
```

The last construction (not counting the theorems Cat.Id$_1$, Cat.Id$_2$, Cat.Ass) is the composition of two functors. Since this is supposed to produce a functor, this step will consist of four parts.

Lean 21

```
-- defining the composition G ∘ F on objects
def CatCmpObj (C : category) (D : category) (E : category) (F
↪  : functor C D) (G : functor D E) :=
fun(X : C.Obj)=>
(G.Obj (F.Obj X))

-- defining the composition G ∘ F on morphisms
def CatCmpHom (C : category) (D : category) (E : category) (F
↪  : functor C D) (G : functor D E) :=
fun(X : C.Obj)=>
fun(Y : C.Obj)=>
fun(f : C.Hom X Y)=>
(G.Hom) (F.Obj X) (F.Obj Y) (F.Hom X Y f)

-- proving the identity law for the composition G ∘ F
def CatCmpIdn (C : category) (D : category) (E : category) (F
↪  : functor C D) (G : functor D E) :=
fun(X : C.Obj)=>
(congrArg (G.Hom (F.Obj X) (F.Obj X)) (F.Idn X) ) | (G.Idn
↪  (F.Obj X))

-- proving the compositionality law for the composition G ∘ F
def CatCmpCmp (C : category) (D : category) (E : category) (F
↪  : functor C D) (G : functor D E) :=
fun(X : C.Obj)=>
fun(Y : C.Obj)=>
fun(Z : C.Obj)=>
fun(f : C.Hom X Y)=>
fun(g : C.Hom Y Z)=>
((Eq.trans)
(G.Cmp (F.Obj X) (F.Obj Y) (F.Obj Z) (F.Hom X Y f) (F.Hom Y Z
↪  g))
(congrArg (G.Hom  (F.Obj X) (F.Obj Z)) (F.Cmp X Y Z f g)))

-- defining the composition in the category Cat
def CatCmp (C : category) (D : category) (E : category) (F :
↪  functor C D) (G : functor D E) : functor C E :=
{Obj := CatCmpObj C D E F G, Hom := CatCmpHom C D E F G,Idn :=
↪  CatCmpIdn C D E F G, Cmp := CatCmpCmp C D E F G }
```

Functor composition gets the notation • (U2202). Note that we will use a similar but distinct unicode symbol • (U2219) for horizontal composition of natural transformations.

Lean 22

```
-- notation for the composition in the category Cat
def functor_composition {C : category} {D : category} {E :
↪  category} (F : functor C D) (G : functor D E) : functor C
↪  E := CatCmp C D E F G
notation G "•" F => functor_composition F G
/-
-- this should be able to handle F • X or F • G
-/
```

Now we may proceed to prove the three conditions ensuring that `Cat` is a category:

Lean 23

```
-- proving Cat.Id₁
def CatId₁ (C : category) (D : category) (F : functor C D) :
↪  ((CatCmp C D D) F (CatIdn D) = F) := Eq.refl F
```

Lean 24

```
-- Proof of Cat.Id₂
def CatId₂ (C : category) (D : category) (F : functor C D) :
↪  ((CatCmp C C D) (CatIdn C) F = F) := Eq.refl F
```

Lean 25

```
-- Proof of Cat.Ass
def CatAss (B : category) (C : category) (D : category) (E :
↪  category) (F : functor B C) (G : functor C D) (H : functor
↪  D E) : (CatCmp B C E F (CatCmp C D E G H) = CatCmp B D E
↪  (CatCmp B C D F G) H) :=
Eq.refl (CatCmp B C E F (CatCmp C D E G H))
```

Lean 26

```
-- The category of categories
def Cat : category := {Obj := category, Hom := functor, Idn :=
↪  CatIdn, Cmp := CatCmp, Id₁:= CatId₁, Id₂:= CatId₂, Ass :=
↪  CatAss}
```

7. op

Lean 27

```
/-
def OppObjObj (C : category) := C.Obj
def OppObjHom (C : category) (X : OppObjObj) (Y :
↪  OppObjObj) := C.Hom Y X
def OppObjIdn (C : category) (X : OppObjObj) :=
↪  C.Idn X
def OppObjCmp (C : category) (X : OppObjObj) (Y :
↪  OppObjObj) (Z : OppObjObj) (f : OppObjHom X Y)
↪  (g : OppObjHom Y Z) : OppObjHom X Z :=
-/
-- def OppObj (C : category) : category := {Obj := OppObjObj,
↪  Hom := OppObjHom, Idn := OppObjIdn, Cmp := OppObjCmp}
```

Lean 28

```
/-
def OppHomObj (C : category) (D : category) (F :
↪  functor C D)
def OppHomHom (C : category) (D : category) (F :
↪  functor C D)
def OppHomIdn (C : category) (D : category) (F :
↪  functor C D)
def OppHomCmp (C : category) (D : category) (F :
↪  functor C D)
def OppHom (C : category) (D : category) (F :
↪  functor C D)
-/
```

Lean 29

```
/-
def OppIdn
-/
```

Lean 30

```
/-
def OppCmp
-/
```

Lean 31

```
def Opp : Cat ⟶ Cat := sorry --{}
```

Lean 32

```
notation C "ᵒᵖ" => Opp.Obj C
```

Chapter II : the cartesian closed category of categories

Section	Description
$C \times D$	the product category $C \times D$ for C, D : category
$[C, D]$	the functor category $[C, D]$ for C, D : category
$C \times - : \mathrm{Cat} \to \mathrm{Cat}$	the functor $C \times - : \mathrm{Cat} \to \mathrm{Cat}$ for C : category
$[C , -] : \mathrm{Cat} \to \mathrm{Cat}$	the functor $[C, -] : \mathrm{Cat} \to \mathrm{Cat}$ for C : category
$- \times C \vdash [C, -], \eta, \varepsilon$	the adjunction $- \times C \vdash [C, -], \eta, \varepsilon$ for C : category
$- \times C, \tau, \Delta$	the comonad $- \times C, \tau, \Delta$ for C : category
$[C, -], \iota, \mu$	the monad $[C, -], \iota, \mu$ for C : category
$(- \times C) \bullet (- \times D) \cong (- \times D) \bullet (- \times C)$	commutativity of $- \times C$ and $- \times D$ for C, D : category
$[C,-] \bullet [D,-] \cong [D,-] \bullet [C,-]$	commutativity of $[C, -]$ and $[D, -]$ for C, D : category
$\circledast$, $\circledast \times - \cong \mathbf{1}\ \mathrm{Cat}$, $[\circledast, -] \cong \mathbf{1}\ \mathrm{Cat}$	the unit category $\circledast$ and identity laws
Fox's Theorem	a characterization of cartesian closed categories

Our next two goals are to define the cartesian product of two categories and the category of functors between two categories. These operations, each a function of two categories, will be called `C × D` and `C ⟶ D`.

Being a category, the construction of `C × D` will take seven steps:

defining `(Prd C D)`
`category`
`(Prd C D).Obj` for `C, D : category`
Defining `(Prd C D).Hom` for `C, D: category`
Defining `(Prd C D).Idn` for `C, D: category`
Defining `(Prd C D).Cmp` for `C, D: category`
Defining `(Prd C D).Id₁` for `C, D: category`
Defining `(Prd C D).Id₂` for `C, D: category`
Defining `(Prd C D).Ass` for `C, D: category`
Defining `Prd C D` for `C, D: category`

We begin by defining the Obj, Hom, Idn, and Cmp components:

Lean 33

