MyMathlibProject/AlgebraicStructure.lean

```lean
import Mathlib
set_option linter.style.emptyLine false
/-
This file demonstrate how to define an algebraic structure
in Lean 4. We start with basic element like Group, Ring
and Filed. Then we will extend our focus to more abstract
algebraic content say category and functor.
-/

-- So far : Group (non-commutative : a * b ≠ b * a in general)

/-
You could find my personal website here:
https://wanghongwei-academicpage.github.io/
Im so glad if you could contact me if you find any mistakes or just wanna discuss
some interesting ideas with me :)

Feb 3, 2026
-/
/-
## Groups
-/

-- f ((a , b)) ↦ c
-- f ∘ (g ∘ h)

/-
A set G is a group if there is binary operation *   G × G → G : g₁ * g₂ = g₃
1. assoc (a * b) * c = a * (b * c)
2. id: e * a = a and a * e = a
3. in: ∀ a ∈ S, ∃ a^⁻¹ such that a^⁻¹ * a = e
-/


class Group₁ (α : Type*) where
-- Here `Type*` means this structure can exsits in any universe. can we
-- claim our structure called `α `.

  mul : α → α → α
  -- (α × α) → α
  one : α
  inv : α → α

-- Once we have our stucture name and universe, we should give out all
-- `elements' type ` in this structure. One thing you should notice
-- that `opreation(mul), id, in` all got there type based on `α`. which
-- means here we consider the opearation as one of the element. Or,
-- we can consider any elements in group besides `id` and `in` are the
-- results of other two elements.


```

```
  mul_assoc : {x y z : α} →  mul (mul x y) z = mul x (mul y z)
  mul_one : {x : α} →  mul x one = x
  one_mul : {x : α} →  mul one x = x
  inv_mul_one : {x : α} →  mul (inv x) x = one

-- After we claim all the type of `elements` in this structure, then
-- we need to tell Lean how these terms works out with others, based
-- on the Group def.

#check Ring


example {G : Type*} [Group₁ G] (a : G) : Group₁.mul a a = Group₁.one :=
  by sorry


variable {G H : Type 0} [Group₁ G] [Group₁ H] (g₁ : G) (h₁ h₂ : H)



/-
class Matrix

mul
theorem M_n2isagroup {G : Type*} [Matrix G] : G → Group :=
  by sorry
-/



/--
## Rings




Def(Ring): We say a set `R` is a ring if
1. (R, +) is an Abelian (commutative) Group (a + b = b + a)
2. (R, *) is a monoid
3. `*` is distributes over `+`
-/
-- [Group₁ α]




class Ring₁ (α : Type*) where
  add : α → α → α
  zero : α
  neg : α → α

  add_assoc : {x y z : α} → add (add x y) z = add x (add y z)
  add_zero : {x : α} → add x zero = x
  zero_add : {x : α} → add zero x = x
```

```
  neg_add_zero : {x : α} → add (neg x) x = zero
  add_comm : {x y : α} → add x y = add y x


  mul : α → α → α
  one : α

  mul_assoc : {x y z : α} → mul (mul x y) z = mul x (mul y z)
  mul_one : {x : α} → mul x one = x
  one_mul : {x : α} → mul one x = x


  left_dis :
    {x y z : α} → mul x (add y z) = add (mul x y) (mul x z)

  right_dis:
    {x y z : α} → mul (add x y) z = add (mul x z) (mul y z)



-- Before we define Fields, lets do it more clever
/-
Now you may see more clearly how the ring is constructed by groups:
1. We copy the definition of group under `+` equipped commutativity.
2. We copy the definition of group under `*` cancel everthing about `*`
3. `*` is distribute over `+`
So can we use `Group₁` to define `Ring₁` just by copy it twice and
add or cancel some of there defs?
The answer is `Yes`! Because we have `extend` which can inherite all
the definition already have. But `No` because `extend` is directely
copy all and we are not able to edit after we define.
