Lecture 10. Functors and monads

Functional Programming

Utrecht University

Goals

- Understand the concept of *higher-kinded* abstraction
- Introduce two common patterns: *functors* and *monads*
- Simplify code with monads

Chapter 12 from Hutton's book, except 12.2

Functors

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 optional values

Optional values are represented with `Maybe`

```
data Maybe a = Nothing | Just a
```

They admit a similar map operation:

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

Map over optional values

Optional values are represented with `Maybe`

```haskell
data Maybe a = Nothing | Just a
```

They admit a similar map operation:

```haskell
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
```

It is similar to the “safe dot” operator in some languages

```
int Total(Order o) {  // o might be null
    return o?.Amount * o?.PricePerUnit;
}
```

Map over binary trees

Remember binary trees with data in the inner nodes:

```haskell
data Tree a = Leaf
            | Node (Tree a) a (Tree a)
            deriving Show
```

What does a map operation over trees look like?

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
```

What does a map operation over trees look like?

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

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
```

What does a map operation over trees look like?

```
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 binary trees

`mapTree` also applies a function over all elements, but now contained in a binary tree

```
> t = Node (Node Leaf 1 Leaf) 2 Leaf
```

```
> mapTree (+1) t
Node (Node Leaf 2 Leaf) 3 Leaf
```

```
> mapTree even t
Node (Node Leaf False Leaf) True Leaf
```

Maps have similar types

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

```haskell
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

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

You can easily generalize it by using `fmap`

```haskell
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
```

Surprising functors, take 1

Functions with a fixed input are also functors

- Remember that `r -> s` is also written `(->) r s`

Question

What type should we write in the `Functor` instance?

Surprising functors, take 1

Functions with a fixed input are also functors

- Remember that `r -> s` is also written `(->) r s`

Question

What type should we write in the `Functor` instance?

Answer

We need something which requires a parameter

- Thus we drop the last one from the arrow, `(->) r`

Surprising functors, take 1

```haskell
instance Functor ((->) r) where
  -- fmap :: (a -> b) -> (r -> a) -> (r -> b)
  fmap ab ra = \r -> ab (ra r)
```

The map operation for functions is composition!

Surprising functors, take 2

`IO` actions form also a functor

```
instance Functor IO where
...
```

Surprising functors, take 2

`IO` actions form also a functor

```haskell
instance Functor IO where
  -- fmap :: (a -> b) -> IO a -> IO b
  fmap f a = do x <- a
                return (f x)
```

This removes the need for a lot of names

```haskell
do x <- getChar         ===>  toUpper <$> getChar
   return (toUpper x)
```

and it is much easier to read and follow!

Functor laws

Valid `Functor` instances should obey two laws

identity	`fmap id = id`
distributivity over composition	`fmap (f.g) = fmap f . fmap g`

These laws guarantee that `fmap` preserves the structure

A wrong `Functor`

Could you find an instance which respects the type of `fmap` but not the laws?

A wrong Functor

Could you find an instance which respects the type of `fmap` but not the laws?

```
instance Functor [] where
  -- Applies the function over all elements,
  -- but also reverses the list
  fmap _ []     = []
  fmap f (x:xs) = fmap f xs ++ [f x]
```

A wrong Functor

Could you find an instance which respects the type of `fmap` but not the laws?

```haskell
instance Functor [] where
  -- Applies the function over all elements,
  -- but also reverses the list
  fmap _ []     = []
  fmap f (x:xs) = fmap f xs ++ [f x]
```

```haskell
fmap id [1,2]  = [2,1]
              /= [1,2] = id [1,2]
```

Another wrong Functor

Things can go wrong in many different ways

```
instance Functor [] where
  -- Always returns an empty list
  fmap _ _ = []
```

```
fmap id [1,2]  = []
              /= [1,2] = id [1,2]
```

Monads

Case study: evaluation of arithmetic expressions

```
data ArithOp   = Plus | Minus | Times | Div
data ArithExpr = Constant Integer
               | Variable Char
               | Op ArithOp ArithExpr ArithExpr
```

Case study: evaluation of arithmetic expressions

```
data ArithOp   = Plus | Minus | Times | Div
data ArithExpr = Constant Integer
               | Variable Char
               | Op ArithOp ArithExpr ArithExpr

eval :: Map Char Integer -> ArithExpr
     -> Maybe Integer
```

Case study: evaluation of arithmetic expressions

```
data ArithOp   = Plus | Minus | Times | Div
data ArithExpr = Constant Integer
               | Variable Char
               | Op ArithOp ArithExpr ArithExpr

eval :: Map Char Integer -> ArithExpr
     -> Maybe Integer

eval m (Op Plus x y)
  = case eval m x of
      Nothing -> Nothing
      Just x' -> case eval m y of
                   Nothing -> Nothing
                   Just y' -> Just (x' + y')
...
```