```
-- defining the objects of the Prd C × D
def PrdObjObj (C : category) (D : category) := (C.Obj) ×
↪ (D.Obj)

-- defining the morphisms of C × D
def PrdObjHom (C : category) (D : category) (X : PrdObjObj C
↪ D) (Y : PrdObjObj C D) := (C.Hom X.1 Y.1) × (D.Hom X.2
↪ Y.2)

-- defining the identity functor on an object in C × D
def PrdObjIdn (C : category) (D : category) (X : PrdObjObj C
↪ D) := ((C.Idn X.1),(D.Idn X.2))

-- defining the composition on morphisms in C × D
def PrdObjCmp (C : category) (D : category) (X : PrdObjObj C
↪ D) (Y : PrdObjObj C D) (Z : PrdObjObj C D) (f : PrdObjHom
↪ C D X Y) (g : PrdObjHom C D Y Z) : PrdObjHom C D X Z :=
↪ (g.1 ∘ f.1, g.2 ∘ f.2)
```

Next we prove the first identity law for the category C × D:

Lean 34

```
-- proving the first identity law for morphisms in C × D
theorem PrdObjId₁ (C : category) (D : category) (X : PrdObjObj
↪  C D) (Y : PrdObjObj C D) (f : PrdObjHom C D X Y) :
  PrdObjCmp C D X Y Y f (PrdObjIdn C D Y) = f  := sorry

/-
-- Eq.trans (PrdObjId₁₀ C D X Y f) (Eq.trans
↪  (PrdObjId₁₁ C D X Y f) (PrdObjId₁₂ C D X Y f))

theorem PrdObjId₁₀ (C : category) (D : category)
↪  (X : PrdObjObj C D) (Y : PrdObjObj C D) (f :
↪  PrdObjHom C D X Y) :
  PrdCmp C D X Y Y f (PrdIdn C D Y) = (C.Cmp X.1
↪  Y.1 Y.1 f.1 (C.Idn Y.1), D.Cmp X.2 Y.2 Y.2 f.2
↪  (D.Idn Y.2)) := Eq.refl (C.Cmp X.1 Y.1 Y.1 f.1
↪  (C.Idn Y.1), D.Cmp X.2 Y.2 Y.2 f.2 (D.Idn
↪  Y.2))
theorem PrdObjId₁₁ (C : category) (D : category)
↪  (X : PrdObjObj C D) (Y : PrdObjObj C D) (f :
↪  PrdObjHom C D X Y) :
  (C.Cmp X.1 Y.1 Y.1 f.1 (C.Idn Y.1), D.Cmp X.2
↪  Y.2 Y.2 f.2 (D.Idn Y.2)) = (f.1, f.2) :=
  by rw [show (f.fst, f.snd) = _ by rw [← C.Id₁
↪  X.1 Y.1 f.1, ← D.Id₁ X.2 Y.2 f.2]]
theorem PrdObjId₁₂ (C : category) (D : category)
↪  (X : PrdObjObj C D) (Y : PrdObjObj C D) (f :
↪  PrdObjHom C D X Y) :
  (f.1, f.2) = f := Eq.refl f
-/
```

and then the second:

Lean 35

```
-- proving the second identity law for morphisms in C × D
theorem PrdObjId₂ (C : category) (D : category) (X : PrdObjObj
↪  C D) (Y : PrdObjObj C D) (f : PrdObjHom C D X Y) :
↪  PrdObjCmp C D X X Y (PrdObjIdn C D X) f = f  := sorry
/-
-- Eq.trans (PrdObjId₂₀ C D X Y f) (Eq.trans
↪  (PrdId₂₁ C D X Y f) (PrdId₂₂ C D X Y f))
theorem PrdId₂₀ (C : category) (D : category) (X :
↪  PrdObjObj C D) (Y : PrdObjObj C D) (f :
↪  PrdObjHom C D X Y) :
  PrdCmp C D X X Y (PrdIdn C D X) f = (C.Cmp X.1
↪  X.1 Y.1 (C.Idn X.1) f.1, D.Cmp X.2 X.2 Y.2
↪  (D.Idn X.2) f.2) :=
  Eq.refl (C.Cmp X.1 X.1 Y.1 (C.Idn X.1) f.1,
↪  D.Cmp X.2 X.2 Y.2 (D.Idn X.2) f.2)
theorem PrdId₂₁ (C : category) (D : category) (X :
↪  PrdObjObj C D) (Y : PrdObjObj C D) (f :
↪  PrdObjHom C D X Y) :
  (C.Cmp X.1 X.1 Y.1 (C.Idn X.1) f.1, D.Cmp X.2
↪  X.2 Y.2 (D.Idn X.2) f.2) = (f.1, f.2) :=
  by rw [show (f.fst, f.snd) = _ by rw [← C.Id₂
↪  X.1 Y.1 f.1, ← D.Id₂ X.2 Y.2 f.2]]
theorem PrdId₂₂ (C : category) (D : category) (X :
↪  PrdObjObj C D) (Y : PrdObjObj C D) (f :
↪  PrdObjHom C D X Y) :
  (f.1, f.2) = f := Eq.refl f
-/
```

and finally associativity:

Lean 36

```
-- proving associativity for morphisms in C × D
theorem PrdObjAss (C : category) (D : category) (W : PrdObjObj
↪  C D) (X : PrdObjObj C D) (Y : PrdObjObj C D) (Z :
↪  PrdObjObj C D) (f : PrdObjHom C D W X) (g : PrdObjHom C D
↪  X Y) (h : PrdObjHom C D Y Z) : PrdObjCmp C D W X Z f
↪  (PrdObjCmp C D X Y Z g h) = PrdObjCmp C D W Y Z (PrdObjCmp
↪  C D W X Y f g) h := sorry
```

Assembling these gives the definition of PrdObj:

Lean 37

