Limits in the category of categories

Joseph Hua

February 3, 2022

1 Introduction

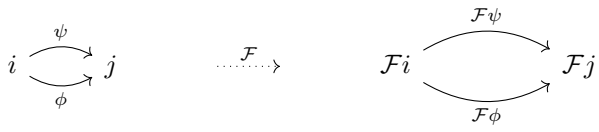
In this project, I show that `Cat.{u u}`, the category of small categories is complete, where "small" in `lean` is taken to mean the universe levels (sizes) for objects and morphisms is the same. I also show that `Cat.{v u}`, the category of all categories (up to appropriate universe levels) has all finite limits. Since the category of categories is a bicategory, and it is quite rare in practice to work with strict equality of functors, the "morally correct" limits to consider should really be 2-limits, where the functors are considered up to natural isomorphism. However, these are not considered here, since existence of 1-limits holds anyway. This should deduce the 2-limit case, as equal funtors are trivially naturally isomorphic.

An obvious next step is to show results for colimits, which I have not yet done.

2 Relevant definitions and results in mathlib

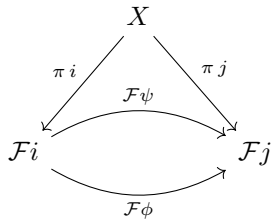
The category theory library in `mathlib` already contains definitions for limits, using cones over diagrams.

> **Definition – Diagrams, cones and limits**
> A diagram in a category $\mathcal{C}$ is simply a functor from some category $F : \mathcal{I} \to \mathcal{C}$, and $\mathcal{I}$ is called the shape. Diagrams are not explicitely defined in `lean`, as this would lead to extra unnecessary definitional equalities.



A diagram for equalizers

> A cone over a diagram $F : \mathcal{J} \to \mathcal{C}$ consists of an apex X an object in $\mathcal{C}$ and a natural transformation from (the constant functor at) X to F. This natural transformation captures the data of legs from X into the diagram that commute with the maps in the diagram.
>
> ```
> structure cone (F : J ⥤ C) :=
> (X : C) (π : (const J).obj X ⟶ F)
> ```
>
> Here `lean` recognizes that `F` is a functor, so it infers that the categorical map into `F` should be in the functor fategory, i.e. a natural transformation.



A cone over the equalizer diagram, or a "fork"

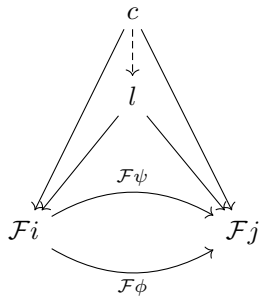
Finally, the data of a cone l being the limit of a diagram consists of

- (`lift`) For any cone c, a map `lift` : $c \to l$ in the category $\mathcal{C}$.
- (`fac`) A proof that `lift` commutes with everything else in the diagram.
- (`uniq`) Any other map $c \to l$ commuting with everything is equal to `lift`.

Put another way, a limit of a diagram is the terminal object when considering the cones as a category.

```
structure is_limit (t : cone F) :=
(lift  : Π (s : cone F), s.X → t.X)
(fac'  : ∀ (s : cone F) (j : J), lift s ≫ t.π.app j = s.π.app j . obviously)
(uniq' : ∀ (s : cone F) (m : s.X → t.X) (w : ∀ j : J, m ≫ t.π.app j = s.π.app j),
m = lift s . obviously)
```

Note that `lift` is data, whereas the others are propositional. Also note that `fac'` and `uniq'` are appended with `obviously`. This means that when we construct a limit, we could only supply the `lift`, and let `lean` figure the rest out. Many of the structures in the category theory library use this so that making instances of categories, functors, natural transformations, and limits less work.



l is the equalizer of $\mathcal{F}\phi$ and $\mathcal{F}\psi$

All of the following are results from `mathlib` that I will be using in my work.

- (Categorical) definitions of binary and arbitrary products, equalizers, and pullback. Browsable here.
- A proof that categories with products and equalizers has all small limits. A proof that categories with finite products and equalizers is complete. A proof that categories with binary products and terminal objects has finite products. Browsable here.
- The product of categories (as a category, *not* as an object in `Cat.{v u}`). Browsable here.
- The binary product of categories (as a category. *not* as an object in `Cat.{v u}`). Browsable here.

3 The process of giving a limit

In this section I walk through the general process of giving a limit, by following my work on binary products. This was the first example I worked on, and I think it demonstrates the process well, having the fewest technicalities.

We begin by giving the object in the category of categories that we are interested in constructing. The definition of the category of categories essentially consists of two pieces of data, α the underlying type, and `str` an `instance` of α as a category. The second piece of data is given the parameter `obviously`, so we need not specify it in our construction:

```
def prod (C D : Cat.{v u}) : Cat.{v u} := { α := (C.α × D.α) }
```

The underlying type of the product is simply of product of the underlying types of `C` and `D`. The library already has an `instance` of this type as a category, so the type is all we need.

We promote this to a cone, by defining the natural transformation from the constant functor at `prod` to the diagram. The diagram in question is called `pair` and its shape (a category with just two objects with no morphisms) is (a type that looks exactly like) `Bool`. A `mathlib` lemma `map_pair` (specialized to our use case) that says to make a natural transformation into a `Bool` shaped diagram, it suffices to just give a map into each of the two objects in the diagram. Hence we have a slick definition

```
def cone (C D : Cat.{v u}) : cone (pair C D) :=
{ X := prod C D,
  π := map_pair (fst _ _) (snd _ _) }
```

We intend for this cone to be the limit. Which consists of a `lift`, factorizations through the `lift`, and uniqness of the lift.

Constructing the lift generally amounts to constructing maps for objects and morphisms by destructing the definition of the limit cone. In this particular case, since the limit cone is a product of categories, to make objects and morphisms in it we need to give pairs of objects and morphisms from each category. Since cones over this diagram only really consist of functors from the apex into `C` and `D`, giving the `lift` is a corollary of this definition:

```
def lift (f : E → C) (g : E → D) : (E → prod C D) :=
{ obj := λ z, ⟨ f.obj z , g.obj z ⟩,
  map := λ _ _ h, ⟨ f.map h , g.map h ⟩ }
```

The remaining two parts of this construction involve proving an equality bewteen two functors, which deserves some interesting design decisions. We leave this discussion to the next section.

4 Equality between functors

In the introduction it was mentioned that strict equality between functors was not "morally correct". This is exemplified when we try to prove such an equality. Naively, extensionality for functors should say two functors $FG : \mathcal{C} \to \mathcal{D}$ should be equal if and only if they are equal on objects and on morphisms. However, type theory complicates this issue for morphisms. If x and y are objects in $\mathcal{C}$ and we have proofs that $Fx = Gx$ and $Fy = Gy$, it doesn't make sense to ask for a morphism $f : x \to y$ whether $Ff = Gf$, because these are terms of types that are not definitionally equal. Namely $Ff : Fx \to Fy$ and $Gf : Gx \to Gy$.

The obvious workaround to this issue is to instead ask for a commutative square