1 Introduction

2 Category theory preliminaries

Before starting to develop the Statebox 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. We start with the definition of category.

```
module Category

%access public export
%default total

data Category : (obj : Type) -> (mor : obj -> obj -> Type) -> Type where
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)
  -> Category obj mor
```

We define a category as a type. It requires to specify a set of objects and, for each object, a set of morphisms implemented as functions that take two objects in input and return a type as output. Category has one constructor, which specifies identities on objects as functions (to each object corresponds an identiy morphism). Similarly it specifies compositions of morphisms (provided three objects and two morphism between them with matching domain/codomain, a composition object is produced).

At the moment nothing ensures that Category has is a category, because identity and associativity laws are not enforced. To solve this, we start by implementing the laws:

```
LeftIdentity :
  {obj : Type}
  -> {mor : obj -> obj -> Type}
  -> {a, b : obj}
  -> (f : mor a b)
  -> Category obj mor
  -> Type
LeftIdentity {obj} {mor} {a} {b} f (MkCategory identity compose) =
  compose a a b (identity a) f = f


RightIdentity :
  {obj : Type}
  -> {mor : obj -> obj -> Type}
  -> {a, b : obj}
```

```
    → (f : mor a b)
    → Category obj mor
    → Type
  RightIdentity {obj} {mor} {a} {b} f (MkCategory identity compose) =
    compose a b b f (identity b) = f
```

The left identity law takes a Category and one of its morphisms, and produces an equation proving that composing the morphism on the left with the identity morphism on its domain amounts to doing nothing. Right identity law is defined analogously:

It remains to implement associativity, which is done as follows:

```
  Associativity :
    {obj : Type}
    → {mor : obj → obj → Type}
    → {a, b, c, d : obj}
    → {f : mor a b}
    → {g : mor b c}
    → {h : mor c d}
    → Category obj mor
    → Type
  Associativity {obj} {mor} {a} {b} {c} {d} {f} {g} {h} (MkCategory identity compose) =
    compose a b d f (compose b c d g h) = compose a c d (compose a b c f g) h
```

Unsurprisingly, associativity takes a category and produces a proof that for each triple of morphisms with matching domains and codomains, the order of composition does not matter.

We can now combine the Category type with the laws just implemented to obtain a VerifiedCategory type, which is a Category obeying the unit and associativity laws:

```
  data VerifiedCategory : (obj : Type) → (mor : obj → obj → Type) → Type where
    MkVerifiedCategory :
      (cat : Category obj mor)
      → ((a, b : obj) → (f : mor a b) → LeftIdentity f cat)
      → ((a, b : obj) → (f : mor a b) → RightIdentity f cat)
      → ((a, b, c, d : obj) → (f : mor a b) → (g : mor b c) → (h : mor c d) → Associativity {f} {g} {
      → VerifiedCategory obj mor
```

As you can see, here the constructor requires that morphism obey the associativity and unit laws we defined before. We conclude the section by defining InnerCategory, which takes a VerifiedCategory and strips the verified part away, producing the underlying Category structure. InnerCategory greatly simplifies life when we have to define more complicated mathematical objects such as functors.

```
  innerCategory : VerifiedCategory obj mor → Category obj mor
  innerCategory (MkVerifiedCategory cat _ _ _) = cat
```