Functional Programming for Logicians - Lecture 4

IO, Trees, Randomness

Malvin Gattinger

17 January 2022

```
module L4 where

import Data.Char
import System.Random
```

IO

Trees

Randomness

Project Practicalities

IO

Summary Lecture 3

Recall that:

- A `Functor` is something that we can `fmap` over.
- An `Applicative` is a `Functor` plus `pure` and `<*>`.
- A `Monad` is an `Applicative` plus `>>=`.

Examples: `Maybe`, `[ ]` and `IO` are monads!

Real World Haskell *

Here are some useful standard IO functions:

```
λ> :t readFile
readFile :: FilePath -> IO String
λ> :t writeFile
writeFile :: FilePath -> String -> IO ()
λ> :t getLine
getLine :: IO String
λ> :t putStr
putStr :: String -> IO ()
λ> :t putStrLn
putStrLn :: String -> IO ()
```

(* This is also the title of the Haskell book by Bryan O'Sullivan, Don Stewart, and John Goerzen. See http://book.realworldhaskell.org/.)

Hello World 2.1

```haskell
dialogue :: IO ()
dialogue = do
  putStrLn "Hello! Who are you?"
  name <- getLine
  putStrLn $ "Nice to meet you, " ++ name ++ "!"
  let capName = map toUpper name
  putStrLn $ "Or should I say " ++ capName ++ "?"
```

Hello World 2.1

```haskell
dialogue :: IO ()
dialogue = do
  putStrLn "Hello! Who are you?"
  name <- getLine
  putStrLn $ "Nice to meet you, " ++ name ++ "!"
  let capName = map toUpper name
  putStrLn $ "Or should I say " ++ capName ++ "?"
```

Note that it is not possible to "get out ouf `IO`" again.

Do not try to write functions like `f :: IO String -> String`.

Hello World 2.1

```haskell
dialogue :: IO ()
dialogue = do
  putStrLn "Hello! Who are you?"
  name <- getLine
  putStrLn $ "Nice to meet you, " ++ name ++ "!"
  let capName = map toUpper name
  putStrLn $ "Or should I say " ++ capName ++ "?"
```

Note that it is not* possible to "get out ouf `IO`" again.

Do not try to write functions like `f :: IO String -> String`.

(* It is, but `unsafePerformIO` is called like that for a reason.)

`sequence`

We already called `<*>` the "sequence operator".

There is also a more general function: `sequence`.

It converts a list of actions into a single action that gives a list.

sequence

We already called <*> the “sequence operator”.

There is also a more general function: `sequence`.

It converts a list of actions into a single action that gives a list.

Example:

```
λ> sequence [Just 4, Just 12, Just 43]
Just [4,12,43]
```

sequence

We already called <*> the "sequence operator".

There is also a more general function: `sequence`.

It converts a list of actions into a single action that gives a list.

Example:

```
λ> sequence [Just 4, Just 12, Just 43]
Just [4,12,43]
```

Definition:

```
mysequence :: Monad m => [m a] -> m [a]
mysequence []     = return []
mysequence (x:xs) = (:) <$> x <*> sequence xs
```

sequence with IO

```
λ> :t replicate
replicate :: Int -> a -> [a]
λ> sequence (replicate 3 getLine)
bob
alice
carol
["bob","alice","carol"]
```

sequence with IO

```
λ> :t replicate
replicate :: Int -> a -> [a]
λ> sequence (replicate 3 getLine)
bob
alice
carol
["bob","alice","carol"]
```

Similar to `foldl`, also `sequence` actually has a more general type:

```
λ> :t sequence
sequence :: (Monad m, Traversable t) => t (m a) -> m (t a)
```

sequence with IO

```
λ> :t replicate
replicate :: Int -> a -> [a]
λ> sequence (replicate 3 getLine)
bob
alice
carol
["bob","alice","carol"]
```

Similar to `foldl`, also `sequence` actually has a more general type:

```
λ> :t sequence
sequence :: (Monad m, Traversable t) => t (m a) -> m (t a)
```

We will now look at another example of something `Traversable`.