```
-- defining the PrdObj of two categories
def PrdObj (C : category) (D : category) : category := {Obj :=
↪  PrdObjObj C D, Hom := PrdObjHom C D, Idn := PrdObjIdn C D,
↪  Cmp := PrdObjCmp C D, Id₁ := PrdObjId₁ C D, Id₂:= PrdObjId₂
↪  C D, Ass := PrdObjAss C D}
```

Lean 38

```
notation C "×_Cat" D => PrdObj C D
```

We will add notation for the product later.

9. (Hom C).Obj D

Next we would like to define the category of functors `(Hom C).Obj D` for a category `C` and a category `D`. This amounts to the following:

defining (Hom C D)
`category`
`(Hom C D).Obj` for `C, D: category`
Defining `(Hom C D).Hom` for `C, D: category`
Defining `(Hom C D).Idn` for `C, D: category`
Defining `(Hom C D).Cmp` for `C, D: category`
Defining `(Hom C D).`$\mathtt{Id}_1$ for `C, D: category`
Defining `(Hom C D).`$\mathtt{Id}_2$ for `C, D: category`
Defining `(Hom C D).Ass` for `C, D: category`
Defining `Hom C D` for `C, D: category`

We can start with `(Hom C).Obj D`, which we name `HomObj`. `HomObj` will be component of a component of `Hom`. We have already defined `HomObj`. HomHom is

10. Prd C : Cat $\rightarrow$ Cat

Our next goal is to define the functor

11. Hom C : Cat ⟶ Cat

We have constructed the category [C, D]_Cat for categories C and D. Next we construct the functor Hom C : Cat → Cat for each category C, which on an object D : category takes the value

() Defining (Hom C F).Obj for a category C and a functor F : $D_1 \to D_2$

Defining (Hom C F).Hom for a category C and a functor F : $D_1 \to D_2$

Defining (Hom C F).Idn for a category C and a functor F : $D_1 \to D_2$

Defining (Hom C F).Cmp for a category C and a functor F : $D_1 \to D_2$

Proving the identity law for the functor Hom C

Proving compositionality for the functor Hom C

We begin with (i)(a) and (i)(b):

Lean 39

```
--  defining Hom C F on objects
def HomHomObj (C : category) (D₁ : category) (D₂ : category) (F
↪  : functor D₁ D₂) (G : functor C D₁) := Cat.Cmp C D₁ D₂ G F

-- defining Hom C F on morphisms
def HomHomHom (C : category) (D₁ : category) (D₂ : category) (F
↪  : functor D₁ D₂) (G₁ : functor C D₁) (G₂ : functor C D₁) (g
↪  : G₁ ⟹ G₂) : (F ● G₁) ⟹ (F ● G₂) := sorry
```

We show the identity and compositionality laws for Hom C:

Lean 40

```
-- proving the identity law for Hom C F
-- def HomHomIdn (C : category) (D₁ : category) (D₂ : category)
↪  (F : functor D₁ D₂) := sorry

--  proving the compositionality law for Hom C F
-- def HomHomCmp := sorry
```

And finally

Lean 41

```
-- defining Hom C F
-- def HomHom (C : category) (D₁ : category) (D₂ : category) (F
↪  : D₁ ⟶ D₂) : (HomObj C D₁) ⟶ (HomObj C D₂) := sorry
```

Next we prove the identity and compositionality laws for `Hom :  category → (Cat ⟶ Cat)`:

Lean 42

```
--  proving the identity law for Hom C
-- def HomIdn (C : category) () :  := sorry

--  proving the compositionality law for Hom C
--  def HomCmp (C : category) () :  := sorry
```

Our work assembles into the desired functor `Hom C : Cat ⟶ Cat`

Lean 43

```
-- defining the functor Hom C : Cat ⟶ Cat
def Hom (C : category) : Cat ⟶ Cat := sorry
```

Lean 44

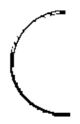
```
notation "[" "-" "," C "]_Cat" => Hom C
```

12. X × - ⊢ [X, -], η, ε

Lean 45

```
-- Defining the unit of the prd-hom adjunction
def Pair (C : category) : (1 Cat) ⟹ (Hom C) • (Prd C) :=
↪  sorry
```

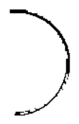
Graphic



Lean 47

```
-- Defining the counit of the prd-hom adjunction
def Eval (C : category) : ((Prd C) • (Hom C)) ⟹ (1 Cat) :=
↪  sorry
```

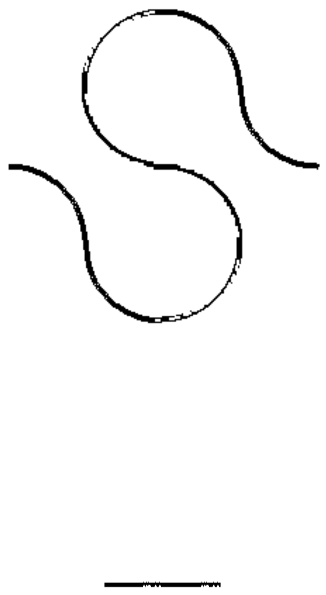
Graphic



Lean 49

```
-- first triangle identity of the prd-hom adjunction
/-
-/
```

Graphic



Lean 51

```
-- first triangle identity of the product-hom adjunction
/-
-/
```

Graphic

13. Prd X, τ, Δ

n-dimensional coordinates (arrays) are an ordinary construction, but we would do well to take careful note of its properties. The cartesian product `X × Y` records the information of two mathematical objects at once in the way that `x : X` and `y : Y` can be recovered from the pair `p : X× Y := (x, y)`as`x = p.fst` and `y = p.snd`. Above all, we might notice two particular other features which stand out about the cartesian product most: the terminal map `τ : X → ⊛`, which sends all elements of `X` to the only element of ⊛, and the diagonal map: `Δ : X → X × X`, `λ (x : X) => (x, x)`. These two maps and their properties reflect the so called *comonadicity* of cartesian product. Cartesian coordinates, while arising from the simple effort to combine pieces of information into array, already features the idiosyncracy of the diagonal and terminal maps.

Lean 53

```
-- ε : X × Y ⟶ Y
def Term (X : category) : (Prd X) ⟹ (1 Cat) := sorry
notation "ε" => Term
```

Graphic

Graphic for the counit of the Prd

Lean 55

```
-- Δ : X × Y ⟶ X × X × Y
def Diag (X : category) : (Prd X) ⟹ ((Prd X) • (Prd X)) :=
↪  sorry
```

Lean 56

```
-- notation for the comultiplication for product with X
notation "Δ" => Diag
```

Graphic

Graphic for the comultiplication of product with X

Lean 58

