Presheaves over the category of elements of a presheaf

Joseph Hua

March 30, 2022

1 Conventions

We call functors $P : \mathcal{C} \Rightarrow \mathcal{U}$ for some universe $\mathcal{U}$ a presheaf, although the convention is usually dual, with $\mathcal{C}^{op}$ instead. Since the proofs work exactly the same for the dual case, this is justified.

2 The goal

In the file `over_presheaf.lean` we prove a category-theoretic lemma that says if $P : \mathcal{C} \Rightarrow \mathcal{U}$ is a presheaf on category $\mathcal{C}$ then the category of presheaves over the category of elements of P is equivalent to the over category of P.

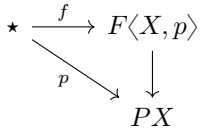
$$\mathrm{Psh}(\Sigma P) \cong \mathrm{Psh}(C)/P$$

In lean this is stated as

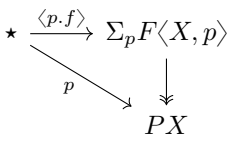
```
def equivalence : (P.elements ⥤ Type u₀) ≌ over P :=
{ functor := _,
  inverse := _,
  iso := _,
  counit_iso := _ }
```

Since presheaves on a category $\mathcal{C}$ is defined as the functors from $\mathcal{C}$ into the universe, which in type theory is taken to be `Type u₀` for some universe level u_0. There are four[1] fields required to define an equivalence of categories: the functors forwards and back, and proof that they are left and right inverses.

The idea is that a presheaf on $\mathcal{C}$ takes any object $X : \mathcal{C}$ to the set of generalized elements of X, which are points $p : \star \to PX$ ($\star : \mathcal{U}$ is the terminal object in the universe). Then a presheaf F on ΣP takes any object $\langle X : \mathcal{C}, p : PX \rangle$ in the category of elements of P to a generalized elements of X that commute with p, forming points $f : \star \to F\langle X, p\rangle$.



If we collect these generalized elements under F across all $p : \star \to X$ then we can get a map $\Sigma_{p:\star\to X} F\langle X, p\rangle \to PX$ with each $F\langle X, p\rangle$ the fiber of a point p.



[1] There is an extra one, the "triangle law", but it is automatically generated by `obviously` in this case.

Thus to make a functor forward, we take a functor $F : \mathrm{Psh}(\Sigma P) \to \mathcal{U}$ and make a presheaf by

$$(X : \mathcal{C}) \mapsto \Sigma_{p:\star \to X} F\langle X, p\rangle : \mathcal{U}$$

It is in the over category of P since we have a map down to each generalized set PX with fibers $F\langle X, p\rangle$.

To make an inverse, we take a presheaf $Q : \mathcal{C} \Rightarrow \mathcal{U}$ over P, meaning we have a natural transformation $\alpha : Q \Rightarrow P$, and produce a presheaf on ΣP by taking an object $\langle X, p\rangle : \Sigma P$ and picking out the generalized elements of X under Q that commute with p. This is the pullback

$$\begin{array}{ccc} \alpha_X^{-1} p & \longrightarrow & \star \\ \downarrow & \lrcorner & \downarrow p \\ QX & \longrightarrow & PX \end{array}$$

3 Forwards

To construct the functor forwards, we must functorially construct for each functor `F : P.elements ⇒ Type u₀` an object of the over category `over P`. An object of the over category `over P` consists of a presheaf `to_presheaf_obj F : C ⇒ Type u₀` and a natural transformation down to `P`. As described above, we define the presheaf part by taking the sum of generalized elements from F

```
def to_presheaf_obj : C ⇒ Type u₀ :=
{ obj := λ X, Σ p : P.obj X, F.obj ⟨ X , p ⟩,
  map := λ X Y h, λ ⟨ p , f ⟩, ⟨ P.map h p , F.map (hom_mk h rfl) f ⟩,
  map_id' := _,
  map_comp' := _ }
```

For mapping morphisms, the obvious thing to do is to send the point p along by the image under P of the morphism h, and send f along the image under F of the morphism in the category of elements induced by h.

