19/20

Interactive Theorem Proving Assignment 4
Martin Skilleter (u6679334)
31 May 2019

I collaborated with the following people while completing this assignment: Isabel Longbottom (u6702144) and Aaron Bryce (u6677442).

```
import tactic.basic

/-
It's time to learn about universes!

This assignment is hopefully fairly quick, but will walk you through the
universe polymorphism issues involved in defining categories in Lean.
-/

-- Here's a simple attempt at defining a category.
-- The parameter `C` describes the objects of the category.
class category (C : Type) :=
(hom : C → C → Type)
(id : Π X : C, hom X X)
(comp : Π {X Y Z : C}, hom X Y → hom Y Z → hom X Z)
(comp_id : Π {X Y : C} (f : hom X Y), comp f (id Y) = f)
(id_comp : Π {X Y : C} (f : hom X Y), comp (id X) f = f)
(assoc : Π {W X Y Z : C} (f : hom W X) (g : hom X Y) (h : hom Y Z), comp (comp
f g) h = comp f (comp g h))

-- However, this definition is no good: we can't define the category of types:
instance category_of_types_broken : category Type :=
{ hom := λ X Y, X → Y }

-- To fix this, you're going to need to modify the definition above,
-- to fix the universe levels.

-- Question 0: Explain why the definition above wasn't useful, perhaps
-- mentioning Russell's paradox.

/- Answer: Because of the hierarchy of the universes, Type cannot be of type
Type.
Instead, it must be of Type 1 (otherwise, we would have an analogue of Russell's
Paradox:
the set of all sets being an element of itself). In a dependent type theory
context,
this means that nothing can be of Type itself, which is implemented by setting
Type to be of
Type 1, Type 1 of Type 2 etc. This is not reflected in the definition,
as the universe level of the category above is fixed. It should instead be
polymorphic, so
```

huh? ℕ : Type ...

Actually this isn't the case. You will have Girard's paradox, but even "type in type" solves Russell's paradox.

```
that elements belonging to higher universe levels can be made into categories.

One possible solution to this would be to change Type to Type 1 (at the very
least, this would
allow the category of types to be defined. However, we would then have an
identical problem to
the one above when we tried to create a category for a type in a universe level
higher
than 1).
-/

 -- Here's one attempt:

 class category_1 (C : Type 1) :=
 (hom : C → C → Type)
 (id : Π X : C, hom X X)
 (comp : Π {X Y Z : C}, hom X Y → hom Y Z → hom X Z)
 (comp_id : Π {X Y : C} (f : hom X Y), comp f (id Y) = f)
 (id_comp : Π {X Y : C} (f : hom X Y), comp (id X) f = f)
 (assoc : Π {W X Y Z : C} (f : hom W X) (g : hom X Y) (h : hom Y Z), comp (comp
 f g) h = comp f (comp g h))

 -- Question 1: fill in the remaining fields of this definition.
 instance category_of_types : category_1 Type :=
 { hom := λ X Y, X → Y,
  id := λ X, λ x, x,
  comp := λ X Y Z, λ f g, λ x, g (f x),
  comp_id := by {intros X Y f, simp},
  id_comp := by {intros X Y f, simp},
  assoc := by {intros W X Y Z f g h, simp}
 }

 -- However, this variant has its own problems. A standard solution to Russell's
 -- paradox about "the set of all sets" (resolved in Lean's dependent type
 theory
 -- by the typing judgement `Type : Type 1`) is to consider all subsets of some
 -- fixed 'big' set, rather than "all sets".
```

(in fact, I think `λx, rfl` also works instead of `by simp`.