```
-- proof of the first identity law of the comultiplication
/-

-/
```

Graphic

Graphic for the first identity law of comultiplication

Lean 60

```
-- proof of the second identity law of the comultiplication
/-

-/
```

Graphic

Graphic for the second identity law of comultiplication

Lean 62

```
-- proof of the coassociativity of the comultiplication
/-

-/
```

Graphic

Graphic for coassociativity of the comultiplication

14. Hom X, ι, μ

Lean 64

```
-- Construction of the unit for Hom X
def Const : (1 Cat) ⥤ (Hom X) := sorry
```

Lean 65

```
-- notation
/-

-/
```

Graphic

Graphic for the unit for Hom X

Lean 67

```
-- Construction of the multiplication for [X, -]
def double : (Hom X) ⥤ (Hom X) • (Hom X) := sorry
```

Graphic

Graphic for the multiplication for Hom X

Lean 69

```
--
/-

-/
```

Graphic

Graphic for the first identity law of multiplication

Lean 71

Graphic

Graphic for the second identity law of multiplication

Lean 73

```
-- proving associativity for the comonad (Hom X)
/-

-/
```

Graphic

Graphic for associativity of the comultiplication

15. (Prd X) • (Prd Y) ≅ (Prd Y) • (Prd X)

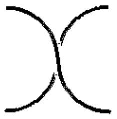
Lean 75

```
-- proof of the commutativity of categorical Prd
def Tw₁ (C : category) (D : category) : ((Prd C) • (Prd D)) ⟹
↪  ((Prd D) • (Prd C)) := sorry
```

Lean 76

```
-- notation "τ₁" => Tw₁
```

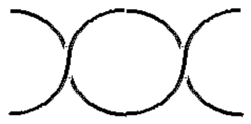
Graphic



Lean 78

```
-- proving that the twist map is its own inverse
-- def (C : category) (D : category) : (τ ∘ τ = (Idn (C × D)))
↪  := sorry
```

Graphic

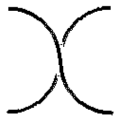


16. (Hom X) ● (Hom Y) ≅ (Hom Y) ● (Hom X)

Lean 80

```
-- defining the twist map (Hom X) • (Hom Y) ≅ (Hom Y) • (Hom
↪  X)
def Tw₂ (C : category) (D : category) : ((Hom C) • (Hom D)) ⟶
↪  ((Hom D) • (Hom C)) := sorry
-- notation "τ₂" => Twist
```

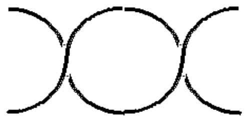
Graphic



Lean 82

```
-- proof that the twist map is its own inverse
-- def (C : category) (D : category) : (τ ∘ τ = (Idn (C × D)))
↪  := sorry
```

Graphic



17. ⊛

1. Defining the category ⊛
2. Notation for the category ⊛

Lean 84

```
-- defining the category ⊛
def PntObj : Type := Unit
def PntHom (_ : PntObj) (_ : PntObj) : Type := Unit
def PntIdn (X : PntObj) : PntHom X X := Unit.unit
def PntCmp (X : PntObj) (Y : PntObj) (Z : PntObj) (_ : PntHom
↪  X Y) (_ : PntHom Y Z) : PntHom X Z := Unit.unit
def PntId₁ (X : PntObj) (Y : PntObj) (f : PntHom X Y) : PntCmp
↪  X Y Y f (PntIdn Y) = f := sorry
def PntId₂ (X : PntObj) (Y : PntObj) (f : PntHom X Y) : PntCmp
↪  X X Y (PntIdn X) f = f := sorry
def PntAss (W : PntObj) (X : PntObj) (Y : PntObj) (Z : PntObj)
↪  (f : PntHom W X) (g : PntHom X Y) (h : PntHom Y Z) :
↪  PntCmp W Y Z (PntCmp W X Y f g) h = PntCmp W X Z f (PntCmp
↪  X Y Z g h) := sorry
def Pnt : category := {Obj := PntObj, Hom := PntHom, Idn :=
↪  PntIdn, Cmp := PntCmp, Id₁ := PntId₁, Id₂ := PntId₂, Ass :=
↪  PntAss}
```

Lean 85

```
-- notation for the category ⊛
notation "⊛" => Pnt
```

18. Prd ⊛ ≅ 1 Cat

1. Proving that (Prd ⊛) ≅ Idn
   1. (a) defining (Prd ⊛) ⥤ (**1** Cat)
   2. (b) defining (**1** Cat) ⥤ (Prd ⊛)
   3. (c) Proving the first inverse law for (Prd ⊛) ⥤ (**1** Cat) and (**1** Cat) ⥤ (Prd ⊛)
   4. (d) Proving the second inverse law for (Prd ⊛) ⥤ (**1** Cat) and (**1** Cat) ⥤ (Prd ⊛)

Lean 86

```
-- defining (Prd ⊛) ⥤ (1 Cat)
```

Lean 87

```
-- defining (1 Cat) ⥤ (Prd ⊛)
```

Lean 88

```
-- proving the first inverse identity
```

Lean 89

```
-- proving the second inverse identity
```

19. Hom ⊛ ≅ 1 Cat

1. Proving that Hom ⊛ ≅ Idn
   (a) Defining (Hom ⊛) ⥤ (𝟙 Cat)
   (b) Defining (𝟙 Cat) ⥤ (Hom ⊛)
   (c) Proving the first inverse law for (Hom ⊛) ⥤ (𝟙 Cat) and (𝟙 Cat) ⥤ (Hom ⊛)
   (d) Proving the second inverse law for (Hom ⊛) ⥤ (𝟙 Cat) and (𝟙 Cat) ⥤ (Hom ⊛)

Our next goal is to prove that Hom ⊛ is naturally isomorphic to **1** Cat.

Lean 90

```
-- defining (Hom ⊛) ⥤ (1 Cat)
-- def IdnHomObj (C : category) : Hom ⊛ C
-- def IdnHomHom
-- def IdnHomIdn
-- def IdnHomCmp
```

Lean 91

```
-- defining (1 Cat) ⥤ (Hom ⊛)
-- def IdnHom
-- def
-- def
-- def Idn
```

Lean 92

```
-- proving the first inverse identity
```

Lean 93

```
-- proving the second inverse identity
-- def Hom ⊛ ≅ X
```

20. Fox's Theorem

Lean 94

```
-- Defining the Prd F × G of two Hom one way
def FunPrd₁ {C₁ : category} {C₂ : category} {D₁ : category} {D₂
↪ : category} (F : C₁ ⟶ C₂) (G : D₁ ⟶ D₂) : (PrdObj C₁ D₁)
↪ ⟶ (PrdObj C₂ D₂) := sorry
```

Graphic

Graphic for the Prd of Hom

Lean 96

