Limits in the category of categories

Joseph Hua

February 4, 2022

1 Introduction

All of my work is in two files, `to_mathlib.lean` (which contains lemmas that would be useful to have in various places in `mathlib`) and `limits.lean` (which should ideally go in a folder for `Cat`, the category of categories). Some of the the content in those files is *not* my own, and I have pointed this out each time in a comment.

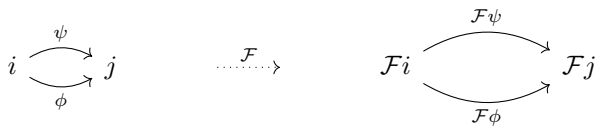
In this project, I show that `Cat.{u u}`, the category of small categories is complete, where "small" in `lean` is taken to mean the universe levels (sizes) for objects and morphisms is the same. I also show that `Cat.{v u}`, the category of all categories (up to appropriate universe levels) has all finite limits. Since the category of categories is a bicategory, and it is rare in practice to work with strict equality of functors, the "morally correct" limits to consider should really be 2-limits, considering functors up to natural isomorphism. However, I construct 1-limits, since they exist 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`

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 leads to extra unnecessary definitional equalities.

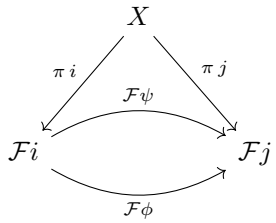


A diagram for equalizers

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 morphism (π) † into `F` should be in the functor fategory, i.e. a natural transformation.



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

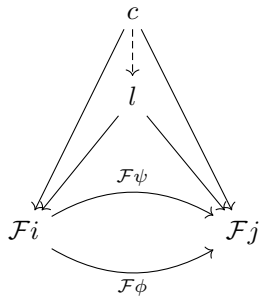
Finally, the data of a cone l being the limit of a diagram consists of

- (`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 $\mathcal{F}\psi$

[†]Unfortunately the arrows for morphisms (`\hom` ⟶) and for functions (`\to` →) look the same.

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.

We begin by giving the object in the category of categories that we are interested in constructing. An object in the category of categories consists of two pieces of data, α the underlying type, and `str` an `instance` of α as a category. The underlying type of the product is simply of product of the underlying types of `C` and `D`.

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

The second piece of data is given the parameter `obviously`, so we need not specify it: the library already has an `instance` of this type as a category.

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

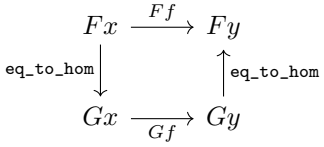
4.1 `heq`

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 $F, G : \mathcal{C} \to \mathcal{D}$ are 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 given $f : x \to y$ whether $Ff = Gf$, because $Ff : Fx \to Fy$ and $Gf : Gx \to Gy$ are terms of types that are not definitionally equal.

The obvious workaround to this issue is to instead ask for a commutative square where the sides are given

by the equalities we have (equal objects are certainly isomorphic).



Whilst this method works - I used it in my first round of proofs, it can be improved with a clever trick.

Instead of considering strict equality ($=$) on morphisms, Xu Junyan on Zulip suggested using `hequality` ($==$) instead. This different version of equality is also a proposition whose only proof is `refl` (that the two terms are definitionally the same - in particular they are of the same type), but this question can be asked even when their ambient types are different. Hence in the above example I could simply write $Ff == Gf$, though I would not be able to prove it until I make the two sides definitionally equal.

```
inductive heq {α : Sort u} (a : α) : Π {β : Sort u}, β → Prop
| refl [] : heq a
```

The benefit of this definition is that one doesn't need to deal with the extra compositions the original workaround introduces. In the end whatever types we are working with should be rewritten to be definitionally equal anyway, so it is more straight forward to first imagine that they are equal, and realise that our imagination was correct later on.

4.2 The simplest case

I will list a few of methods I used to show that two functors are equal.

The simplest case: after appropriate definitional simplification the functors are definitionally equal on objects and morphisms.

A useful result from the library `functor.hext` says that two functors are equal if they are equal on objects and `hequal` on morphisms. In this situation, applying `functor.hext` works well. The following is an example of this (for arbitrary products). The `lift` is defined *so that* when it is destructed via composition with the legs of the cone it returns the right side.

```
lemma fac (c : limits.cone F) (i : discrete I) :
  lift c ≫ (cone F).π.app i = c.π.app i :=
functor.hext (λ _, rfl) (λ _ _ _, heq_of_eq rfl)
```

4.3 Factoring through the lift

When one side of the equality has data but the other does not, we might need to introduce data on the side without. Here is an example (for equalizers):

```
lemma uniq (c : category_theory.limits.fork F G) (m : c.X → (fork F G).X)
  (h : ∀ (j : walking_parallel_pair), m ≫ (fork F G).π.app j = c.π.app j) :
  m = is_limit_fork.lift c :=
by { rw equalizer.self_eq_lift m, congr, exact h walking_parallel_pair.zero }
```

The first rewrite refactors the map `m` into the limit as a lift of itself composed with the maps into the diagram. This introduces data on the left, putting `m` into the same form as the right side (after unfolding what `is_limit_fork.lift` is).