So we need some structure smaller
-/


/-
## Monoids
-/

/-
Def (Monoids): We say a Set S is a monoid if it satisfies two axioms above with
some binary operation `*`:
1. Assoc: ∀ x y z ∈ S, (x * y) * z = x * (y * z)
2. Id : ∃ x ∈ S, s.t. ∀ y ∈ S, x * y = y and y * x = y
-/


structure MulMonoid₁ (α : Type*) where
  mul : α → α → α
  one : α

  mul_assoc : {x y z : α} → mul (mul x y) z = mul x (mul y z)
  one_mul : {x : α} → mul one x = x
  mul_one : {x : α} → mul x one = x
```

```
/-
ext [x, y] → x , y
add_x : α → α → α
zero_x : α
add_y : α → α → α
zero_y : α
-/
structure AddMonoid₁ (α : Type*) where
  add : α → α → α
  zero : α

  add_assoc : {x y z : α} → add (add x y) z = add x (add y z)
  zero_add : {x : α} → add zero x = x
  add_zero : {x : α} → add x zero = x

/-
Now you can see the power of `extend`
-/

structure MulGroup₁ (α : Type*)
  extends MulMonoid₁ α where
  inv : α → α
  inv_mul_cancel : {x : α} → mul (inv x) x = one


structure AddGroup₁ (α : Type*)
  extends AddMonoid₁ α where
  neg : α → α
  neg_add_cancel : {x : α} → add (neg x) x = zero


structure AddCommGroup₁ (α : Type*)
  extends AddGroup₁ α where
    add_comm : {x y : α} → add x y = add y x


/-
Ring `R` is Abliean group under `+` and Monoid under `*`, with `*`distributes on `+`
-/

/-

-/


#check Ring

structure ExtRing₁ (α : Type*)
  extends AddCommGroup₁ α, MulMonoid₁ α where
    left_distrib : {x y z : α} →
      mul x (add y z) = add (mul x y) (mul x z)
```

```
    right_distrib : {x y z : α} →
      mul (add x y) z = add (mul x z) (mul y z)

structure ExtCommRing₁ (α : Type*)
  extends ExtRing₁ α where
    mul_comm : {x y : α} → mul x y = mul y x
/-
CommRing ⊆ Ring ⊆ AddComm_Group ⊆ Add_Group ⊆ Add_Monoid
CommRing ⊆ Ring ⊆ Mul_Group ⊆ Mul_Monoid
-/

-- Now we can see the boss of our game

/-
## Fields
-/

/-
Def (Field): We say a set `F` is a field if there exsits two
element `add_identity_zero: 0 ∈ F, mul_identity_one: 1 ∈ F` and satisfies
these axioms below with two binary opeartion `+` and `*`

1. add_assoc : ∀ x, y, z ∈ F, (x + y) + z = x + (y + z)
2. add_comm : ∀ x, y ∈ F, x + y = y + x
3. add_zero_equal_comm : ∀ x ∈ F, x + 0 = 0 + x = x
4. add_neg_cancel_comm : ∀ x ∈ F, ∃ -x ∈ F, s.t.
  x + -x = -x + x = 0

5. mul_assoc : ∀ x, y, z ∈ F, (x * y) * z = x * (y * z)
6. mul_comm : ∀ x, y ∈ F, x * y = y * x
7. mul_one_equal_comm : ∀ x ∈ F, x * 1 = 1 * x = x
8. mul_inv_cancel_comm : ∀ x ∈ F \ {0}, ∃ x⁻¹ ∈ F, s.t.
  x * x⁻¹ = x⁻¹ * x = 1

9. dis_mul_add : ∀ x, y, z ∈ F, x * (y + z) = x * y + x * z

10. zero_neq_one : 0 ≠ 1
-/

-- Now lets formalize this definition flattly.
structure Fields₁ (α : Type*) where
  zero : α
  one : α
  add : α → α → α
  mul : α → α → α

  add_assoc : {x y z : α} → add (add x y) z = add x (add y z)
  add_comm : {x y : α} →  add x y = add y x
  add_zero : {x : α} → add x zero = x
  neg : α → α
  add_neg_cancel : {x : α} → add x (neg x) = zero

  mul_assoc : {x y z : α} → mul (mul x y) z = mul x (mul y z)
  mul_comm : {x y : α} → mul x y = mul y x
```

```
  mul_one : {x : α} → mul x one = x
  inv : α → α
  mul_inv_cancel : {x // x ≠ zero} → mul x (inv x) = one

  dis_mul_add : {x y z : α} →
    mul x (add y z) = add (mul x y) (mul x z)

  zero_neq_one : zero ≠ one

#check Fields₁



-- Now lets use `extends`

/-
We say a Ring `F` is a field if `(F \ {0}, *)` is a
commutative Group where `0 ≠ 1`
-/

structure ExtFields₁ (α : Type*)
  extends ExtCommRing₁ α where


  inv : {x // x ≠ zero} → {x // x ≠ zero}
  mul_inv_cancel : {x // x ≠ zero} → mul x.1 (inv x) = one

  zero_neq_one : 0 ≠ 1

-- mul  α →   α\ {0} → ?