```
-- Defining the Prd F × G of two Hom the other way
def FunPrd₂ {C₁ : category} {C₂ : category} {D₁ : category} {D₂
↪ : category} (F : C₁ ⟶ C₂) (G : D₁ ⟶ D₂) : (PrdObj C₁ D₁)
↪ ⟶ (PrdObj C₂ D₂) := sorry
```

Graphic

Graphic for the Prd of Hom

Lean 98

```
-- Showing that the two Prds are equal
theorem FunPrdEqn {C₁ : category} {C₂ : category} {D₁ :
↪ category} {D₂ : category} (F : C₁ ⟶ C₂) (G : D₁ ⟶ D₂) :
↪ FunPrd₁ F G = FunPrd₂ F G := sorry
```

Lean 99

```
-- notation for the functor Prd
notation F "×" G => FunPrd₁ F G
```

Lean 100

```
-- Defining the canonical map in the universal property of Prd
-- def
```

Graphic

Graphic for the equivalence

Lean 102

```
-- Proving the uniqueness of the canonical map in the
↪  universal property of Prd
/-
theorem (uniqueness of the canonical map)
-/
```

Graphic

Graphic for the proof of Fox’s theorem

Lean 104

```
--
/-

-/
```

Python 1

```
from PIL import Image, ImageDraw
from PIL import ImageFont
from math import *
```

Python 2

```
def index(list):
    return range(len(list))
```

Python 3

```python
class vector:
    def __init__(self, entries):
        self.entries = entries
        self.length = len(entries)

    def __getitem__(self, i):
        return self.entries[i]

    def __add__(self, other):
        entries = []
        for i in range(len(self.entries)):
            entries.append(self[i] + other[i])
        return vector(entries)

    def __rmul__(self, a):
        entries = []
        for i in range(len(self.entries)):
            entries.append(a*self[i])
        return vector(entries)

    def height(self):
        return self.entries[1] - self.entries[0]

    def average(self):
        (1/float(len(self.entries)))*sum([self.entries])

    def print(self):
        print(self.entries)
```

Python 4

```python
def exp(theta):
    return vector([cos(theta), sin(theta)])
```

Python 5

```python
def interpolation(t):
    if 0 <= t and t <= 0.5:
        return 0.5 - sqrt(0.25-t*t)
    if 0.5 < t <= 1:
        s = 1-t
        return 0.5 + sqrt(0.25-s*s)
    return (-2.77333)*t*t*t + 4.16*t*t + (-0.386667)*t
```

Python 6

```
def curve(coordinates, border):
    for counter in range(len(coordinates) - 1):
        a = coordinates[counter][0]
        b = coordinates[counter+1][0]
        i = coordinates[counter][1]
        j = coordinates[counter+1][1]
        draw.line((j, b, i, a), fill=(256, 256, 256),
        ↪  width=12)
        draw.line((j, b, i, a), fill=(0, 0, 0), width=4)
    image.save('test.png')
```

Python 7

```
def squiggle(v1, v2, border):
    coordinates = []
    l = 100
    for r in range(l):
        t = float(r)/l
        s = interpolation(t)
        x = s*v1[0] + (1-s)*v2[0]
        y = t*v1[1] + (1-t)*v2[1]
        coordinates.append([x, y])
    curve(coordinates, border)
```

Python 8

```
def arc(v, r, theta1, theta2, u, border):
    n = floor((theta2 - theta1)*300)
    coordinates = []
    for i in range(n):
        t = float(float(i) * (theta2 - theta1)) / float(n)
        if u ==0:
            w = v +(-1)* r*exp(t)
        if u == 1:
            w = v + r*exp(t)
        coordinates.append(w)
    curve(coordinates, border)
```

Python 9

```
def bubble(v, color):
    draw.ellipse((v[1]-13, v[0]-13, v[1]+13, v[0]+13), fill =
    ↪  (0,0 ,0))
    draw.ellipse((v[1]-10, v[0]-10, v[1]+10, v[0]+10), fill =
    ↪  color)
```

Python 10

```
class cell:
    def __init__(self, entries, border, bubble, color):
        self.symbol = None
        self.entries = entries
        self.border = border
        self.x1    = entries[0]
        self.y11   = entries[1]
        self.y12   = entries[2]
        self.x2    = entries[0] +1
        self.y21   = entries[3]
        self.y22   = entries[4]
        self.y1 = self.y12 - self.y11
        self.y2 = self.y22 - self.y21
        self.width = 1
        self.bubble = bubble
        self.color = color

    def print(self):
        print("x1: ", self.x1)
        print("y11: ", self.y11)
        print("y12: ", self.y12)
        print("x2: ", self.x2)
        print("y21: ", self.y21)
        print("y22: ", self.y22)

    def copy(self):
        x1 = self.x1
        y11 = self.y11
        y12 = self.y12
        y21 = self.y21
        y22 = self.y22
        border = self.border
        bubble = self.bubble
        color = self.color
        return cell([x1, y11, y12, y21, y22], border, bubble,
        ↪  color)
```

Python 11

```
def copy_cell(l):
    cells = []
    for c in l:
        d = c.copy()
        cells.append(d)
    return cells
```

Python 12

```
def xshift(c, t):
    x1 = c.x1 + t
    x2 = c.x2 + t
    y11 = c.y11
    y12 = c.y12
    y21 = c.y21
    y22 = c.y22
    border = c.border
    bubble = c.bubble
    color = c.color
    return cell([x1, y11, y12, y21, y22], border, bubble,
     ↪  color)
```

Python 13

```
def yshift(c, y1, y2):
    y11 = c.y11 + y1
    y12 = c.y12 + y1
    y21 = c.y21 + y2
    y22 = c.y22 + y2
    x1 = c.x1
    border = c.border
    bubble = c.bubble
    color = c.color
    return cell([x1, y11, y12, y21, y22], border, bubble,
     ↪  color)
```

Python 14

```
class diagram:
    def __init__(self, heights, cells):
        self.color = color
        self.heights = heights
        self.width = len(heights)
        self.height = 0
        for h in self.heights:
            if h > self.height:
                self.height = h
        self.height = self.height
        self.cells = cells
```

Python 15

```
def __mul__(self, other):
    assert self.heights[-1] == other.heights[0]

    heights = self.heights[0:-1].copy() +
    ↪  other.heights.copy()

    cells = copy_cell(self.cells)
    for cell in copy_cell(other.cells):
        cells.append(xshift(cell, self.width - 1))

    return diagram(heights, cells)
```

Python 16

```
def __add__(self, other):
    cells = self.cells.copy()
    for i in range(len(other.cells)):
        x1 = other.cells[i].x1
        x2 = other.cells[i].x2
        h1 = self.heights[x1]
        h2 = self.heights[x2]
        cells.append(yshift(other.cells[i], h1, h2))
    p = len(self.heights)
    return diagram([self.heights[i] + other.heights[i] for
    ↪  i in range(p)], cells)
```

Python 17

