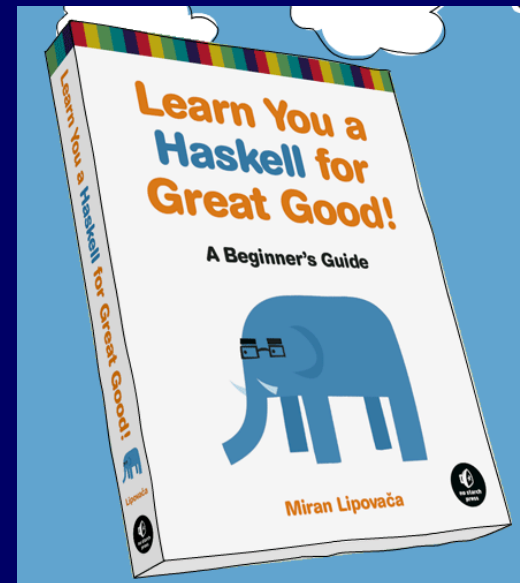
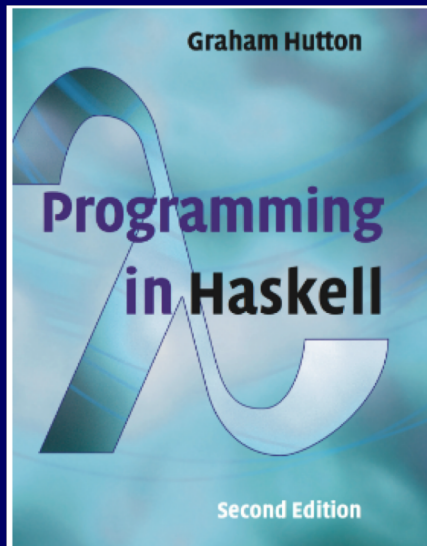


PROGRAMMING IN HASKELL

Functors and monads



Based on lecture notes by Graham Hutton,
the book “Learn You a Haskell for Great Good”
(and a few other sources)

Functors

Functors are a typeclass, just like Ord, Eq, Show, and all the others. This one is designed to hold things that can be mapped over; for example, lists are part of this typeclass.

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

Essentially, fmap promotes an “ordinary” function, that takes $a \rightarrow b$, to a function that works over values in a context.

Map over Lists

`map f xs` applies `f` over all the elements of the list `xs`

`map :: (a -> b) -> [a] -> [b]`

`map _ [] = []`

`map f (x:xs) = f x : map f xs`

```
>map (+1) [1,2,3]  
[2,3,4]
```

```
>map even [1,2,3]  
[False,True,False]
```

Map over Binary Trees

Remember binary trees with data in the inner nodes:

```
data Tree a = Leaf
```

```
          | Node (Tree a) a (Tree a)
```

```
          deriving Show
```

They admit a similar map operation:

```
mapTree :: (a -> b) -> Tree a -> Tree b
```

```
mapTree _ Leaf = Leaf
```

```
mapTree f (Node l x r) =
```

```
    Node (mapTree f l) (f x) (mapTree f r)
```

Map over optional values

Optional values are represented with Maybe data

Maybe a = Nothing | Just a

How does a map operation over optional values look like?

Map over optional values

Optional values are represented with Maybe data

Maybe a = Nothing | Just a

How does a map operation over optional values look like?

```
mapMay :: (a -> b) -> Maybe a -> Maybe b  
mapMay _ Nothing = Nothing  
mapMay f (Just x) = Just (f x)
```

Map over optional values

mapMay applies a function over a value, only if it is present

```
>mapMay (+1) (Just 1)  
Just 2
```

```
>mapMay (+1) Nothing  
Nothing
```

Maps have similar types

`map :: (a -> b) -> [a] -> [b]`

`mapTree :: (a -> b) -> Tree a -> Tree b`

`mapMay :: (a -> b) -> Maybe a -> Maybe b`

`mapT :: (a -> b) -> T a -> T b`

The difference lies in the type constructor

- ▶ `[]` (list), `Tree`, or `Maybe`
- ▶ Those parts need to be applied to other types