Validation of data

```haskell
data Record = Record Name Int Address

-- These three validate input from the user
validateName :: String -> Maybe Name
validateAge  :: String -> Maybe Int
validateAddr :: String -> Maybe Address

-- And we want to compose them together
case validateName nm of
  Nothing  -> Nothing
  Just nm' -> case validateAge ag of
    Nothing  -> Nothing
    Just ag' -> case validateAddr ad of
      Nothing  -> Nothing
      Just ad' -> Just (Record nm' ag' ad')
```

Looking for similarities

The same pattern occurs over and over again

```
case maybeValue of
  Nothing -> Nothing
  Just x  -> -- return some Maybe which uses x
```

Looking for similarities

The same pattern occurs over and over again

```haskell
case maybeValue of
  Nothing -> Nothing
  Just x  -> -- return some Maybe which uses x
```

Higher-order functions to the rescue!

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

Shorter code for the examples

For the arithmetic expression evaluator:

```
eval m (Op Plus x y)
  = eval m x `next` (\x' ->
     eval m y `next` (\y' ->
      Just (x' + y') ) )
```

For data validation:

```
validateName nm `next` (\nm' ->
 validateAge ag  `next` (\ag' ->
  validateAddr ag `next` (\ad' ->
   Just (Record nm' ag' ad') )))
```

Does it sound familiar?

Remember the “bind” operation for input/output actions

```
(>>=) :: IO    a -> (a -> IO    b) -> IO    b
```

Now, compare it to the `next` operation for `Maybe`

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

Another example of *higher-kinded abstraction*

`return` for optional values

The other basic operation for `IO` was `return`

```haskell
return :: a -> IO a
```

This function embeds a pure value into the `IO` world

`return` for optional values

The other basic operation for `IO` was `return`

```
return :: a -> IO a
```

This function embeds a pure value into the `IO` world

Optional values provide a similar function

```
Just   :: a -> Maybe a
```

`return` for optional values

The other basic operation for `IO` was `return`

```
return :: a -> IO a
```

This function embeds a pure value into the `IO` world

Optional values provide a similar function

```
Just   :: a -> Maybe a
```

Maybe it is about time to introduce a new type class…

(>>=) + return = monad

A **monad** is a type constructor which provides the previous two operations

- Subject to some laws that we shall introduce later
- In addition, every monad is also a functor

```
class Functor m => Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b

instance Monad Maybe where
  return = Just
  (>>=)  = next

instance Monad IO where
  -- Hidden from us, mere mortals
```

do-notation for generic monads

The do-notation introduced for `IO` works for any monad

```
do x1 <- a1                  a1 >>= (\x1 ->
   x2 <- a2                   a2 >>= (\x2 ->
   ...             ===>        ...
   xn <- an                     an >>= (\xn ->
   expr                          expr) ... ))
```

Rule of thumb for writing monadic code: do not think about nested `(>>=)` at all, just use do

Shorter (and nicer) code for the examples

For the arithmetic expression evaluator:

```
eval m (Op Plus x y) = do x' <- eval m x
                          y' <- eval m y
                          return (x' + y')
```

For data validation:

```
do nm' <- validateName nm
   ag' <- validateAge  ag
   ad' <- validateAddr ad
   return (Record nm' ag' ad')
```

Tricky monadic questions

What does the following code do?

```
f :: Maybe Int -> Maybe Int
f m = do x <- m
         return 3
         return (x + 1)
```

Tricky monadic questions

What does the following code do?

```
f :: Maybe Int -> Maybe Int
f m = do x <- m
         return 3
         return (x + 1)
```

Solution

Adds 1 to the value in `m`, if present

- `return` does **not** break evaluation
- So it does not always return 3

Tricky monadic questions

```haskell
f :: Maybe Int -> Maybe Int
f m = do x <- m
         return 3
         return (x + 1)
```

The behavior is clear by looking at the translation

- `<-` are turned into nested `(>>=)`
- `return` for `Maybe` is `Just`

```haskell
f m = m >>= \x ->
        Just 3 >>= \_ ->  -- "gets" the 3
          Just (x + 1)
```

Tricky monadic questions

Is the following code type correct at all?

```haskell
g :: Maybe Int -> Maybe Int
g m = do x <- return 3
         y <- m
         return (x + y)
```

Tricky monadic questions

Is the following code type correct at all?

```haskell
g :: Maybe Int -> Maybe Int
g m = do x <- return 3
         y <- m
         return (x + y)
```

And what about the following variation?

```haskell
g' :: Maybe Int -> Maybe Int
g' m = do x <- Just 3
          y <- m
          return (x + y)
```

Tricky monadic questions

Does this code compile?

```haskell
h :: Maybe Int -> IO Int -> Maybe Int
h x y = do x' <- x
           y' <- y
           return (x' + y')
```

Tricky monadic questions

Does this code compile?

```
h :: Maybe Int -> IO Int -> Maybe Int
h x y = do x' <- x
           y' <- y
           return (x' + y')
```