```
def coordinate(self, v, unit,  W, H):
    h = unit*self.heights[v[0]]
    w = unit*self.width
    x = unit*float(v[0])
    y = unit*float(v[1])
    a = H/2-h/2 + unit*v[1]
    b = W/2-w/2 + unit*v[0] + unit/2
    return vector([a, b])
```

Python 18

```
def print(self, name, unit = 110, border = 0, radius =
↪  10):
    file_name = name + ".png"
    global image
    W = unit*self.width + 100
    H = unit*self.height + 100
    image = Image.new('RGBA', (W, H), (256,256,256))
    global draw
    draw = ImageDraw.Draw(image)
    for cell in self.cells:
        v11 = self.coordinate(vector([cell.x1, cell.y11]),
        ↪  unit, W, H)
        v12 = self.coordinate(vector([cell.x2, cell.y21]),
        ↪  unit, W, H)
        v21 = self.coordinate(vector([cell.x1, cell.y12]),
        ↪  unit, W, H)
        v22 = self.coordinate(vector([cell.x2, cell.y22]),
        ↪  unit, W, H)
        m1 = (1/2)*(v11 + v21)
        m2 = (1/2)*(v12 + v22)
        o = (1/2)*(m1 + m2)
```

Python 19

```
        if cell.y1 == 0 and cell.y2 == 1:
            squiggle(m1, o, cell.border)
            if cell.bubble == 1:
                bubble(o, self.color)
            image.save(file_name)
```

Python 20

```
        if cell.y1 == 0 and cell.y2 == 2:
            arc(m2, unit/2, -0.05, pi+0.05, 0,
            ↪  cell.border)
            if cell.bubble == 1:
                bubble(o+ vector([-unit/2, 0]),
                ↪  cell.color)
            image.save(file_name)
```

Python 21

```python
if cell.y1 == 1 and cell.y2 == 0:
    squiggle(o, m2, cell.border)
    if cell.bubble == 1:
        bubble(o, cell.color)
    image.save(file_name)
```

Python 22

```python
if cell.y1 == 1 and cell.y2 == 1:
    squiggle(m1, m2, cell.border)
    if cell.bubble == 1:
        bubble(o, cell.color)
    image.save(file_name)
```

Python 23

```python
if cell.y1 == 1 and cell.y2 == 2:
    a = (1/4)*(3*v22 + v12)
    b = (1/4)*(v22 + 3*v12)
    squiggle(o, a, cell.border)
    squiggle(o, b, cell.border)
    squiggle(m1, o, cell.border)
    p = vector([o[0], o[1]])
    if cell.bubble == 1:
        bubble(p, cell.color)
    image.save(file_name)
```

Python 24

```python
if cell.y1 == 2 and cell.y2 == 0:
    arc(m1, unit/2, -0.05, pi+0.05, 1,
     ↪  cell.border)
    if cell.bubble == 1:
        bubble(o + vector([unit/2, 0]),
         ↪  cell.color)
    image.save(file_name)
```

Python 25

```
if cell.y1 == 2 and cell.y2 == 1:
    a = (1/4)*(3*v21 + v11)
    b = (1/4)*(v21 + 3*v11)
    squiggle(a, o, cell.border)
    squiggle(b, o, cell.border)
    squiggle(o, m2, cell.border)
    p = vector([o[0], o[1]])
    if cell.bubble == 1:
        bubble(p, cell.color)
    image.save(file_name)
```

Python 26

```
if cell.y1 == 2 and cell.y2 == 2:
    w11 = (1/2)*(v11 + m1)
    w12 = (1/2)*(v12 + m2)
    w21 = (1/2)*(v21 + m1)
    w22 = (1/2)*(v22 + m2)
    if cell.border == 1:
        squiggle(w12, w21, 1)
        squiggle(w11, w22, 1)
    if cell.border == 2:
        squiggle(w11, w22, 1)
        squiggle(w12, w21, 1)
    if cell.bubble == 1:
        bubble(o, cell.color)
    image.save(file_name)
```

Python 27

```
def make_cell(i, j, color =(0, 0, 0), border = 0, bubble =0):
    return diagram([i, j], [cell([0, 0, i, 0, j], border,
    ↪  bubble, color)])
```

Python 28

```
blue = (102, 178, 255)
green = (0, 204, 102)
yellow = (250, 250, 50)
brown = (30, 150, 69)
color = blue
```

Python 29

```
counit = make_cell(2, 0, blue,1, 0)
unit = make_cell(0, 2, blue, 1, 0)
Unit = make_cell(0, 1, blue, 1, 0)
Counit = make_cell(1, 0, blue, 1, 0)
Multiplication = make_cell(2, 1, blue, 1, 0)
Comultiplication = make_cell(1, 2, blue, 1, 0)
IdF = make_cell(1, 1, blue, 1, 1)
IdG = make_cell(1, 1, blue, 1, 0)
identity = make_cell(1, 1, blue, 0, 0)
Ltriangle = (unit + IdG)*(IdF + counit)
Rtriangle = (IdG + unit)*(counit + IdF)
Lbraid = make_cell(2, 2, blue, 1, 0)
Rbraid = make_cell(2, 2, blue, 2, 0)
a1 = unit+identity+identity
a2 = identity + counit +identity
a3 = unit + identity+identity
a4 = identity + identity + counit
Multiplication.print("test10")

#To do:
#Make each cell have its own color
#Get each cell to have a plus or a minus sign
#Strings with a grade of color for permeable membranes
#Strings which are colored or which stand for terms
```

asdfasdfasdf $\}$— asdfasdf

Chapter II: the (strict) twocategory of categories

Section	Description
`twocategory`	the (strict) twocategory structure
`Two`	the twocategory of categories
$\bullet$	notation for horizontal composition

21. twocategory

Lean 105

```
-- definition of a (strict) twocategory
structure twocategory where
  TwoObj : Type
  TwoHom : TwoObj → TwoObj → category
  TwoIdn : (C : TwoObj) → ⊛ ⟶ (TwoHom C C)
  TwoCmp : (C : TwoObj) → (D : TwoObj) → (E : TwoObj) →
  ↪  (PrdObj (TwoHom C D) (TwoHom D E)) ⟶ (TwoHom C E)
--  TwoId₁ : (C : Obj) → (D : Obj) → (TwoCmp C D D) ● ((Idn 1)
↪  × (𝟙 )) =
--  TwoId₂ : (C : Obj) → (D : Obj) → (Cmp C C D) ● ((Idn D) ×
↪  1) =
--  Ass : (B : Obj) → (C : Obj) → (D : Obj) → (E : Obj) →
↪  (((Cmp B C E) ● (FunPrd₁ (𝟙 (Hom B C)) (Cmp C D E))) = (Cmp
↪  B D E ● (FunPrd₁ (Cmp B C D) (𝟙 (Hom D E)))))
```

Lean 106