Functors

A type constructor which has a “map” is called a functor

```
class Functor f where
```

```
  fmap :: (a -> b) -> f a -> f b
```

```
instance Functor [] where
```

```
  -- fmap :: (a -> b) -> [a] -> [b]
```

```
  fmap = map
```

```
instance Functor Maybe where
```

```
  -- fmap :: (a -> b) -> Maybe a -> Maybe b
```

```
  fmap = mapMay
```

Higher Kinded Abstraction

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

In Functor the variable f stands for a type constructor

- ▶ A “type” which needs to be applied
- ▶ This is called higher-kinded abstraction
 - ▶ Not generally available in a programming language
 - ▶ Haskell, Scala and Rust have it
 - ▶ Java, C# and Swift don't

Functors generalize maps

Suppose you have a function operating over lists

```
inc :: [Int] -> [Int]
inc xs = map (+1) xs
```

You can easily generalize it by using fmap

```
inc :: Functor f => f Int -> f Int
inc xs = fmap (+1) xs
```

Note that in this case the type of elements is fixed to Int, but the type of the structure may vary

(<\$>) instead of fmap

Many Haskellers use an alias for fmap

```
(<$>) = fmap
```

This allows writing maps in a more natural style, in which the function to apply appears before the arguments

```
inc xs = (+1) <$> xs
```

Back to trees

```
data Tree a = EmptyTree
  | Node a (Tree a) (Tree a)
  deriving (Show,Read,Eq)
```

An example run:

```
ghci> let nums = [8,6,4,1,7,3,5]
```

```
ghci> let numsTree = foldr treeInsert EmptyTree  
      nums
```

```
ghci> numsTree
```

```
Node 5 (Node 3 (Node 1 EmptyTree EmptyTree) (Node 4 EmptyTree EmptyTree)) (Node 7 (Node 6 EmptyTree EmptyTree) (Node 8 EmptyTree EmptyTree))
```

Back to functors:

If we looked at fmap as though it were only for trees, it would look something like:

(a \rightarrow b) \rightarrow Tree a \rightarrow Tree b

We can certainly phrase this as a functor, also:

instance Functor Tree where

```
fmap f EmptyTree = EmptyTree
```

```
fmap f (Node x leftsub rightsub) =  
    Node (f x) (fmap f leftsub)  
              (fmap f rightsub)
```

Using the tree functor:

```
ghci> fmap (*2) EmptyTree  
EmptyTree
```

```
ghci> fmap (*4) (foldr treeInsert  
                  EmptyTree [5,7,3,2,1,7])
```

```
Node 28 (Node 4 EmptyTree (Node 8 EmptyTree (N  
ode 12 EmptyTree (Node 20 EmptyTree EmptyTree  
)))) EmptyTree
```


Functor laws

There are two laws any functor MUST follow if you define them:

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

- If we can show that some type obeys both functor laws, we can rely on it having the same fundamental behaviors as other functors when it comes to mapping.
- We can know that when we use `fmap` on it, there won't be anything other than mapping going on behind the scenes and that it will act like a thing that can be mapped over, i.e. a functor.
- This leads to code that is more abstract and extensible, because we can use laws to reason about behaviors that any functor should have and make functions that operate reliably on any functor.

Takeaway: WHY?

- The availability of the `fmap` method relieves us from having to recall, read, and write a plethora of differently named mapping methods (`maybeMap`, `treeMap`, `weirdMap`, and `infinitem`). As a consequence, code becomes both cleaner and easier to understand. On spotting a use of `fmap`, we instantly have a general idea of what is going on. Thanks to the guarantees given by the functor laws, this general idea is surprisingly precise.
- Using the type class system, we can write `fmap`-based algorithms which work out of the box with any functor - be it `[]`, `Maybe`, `Tree` or whichever you need. Indeed, a number of useful classes in the core libraries inherit from `Functor`.