Lecture 14. Monadic utilities and traversables

Functional Programming 2018/19

Alejandro Serrano

Goals

- Look at some utilities for monadic code
  - How to write functions working on monads
- In particular, learn about *traversable* functors

Chapter 14.3 from Hutton's book

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

The functor - applicative - monad hierarchy

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

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

class Applicative f => Monad f where
  -- return is the same as Applicative's pure
  (>>=) :: f a -> (a -> f b) -> f b
```

Monadic utilities

The "final M" family

Many standard functions have monadic counterparts

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

filter  :: (a ->   Bool) -> [a] ->   [b]
filterM :: (a -> m Bool) -> [a] -> m [b]

foldl :: (b -> a ->   b) -> b -> [a] ->   [b]
foldM :: (b -> a -> m b) -> b -> [a] -> m [b]

zipWith  :: (a -> b ->   c) -> [a] -> [b] ->   [c]
zipWithM :: (a -> b -> m c) -> [a] -> [b] -> m [c]
```

Cooking `filterM`

1. Define the type

```
filterM :: (a -> m Bool) -> [a] -> m [b]
```

2. Enumerate the cases

```
filterM p []     = _
filterM p (x:xs) = _
```

3. Define the simple (base) cases
   - Remember that we work in a monadic context

```
filterM _ [] = return []
```

Cooking `filterM`

4. Define the other (recursive) cases
   - We cannot use a guard, because `p` does not return `Bool`

```
filterM p (x:xs) | p x = _   -- Does not work
```

   - For the same reason, we cannot use an `if` directly

```
filterM p (x:xs) = if p x then _ else _   -- Nope
```

   - `p` returns its value wrapped in a monad
   - We unwrap it by means of `<-` in a `do`-block

```
filterM p (x:xs) = do q <- p x
                      ...
```

Cooking `filterM`

4. Define the other (recursive) cases
   - Let us try to write the rest as with `filter`

```
filterM p (x:xs) = do
  q <- p x
  if q
     then x      :   filterM p xs
       -- :: a       :: m [a]
     else filterM p xs
```

   - Wrong: `x` is pure but the result of `filterM` is monadic

Cooking `filterM`

4. Define the other (recursive) cases
   - Solution 1: unwrap the result of `filterM` with `<-`

```
filterM p (x:xs) = do
  q <- p x
  r <- filterM p xs
  if q then return (x:r) else return r
```

   - Solution 2: lift the pure part with applicatives

```
filterM p (x:xs) = do
  q <- p x
  if q then (x:) <$> filterM p xs
       else          filterM p xs
```

Cooking `filterM`

Having to unwrap `p x` distracts a bit

- We just need a "lifted" `if-then-else`
- Why not define our own `ifM` which does just that?

```
ifM :: m Bool -> m a -> m a -> m a
ifM q t e = do q' <- q
               if q' then t else e

filterM _ []     = return []
filterM p (x:xs) = ifM (p x)
                       ((x:) <$> filterM p xs)
                       (filterM p xs)
```


Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Cooking zipWithM

Using a do-block:

```
zipWithM _ []     _      = return []
zipWithM _ _      []     = return []
zipWithM f (x:xs) (y:ys) = do
  z <- f x y
  r <- zipWithM f xs ys
  return (z:r)
```

Cooking `zipWithM`

Using a do-block:

```
zipWithM _ []     _      = return []
zipWithM _ _      []     = return []
zipWithM f (x:xs) (y:ys) = do
  z <- f x y
  r <- zipWithM f xs ys
  return (z:r)
```

Using applicative style:

```
zipWithM _ []     _      = return []
zipWithM _ _      []     = return []
zipWithM f (x:xs) (y:ys)
  = (:) <$> f x y <*> zipWithM f xs ys
```

Traversables

Functors generalize maps

We started with a `map` function for lists

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

which we generalized to arbitrary functors

```
mapTree :: (a -> b) -> Tree  a -> Tree  b
mapMay  :: (a -> b) -> Maybe a -> Maybe b
...
fmap :: Functor f => (a -> b) -> f a -> f b
```

Can we do something similar for `mapM`?