```
-- notation for a twocategory
/-

-/
```

22. Two

Next we define "categories", the (strict) twocategory of categories. We have already defined categories.Obj := category and categories.Hom := functor, so we may start by defining the Idn component:

Lean 107

```
-- defining categories.Idn.Obj
def TwoIdnObj (C : category) (_ : Unit) := Cat.Idn C
```

Lean 108

```
-- defining the functor categories.Idn.Hom on morphisms
def TwoIdnHom (C : category) (_ : Unit) (_ : Unit) (_: Unit)
↪  := (HomObj C C).Idn (Cat.Idn C)
```

Lean 109

```
-- proving the identity law for the functor categories.TwoIdn
-- def TwoIdnIdn (C : category) (_ : Unit) (_ : Unit) (_:
↪  Unit) := (HomObj C C).Idn (Cat.Idn C)
```

Lean 110

```
-- proving compositionality for the functor categories.TwoIdn
-- def Two.Idn.Cmp (C : category) (_ : Unit) (_ : Unit) (_:
↪  Unit) := (HomObj C C).Idn (Cat.Idn C)
```

Lean 111

```
-- def categories.Idn
def TwoIdn (C : category) : ⊛ ⟶ (HomObj C C) := sorry
```

Next we define Two.Cmp:

Lean 112

```
--  defining Two.Cmp.Obj
/-
-/
```

Lean 113

```
--  defining Two.Cmp.Hom
/-
def TwoTwoHom (C : Obj) (D : Obj) (E : Obj)  :
↪  FG.1 FG.2
def TwoTwoHom (C : Obj) (D : Obj) (E : Obj) (f :
↪  ((Hom C D) × (Hom D E)).Hom )
def CatsHom (C : Obj) (D : Obj) (E : Obj)
(F₁G₁ : ((Hom C D) × (Hom D E)).Obj) (F₂G₂ : ((Hom
↪  C D) × (Hom D E)).Obj)
-/
```

Lean 114

```
-- proving the identity law equation for Two.TwoCmp
/-
def
-/
```

Lean 115

```
-- proving compositionality for the functor Two.Cmp
-- def TwoCmpCmp : (C : category) → (D : category) → (E :
↪  category) → (PrdObj (HomObj C D) (HomObj D E)) ⟶ (HomObj C
↪  E) := sorry
```

Lean 116

```
--  Two.Cmp : (C : Obj) → (D : Obj) → (E : Obj) → (Hom C D) ×
↪  (Hom D E) ⟶ (Hom C E)
def TwoCmp : (C : category) → (D : category) → (E : category)
↪  → (PrdObj (HomObj C D) (HomObj D E)) ⟶ (HomObj C E) :=
↪  sorry
```

Now that we have constructed the first four constituents of the twocategory Two, we proceed to prove that they satisfy the three conditions Id_1, Id_2, and Ass:

Lean 117

```
-- Id₁ : (C : Obj) → (D : Obj) → (Cats.Id₁)
/-
def TwoId₁ : (C : category) → (D : category) → (F
↪ : functor C D) →
-/
```

Lean 118

```
-- Id₂ : (C : Obj) → (D : Obj) → (F : (Hom C D).Obj) → ...
↪ (Cats.Id₁)
/-
def TwoId₂ : (C : category) → (D : category) → (F
↪ : functor C D) →
-/
```

Lean 119

```
-- proving associativity of composition for the twocategory of
↪ Two
/-
def TwoAss
-/
```

23. •

Lean 120

```
-- notation for horizontal composition
/-
class horizontal_composition (C : category) (D :
↪  category) (E : category) (F₁ : C ⟶ D) (F₂ : C →
↪  D) (G₁ : D → D) (G₂ : D → E) where
  f : (F₁ ⟹ F₂) → (G₁ ⟹ G₂) → ((G₁ ● F₁) ⟹ (G₂ ● F₂))

def f (p : Prop) : Prop := ¬p
def g (n : Nat): Nat := n + 1
-/
```

Lean 121

```
/-
class Elephant (T : Type) where
  fn : T → T

instance prop_elephant : Elephant Prop where
  fn := f

instance int_elephant : Elephant Nat where
  fn := g

def elephant {T : Type} [E : Elephant T] (t : T) :
↪  T := E.fn t

#check elephant (2 : Nat)
#reduce elephant (2 : Nat)
#eval elephant (2 : Nat)

#check elephant True
#reduce elephant True

#check elephant (0 : Nat)

notation "" t => elephant t
#eval  (2 : Nat)
-/
```

Lean 122

```
/-
class composition (C : category) (D : category)
↪  (F : functor C D) (X : Type p₁) (Y : Type p₂) (T
↪  : Type p₁ → Type p₂ → Type p₃) (Z : T X Y) where
  f : X → Y → Z

instance functor_application_on_objects (C :
↪  category) (D : category) : composition
↪  (functor C D) C.Obj (Type p₃) D.Obj where
  f := fun(F : functor C D) => fun(X : C.Obj) =>
↪  F.Obj X

instance functor_application_on_morphisms (C :
↪  category) (D : category) (X : C.Obj) (Y :
↪  C.Obj) : composition (functor C D) () () where
  f :=

instance functor_composition

instance natural_transformation_whisker₁

instance natural_transformation_whisker₂

instance horizontal_composition

-/

/-
notation X × Y => horizontal_composition X Y


notation "" t => elephant t
#eval  (2 : Nat)
-/
```

Chapter III: the Yoneda lemma

Section	Description
Set	the category of sets
よ	the Yoneda embedding and the Yoneda lemma

24. よ

Lean 123

```
-- definition of the yoneda embedding
def yoneda_embedding (C : category) : Cᵒᵖ ⟶ Set := sorry
```

Lean 124

```
-- notation for the Yoneda embedding
notation "よ" => yoneda_embedding
```

Lean 125

```
-- definition of the contravariant yoneda embedding
/-

-/
```

Lean 126

```
/-
def (C : category) (F : Cᵒᵖ ⟹ Set) : [X, -] ⟹ F ≅
↪  F • X := sorry
-/
```

Lean 127

```
/-
def (C : category) (F : Cᵒᵖ ⟹ Set) : [X, -] ⟹ F ≅
↪  F • X := sorry
-/
```

Lean 128

