How to prove that something is a functor
an illustrated guide

January 31, 2021

Let's start with the usual boilerplate declaration:

```
module Streams where

open import Function
open import Data.Product

open import Relation.Binary.PropositionalEquality
open Relation.Binary.PropositionalEquality.≡-Reasoning
```

This is the minimal amount of code necessary to go on. Function contains the definition of function composition, identity, etc.; Product is the library for (indeed) categorical products; the usual PropositionalEquality allows for $\equiv$-reasoning.

The first piece of code we need is a record with the definition of functor: a functor is a map of categories sending objects to objects, endowed with an *action on morphisms*

$$\texttt{fmap} : \hom(A, B) \to \hom(FA, FB)$$

depending on the two implicit arguments A, B and sending an arrow $u : A \to B$ in the domain to an arrow $Fu : FA \to FB$ in the codomain (do not pay attention to the fact that aleft functors in Agda have the same domain and codomain Set).

```
record Functor (M : Set → Set) : Set₁ where
  field
    -- definition of fmap
    fmap : {A B : Set} → (A → B) → M A → M B
    -- functor laws
    fun-idty : ∀ {A : Set} {t : M A} → fmap {A} id t ≡ id t
    -- ^ M (id A) = id_{M a} for aleft object a
    fun-comp : ∀ {A B C : Set} {f : B → C} {g : A → B} {x : M A} →
      (fmap (f ∘ g)) x ≡ ((fmap f) ∘ (fmap g)) x
    -- M (f.g) = (M f).(M g) for aleft composable morphisms f,g
```

The next step to define the Stream functor is of course to define the coproduct, which is part of the definition

$$FS = \{\bot\} \cup A \times S$$

The coproduct of two objects A, B has two constructors, the *left injection* $i_L : A \to A \cup B$ and the *right injection* $i_R : B \to A \cup B$ (of course left and right are not well-defined concepts here, because the coproduct is symmetric; but still).

```
data _∪_ (A B : Set) : Set where
  left : A → A ∪ B
  rigt : B → A ∪ B
```

Finally, we define the 'terminal object' $\bot$, having of course a single term t, and then we're ready to define the F of our dreams.

```
data ⊥ : Set where
  t : ⊥

F : {A : Set} → Set → Set
F {A} S = ⊥ ∪ (A × S)
```

The proposition we have to prove, that F so defined is a functor, is easily stated and proved:

```
isFun : {A : Set} → Functor (F {A})
isFun {A} =
  record
    { fmap = fmap
    ; fun-idty = fun-idty
    ; fun-comp = fun-comp
    }
    where
      fmap : {X Y : Set} → (X → Y) → F {A} X → F {A} Y
      fmap u (left fx) = left fx
      fmap u (rigt fx) = rigt (proj₁ fx , u (proj₂ fx))
      fun-idty : ∀ {Z : Set} {x : F {A} Z} → fmap {Z} {Z} id x ≡ id x
      fun-idty {_} {left _} = refl
      fun-idty {_} {rigt _} = refl
      fun-comp : {X Y Z : Set} {u : Y → Z} {g : X → Y} {x : F {A} X} →
        (fmap {X} {Z} (u ∘ g)) x ≡ (fmap u ∘ fmap g) x
      fun-comp {u = u} {g = g} {left _} = refl
      fun-comp {u = u} {g = g} {rigt _} = refl
```

We can use a similar technique to prove that many other old friends in functional programming are functors: for example, the Maybe construction,

```
data Maybe (A : Set) : Set where
  nothing : Maybe A
```

```
  just : (a : A) → Maybe A

maybeIsFunctor : ∀ {A : Set} → Functor Maybe
maybeIsFunctor {A} = record
  { fmap = fmap
  ; fun-idty = fun-idty
  ; fun-comp = fun-comp
  }
  where
    fmap : {A B : Set} → (A → B) → Maybe A → Maybe B
    fmap {A} {B} f nothing = nothing
    fmap {A} {B} f (just a) = just (f a)
    --
    fun-idty : {A : Set} {x : Maybe A} → fmap {A} id x ≡ id x
    fun-idty {A} {nothing} = refl
    fun-idty {A} {just a} = refl
    --
    fun-comp : {A B C : Set} {f : B → C} {g : A → B} {x : Maybe A} →
      (fmap (f ∘ g)) x ≡ (fmap f ∘ fmap g) x
    fun-comp {A} {B} {C} {f} {g} {nothing} = refl
    fun-comp {A} {B} {C} {f} {g} {just a} = refl
```

The State functor is even a simpler one:

```
stateM : {S : Set} → Set → Set
stateM {S} X = S → S × X

stateIsFunctor : {B : Set} → Functor (stateM {B})
stateIsFunctor {B} =
  record
    { fmap = fmap
    ; fun-idty = refl
    ; fun-comp = refl
    -- the functor identities truly are simpler!  Can you figure out why?
    -- (hint:  does Maybe need case split?)
    }
  where
    fmap : {A B S : Set} → (A → B) → stateM {S} A → stateM {S} B
    fmap {A} {B} {S} u x = \t → (proj₁ (x t) , u (proj₂ (x t)))
```

Certainly, this description is hiding the most interesting part of the story under the carpet; Maybe and State are not just functors, they are monads. We will get back to it.

Now: have fun yourself, with the *continuation* monad:

```
contM : ∀ {B : Set} → Set → Set
contM {B} A = (A → B) → B
```

Fill the holes for

```
-- contIsFunctor :  ∀ {B : Set} → Functor (contM {B})
-- contIsFunctor {B} =
-- record
-- { fmap = {!  !}
-- -- ids
-- ; fun-idty = {!  !}
-- ; fun-comp = {!  !}
-- }
```

(just take the source of this file, uncomment the lines above, and tinker)