4.4 Explicitely using the underlying type

To show that legs of any cone factor through the binary product `lift`, I cased on the object in the shape for which I am concerned. This is basically induction on `Bool`, and the resulting cases are fairly trivial.

```
lemma fac (c : limits.cone (pair C D)) (j : walking_pair) :
  lift c ≫ (cone C D).π.app j = c.π.app j :=
walking_pair.cases_on j (lift_fst _) (lift_snd _)
```

5 Equalizers with `heq`

My definition of the equalizers of two functors $F, G : \mathcal{C} \to \mathcal{D}$ takes the underlying type to be a subtype of objects x in the source category $\mathcal{C}$ such that $Fx = Gx$. The morphisms are taken to be the subtype of morphisms f in each hom set such that $Ff == Gf$. The immediate (and so far only; the rest was straight-forward) issue that arises from using `heq` is composition in this category: if f and g are composable morphisms in $\mathcal{C}$ such that $Ff == Gf$ and $Fg == Gg$, do we have that $F(f \gg g) == G(f \gg g)$? This fact is not immediately obvious. To prove this I would like to rewrite the first `heq`uality. Since Ff is not a variable, and induction/-cases on an equality requires that the first term is a variable, we cannot case on the proof of $Ff == Gf$. One solution to this is generalizing the appropriate variables, using the `generalize_hyp` tactic, given here (due to Xu Junyan). My explanations are in comments:

```
lemma map_comp_heq (hx : F.obj x = G.obj x) (hy : F.obj y = G.obj y) (hz : F.obj z = G.obj z)
  (hf : F.map f == G.map f) (hg : F.map g == G.map g) :
  F.map (f ≫ g) == G.map (f ≫ g) :=
begin
  rw [F.map_comp, G.map_comp],
  -- cannot case directly on hf, since F.map f is not a variable -> generalize F.map f
  generalize_hyp : F.map f = Ff at ⊢ hf, generalize_hyp : G.map f = Gf at ⊢ hf,
  generalize_hyp : F.map g = Fg at ⊢ hg, generalize_hyp : G.map g = Gg at ⊢ hg,
  -- however, still cannot case on hf, since the types on the left and right are not
    definitionally equal
  -- To make them definitionally equal, must case on x.2 y.2 z.2
  -- Similarly, to case on x.2 and z.2 must generalize F.obj x, F.obj y, F.obj z
  generalize_hyp : F.obj x = Fx at ⊢ Ff Gf hx,
  generalize_hyp : F.obj y = Fy at ⊢ Fg Ff hy,
  generalize_hyp : F.obj z = Fz at ⊢ Fg Gg hz,
  -- now able to substitute -> all definitionally equal -> cases on hf hg
  subst hx, subst hy, subst hz, cases hf, cases hg,
  exact heq_of_eq rfl,
end
```

However, I was then informed that the tactic `congr'` (but not `congr`) does almost all of the work.

```
  lemma map_comp_heq (hx : F.obj x = G.obj x) (hy : F.obj y = G.obj y) (hz : F.obj z = G.obj z)
  (hf : F.map f == G.map f) (hg : F.map g == G.map g) :
  F.map (f ≫ g) == G.map (f ≫ g) :=
  by { rw [F.map_comp, G.map_comp], congr' }
```

If I understand correctly, this is because `congr` has the tools for casing on the equalities, but "agreessively" first checks for syntactic similarities, resulting in goals such as $F = G$, which cannot be proven. On the other hand `congr'` first tries to apply equalities in the hypothesis, and succeeds.

6 Universe levels

As this is work on category theory in `lean`, universe levels deserve attention. The first detail is sensible, and not a `lean` specific issue - that `Cat` needs to be a small category to be closed under arbitrary limits. The second was a design issue related to terminal objects.

6.1 Arbitrary products

Suppose that `Cat.{v u}` (not small) in general had all arbitrary limits. Then in particular `Cat.{v u}` would be closed under morphism sized products (this is the definition of having all limits in `lean`). Since morphisms are functors, which live in `Type max (v u)` this means for any discrete category `I : Type max (v u)` (discrete means it has no morphisms between objects), any diagram of shape `I` has a limit. (Informal) Our construction for the limit of this diagram will likely need to mention the indexing category `I`, and in doing so will be a type with universe level at least `max (v u)`. However, the underlying types of objects in `Cat.{v u}` need to be in `Type u`.

Instead forcing `v` and `u` to be (externally) equal, then taking the `max` keeps us in at the same level, and there is no issue.

6.2 The terminal category

The terminal category in `Cat.{v u}` should have an underlying type that looks like a singleton type, which is called `punit : Type u` in the library. Intuitively there should be only one morphism in this category, namely the identity on the unique object `punit.star`.

The library contructs such a category more generally, by taking any type `X : Type u` and taking the hom sets between its terms `x y : X` to be the type `x = y`. The issue here is that this hom set is a proposition, so it must be lifted to the appropriate type by universe lifting.

```
x = y : Prop : Type 0 : Type 1 : ...
```

The issue here is that it is hard to keep track of the universe we are lifting it to. The library gets around this by requiring `ulift (x = y) : Type u`, so that the terminal category is a small category. However, this is not ideal here, since we want the hom set to be of level `v`. Thus I contruct another version of the terminal category. Since there is exactly one morphism between the obvious point, I also take the unique hom set to be `punit : Type v`, but at level `v` (`punit` is polymorphic).

```
def terminal : Cat.{v u} :=
{ α := punit,
  str := { hom := λ _ _, punit,
           -- this punit : Type v should to be thought of as ‘star → star‘
           id := λ _, punit.star, -- there is only one map from ‘star → star‘
           comp := λ _ _ _ _ _, punit.star } }
           -- the composition of this trivial map with itself is itself
```