PROGRAMMING IN HASKELL

Chapter 12 – Functor Classes, Applicatives, and Monads

Introduction

We look at several classes for functions (methods) that are generic over a range of parameterised types such as lists, trees, and input/output action.

These are functors, applicatives, and monads.

Generic, higher-order notions of mapping, function application, and effectful programming.

Every monad must be applicative, and every applicative must be a functor. So, we start with functor.

Parameterised types

Recall from the discussion of types and classes:

We have *types* such as Int, Bool, Char,....

We can also declare types:

```
type String = [Char]
```

Types declarations can also be parameterised with respect to other types:

```
type Pair a =(a,a)
```

Class declarations

Recall from the discussion of types and classes:

a *class* (e.g. Eq, Ord, Show,...)

is a collection of *types* (e.g. Int, Bool, Char,...)

that support overloaded operations called

methods (e.g. <=, >, =>, min, max,...).

Class declarations

We can declare a class:

```
class Eq a where
      (==), (/=) :: a -> a -> Bool
```

For a to be an instance of the class Eq, it must support equality and inequality operators (methods, functions), which are of the specified type.

As we saw with Pair, types can be parameterised.

Towards functors

Example of abstracting a pattern.

```
inc :: [Int] -> [Int]
inc []          = []
inc (n:ns)      = n + 1 : inc ns
```

```
sqr :: [Int] -> [Int]
sqr []          = []
sqr (n:ns)      = n^2 : sqr ns
```

Towards functors

Abstracting using the library function map.

```
map :: (a -> b) -> [a] -> [b]
map f []         = []
map f (x:xs)     = f x : map f ns
```

We can use this represent the others:

```
inc = map (+1)
sqr = map (^2)
```

Abstracting to Functors

Mapping may apply to a wide range of parameterised datatypes; generalise to fmap, which is polymorphic.

These are the class of types that support such a mapping – functors.

```
class Functor f where
      fmap :: (a -> b) -> f a -> f b
```

After Functor, we get a type constructor f, e.g. [] or Maybe.

Functors

f is not a concrete type (a type that a value can hold, like Int, Bool or Maybe String), but a type constructor that takes one type parameter.

Maybe Int is a concrete type, but Maybe is a type constructor that takes one type as the parameter.

f a and f b are parameterised type constructors, e.g. if f is [ ] and a is Int, then f a is [Int].

```
class Functor f where
      fmap :: (a -> b) -> f a -> f b
```

Functors

map is just fmap that only works on lists.

We pass [] as a type constructor over any concrete type, e.g. Int, Char, ...

```
instance Functor [] where
     fmap = map
```

map is an instance of the Functor class with respect to lists (leaving aside what map does and the types of elements of the list).

Functors

```
map :: (a -> b) -> [a] -> [b]
ghci> fmap (*2) [1..3]
[2,4,6]
ghci> map (*2) [1..3]
[2,4,6]
```

Same output.

Functors

Functions where the parameter can act (informally) like a “box” or a “computational context” can be functors.

A box or context has “compartments” that can hold (or not) some information.

A list is a “context”.

fmap takes a function a -> b, applies the function to the contents of the first context, and produces the second context and its contents.

Functors

```
instance Functor Maybe where
      -- fmap :: (a -> b) -> Maybe a -> Maybe b
      fmap g (Just x) = Just (g x)
      fmap _ Nothing = Nothing
```

Maybe is a context that can either hold nothing or something: where nothing, the value is Nothing; where an item, like ”Bill", the value is Just ”Bill”.

Other types that can act like a context and so can be functors: Tree, Either a, data, IO, and others (that we have not looked at).

Functor Laws

An interesting aspect of Haskell functions is that equational laws can be defined about the behaviour of functions:

```
fmap id          = id
fmap (g . h)     = fmap g . fmap h
```

fmap must preserve the identity function and function composition.
This also ensures that fmap preserves the structure of the argument list wrt order and contents.

Applicatives

Continuing to have higher level abstraction of functions, suppose rather than applying an fmap function to a single argument, we want a hierarchy of fmap functions that apply map to any number of arguments.

Applicatives

```
fmap0 :: a -> f a
fmap1 :: (a -> b) -> f a -> f b
fmap2 :: (a -> b -> c) ->
     f a -> f b -> f c
fmap3 :: (a -> b -> c -> d) ->
     f a -> f b -> f c -> f d
etc

fmap0 is degenerate, without arguments.
fmap1 is our earlier fmap.
```

Applicatives

We want a general solution.
It uses the following two basic functions:

```
pure :: a -> f a
(<*>) :: f (a -> b) -> f a -> f b
```

pure converts a value of type a to a structure (context) of type f a, i.e. f parameterised to a.

<*> is a generalised form of (mapping) function application, where the argument function, argument value, and result value are all in f structures (contexts).

Applicatives

<*> is a function on two arguments. So, it can be written in a curried form with implicit brackets (functions are left associative):

```
(((g <*> x) <*> y) <*> z)

same as

g <*> x <*> y <*> z
```

Consider (g <*> x) has output g_x that is input to (g_x <*> y) and so on.

Applicatives

```
fmap0 :: a -> f a
fmap0 = pure

fmap1 :: (a -> b) -> f a -> f b
fmap1 g x = pure g <*> x

fmap2 :: (a -> b -> c) ->
      f a -> f b -> f c
fmap2 g x y = pure g <*> x <*> y

fmap3 :: (a -> b -> c -> d) ->
      f a -> f b -> f c -> f d
fmap3 g x y z = pure g <*> x <*> y <*> z
```

Applicatives type declaration

```
instance Functor f => Applicative f where
     pure :: a -> f a
     (<*>) :: f (a -> b) -> f a -> f b
```

Applicatives are functors where pure and <*> hold.

One need not define flexible mapping functions, but once a mapping function is defined as an instance of the applicable class, the functions are constructed on the fly.

Applicative Example

The instance on list types:

```
instance Applicative [] where
     pure x = [x]
     gs <*> xs = [g x | g <- gs, x <- xs]
```

pure transforms a value (x) into a singleton list ([x]).
<*> takes a list of functions and a list of arguments, then applies each function to each argument in turn, returning all the results in a list.

Applicative Example

List comprehension:

```
prods :: [Int] -> [Int] -> [Int]
prods xs ys = [x*y | x <- xs, y <- ys]
```

Applicative:

```
prods :: [Int] -> [Int] -> [Int]
prods xs ys = pure (*) <*> xs <*> ys
```

This definition avoids having to name the intermediate results as in list comprehension.

Applicatives

It is said that the applicative style supports a form of non-deterministic programming in which we can apply functions to multi-valued arguments without the need to manage the selection of the values or the propogation of failure, as this is taken care of by the applicative machinery.

This is particularly useful for IO types, where there can be errors or open numbers of arguments.

Applicative Laws

```
pure id <*> x       =     x
pure (g x)          =     pure g <*> pure x
x <*> pure y        =     pure (\g -> g y) <*> x
x <*> (y <*> z)     =     (pure (.) <*> x <*> y) <*> z
```

- pure preserves identity.
- pure preserves function application.
- the order of application of function to argument does not matter.
- <*> is associative (depending on type).

These are said to formalise the notion of pure, which embeds values of type a into the pure fragment of a context of type f a.

Monads