-- mul α → α →  α






-----------------------------------------
-----------------------------------------
-----------------------------------------
-----------------------------------------
-----------------------------------------
-----------------------------------------



/-!
## Monoids
We begin our journey of algebra by introducing Monoids.

Courses in abstract algebra often start with groups and then progress to rings, fields, and vector spaces. This involves
some contortions when discussing multiplication on rings since the multiplication operation does not come from a group
```

```lean
structure but many of the proofs carry over verbatim from group theory to this new setting. The most common fix,
when doing mathematics with pen and paper, is to leave those proofs as exercises. A less efficient but safer and more
formalization-friendly way of proceeding is to use monoids. A monoid structure on a type M is an internal composition
law that is associative and has a neutral element. Monoids are used primarily to accommodate both groups and the
multiplicative structure of rings. But there are also a number of natural examples; for instance, the set of natural numbers
equipped with addition forms a monoid

-/
example {M : Type*} [Monoid M] (x : M) : x * 1 = x :=
  mul_one x

example {M : Type*} [AddCommMonoid M] (x y : M) : x + y = y + x :=
  add_comm x y

/-
The type of morphisms between monoids M and N is called MonoidHom M N and written M →* N.
Lean will automatically see such a morphism as a function from M to N when we apply it to
elements of M. The additive version is called AddMonoidHom and written M →+ N.
-/

example {M N : Type*} [Monoid M] [Monoid N] (x y : M) (f : M →* N) :
  f (x * y) = f x * f y := f.map_mul x y



-- variable {M N : Type*} [Monoid M] [Monoid N] (x y : M) (f : M →* N)


-- #check (M →* N)
-- Also the maps between monoids also follow the composition as `AddMonoidHom.map`.

--example {M N P : Type*} [Monoid M] [Monoid N] [Monoid P] (f : M →+ N)
-- (g : N →+ P) : M →+ P := by exact g comp.f


/-
## Groups
Now we will see how to use the content of Group structure, which is
already been defined in Mathlib4
You can use the definition we used last lecture to define a group, as
give out all the axioms of gorups. But we coul do this better, by what we
have known about the stucture to Monoind. The fact that a group is actually
a monoind with the mulitple inverse tructure allowed/
-/
#check Group

example {G : Type*} [Group G] (x : G) : x * x⁻¹ = 1 :=
```

```lean
  mul_inv_cancel x

#check mul_inv_cancel
#check Group

-- You could go the definition of group in Mathlib4 to see how we define
/-
 class Group (G : Type u) extends DivInvMonoid G where
   protected inv_mul_cancel : ∀ a : G, a⁻¹ * a
-/


-- If you still remember the tactic `ring`, we can also use `group`  or `abel` to prove anything about the group or commutative gorup structures' identity.

example {G : Type*} [Group G] (x y z : G) : x * (y * z) * (x * z)⁻¹
* (x * y * x⁻¹)⁻¹ = 1 :=by
  group

example {G : Type*} [AddCommGroup G] (x y : G) : x + y = y + x := by
  abel

#check AddCommGroup


-- The morphims betwwen the groups is just the maps betwwen Moniond

example {G H : Type*} [Group G] [Group H] (x y : G) (f : G →* H) :
  f (x * y) = f x * f y := by
    exact f.map_mul x y

-- But by the inverse structure we do get some more properties

example {G H : Type*} [Group G] [Group H] (f : G →* H) (x : G) :
  f (x⁻¹) = (f x)⁻¹ := by
    exact f.map_inv x

/-
There is also a type `MulEquiv` of groups or monoids isomorphisms denoted by
`≃*` and `≃+`. The inverse of `f : G ≃* H` is `MulEquiv.symm f : H ≃*G`
-/

example {G H : Type*} [Group G] [Group H] (f : G ≃* H) :
  f.trans f.symm = MulEquiv.refl G :=
    f.self_trans_symm

/-
Isomorphim can be considered as a bijective function betwwen two groups,
thus ,we can build an iso as below (doing so makes the inverse noncomputable)
-/
noncomputable example {G H : Type*} [Group G] [Group H]
  (f : G →* H) (h : Function.Bijective f) :
    G ≃* H :=
      MulEquiv.ofBijective f h
```

```lean

#check MulEquiv.ofBijective




/-
## Subgroups
A subgrps of `G` is also a bundled structure consisting of a set `G` with
the relevant closure properties
-/

example {G : Type*} [Group G] (H : Subgroup G) {x y : G}
 (hx : x ∈ H) (hy : y ∈ H) : x * y ∈ H :=
  H.mul_mem hx hy

#check mul_mem


/-
One thing should be remarked that `Subgroup G` is the type of subgrps of `G`, not a predicate `IsSubgroup H` where `H` is an element of `Set G.Subgroup G`
To show two subgroups are the same if and only if they have the same elements.
