1 Introduction

2 Category theory preliminaries

This library defines various concepts of category theory. Every subsection will document in detail how a module has been implemented.

2.1 The module `Category`

We start with the most basic thing we can do, namely the definition of category. First things first, we start by defining our module.

```
module Basic.Category
 % access public export
 % default total
```

Then, we implement the basic elements a category consists of.

```
record Category where
```

A `record` in Idirs is just the product type of several values, which are called the fields of the record. It's a convenient syntax because Idris provides field access and update functions automatically for us. We add also the constructor `MkCategory` to be able to construct concrete values of type `Category`:

```
constructor MkCategory
```

2.1.1 The elements

At its most basic level, a category is a collection of things, called *objects*. To specify what these things are, we add to our definition a field to keep track of them:

```
obj : Type
```

You can think about `obj` as a collection of dots, which will be items we want to talk about in our category.

Next thing up, we need ways to go from one dot to the other, so that we can wander around our objects. This introduces some dynamics inside our category, which allows us to talk about movement and evolution of systems.

In practice, we need to describe, for any pair of objects a and b, the collection of *arrows* (also called *morphisms*) going from a to b. An arrow from a to b is sometimes mathematically denoted as $f : a \to b$ or more compactly as $a \xrightarrow{f} b$. Moreover, if we denote a category as $\mathcal{C}$, a convenient notation to denote *all* the arrows from object a to object b is $\mathcal{C}(a, b)$.

To translate this to Idris, let's add another field to our Category:

```
mor : obj → obj → Type
```

Now, for any pair of objects `a, b : obj`, we can talk about the collection `mor a b` of arrows going from a to b. This faithfully models, on the implementation side, what $\mathcal{C}(a, b)$ is on the theoretical side.

2.1.2 The operations

Now that we have arrows in our category, allowing us to go from one object to the other, we would like to start following consecutive arrows; I mean, if an arrow leads us to b, we would like to continue our journey by taking any other arrow starting at b. Nobody stops us from doing that, but it would be really cumbersome

if we must keep track of every single arrow whenever we want to describe a path from one dot to another. The definition of category comes in our help here, providing us with an operation to obtain arrows from paths, called *composition*.

```
compose : (a, b, c : obj)
  → (f : mor a b)
  → (g : mor b c)
  → mor a c
```

Furthermore, the constructor `MkCategory` asks to determine:

```
identity : (a : obj) → mor a a
```

Which for now is nothing more than a function that, for each object a, returns a morphism $a \to a$.

2.1.3 The laws

The part of the construction covered above defines the components of a category, but as they stand nothing ensures that the category axioms hold. For instance, there is nothing in principle that tells us that composing an identity with a morphism returns the morphism itself. This is the role of the remaining definition of the constructor `MkCategory`, ensuring that such axioms are enforced:

```
leftIdentity : (a, b : obj) → (f : mor a b) → compose a a b (identity a) f = f
rightIdentity : (a, b : obj) → (f : mor a b) → compose a b b f (identity b) = f
associativity : (a, b, c, d : obj)
              → (f : mor a b)
              → (g : mor b c)
              → (h : mor c d)
              → compose a b d f (compose b c d g h) = compose a c d (compose a b c f g) h
```

These lines are a bit different in concept: They eat type, but produce *equations*, effectively imposing further constraints on the components we defined before. Let's review them in detail.

- `leftIdentity` takes a couple of objects (specified as `a, b:obj`) and a morphism between them (specified as `f : mor a b`) and produces an equation proving that composing the morphism on the left with the identity on its domain amounts to doing nothing. This is akin to the commutativity of the familiar diagram:

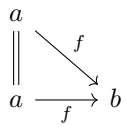


Figure 1: The equation $id_a; f = f$

- Right identity law is defined analogously by `rightIdentity`, modelling the commutative diagram:

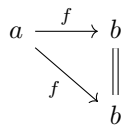


Figure 2: The identity laws $id_a; f = f$ and $f; id_b = f$

In Idris, we can state the two identity laws as follows:

```
leftIdentity : (a, b : obj)
  → (f : mor a b)
  → compose a a b (identity a) f = f
```

and

```
rightIdentity : (a, b : obj)
  → (f : mor a b)
  → compose a b b f (identity b) = f
```

In short, this amounts to say that $(\texttt{identity a}); \texttt{f} = \texttt{f} = \texttt{f}; (\texttt{identity b})$ for any morphism $\texttt{f} : \texttt{a} \to \texttt{b}$. As a technical side note, I'd like to emphasise here how Idris allows us to encode equality in the type system; from a practial point of view, equality in Idris is a type which has only one inhabitant, called `Refl`, which corresponding to reflexivity, and stating that $\texttt{x} = \texttt{x}$ for any possible `x`.