Trees

Example: trees

Binary-branching trees with things of type `a` at the leafs:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
     deriving (Eq,Ord,Show)
```

Example: trees

Binary-branching trees with things of type a at the leafs:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
     deriving (Eq,Ord,Show)
```

Example:

```
numberTree :: Tree Int
numberTree = Branch (Leaf 1) (Branch (Leaf 2) (Leaf 3))
```

instance Functor Tree

Let us define `fmap` for trees.

```
instance Functor Tree where
-- fmap :: (a -> b) -> Tree a -> Tree b
```

instance Functor Tree

Let us define fmap for trees.

```
instance Functor Tree where
-- fmap :: (a -> b) -> Tree a -> Tree b
   fmap f (Leaf x)            = Leaf (f x)
   fmap f (Branch left right) = Branch (fmap f left)
                                       (fmap f right)
```

instance Functor Tree

Let us define fmap for trees.

```
instance Functor Tree where
-- fmap :: (a -> b) -> Tree a -> Tree b
   fmap f (Leaf x)            = Leaf (f x)
   fmap f (Branch left right) = Branch (fmap f left)
                                       (fmap f right)
```

Example:

```
λ> numberTree
Branch (Leaf 1) (Branch (Leaf 2) (Leaf 3))
λ> fmap (*10) numberTree
Branch (Leaf 10) (Branch (Leaf 20) (Leaf 30))
```

instance Functor Tree

Let us define `fmap` for trees.

```
instance Functor Tree where
-- fmap :: (a -> b) -> Tree a -> Tree b
   fmap f (Leaf x)            = Leaf (f x)
   fmap f (Branch left right) = Branch (fmap f left)
                                       (fmap f right)
```

Example:

```
λ> numberTree
Branch (Leaf 1) (Branch (Leaf 2) (Leaf 3))
λ> fmap (*10) numberTree
Branch (Leaf 10) (Branch (Leaf 20) (Leaf 30))
```

Note that `fmap` will never change the shape of the tree.

How does this compare to `fmap` for `Maybe`? 

instance Applicative Tree

```haskell
instance Applicative Tree where
-- pure ::  a -> Tree a
   pure          = Leaf
-- (<*>) :: Tree (a -> b) -> Tree a -> Tree b
   (<*>) ftree (Leaf x)        = fmap ($ x) ftree
   (<*>) ftree (Branch xl xr) = Branch (ftree <*> xl)
                                       (ftree <*> xr)
```

What does this do?

instance Applicative Tree

```
instance Applicative Tree where
-- pure ::  a -> Tree a
   pure          = Leaf
-- (<*>) :: Tree (a -> b) -> Tree a -> Tree b
   (<*>) ftree (Leaf x)         = fmap ($ x) ftree
   (<*>) ftree (Branch xl xr) = Branch (ftree <*> xl)
                                       (ftree <*> xr)
```

What does this do?

Example:

```
λ> Branch (Leaf (+1)) (Leaf (+10)) <*> Branch (Leaf 3) (Leaf 4)
```

instance Applicative Tree

```
instance Applicative Tree where
-- pure ::  a -> Tree a
   pure          = Leaf
-- (<*>) :: Tree (a -> b) -> Tree a -> Tree b
   (<*>) ftree (Leaf x)         = fmap ($ x) ftree
   (<*>) ftree (Branch xl xr) = Branch (ftree <*> xl)
                                       (ftree <*> xr)
```

What does this do?

Example:

```
λ> Branch (Leaf (+1)) (Leaf (+10)) <*> Branch (Leaf 3) (Leaf 4)
Branch (Branch (Leaf 4) (Leaf 13)) (Branch (Leaf 5) (Leaf 14))
```

Alternative instance Applicative Tree

We could also use the following <*> function, recursing first on the tree of functions and then making full copies of the value tree.

```
star :: Tree (a -> b) -> Tree a -> Tree b
star (Leaf f)         atree = fmap f atree
star (Branch fl fr) atree = Branch (fl `star` atree)
                                   (fr `star` atree)
```

Alternative `instance Applicative Tree`

We could also use the following <*> function, recursing first on the tree of functions and then making full copies of the value tree.

```
star :: Tree (a -> b) -> Tree a -> Tree b
star (Leaf f)         atree = fmap f atree
star (Branch fl fr) atree = Branch (fl `star` atree)
                                   (fr `star` atree)
```

Exercise: Check which way to implement `instance Applicative Tree` fulfils the `Functor` and `Applicative` laws?

instance Monad Tree

Wrapping up a thing into a tree is done using `Leaf`.

Binding a tree to `f` means we apply f to all its leafs.

```
instance Monad Tree where
-- return :: a -> Tree a
   return = Leaf
-- (>>=) :: Tree a -> (a -> Tree b) -> Tree b
   (>>=) (Leaf x)            f = f x
   (>>=) (Branch left right) f = Branch (left >>= f) (right >>= f)
```

instance Monad Tree

Wrapping up a thing into a tree is done using `Leaf`.