Solution

No, a do block works only with *one* monad

- The first `<-` and `return` require `Maybe`
- The second `<-` requires `IO`

The List monad

Building the Monad `[]` instance

Let us try to write the methods from their types

```
return :: a -> [a]
return x = _
```

Building the Monad `[]` instance

Let us try to write the methods from their types

```
return :: a -> [a]
return x = _
```

We only have two options:

- Return the empty list, `[]`
- Return the given element repeated some amount of times, `[x, ...]`

In this case, we settle for `[x]`, a singleton list

- It is the only possibility to satisfy the laws
  - But I will not show you why

Building the Monad [] instance

```haskell
(>>=) :: [a] -> (a -> [b]) -> [b]
xs >>= f = ...
```

Building the Monad `[]` instance

```haskell
(>>=) :: [a] -> (a -> [b]) -> [b]
xs >>= f = ...
```

1. We have a list of as and a function which operate in one
   - The natural instinct is to map one over the other
2. But `map f xs :: [[b]]`, a list of lists
3. Luckily, we have `concat :: [[a]] -> [a]`

```haskell
xs >>= f = concat (map f xs)
```

What does the List monad model?

```haskell
[1,2,3] >>= \x ->        do x <- [1,2,3]
 [4,5,6] >>= \y ->          y <- [4,5,6]
  return (x + y)            return (x + y)
= -- definition of (>>=) and return
[5,6,7,6,7,8,7,8,9]
=
[1+4,1+5,1+6,2+4,2+5,2+6,3+4,3+5,3+6]
```

Lists model search and non-determinism

```
[1,2,3] >>= \x ->        do x <- [1,2,3]
 [4,5,6] >>= \y ->          y <- [4,5,6]
  return (x + y)            return (x + y)
= -- definition of (>>=) and return
[5,6,7,6,7,8,7,8,9]
=
[1+4,1+5,1+6,2+4,2+5,2+6,3+4,3+5,3+6]
```

The list monad applies the function over all choices of elements from each list

- For that reason we call `[]` the **search** monad
- Each variable can be thought as having more than one value assigned to it
  - This is called **non-determinism**

Case study: sum and Pythagorean triples

Given three numbers x, y, z, we say that they form

- A *sum triple* if $x + y = z$
- A *Pythagorean triple* if $x^2 + y^2 = z^2$

`triples xs` computes, given a list of numbers `xs`, those subsets of elements which form a triple

```
> triples [1,2,3]
[(1,2,3),(2,1,3)]
```

We are going to build it using the monadic interface to lists

Cooking sumTriple

A first approximation to sum triples is:

```
sumTriples xs = do x <- xs
                   y <- xs
                   z <- xs
                   if x + y == z
                      then return (x,y,z)
                      else []
```

The value `[]` denotes failure while searching

- No value is produced from ranging over an empty list

```
[] >>= f  =  []  =  xs >>= \_ -> []
```

Introducing guard

This pattern is very common to perform search

```
guard :: Bool -> [()]
guard True  = [()]
guard False = []
```

We do not really care of the value returned by `guard`

- The important bit is that when the condition is false, we produce no more results

```
sumTriples xs = do x <- xs
                   y <- xs
                   z <- xs
                   guard (x + y == z)
                   return (x,y,z)
```

Cooking `triples`

Assuming we have `sumTriples` and `pytTriples`

```
triples :: [Int] -> [(Int, Int, Int)]
triples xs = sumTriples xs ++ pytTriples xs
```

Concatenation combines solutions from multiple sources

- In a search, it works as a disjunction

Monads with failure

Other monads exhibit the same pattern of failure and combination of results

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

Monads with failure

Other monads exhibit the same pattern of failure and combination of results

```haskell
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

The simplest case is `Maybe`: try to implement `mzero` and `mplus`!

Monads with failure

Other monads exhibit the same pattern of failure and combination of results

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

The simplest case is `Maybe`, with `Nothing` representing failure

```
instance MonadPlus Maybe where
  mzero = Nothing
  mplus (Just x) _        = Just x
  mplus _        (Just y) = Just y
  mplus Nothing  Nothing  = Nothing
```

do versus comprehensions

If I had told you to write `sumTriples` without imposing monadic notation, the result would have been

```
                                  [ (x,y,z)
do x <- xs                            | x <- xs
   y <- xs                            , y <- xs
   z <- xs                            , z <- xs
   guard (x + y == z)                 , x + y == z ]
   return (x,y,z)
```

do-notation and comprehensions are exactly the same!

- GHC provides *monad comprehensions* under a flag
- Other languages, such as Scala, only provide comprehensions for working with monads

Summary

- With higher-order functions and higher-kinded abstraction many patterns become mere functions
  - Higher-kinded abstraction refers to making a type constructor vary, in contrast to “full” types
- `Functor` generalizes the idea of “map”
- `Monads` encode the notion of “sequential computation”

Later in the course

- More examples of monads
- Utility functions for monads
- Another abstraction: applicatives