- Finally, the line

```
associativity : (a, b, c, d : obj)
  → (f : mor a b)
  → (g : mor b c)
  → (h : mor c d)
  → compose a b d f (compose b c d g h)
  = compose a c d (compose a b c f g) h
```

Imposes the familiar associativity law. It takes four objects and three morphisms between them, and produces an equation stating that the order of composition does not matter. This effectively models the commutative diagram:

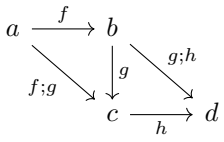


Figure 3: The associativity law $f; (g; h) = (f; g); h$

2.1.4 Conclusion

Summing up and putting it all together, our definition of `Category` now looks like this:

```
module Basic.Category
%access public export
%default total
record Category where
  constructor MkCategory
  obj          : Type
  mor          : obj → obj → Type
  identity     : (a : obj) → mor a a
  compose      : (a, b, c : obj)
    → (f : mor a b)
    → (g : mor b c)
    → mor a c
  leftIdentity : (a, b : obj)
    → (f : mor a b)
    → compose a a b (identity a) f = f
  rightIdentity : (a, b : obj)
    → (f : mor a b)
    → compose a b b f (identity b) = f
  associativity : (a, b, c, d : obj)
    → (f : mor a b)
    → (g : mor b c)
    → (h : mor c d)
    → compose a b d f (compose b c d g h) = compose a c d (compose a b c f g) h
```

2.2 The module `Functor`

In the last Section, we have defined a type `Category`, which is great since many different structures can be formulated as instances of categories. On the other hand, categories are structures themselves, and are themselves the object of a category, the category of categories! The goal of this Section is to define morphisms between categories, formally called *functors*.

A functor maps a category to another category in the very same way a function maps a set to another set. But, since categories have more structure than sets, we want this structure to be preserved in the process. As before we start declaring `Functor` as a module. Notice how, being `Functor` defined for categories, we import the module `Category` we already defined in the previous Section.

```
module Basic.Functor
import Basic.Category
%access public export
%default total
```

We will again employ `record` to implement functors, as we did for categories.

```
record CFunctor (cat1 : Category) (cat2 : Category) where
```

Notice how we had to use the name `CFunctor` to avoid clashings with `Functor`, which is a type already defined by Idris. We will moreover use `constructor` again to construct concrete values of type `CFunctor`.

```
constructor MkCFunctor
```

2.2.1 The components

All the preliminary work is done, let's now start with the real stuff. As we said, a functor between categories $\mathcal{C}$ and $\mathcal{D}$, formally denoted with $F : \mathcal{C} \to \mathcal{D}$, is a mapping. We begin by noticing how a category consists of objects and morphisms, so our functor will have to act both on objects *and* morphisms.

We start with the objects, where we declare our functor to be simply a map: If we have $F : \mathcal{C} \to \mathcal{D}$ then we map every object a of $\mathcal{C}$ to an object Fa of $\mathcal{D}$. In Idris, this becomes:

```
mapObj : obj cat1 → obj cat2
```

As we would expect, `mapObj` takes an object of `cat1`, extracted from the `Category` type using the `obj` value, and maps it to an object of `cat2`.

For morphisms, we do pretty much the same. Notice however that, given a functor $F : \mathcal{C} \to \mathcal{D}$, a morphism $f : a \to b$ has a domain and a codomain, namely the objects a, b of $\mathcal{C}$. We want to map f to a morphism of $\mathcal{D}$, but from what to what? Since we want to respect the structure it makes sense to use what we already defined for objects, and map $f : a \to b$ to a morphism $Ff : Fa \to Fb$.

In our implementation we model this as follows:

```
mapMor : (a, b : obj cat1)
  → mor cat1 a b
  → mor cat2 (mapObj a) (mapObj b)
```

`mapMor` is just a map between morphisms of `cat1` and morphisms of `cat2` which is consistent with respect to the map we already defined on objects This time though our implementation forces us to specify also objects `a` and `b` of `cat1` since the way we defined `mor` expects them.