probably better to write lambdas.

```
-- Question 2: Decide whether `C` below should be `category` or `category_1`,
-- and fill in the remaining fields.
instance category_of_subsets (X : Type) : category (set X) :=
{ hom := λ P Q, { x // P x } → { x // Q x },
 id := λ P, λ x, x,
 comp := λ P Q R, λ p q, λ x, q (p x),
 comp_id := by {intros P Q f, simp},
 id_comp := by {intros P Q f, simp},
 assoc := by {intros P Q R f g h, simp}
}

-- We've now got a conundrum: for some reasonable examples, we want the
-- objects and morphisms to live in the same universe, while for other
-- examples we want the objects to live one universe level higher than
-- the morphisms.

-- In ZFC based category theory, these two variants of the notion of a category
-- are called "small categories" and "large categories".

-- Rather than duplicate everything we want to prove about categories
-- (or worse: we'll need to worry about four different sorts of functors,
-- as the source and target categories could individually be small or large!)
-- in the mathlib category theory library we've made a "universe polymorphic"
-- definition, which is essentially this one:

universes v u

class pcategory (C : Type u) :=
(hom : C → C → Type v)
(id : Π X : C, hom X X)
(comp : Π {X Y Z : C}, hom X Y → hom Y Z → hom X Z)
(comp_id : Π {X Y : C} (f : hom X Y), comp f (id Y) = f)
(id_comp : Π {X Y : C} (f : hom X Y), comp (id X) f = f)
(assoc : Π {W X Y Z : C} (f : hom W X) (g : hom X Y) (h : hom Y Z), comp (comp
f g) h = comp f (comp g h))

-- Question 3:
-- Work out the appropriate values of the universe parameters `p`, `q`, `r`,
`s`
-- below, and then verify that the same fields you used above can be used to
-- complete the following two definitions:
-- (Hint: substitute fixed large values, like p=5, q=10, and read the error
messages...)
```

```
instance category_of_types' : pcategory.{0 1} Type :=
{ hom := λ X Y, X → Y,
 id := λ X, λ x, x,
 comp := λ X Y Z, λ f g, λ x, g (f x),
 comp_id := by {intros X Y f, simp},
 id_comp := by {intros X Y f, simp},
 assoc := by {intros W X Y Z f g h, simp}
}

instance category_of_subsets' (X : Type) : pcategory.{0 0} (set X) :=
{ hom := λ P Q, { x // P x } → { x // Q x },
 id := λ P, λ x, x,
 comp := λ P Q R, λ p q, λ x, q (p x),
 comp_id := by {intros P Q f, simp},
 id_comp := by {intros P Q f, simp},
 assoc := by {intros P Q R f g h, simp}
}

-- Question 4:
-- Complete the following definitions:
universes v₁ u₁ v₂ u₂

structure Functor (C : Type u₁) [pcategory.{v₁ u₁} C] (D : Type u₂)
[pcategory.{v₂ u₂} D] :=
(obj : C → D)
(map : Π {X Y : C}, pcategory.hom X Y → pcategory.hom (obj X) (obj Y))
(id_map : Π X : C, map (pcategory.id X) = pcategory.id (obj X))
(map_comp : Π X Y Z : C, Π f : pcategory.hom X Y, Π g : pcategory.hom Y Z,
map (pcategory.comp f g) = pcategory.comp (map f) (map g))
-- Hint: there are two missing fields here, giving the axioms for functors.

def List : Functor Type Type :=
{ obj := λ α, list α,
 map := λ α β : Type, λ f : pcategory.hom α β, λ L, list.map f L,
 id_map := by {intros X, dsimp [pcategory.id], funext, induction x, simp, dsimp
[list.map], rw [x_ih]},
 map_comp := by {intros X Y Z f g, simp, funext, dsimp [pcategory.comp],
induction x,
                 simp, dsimp [list.map], rw [x_ih],}}
```

map_id would be the canonical name — just read the tokens in order.

it's definitely possible to prove these without resorting to induction!

```
-- Question 5:
-- Complete the following definitions:

structure NaturalTransformation
 {C : Type u₁} [pcategory.{v₁ u₁} C] {D : Type u₂} [pcategory.{v₂ u₂} D]
 (F G : Functor C D) :=
(app : Π X : C, pcategory.hom (F.obj X) (G.obj X))
(naturality : Π X Y : C, Π f : pcategory.hom X Y, pcategory.comp (F.map f)
(app Y) = pcategory.comp (app X) (G.map f))
```

Space

```
namespace NaturalTransformation
def id
 {C : Type u₁} [pcategory.{v₁ u₁} C] {D : Type u₂} [pcategory.{v₂ u₂} D]
 (F : Functor C D) : NaturalTransformation F F :=
```

indent

```
{app := λ X : C, pcategory.id (F.obj X),
naturality := by {intros X Y f, rw [pcategory.comp_id, pcategory.id_comp]}}
```

lambdas

```
def comp
 {C : Type u₁} [pcategory.{v₁ u₁} C] {D : Type u₂} [pcategory.{v₂ u₂} D]
 {F G H : Functor C D}
 (α : NaturalTransformation F G) (β : NaturalTransformation G H) :
NaturalTransformation F H :=
{app := λ X : C, pcategory.comp (α.app X) (β.app X),
naturality := by {intros X Y f, rw [←pcategory.assoc, α.naturality,
pcategory.assoc, pcategory.assoc],
apply congr_arg, rw [β.naturality]}}

@[extensionality] lemma Natural_ext
{C : Type u₁} [pcategory.{v₁ u₁} C] {D : Type u₂} [pcategory.{v₂ u₂} D]
{F G : Functor C D} {N M : NaturalTransformation F G} :
(Π X : C, N.app X = M.app X) → N = M :=
begin
 intros h,
 induction N,
 induction M,
 simp at h,
 simp [h],
 funext,
 exact h x,
end
```

```
@[simp] lemma id_app
{C : Type u₁} [pcategory.{v₁ u₁} C] {D : Type u₂} [pcategory.{v₂ u₂} D] {F :
Functor C D} :
(id F).app = λ X : C, pcategory.id (F.obj X) := rfl
```