3.1 `obj_mk` and `hom_mk`

To expain `hom_mk`: For design purposes, it is convenient to make functions `obj_mk` and `hom_mk` that produce objects and morphisms in the category of elements of P[1]; objects are given by objects $X : \mathcal{C}$ and generalized elements $p : PX$ and morphisms are given by morphisms $h : X \to Y$ and proofs that h respects the points in X and Y.

```
/-- Explicit, typed construction of an object in category of elements -/
def obj_mk (X : C) (p : P.obj X) : P.elements := ⟨ X , p ⟩

/-- Explicit, typed construction of a hom in category of elements -/
def hom_mk {X Y : C} {pX : P.obj X} {pY : P.obj Y} (h : X → Y) (hcomm : P.map h pX = pY) :
  obj_mk X pX → obj_mk Y pY := ⟨ h , hcomm ⟩
```

3.2 `heq` issues

Above we gave the data of what the presheaf on $\mathcal{C}$ does, but `obviously` is not able to show that this respects the identity and composition, because there are definitional equality issues. For the identity we let $X : \mathcal{C}$

[1] We could use brackets $\langle \_, \_ \rangle$ and provide the type each time, but this is harder on the eyes.

be an object and consider whether we are sending the identity on X to the identity on the sigma type. The problem is, the image of the identity is the function

```
λ ⟨ p , f ⟩, ⟨ P.map (𝟙 X) p , F.map (hom_mk (𝟙 X) rfl) f ⟩
```

where the second component has type

```
F.obj ⟨ X , P.map (𝟙 X) p ⟩
```

whereas the identity on the image will have second component

```
f : F.obj ⟨ X , p ⟩
```

It makes sense to ask if the functions are equal, but to prove this we must use functional extensionality, introducing such a pair $\langle p, f \rangle$. It makes sense to ask if their images are equal as terms in the sigma type,

```
Σ p : P.obj X, F.obj ⟨ X , p ⟩
```

but to prove this we must use extensionality on the sigma type, which introduces `heq`, since the types on the second part are not definitionally equal. Hence the proof goes

```
def to_presheaf_obj : C ⥤ Type u₀ :=
{ obj := ...,
  map := ...,
  map_id' := λ X, funext (λ ⟨ p , f ⟩,
    by { ext, { simp [to_presheaf_obj._match_1] }, ??? }),
    map_comp' := ...}
```

The first goal is a true equality and can be closed by simplifying until both sides are just p. The second goal (currently `???`) is the `hequality` and needs care. The goal reduces to asking for a proof that

```
F.map (hom_mk (𝟙 X) rfl) f == f
```

We hope that we can use transitivity of `heq` to break this up into

```
F.map (hom_mk (𝟙 X) rfl) f == F.map (𝟙 ⟨X, p⟩) f == f
```

Where the last `hequality` is a definitional equality, and is true since functor F preserves the identity. The first equality naively looks like two applications of `congr`, but for `heq`. Indeed we make such a lemma

```
lemma hcongr_fun {f₀ : α → β₀} {f₁ : α → β₁} (hβ : β₀ = β₁) (hf : f₀ == f₁) (a : α) :
  f₀ a == f₁ a := by { subst hβ, subst hf }
```

Applying the above removes f, reducing the goal to

```
F.map (hom_mk (𝟙 X) rfl) == F.map (𝟙 ⟨X, p⟩)
```

This is surprisingly straight forward

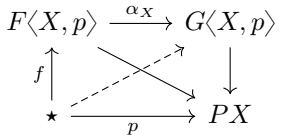
```
lemma map_hom_mk_id_heq_map_id {X : C} {p : P.obj X} {F : P.elements ⥤ Type u₀} :
  F.map (hom_mk (𝟙 X) rfl) == F.map (𝟙 ⟨ X , p ⟩) :=
by { congr', {simp}, {simp} }
```

3.3 The rest of the functor

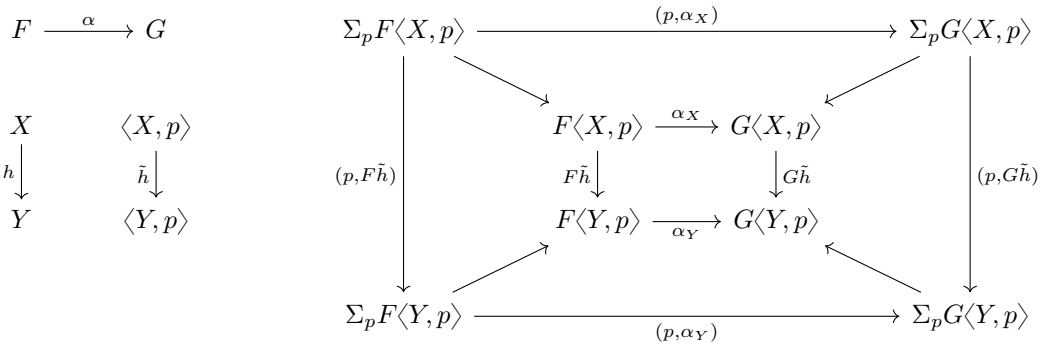
The story for `map_comp'` is similar, and not worth examining in detail. To summarize, the above gave us a presheaf on $\mathcal{C}$ for each presheaf on ΣP. We want to make this construction a functor, rather than just a function. So we also define a function on maps (natural transformations in our case since the objects are functors)

```
def to_presheaf_map (α : F ⟶ G) : to_presheaf_obj F ⟶ to_presheaf_obj G :=
{ app := λ X ⟨ p , f ⟩, ⟨ p , α.app _ f ⟩,
  naturality' := λ X Y h, funext (λ ⟨ p , f ⟩, by { ext,
    { simp [to_presheaf_map._match_1, to_presheaf_obj] },
    { apply heq_of_eq, exact congr_fun
      (@nat_trans.naturality _ _ _ _ _ _ α ⟨ X , p ⟩ ⟨ Y , P.map h p ⟩ ⟨ h , rfl ⟩) f }})}
```