2.2.2 The laws

Now, we recall that a category is not just a collection of objects and morphisms. We have indeed more structure we have to take into account, namely identities and composition. Since we want functors to be defined so that they respect the categorical structure, it makes sense to require that identities get carried to identities. This translates, for each object a of $\mathcal{C}$, in the equation between morphisms of $\mathcal{D}$:

$$Fid_a = id_{Fa}$$

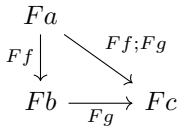
In Idris we represent this by defining a function that, for each object `a` of `cat1`, provides a proof that the identity on `a` in `cat1` is mapped to the identity on `mapObj a` in `cat2`:

```
preserveId : (a : obj cat1)
  → mapMor a a (identity cat1 a) = identity cat2 (mapObj a)
```

Finally, we must account for composition. The idea here is that, if functors respects the categorical structure, it shouldn't matter if we compose two morphisms first and then apply the functor or viceversa. Mathematically, we represent this, for $f : a \to b$ and $g : b \to c$ in $\mathcal{C}$, with the equation between morphisms of $\mathcal{D}$:

$$F(f;g) = Ff;Fg$$

If you like it more, this can also be depicted as a commutative diagram:



In Idris, we proceed in a way similar to what we did for identities:

```
preserveCompose : (a, b, c : obj cat1)
  → (f : mor cat1 a b)
  → (g : mor cat1 b c)
  → mapMor a c (compose cat1 a b c f g)
    = compose cat2 (mapObj a) (mapObj b) (mapObj c) (mapMor a b f) (mapMor b c g)
```

Given three objects `a, b, c : obj cat1`, which are the domains/codomains of the morphisms we are going to compose, and given two of these morphisms `f : mor cat1 a b` and `g : mor cat1 b c` respectively, we produce a proof that `mapMor` and `compose` commute with each other.

2.2.3 conclusion

The code covered above completes our definition of `CFunctor`, which we provide in its entirety below:

```
module Basic.Functor

import Basic.Category

%access public export
%default total

record CFunctor (cat1 : Category) (cat2 : Category) where
  constructor MkCFunctor
  mapObj      : obj cat1 → obj cat2
  mapMor      : (a, b : obj cat1)
    → mor cat1 a b
    → mor cat2 (mapObj a) (mapObj b)
  preserveId : (a : obj cat1)
    → mapMor a a (identity cat1 a) = identity cat2 (mapObj a)
  preserveCompose : (a, b, c : obj cat1)
    → (f : mor cat1 a b)
    → (g : mor cat1 b c)
    → mapMor a c (compose cat1 a b c f g)
            = compose cat2 (mapObj a) (mapObj b) (mapObj c) (mapMor a b f) (mapMor b c g)

functorEq :
  (cat1, cat2 : Category)
  → (fun1, fun2 : CFunctor cat1 cat2)
  → ((a : obj cat1) → (mapObj fun1 a = mapObj fun2 a))
  → ((a, b : obj cat1) → (f : mor cat1 a b) → (mapMor fun1 a b f = mapMor fun2 a b f))
  → fun1 = fun2
functorEq cat1 cat2 fun1 fun2 prfObj prfMor = really_believe_me ()

idFunctor : (cat : Category) → CFunctor cat cat
idFunctor cat = MkCFunctor
  id
  (λa, b ⇒ id)
  (λa ⇒ Refl)
  (λa, b, c, f, g ⇒ Refl)

functorComposition :
  (cat1, cat2, cat3 : Category)
  → CFunctor cat1 cat2
  → CFunctor cat2 cat3
  → CFunctor cat1 cat3
functorComposition cat1 cat2 cat3 fun1 fun2 = MkCFunctor
  ((mapObj fun2) ∘ (mapObj fun1))
  (λa, b ⇒ (mapMor fun2 (mapObj fun1 a) (mapObj fun1 b)) ∘ (mapMor fun1 a b))
  (λa ⇒ trans (cong (preserveId fun1 a)) (preserveId fun2 (mapObj fun1 a)))
  (λa, b, c, f, g ⇒ trans (cong (preserveCompose fun1 a b c f g))
    (preserveCompose fun2
      (mapObj fun1 a)
      (mapObj fun1 b)
      (mapObj fun1 c)
      (mapMor fun1 a b f)
      (mapMor fun1 b c g)))
```

2.3 The module NaturalTransformation

Perhaps unsurprisingly, after having defined categories and functors we switch to the next basic element of category theory, natural transformations. Natural transformations are defined between functors, and hence we start by importing what we did up to now, namely the modules Category and Functor.

```
module Basic.NaturalTransformation

import Basic.Category
import Basic.Functor

%access public export
%default total
```

As we did for the previous modules, to implement `NaturalTransformation` we will resort again to records and constructors. In the following snippet, you can see how the record `NaturalTransformation` is specified by two categories $\mathcal{C}, \mathcal{D}$, implemented as `cat1` and `cat2` respectively, and two functors $F, G : \mathcal{C} \to \mathcal{D}$ between them, implemented as `func1` and `func2`.

```
record NaturalTransformation
  (cat1 : Category)
  (cat2 : Category)
  (fun1 : CFunctor cat1 cat2)
  (fun2 : CFunctor cat1 cat2)
  where
    constructor MkNaturalTranformation
```

