1 Introduction

2 Category theory preliminaries

Before starting to develop the Statebox dependently typed core, we need to implement some basic categorical definitions which will allow us to stay as faithful as possible to the categorical model outlined in the Statebox Monograph. This section is devoted precisely to this task. 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 Category
 %access public export
 %default total
```

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

```
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
```

Let's look at this implementation more in detail, starting from line one.

```
record Category where
```

We define a category as a type, to be precise a `record`. A record type allows to aggregate values together. In our case said values represent all the main ingredients that make up a category – morphisms, objects, etc. These are implemented in the record using the constructor `MkCategory`.

```
constructor MkCategory
obj : Type
mor : obj → obj → Type
```

The main ingredients to model are objects and morphisms: We give objects a type `obj` and morphisms a type `obj → obj → Type` – that is, morphisms are interpreted as functions that take two objects representing domain and codomain and return a type.

Furthermore, the constructor `MkCategory` asks to determine:

- For each object, a selected identity morphism. This is represented by

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

  Which is a function that, for each object a, returns a morphism $a \to a$.

- For each couple of morphisms such that their domain and codomain match up, their composition. This is represented by

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

  Wich is again a function. It asks to determine some objects and a couple of functions having the objects as domain/codomain such that they match up. The result, as we would expect, is the composition of the specified morphisms.

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:

$$\begin{tikzcd}
a \arrow[d, equal] \arrow[dr, "f"] & \\
a \arrow[r, "f"'] & b
\end{tikzcd}$$

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

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

$$\begin{tikzcd}
a \arrow[r, "f"] \arrow[dr, "f"'] & b \arrow[d, equal] \\
 & b
\end{tikzcd}$$

Figure 2: The equation $f; id_b = f$

- 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

$$\begin{tikzcd}
a \arrow[r, "f"] \arrow[dr, "f;g"'] & b \arrow[d, "g"] \arrow[dr, "g;h"] & \\
 & c \arrow[r, "h"'] & d
\end{tikzcd}$$

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