MyMathlibProject/CategoryFormalization.lean

```lean
/-
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
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/
-/

import Mathlib
/-
## Feb 5 / 2026
-/



/-
## 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
  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)


/-
Category structure:
Set Category
ob(Set): {S₁, S₂, S₃ ....}.    for S₁, S₂, Hom(S₁, S₂) = f : S₁ → S₂

Linear Vect Space Category
ob(LV) : {H, G, ....}.   for H, G. Hom(H, G) = L (H, G) = Matrix

Group Category
ob(Grp) : {G₁, G₂, G₃, ...}.  for G₁, G₂ . Hom(G₁, G₂) = homomorphism: G₁ → G₂

-/






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


-- we say a function f : X → X, x ↦ x, we denote the f
-- as 𝟙_X
namespace Mycat


scoped notation3 "𝟙" => id

scoped infixr:10 " ⟶ " => MyCat.hom
```

```
scoped infixr:80 " ≫ " => MyCat.comp

-- a ⟶  b ⟶ c = a ⟶ (b ⟶ c)    (infixr)
-- a ≫  b ≫ c = a ≫ (b ≫ c)
-- ≫
-- a ⟶ ((b ≫ c) ⟶ (d ⟶ e))
-- a ⟶ b ≫ c ⟶ d ⟶ e




/-
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
-/

#check MyCat

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} →  MyCat.hom X Y → MyCat.hom (obj X) (obj Y)

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

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

  -- F.obj X ⟶ G.obj X ∈ Hom_D (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



-- For a fixed cat C, functor F : C → C

-- Set A, B.  f: A → A    g: A → B

--  f : ℕ → ℝ,   g : ℕ → ℝ .    α : f → g.   α_x : f(x) → g(x)

namespace MyNatTrans





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




end MyNatTrans
```