-/

/-
For instance, `ℤ` is a subgroup of `ℚ`, what we really want is to construct a term of type `AddSubgroup ℚ` whose projection to `set ℚ` is `ℤ`.


example : AddSubgroup ℚ where
  carrier := Set.range ((↑) : ℤ  → ℚ )
  add_mem' := by
    rintro _ _ ⟨n, rfl⟩ ⟨m, rfl⟩
    use n + m
    simp
  zero_mem' := by
    use 0
    simp
  neg_mem' := by
    rintro _ ⟨n, rfl⟩
    use -n
    simp

-/


-- Mathlib knows that a subgrp of a group automatically inherits
-- the group structure.

example {G : Type*} [Group G] (H : Subgroup G) :  Group H := by
  exact inferInstance

example {G : Type*} [Group G] (H : Subgroup G) :
```

```
  Group {x : G // x ∈ H} := by
    infer_instance

#check inferInstance

/-
Now lets check that the set underlying the infimum of two subgrps is indeed,
by definition, their intersection.
-/

example {G : Type*} [Group G] (H H' : Subgroup G) :
  ((H ⊓ H': Subgroup G) : Set G) =
    (H : Set G) ∩ (H' : Set G) := rfl

-- And the supremum opreation

example {G : Type*} [Group G] (H H' : Subgroup G) :
  ((H ⊔ H' : Subgroup G) : Set G) =
    Subgroup.closure ((H : Set G) ∪ (H' : Set G)) := by
      rw [Subgroup.sup_eq_closure]



-----------------------------------------------------------------

/-
## Ring: units, morphisms and subrings
-/

/-
The type of ring structure on a type `R` is `Ring R`, and if the ring is
abelian we have `CommRing R`. As we have seen before, `ring` is a powerful
tactic when we dealing with the proof of identity of some elements of `ℤ`,
`ℝ` and Àna
-/

example {R : Type*} [CommRing R] (x y : R) :
  (x + y) ^ 2 = x ^ 2 + 2 * x * y + y ^ 2 := by
    ring


/-
The tactic `ring` is more powerful that you think, it can prove
the identity defined on a structure that even not a ring itself.
It only requires the addition on `R` forms an additive monoind.
In this condition, the type class `ring R` deforms back to `Semiring R`
or `CommSemiring R`.
-/

-- The most typicial example of `Semiring` is `ℕ `

example (x y : ℕ) : (x + y) ^ 2 = x ^ 2 + 2 * x * y + y ^ 2 := by
  ring

```

```lean
/-
When we use the tactic `ring`, the tacitc will check the type class of
input variable `x` and `y`. As long as the type is `Semiring` the tactic
can be processd.
But one thing needs to be remarked is that the `Semiring` is a algebraic
structure is a monoid under `addition`, any identity of `subsitution` can
not be applied
-/

example (x y : ℕ) :(x - y) ^ 2 = x ^ 2 - 2 * x * y + y ^ 2 := by
  --ring
  sorry

-- Same as monoids and groups

example {R S : Type*} [Ring R] [Ring S] (f : R →+* S) (x y : R) :
  f (x + y) = f x + f y := f.map_add x y

example {R S : Type*} [Ring R] [Ring S] (f : R →+* S) :
  Rˣ →* Sˣ   := Units.map f

example {R : Type*} [Ring R] (S : Subring R) : Ring S :=
  inferInstance


/-
## Ideals and Quotients

Mathlib4 only got the theory of ideals for commutative rings, so this section all our works will assume the commutativity
-/

/-
To make a quotient ring, we have to give out the ideal `I` and use
`Ideal.Quotient.mk I` or `I.Quotient.mk` with the dot notation.