2.3.1 The components

Recall that, given functors $F, G : \mathcal{C} \to \mathcal{D}$, a natural transformation $\alpha : F \to G$ is defined by specifying, for each object A of $\mathcal{C}$, a morphism $\alpha_A : FA \to GA$ in $\mathcal{D}$, subject to some properties we will get to later. We define the components of a natural transformation, for each object `a` of `cat1`, as follows:

```
component : (a : obj cat1) → mor cat2 (mapObj fun1 a) (mapObj fun2 a)
```

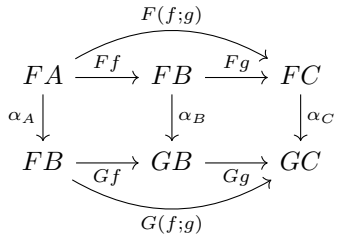
`mapObj fun1 a` means that we are applying `fun1` to the object `a`. This is akin to consider FA. Similarly, `mapObj fun2 a` is akin to consider GA. These two objects, belonging to `cat2` (standing for $\mathcal{D}$ in our mathematical definition), get fed to `mor` producing the homset of morphisms from FA to GA. a term of this type is just the implementation of a morphism $FA \to GA$, and it is precisely what we associate to an object `a`.

2.3.2 The laws

Up to now, we defined, for a natural transformation $\alpha : F \to G$, its components $\alpha_A : FA \to GA$ for each A object of $\mathcal{C}$. These components have to be related with each other by a property, stating that for each morphism $f : A \to B$ in $\mathcal{C}$ the following square commutes:

$$\begin{array}{ccc} FA & \xrightarrow{Ff} & FB \\ {\scriptstyle \alpha_A}\downarrow & & \downarrow{\scriptstyle \alpha_B} \\ FB & \xrightarrow[Gf]{} & GB \end{array}$$

This property lets us interpret a natural transformation as a way to link the result of applying F to the result of applying G in a way that cooperates with the structure, namely morphism composition: In fact, notice how the commutative square above guarantees that given $f : A \to B$ and $g : B \to C$ in $\mathcal{C}$, applying the natural transformation law above to $f; g$ has the same effect of pasting together the commutative squares for f and g, that is, the following diagram commutes:



In Idris, as we expect, this property is expressed by returning a proof of the equation expressed by the diagram above for each morphism f:

```
commutativity : {a, b : obj cat1}
  → (f : mor cat1 a b)
  → compose cat2
      (mapObj fun1 a)
      (mapObj fun2 a)
      (mapObj fun2 b)
      (component a)
      (mapMor fun2 a b f)
    = compose cat2
      (mapObj fun1 a)
      (mapObj fun1 b)
      (mapObj fun2 b)
      (mapMor fun1 a b f)
      (component b)
```

Here, we specify a morphism `f` from `a` to `b` in `cat1`. From this, we can consider $Ff : FA \to FB$, specified by `mapMor fun1 a b f`, and $Gf : GA \to GB$, specified by `mapMor fun2 a b f`. The equation modeled by the diagram above reads:

$$\alpha_A ; Gf = Ff ; \alpha_B$$

α_A and α_B are respectively `component a` and `component b` in our implementation. We can then apply `compose` to obtain the two sides of the equation, leading to the type

```
compose cat2
  (mapObj fun1 a)
  (mapObj fun2 a)
  (mapObj fun2 b)
  (component a)
  (mapMor fun2 a b f)
= compose cat2
  (mapObj fun1 a)
  (mapObj fun1 b)
  (mapObj fun2 b)
  (mapMor fun1 a b f)
  (component b)
```

2.3.3 Conclusion

The code above is everything we need to define what a natural transformation is. In the next sections, we will proceed by making this definition more specific and obtain a natural isomorphism. The code of this section, presented as a unique block, can be found below.

```
module Basic.NaturalTransformation

import Basic.Category
import Basic.Functor

%access public export
%default total

record NaturalTransformation
  (cat1 : Category)
  (cat2 : Category)
  (fun1 : CFunctor cat1 cat2)
  (fun2 : CFunctor cat1 cat2)
  where
    constructor MkNaturalTranformation
    component : (a : obj cat1) → mor cat2 (mapObj fun1 a) (mapObj fun2 a)
    commutativity : (a, b : obj cat1)
      → (f : mor cat1 a b)
      → compose cat2
          (mapObj fun1 a)
          (mapObj fun2 a)
          (mapObj fun2 b)
          (component a)
          (mapMor fun2 a b f)
        = compose cat2
          (mapObj fun1 a)
          (mapObj fun1 b)
          (mapObj fun2 b)
          (mapMor fun1 a b f)
          (component b)
```