Functors and Applicative Functors

Donovan Crichton

August 2022

Preliminaries

- Slides and Examples available at:
  https://github.com/donovancrichton/ANU-FP
- This talk: /FunctorsAndApplicatives

What is a type class?

recall:

```
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
```

Type classes for programmers

- A way to introduce ad-hoc polymorphism to our functions.
- Specify a 'class' of behaviours for particular types.
- Allow operator and function overloading (e.g + on Double and Int)
- We can use any type that *subscribes* to the class.

```haskell
f :: Num a => a -> a
f x = x + x
```

Type classes for mathematicians

- A way to define our own algebra! (almost)
- Recall that an algebra is a triple (A, F, L)
- A is our carrier set.
- F is a set of operations on A.
- L is our set of axioms or laws for F.
- This is why ':doc' in Haskell gives us something interesting.

Functors in Haskell

```
Prelude> :doc Functor
 The 'Functor' class is used for types that can be mapped
 over. Instances of 'Functor' should satisfy the
 following laws:

> fmap id  ==  id
> fmap (f . g)  ==  fmap f . fmap g

The instances of 'Functor' for lists,
'Data.Maybe.Maybe' and 'System.IO.IO'
satisfy these laws.
Prelude> :info Functor
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a -> f b -> f a
  {-# MINIMAL fmap #-}
```

What is a functor?

- A morphism between categories (of course!)
- Inspired by functors from category theory.
- A structure preserving map from objects in FA to objects in FB
- $map : (a \xrightarrow{g} b) \rightarrow F(a) \xrightarrow{F(g)} F(b)$ where F is a functor.

Functors in Idris

```
interface Prelude.Functor : (Type -> Type) -> Type
Functors allow a uniform action over a parameterised type.
@ f a parameterised type
Parameters: f
Constructor: MkFunctor
Methods:
  map : (a -> b) -> f a -> f b
    Apply a function across everything of type 'a' in a
      parameterised type
    @ f the parameterised type
    @ func the function to apply
```

Functors as an algebra

let *map* (also called $\langle\$\rangle$) : $(a \xrightarrow{g} b) \rightarrow F(a) \xrightarrow{F(g)} F(b)$ where F is a functor.

let
$F = (A, \{map\}, \{map(id) = id, map(g \circ h) = map(g) \circ map(h)\})$

So we have our carrier set A, one operation *map*, and two axioms/laws.

Legal Functors in Idris

Here we show a demo on a legal Functor definition in Idris with an example candidate implementation on lists.

Applicative Functors

An Applicative Functor F is another common algebra over a carrier set A. With one binary operation and one unary operation:

$$pure : A \to F(A)$$

$$apply \text{ (also called } \langle * \rangle) : F(A \xrightarrow{g} B) \to F(A) \xrightarrow{F(g)} F(B)$$

And our set of axioms:

$$\begin{aligned}
&\text{Identity:} \\
apply(pure(id), v) &= v \\
&\text{Composition:} \\
apply(pure(\circ, apply(u, apply(v, w))) &= apply(u, (apply(v, w))) \\
&\text{Homomorphism:} \\
apply(pure(f), pure(x)) &= pure(f(x)) \\
&\text{Interchange:} \\
apply(u, pure(y)) &= apply(pure(\lambda f . f(y)), u)
\end{aligned}$$

Applicative Functors 2

Our Algebra for Applicative Functors is formally:
Let $F = (A, \{pure, \langle * \rangle\}, \{id, com, hom, int\})$

Why is the Applicative algebra also a Functor?
We can define *map* as follows:
$g < \$ > x = pure(g)\langle * \rangle x$

Let's see the demo!