-/


-- And the quotient ring can be considered as a surjective map

namespace Ideal.Quotient

example {R : Type*} [CommRing R] (I : Ideal R) : R →+* R ⧸ I :=
  mk I

example {R : Type*} [CommRing R] {a : R} {I : Ideal R} :
  mk I a = 0 ↔ a ∈ I :=
    eq_zero_iff_mem

-- The universal property of quotient ring is `Ideal.Quotient.lift`


example {R S : Type*} [CommRing R] [CommRing S] (I : Ideal R) (f : R →+* S)
  (hI : I ≤ RingHom.ker f) : R ⧸ I →+* S :=
```

```
    lift I f hI

#check lift

-- This exactly lifts to the `first iso theorem of rings`.
noncomputable example {R S : Type*} [CommRing R] [CommRing S] (f : R →+* S) :
  R ⧸ RingHom.ker f ≃+* f.range := by
    simpa using RingHom.quotientKerEquivRange f


/-
Ideals form a complete lattice structure with inclusion, as well as a
semiring structure
-/

variable {R : Type*} [CommRing R] {I J : Ideal R}

example : I + J = I ⊔ J := rfl
example {x : R} : x ∈ I + J ↔ ∃ a ∈ I, ∃ b ∈ J, a + b = x := by
  simp [Submodule.mem_sup]
example : I * J ≤ J := Ideal.mul_le_left
example : I * J ≤ I := Ideal.mul_le_right
example : I * J ≤ I ⊓ J := Ideal.mul_le_inf


/-
We can use ring homos to push ideals forward and pull them back by
using `Ideal.map` and `Ideal.comap`.'
-/

example {R S : Type*} [CommRing R] [CommRing S]
  (I : Ideal R) (J : Ideal S) (f : R →+* S)
  (h : I ≤ Ideal.comap f J) : R ⧸ I →+* S ⧸ J :=
    Ideal.quotientMap J f h




#check Ideal.comap
#check Ideal.quotientMap


end Ideal.Quotient






/-
This is the file written by Hongwei.Wang in winter 2026,
this file is the basic formalization of Category Theory in
Pure Mathematics.

You can find my personl homepage here and it is my pleasure
```

```lean
if you can contact me if you find any mathemaical mistakes
or typos in this file. Also, please feel free to contact me
if you just want to discuss your idea with me.


Personal Homepage:
https://wanghongwei-academicpage.github.io/
-/




/-
## Categories
-/


/-
Def (Category): A Category 𝔸 consits of a collection `Ob_𝔸 ` of objects
and `∀ A, B ∈ ob_𝔸`, there is a collection `Hom_𝔸 (A, B)` of maps or morphisms
from A to B, such that
1. Existence of identity: `∀ X ∈ Ob_𝔸 ` , there is a morphism `X ⟶ X` denoted as `1_X`
2. Composition laws : `∀ X, Y, Z ∈ Ob_𝔸`, such that `f(X) = Y, g(Y) = Z`,
this is equivlent to say `f ∈ Hom_𝔸 (X, Y)` and `g ∈ Hom_A (Y, Z)` then we
have `g ∘ f (X) = Z`, i.e. `g ∘ f ∈ Hom_𝔸 (X, Z)`.

Moreover, the collection of the morphisms satisfy the two more axioms that
3. Associativity : `∀ f ∈ Hom_𝔸 (X, Y), g ∈ Hom_𝔸 (Y, Z), h ∈ Hom_𝔸 (Z, W)`, we have
  `(h ∘ g) ∘ f = h ∘ (g ∘ f) ∈ Hom_𝔸 (X, W)`
4. Identity law: `∀ f ∈ Hom_𝔸 (X, Y)` we have `f ∘ 1_X = f = 1_Y ∘ f `



A category consists of objects living in a universe `u`.
For any two objects `X Y`, the type of morphisms `hom X Y`
lives in a universe `v`.

Since the structure `Category` contains a field whose value
is a type in `Type v`, the category structure itself must
live in universe `v + 1`.

Taking into account the universe of objects as well, a category
with objects in `Type u` and morphisms in `Type v` lives in
universe `max u (v + 1)`.