```
/-
def ([X, -] ⟹ [Y, -]) ≅ [X, Y]
-/
```

Lean 129

```
/-
def ([-, X] ⟹ [-, Y]) ≅ [Y, X]
-/
```

Lean 130

```
-- corollary: the Yoneda embedding is full
/-

-/
```

Lean 131

```
-- corollary: the Yoneda embedding is faithful
/-

-/
```

Lean 132

```
-- corollary: the contravariant Yoneda embedding is full and
↪  faithful
/-

-/
```

Chapter IV: adjunctions, monads, and comonads

Section	Description
`adjunction`	the adjunction structre
`monad`	the monad structure
`comonad`	the comonad structure
`monadicity`	the monadicity of an adjunction
`comonadicity`	the comonadicity of an adjunction

25. adjunction

The relationship that `Hom C` and `Prd C` are in has been depicted with triangle identities, which amounted to pulling a string tight in the string calculus. Now we will analyze the relationship that `Hom C` and `Prd C` had in detail. The graphical depiction of the triangle identities was mentioned, and so we start our analysis there. Some amount of inspection suggests a more general sitation than before:

Lean 133

```
-- definition of an adjunction
structure adjunction where
  C : category
  D : category
  F  : C ⟶ D
  G  : D ⟶ C
  Unit  : (1 C) ⟹ (G ● F)
  Counit  : (F ● G) ⟹ (1 D)
 -- τ₁  : ((1 F) ● η)  =  ((1 F) ● η)   -- ○ (Iso (ℂ.Hom Dom
 ↪  Cod) (Ass F G F)) ○ (((CatHom C D).Idn F) ● η)) = (CatHom
 ↪  D C).Idn left
--  τ₂  : (𝟙 F) = (𝟙 F)   -- ○ (Iso (ℂ.Hom Dom Cod) (Ass F G
↪  F)) ○ (((CatHom C D).Idn F) ● η)) = (CatHom D C).Idn left
```

This is the `adjunction` structure. Here is some notation we will use for it:

Lean 134

```
-- notation for an adjunction
/-
notation C "" D => adjoint C D --adjoint symbol
def F (U : TwoCat) {C : U.Obj} {D : U.Obj} (f :
↪  Adj C D) := f.F

notation f "˙" => F f

def G (U : TwoCat) {C : U.Obj} {D : U.Obj} (f :
↪  Adj C D) := f.G
notation f "." => G f

def adjoint {C : category} {D : category} (F :
↪  )...

notation F "⊢" G => adjoint
-/
```

We depict adjunctions graphically just as we did for `Prd C` and `Hom C`.

26. monad

Lean 135

```
-- definition of a monad
structure monad where
  C : category
  T : C ⟶ C
  Unit : (𝟙 C) ⟹ T
  Mult : (T • T) ⟹ T
--  Id₁  : μ ∘ (η • (𝟙 T)) = 𝟙 T
--  Id₂  : μ ∘ ((𝟙 T) • η) = 𝟙 T
--  Ass  : μ ∘ (μ • (𝟙 T)) = μ ∘ ((𝟙 T) • μ)
```

Lean 136

```
-- notation for a monad
/-
-- notation for monad application
instance comonad_application {C : CatObj} :
↪  horizontalCmp (Com C) (Obj C) (Obj C) where
  ϕ := fun(T₀ : Com C)=>fun(X₀ : Obj
↪  C)=>(T₀.functor.Obj X₀)
-/
```

27. comonad

Lean 137

```
-- definition of a comonad (shouldn't depend on a twocat)
structure comonad where
  C : category
  T : C ⟶ C
  Counit : T ⟹ (1 C)
  Comult : T ⟹ (T • T)
--  Id₁  : (Unt × (Idn T)) • Comul  = (Idn T)
--  Id₂  : ((Idn T) × Unt) • Comul  = (Idn T)
--  Ass  : (Mul × (Idn T)) • (Idn T) = ((Idn T) × Mul) • (Idn
↪  T)
```

Lean 138

```
-- notation for a comonad
/-
def Unit {C : category} (M : comonad C) :=
↪  M.Counit
notation "τ" M => Counit M

def ult {C : category} (M : comonad C) :=
↪  M.Comult
notation "δ" M => Mlt M

-- notation for monad application
instance comonad_application {C : category} :
↪  horizontalCmp (Com C) (Obj C) (Obj C) where
  ϕ := fun(T₀ : Com C)=>fun(X₀ : Obj
↪  C)=>(T₀.functor.on_objects X₀)
-- τ₁
-- τ₂
-- γ
-/
```

28. monadicity

Lean 139

```
-- the monad corresponding to an adjunction
-- def
```

Lean 140

```
-- notation for the monad corresponding to an adjunction
/-
notation
-/
```

Lean 141

```
-- canonical map from eilenberg moore category of the
↪  corresponding monad for an adjunction
/-
def
-/
```

Lean 142

```
-- notation for the canonical map from eilenberg moore
↪  category of the corresponding monad for an adjunction
/-
notation ?
-/
```

Lean 143

```
-- the eilenberg moore adjunction unit
/-
def
-/
```

Lean 144

```
-- eilenberg moore adjunction triangle identity 1
/-
theorem
-/
```

Lean 145

```
-- eilenberg moore adjunction triangle identity 2
/-
theorem
-/
```

Lean 146

```
-- LEAN: def when ! is an iso (monadicity)
/-
def Monadic (f : Adj) : Prop := sorry
-/
```

Lean 147

```
-- defining premonadicity
/-
def Premonadic (f : Adj) : Prop := sorry
-/
```

29. comonadicity

Lean 148

```
-- the comonad corresponding to an adjunction
/-
def
-/
```

Lean 149

```
-- notation for the comonad corresponding to an adjunction
/-
notation !
-/
```

Lean 150

```
-- canonical map into the coeilenberg comoore category of the
↪  corresponding comonad
/-
def
-/
```

Lean 151

```
-- notation for canonical map into coeilenberg comoore
↪  category of the corresponding monad for an adjunction
/-
notation ?
-/
```

Lean 152

```
-- the coeilenberg comoore adjunction unit
/-
def
-/
```

Lean 153

```
-- the coeilenberg comoore adjunction counit
/-
def
-/
```

Lean 154

```
-- coeilenberg comoore adjunction triangle identity 1
/-
theorem
-/
```

Lean 155

```
-- coeilenberg comoore adjunction triangle identity 2
/-
theorem
-/
```

Lean 156

```
-- defining when ? is an iso (comonadicity)
/-
def Comonadic (f : Adj) : Prop := sorry
-/
```

Lean 157

```
-- defining precomonadicity
/-
def Precomonadic (f : Adj) : Prop := sorry
-/
```

About the Author

Elliot Young is a graduate student at New York University, where he studies pure and applied mathematics.