This seems a slightly awkward lemma. Why not provide the argument X?

```
-- Hint: you may find `conv` helpful.
end NaturalTransformation

-- Here's the crux of the whole assignment: understand what the correct values
-- of `p` and `q` are in this definition, then complete the definition.
-- Hint: you may like to prove an extensionality lemma for natural
transformations,
-- and appropriate simp lemmas.
instance {C : Type u₁} [pcategory.{v₁ u₁} C] {D : Type u₂} [pcategory.{v₂ u₂}
D] :
 pcategory.{(max u₁ v₂) (max u₁ u₂ v₁ v₂)} (Functor C D) :=
{ hom := λ F G, NaturalTransformation F G ,
 id := NaturalTransformation.id,
 comp := λ F G H, NaturalTransformation.comp,
 comp_id := by {intros F G N, ext X, dsimp [NaturalTransformation.comp], rw
[pcategory.comp_id]},
 id_comp := by {intros F G N, ext X, dsimp [NaturalTransformation.comp], rw
[pcategory.id_comp]},
 assoc := by {intros F G H I N M O, ext X, dsimp [NaturalTransformation.comp],
rw [pcategory.assoc]}
}

constant C : Type u₁
constant D : Type u₂
variables [C_cat : pcategory.{v₁ u₁} C] [D_cat : pcategory.{v₂ u₂} D]
variables F G : @Functor C C_cat D D_cat
constant N : NaturalTransformation F G

set_option pp.universes true
#check @Functor C C_cat D D_cat -- Type (max u₁ u₂ v₁ v₂)
#check @NaturalTransformation C C_cat D D_cat F G -- Type (max u₁ v₂)
```

✓

```lean
-- Question 6:
-- What universe does `NaturalTransformation F G` live in? Why?
-- Hint: a really good answer will use the word 'impredicativity' somewhere
-- along the way.

/- Answer: NaturalTransformation F G lives in the universe 'max (u₁ v₂)'.
 We begin by noting that Functor C D is of type Type max (u₁ u₂ v₁ v₂).
 This is because the field obj has type Type (max u₁ u₂) (because C : Type u₁
 and D : Type u₂ and so) and the field map has type Type (max v₁ v₂) (because
 the morphisms in the category of C live in the universe v₁,  and the morphisms
 in the category of D live in the universe v₂).
  The other fields are of type Prop = Sort 0 : Type 0, because they are functions
from some type
 into Prop. Impredicativity means that regardless of the universe level
 of the domain, the type of this will always be Type 0. We want this feature
 in our universe levels in Lean because we can interpret a function f : A → B
for some
 types A : Type u and B : Prop by the statement "given some element of A, I can
give you an
 element of B". In the case where B : Prop, an element of B is a proof of B and
so f : A → Prop
 represents the statement "if A has an element then B", so f is itself a
proposition (and hence
 of type Prop), even if this is a lower universe level than A.
  All of these facts together mean that Functor C D is of type
 Type max (max u₁ u₂) (max v₁ v₂) = Type max (u₁ u₂ v₁ v₂).
 [Note : I realised after I had written this that the universe level of Functor
does not affect
  the universe level of NaturalTransformation. However, the explanation about
impredicativity above
  is still used below when we discuss why the field naturality of
NaturalTransformation does not
  affect its universe level, so I left it in.]

 We can now apply the same reasoning as above to determine the universe level of
 NaturalTransformation. Because naturality is a Prop, it will not affect the
universe level of
```

```
 NaturalTransformation (once again by impredicativity), so it suffices to
determine the universe
 level of app. pcategory.hom (F.obj X) (G.obj X) is a morphism in the category D
and so is of type
 Type v₂. Because X : C, app is a function from something of type Type u₁ to
something of type
 Type v₂ and so its type is Type (max u₁ v₂) (meaning it lives in the universe
level max (u₁ v₂)).
 Then NaturalTransformation has the same universe level as app and so lives in
the universe level
 max (u₁ v₂).
-/
```

Good.