Presheaves over the category of elements of a presheaf

Joseph Hua

March 29, 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 u_0) ≌ 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` u_0 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 on ΣP takes any object $\langle X : \mathcal{C}, p : PX \rangle$ in the category of elements of P to the generalized elements of X that commute with p, forming points $f : \star \to F\langle X, p \rangle$.

$$\begin{tikzcd} \star \arrow[d, "f"'] \arrow[dr, "p"] & \\ F\langle X, p \rangle \arrow[r, hook] & PX \end{tikzcd}$$

Thus to make a functor forward, we take a functor $F : \mathrm{Psh}(\Sigma P) \to \mathcal{U}$ and make a presheaf over P by taking any object $X : \mathcal{C}$ and collecting all the generalized elements under F at each generalized element of X. This means

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

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

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

elements of X under Q that commute with p. This is the pullback

$$\begin{tikzcd}
\alpha_X^{-1}p \arrow[r] \arrow[d, hook, dashed] \arrow[dr, phantom, "\lrcorner", very near start] & \star \arrow[d, "p"] \\
QX \arrow[r] & PX
\end{tikzcd}$$

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 $X : \mathcal{C}$ 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 ⟩
```

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

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 `heq`uality 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 `heq`uality 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 Tidying the rest up

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 , nat_trans.app α _ 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 }})}
```

```
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) }
```

Such a