The natural transformation at each object $X : \mathcal{C}$ is given extensionally as a map between sigma types such that for each generalized element p of X under P, the diagram commutes



Since this map is defined component-wise, we use extensionality on the sigma type to check naturality, and it follows from naturality of α, i.e. the outer square commutes because each inner square commutes, where going to the inner square is extensionality. Luckily, no `heq` is needed here, as the types are definitionally equal.

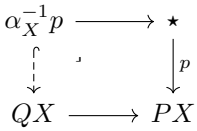


We finally turn this into a functor into the over category, where each natural transformation down to P is given by projection onto the first component of the sigma type $\Sigma_p F\langle X, p\rangle$ for each X.

```
def to_presheaf_over : (P.elements ⥤ Type u₀) ⥤ over P :=
{ obj := λ F, over.mk ({ app := λ X, sigma.fst } : to_presheaf_obj F ⟶ P),
  map := λ F G α, over.hom_mk (to_presheaf_map α) }
```

4 Backwards

To make an inverse functor we fix a preseheaf $Q : \mathcal{C} \to \mathcal{U}$ with $\alpha : Q \Rightarrow P$, and construct a presheaf on ΣP. We first do so on objects, taking any object $\langle X, p\rangle : \Sigma P$ to the pullback



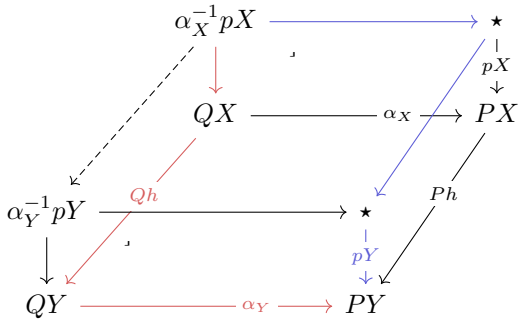
Rather than using the abstract existence of pullbacks in `Type`, we construct it as the preimage of p (as a subtype of QX), under α_X.

4.1 On objects (`inverse`)

4.1.1 On objects (`inverse_obj`)

```
def inverse_obj_obj : P.elements → Type u₀ :=
λ ⟨ X , p ⟩, { q : Q.left.obj X // Q.hom.app X q = p }
```

We extend this to a functor $\Sigma P \Rightarrow \mathcal{U}$. Suppose $pX : PX$ and $pY : PY$. Given a morphism $h : X \to Y$ in the category of elements by having a commuting property `hcomm` : $PhpX = pY$, the pullback property of $\alpha_Y^{-1}pY$ gives a map from $\alpha_X^{-1}pX$, viewed as a cone over the diagram.



In our construction we explicitely give the induced morphism, which is as the "restriction of Qh to $\alpha_X^{-1}pX$", consisting of the map Qh on elements and a proof that the image is sent to pY. The image is sent to pY if and only if the blue and red maps (the legs of the pullback cone) commute. The latter holds because the right face commutes by `homm`, the back face commutes by the pullback property defining $\alpha_X^{-1}pX$ (in the proof this appears as `hqX`), and the bottom face commutes by naturality of α.