-/

universe v u

def x : ℕ := 1

class MyCat (Ob : Type u) : Type (max u (v + 1)) where
  hom   : Ob → Ob → Type v
```

```lean
  id    : (X : Ob) →  hom X X
  comp  : {X Y Z : Ob} →  hom X Y → hom Y Z → hom X Z

  comp_id : {X Y : Ob} → (f : hom X Y) →
    comp (id X) f  = f

  id_comp : {X Y : Ob} → (f : hom X Y) →
    comp f (id Y) = f

  assoc : {W X Y Z : Ob} →
    (f : hom W X) → (g : hom X Y) → (h : hom Y Z) →
    comp (comp f g) h = comp f (comp g h)

-- U should carefully use the `comp`: (f : X ⟶ Y) , (g : Y ⟶ Z) ↦ g ∘ f

namespace MyCat

scoped notation3 "𝟙" => MyCat.id
scoped infixr:10 " ⟶ " => MyCat.hom
scoped infixr:80 " ≫ " => MyCat.comp

/-
The keyword `infixr` means that the notation is right-associative.
For example, `X ⟶ Y ⟶ Z` is parsed as `X ⟶ (Y ⟶ Z)`, and similarly
`f ≫ g ≫ h` is parsed as `f ≫ (g ≫ h)`.

The number following `infixr` specifies the precedence: since
`⟶` has precedence 10 and `≫` has precedence 80, the operator `≫`
binds more tightly than `⟶`. This determines how expressions are
parsed in the absence of parentheses.
-/


variable {Ob : Type u} [MyCat Ob]
variable {X Y Z : Ob}
variable (f : X ⟶ Y) (g : Y ⟶ Z)

#check 𝟙 X
#check f ≫ g



def discreteCat (α : Type u) : MyCat α where
  hom X Y := PLift (X = Y)
  id X := PLift.up rfl
  comp f g := PLift.up (PLift.down f ▸ PLift.down g)
  comp_id f := by
    cases f
    rfl
  id_comp f := by
    cases f
    rfl
  assoc f g h := by
    cases f; cases g; cases h
```

```
    rfl
/-
## Functors
-/

structure MyFun {C : Type u} {D : Type u'} [MyCat.{v} C] [MyCat.{v} D] :
  Type (max (max u v) (max u' v)) where
    obj : C → D
    map : {X Y : C} →  hom X Y → hom (obj X) (obj Y)

    map_id :  {X : C} →  map (id X) = id (obj X)

    map_comp : {X Y Z : C} → (f : hom X Y) → (g: hom Y Z) →
      map (comp f g) = comp (map f) (map g)



variable {A B : Type u} [MyCat A] [MyCat B]
variable (F : MyFun (C := A) (D := B)) {X Y Z : A}
variable (f : X ⟶ Y) (g : Y ⟶ Z)

#check F.map (f ≫ g)


@[simp]
lemma map_id' (X : A) :
  F.map (𝟙 X) = 𝟙 (F.obj X) :=
    F.map_id

@[simp]
lemma map_comp' : F.map (f ≫ g) = F.map f ≫ F.map g :=
  F.map_comp f g

-- Identity functor
def IdFun (C : Type u) [MyCat C] :
  MyFun (C := C) (D := C) :=
    {
      obj := fun X => X
      map := fun f => f
      map_id := by simp
      map_comp := by simp
    }


-- Composition of functor
def comFun {C D E : Type u} [MyCat.{_} C] [MyCat.{_} D] [MyCat.{_} E]
  (F : MyFun (C := C) (D := D)) (G : MyFun (C := D) (D := E)) : MyFun (C := C) (D := E) :=
    {
      obj := fun X => G.obj (F.obj X)
      map := fun {X Y} f => G.map (F.map f)

      map_id := by
        intro X
```

```
        simp [F.map_id, G.map_id]

      map_comp := by
        intro X Y Z f g
        simp [F.map_comp, G.map_comp]
    }

/-
## Natural Transformations
-/

structure MyNatTrans
  {C D : Type u}
  [MyCat.{v} C] [MyCat.{v} D]
  (F G : MyFun (C := C) (D := D)) :
  Type (max u v) where

  -- Component at each object
  app : (X : C) → F.obj X ⟶ G.obj X

  -- Naturality condition
  naturality :
    {X Y : C} → (f : X ⟶ Y) →
      F.map f ≫ app Y = app X ≫ G.map f

namespace MyNatTrans





variable {C D : Type u}
variable [MyCat.{v} C] [MyCat.{v} D]
variable (F : MyFun (C := C) (D := D))




end MyNatTrans
end MyCat
```