Cooking `mapTreeM`

1. Define the type
   - Remember: we want a monadic function as argument

```
mapTreeM :: Monad m
         => (a -> m b) -> Tree a -> m (Tree b)
```

2. Enumerate the cases

```
mapTreeM f Leaf           = _
mapTreeM f (Node l x r) = _
```

3. Define the simple (base) cases
   - Remember: we are now in a monadic context

```
mapTreeM _ Leaf = return Leaf
```

Cooking `mapTreeM`

4. Define the other (recursive) cases
   - Solution 1: using `do`-notation

```
mapTreeM f (Node l x r) = do
  l' <- mapTreeM f l
  x' <- f x
  r' <- mapTreeM f r
  return (Node l' x' r')
```

   - Solution 2: using applicative style

```
mapTreeM f (Node l x r)
  = Node <$> mapTreeM f l
         <*> f x
         <*> mapTreeM f r
```

Cooking `mapTreeM`

5. Generalize and simplify
   - The second implementation only needs `Applicative`, the first uses `do` and thus needs `Monad`
   - Remember: `Applicative` is more general than `Monad`

```
mapTreeM :: Applicative f
         => (a -> f b) -> Tree a -> f (Tree b)
```

Traversables

The generalization of `Functor` to handle functions of the form `a -> f b` is called a **traversable** (functor)

```
class Functor t => Traversable t where
  traverse :: Applicative f
           => (a -> f b) -> t a -> f (t b)
```

- `f` defines the context in which the function run
- `t` defines the data structure which contains the elements to map over

Cooking `printTree`

`printTree t` print the elements of the tree to the screen, in infix order, one per line

1. Define the type

   ```
   printTree :: Show e => Tree e -> IO ()
   ```

2. `IO` is an applicative, `Tree` is a traversable
   - We can just “map” the `print` function!

   ```
   printTree = traverse print
   ```

Cooking `printTree`

3. We run into a problem, the result of `traverse print` is `IO (Tree ())`, not `IO ()`
   - Solution 1: explicitly return ()

     ```
     printTree t = do traverse print t
                      return ()
     ```

   - Solution 2: use `void :: Functor f => f a -> f ()` to discard the value

     ```
     printTree = void . traverse print
     ```

4. This implementation works for any traversable

   ```
   printTraversable :: (Show e, Traversable t)
                    => t e -> IO ()
   ```

Summary

- Haskell has powerful ways to abstract
  - Code and design patterns become functions
- Higher-order functions
  - Maps, folds, filters…
- Higher-kinded abstractions
  - Functors, monads and applicatives for “contexts”
  - Functors and traversables for “containers”

`sequence` for arbitrary traversables is not part of 2018/2019 contents

Note that `sequence` for `IO` *is* part of the contents

Another look at `zipWithM`

What happens if we just zip the function?

```
zipWith :: (a -> b ->   c) -> [a] -> [b] -> [c]
f       :: (a -> b -> m r)
xs      :: [a]
ys      :: [b]
-- By making c = m r in the type of zipWith
zipWith f xs ys :: [m r]
```

Alas, what we require is `m [r]`

`sequence` to the rescue

During the lecture on `IO`, we introduced a function

```
sequence :: [IO a] -> IO [a]
```

This function actually works on any monad

```
sequence :: Monad m => [m a] -> m [a]
```

We can write `zipWithM` with its help

```
zipWithM f xs ys = sequence (zipWith f xs ys)
```

Generalizing `sequence`

A traversable admits a generic version of `sequence`

```
sequence :: Monad m => [m a]    -> m [a]
-- Generalized to traversables
sequence :: (Traversable t, Applicative f)
                          => t (f a) -> f (t a)
```

Cooking generic `sequence`

Let us try to find the implementation by looking at the types

```
traverse :: (  a -> f b) -> t a     -> f (t b)
sequence ::                 t (f r) -> f (t r)
```

Cooking generic `sequence`

Let us try to find the implementation by looking at the types

```
traverse :: (  a -> f b) -> t a     -> f (t b)
sequence ::                 t (f r) -> f (t r)
```

Solution: let us make `a = f r` and `b = r`

```
traverse :: (f r -> f r) -> t (f r) -> f (t r)
```

How do we get a function of type `f r -> f r`?

Cooking generic `sequence`

Let us try to find the implementation by looking at the types

```
traverse :: (  a -> f b) -> t a     -> f (t b)
sequence ::                 t (f r) -> f (t r)
```

Solution: let us make `a = f r` and `b = r`

```
traverse :: (f r -> f r) -> t (f r) -> f (t r)
```

How do we get a function of type `f r -> f r`?

```
sequence = traverse id
```