4.1.2 Functoriality (`inverse_obj`)

```
def inverse_obj_map : Π (X Y : P.elements) (h : X → Y),
  inverse_obj_obj Q X → inverse_obj_obj Q Y :=
λ ⟨ X , pX ⟩ ⟨ Y , pY ⟩ ⟨ h , hcomm ⟩ ⟨ qX , hqX ⟩,
  ⟨ Q.left.map h qX , by { convert hcomm, rw ← hqX, exact congr_fun (Q.hom.naturality h) qX } ⟩
```

Again, `obviously` cannot generate code for this functor preserving the identity and composition. This time it is because we need to use functional extensionality and case on the points in the sigma type, which is straight forward.

```
def inverse_obj : P.elements ⇒ Type u₀ :=
{ obj := inverse_obj_obj Q,
  map := λ _ _, inverse_obj_map Q _ _,
  map_id' := λ ⟨ X , p ⟩, funext $ λ ⟨ q , hq ⟩, by { simp ... },
  map_comp' := λ ⟨ X , pX ⟩ ⟨ Y , pY ⟩ ⟨ Z , pZ ⟩ ⟨ f , fcomm ⟩ ⟨ g , gcomm ⟩,
    funext $ λ ⟨ qX , hqX ⟩, by { simp ... }
```

4.2 Functoriality `inverse`

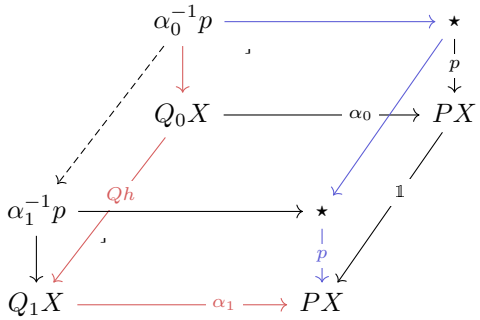
We need to construct for each natural transformation $\nu : Q_0 \Rightarrow Q_1$ of presheaves over the presheaf P, a natural transformation `inverse_obj` $Q_0 \Rightarrow$ `inverse_obj` Q_1. To make such a natural transformation, we need a map for each object $\langle X, p\rangle : \Sigma P$,

```
{ q : Q₀.left.obj X // Q₀.hom.app X q = p } → { q : Q₁.left.obj X // Q₁.hom.app X q = p }
```

which we write as

$$\alpha_0^{-1}(p) \to \alpha_1^{-1}(p)$$

Via pullbacks it is given by the universal property of $\alpha_1^{-1}(p)$ in the following diagram:



As before it suffices that the right, back, and bottom sides commute, which respectively hold trivially, by the pullback property of $\alpha_0^{-1}(p)$, and by ν being a morphism in the over category (ν.w in the code).

```
def inverse : over P ⇒ P.elements ⇒ Type u₀ :=
{ obj := inverse_obj,
  map := λ Q₀ Q₁ ν,
    { app := λ ⟨ X , p ⟩ ⟨ q , hq ⟩, ⟨ ν.left.app X q ,
        by {convert hq, exact congr_fun (congr_fun (congr_arg nat_trans.app ν.w) X) q } ⟩,
      naturality' := λ ⟨ X , pY ⟩ ⟨ Y , pY ⟩ ⟨ h , hcomm ⟩, funext ( λ ⟨ q , hq ⟩,
      subtype.ext (congr_fun (ν.left.naturality h) q)) } }
```

Naturality requires `funext` and `subtype.ext`, but otherwise follows from naturality of ν.

5 Left and right inverses

It remains show that the composed functors back and forth are naturally isomorphic to the identity functors. Here `obtain` is able to bear a lot of the workload.

Our first goal is to show that the identity functor on the presheaf category on ΣP is naturally isomorphic to the composition. This involves making natural transformations back and forth, both of which are also natural transformations on objects. These maps are easy to define once we case on the objects; though the goals are unsightly, by going tactic mode and using `dsimp` they become very simple, so that is the recommended first approach to making these definitions.

```
def unit_iso_hom : 𝟙 (P.elements ⇒ Type u₀) ⟶ to_presheaf_over ⋙ inverse :=
{ app := λ F, { app := λ ⟨ X , p ⟩ f, ⟨ ⟨ p , f ⟩ , rfl ⟩ } }

def unit_iso_inv : to_presheaf_over ⋙ inverse ⟶ 𝟙 (P.elements ⇒ Type u₀) :=
{ app := λ F, { app := λ ⟨ X , p ⟩ ⟨ ⟨ p' , f ⟩ , hq ⟩, eq.mp (by {congr, exact hq}) f }}

def unit_iso : 𝟙 (P.elements ⇒ Type u₀) ≅ to_presheaf_over ⋙ inverse :=
{ hom := unit_iso_hom,
  inv := unit_iso_inv }
```

Note that `obtain` worked out all the naturality conditions needed. For the other direction we make use of `ext` whereever possible, and note that in the cases of extensionality on sigma types no `heq` is needed. Again, nothing interesting is going on here so we omit the code.