Binding a tree to `f` means we apply f to all its leafs.

```
instance Monad Tree where
-- return :: a -> Tree a
   return = Leaf
-- (>>=) :: Tree a -> (a -> Tree b) -> Tree b
   (>>=) (Leaf x)            f = f x
   (>>=) (Branch left right) f = Branch (left >>= f) (right >>= f)
```

Example:

```
λ> Branch (Leaf 5) (Leaf 8) >>= (\n -> Branch (Leaf (n+100)) (Leaf (n+10)))
Branch (Branch (Leaf 105) (Leaf 15)) (Branch (Leaf 108) (Leaf 18))
```

Foldable and Traversable

```
type Traversable :: (* -> *) -> Constraint
class (Functor t, Foldable t) => Traversable t where
  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
  sequenceA :: Applicative f => t (f a) -> f (t a)
  mapM :: Monad m => (a -> m b) -> t a -> m (t b)
  sequence :: Monad m => t (m a) -> m (t a)
  {-# MINIMAL traverse | sequenceA #-}
```

Foldable and Traversable

```
type Traversable :: (* -> *) -> Constraint
class (Functor t, Foldable t) => Traversable t where
  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
  sequenceA :: Applicative f => t (f a) -> f (t a)
  mapM :: Monad m => (a -> m b) -> t a -> m (t b)
  sequence :: Monad m => t (m a) -> m (t a)
  {-# MINIMAL traverse | sequenceA #-}

type Foldable :: (* -> *) -> Constraint
class Foldable t where
  Data.Foldable.fold :: Monoid m => t m -> m
  foldMap :: Monoid m => (a -> m) -> t a -> m
  Data.Foldable.foldMap' :: Monoid m => (a -> m) -> t a -> m
  foldr :: (a -> b -> b) -> b -> t a -> b
  Data.Foldable.foldr' :: (a -> b -> b) -> b -> t a -> b
  foldl :: (b -> a -> b) -> b -> t a -> b
  Data.Foldable.foldl' :: (b -> a -> b) -> b -> t a -> b
  foldr1 :: (a -> a -> a) -> t a -> a
  foldl1 :: (a -> a -> a) -> t a -> a
  Data.Foldable.toList :: t a -> [a]
  null :: t a -> Bool
  length :: t a -> Int
  elem :: Eq a => a -> t a -> Bool
  maximum :: Ord a => t a -> a
  minimum :: Ord a => t a -> a
  sum :: Num a => t a -> a
  product :: Num a => t a -> a
  {-# MINIMAL foldMap | foldr #-}
```

instance Foldable Tree

```haskell
instance Foldable Tree where
-- foldr :: (a -> b -> b) -> b -> Tree a -> b
```

instance Foldable Tree

```haskell
instance Foldable Tree where
-- foldr :: (a -> b -> b) -> b -> Tree a -> b
  foldr f y (Leaf x)     = f x y
  foldr f y (Branch l r) = foldr f (foldr f y l) r
```

instance Foldable Tree

```haskell
instance Foldable Tree where
-- foldr :: (a -> b -> b) -> b -> Tree a -> b
  foldr f y (Leaf x)     = f x y
  foldr f y (Branch l r) = foldr f (foldr f y l) r
```

```haskell
instance Traversable Tree where
-- traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
```

instance Foldable Tree

```haskell
instance Foldable Tree where
-- foldr :: (a -> b -> b) -> b -> Tree a -> b
  foldr f y (Leaf x)     = f x y
  foldr f y (Branch l r) = foldr f (foldr f y l) r
```

```haskell
instance Traversable Tree where
-- traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
  traverse g (Leaf x)     = Leaf <$> g x
  traverse g (Branch l r) = Branch <$> traverse g l <*> traverse g r
```

What is it good for?

Now `foldr` and many more functions work on our trees:

```
λ> numberTree
Branch (Leaf 1) (Branch (Leaf 2) (Leaf 3))
λ> length numberTree
3
λ> sum numberTree
6
```

What is it good for?

Now `foldr` and many more functions work on our trees:

```
λ> numberTree
Branch (Leaf 1) (Branch (Leaf 2) (Leaf 3))
λ> length numberTree
3
λ> sum numberTree
6
```

And we can execute a tree of IO actions!

```
λ> sequence $ Branch (Leaf (putStrLn "kwik"))
                     (Branch (Leaf (putStrLn "kwek"))
                             (Leaf (putStrLn "kwak")))
kwik
kwek
kwak
Branch (Leaf ()) (Branch (Leaf ()) (Leaf ()))
```

Other Trees

Exercise 3.4: Add things of type a at intermediate nodes.

Other Trees

Exercise 3.4: Add things of type `a` at intermediate nodes.

Non-binary trees with arbitrary branching:

```
data ArbTree a = Node a [ArbTree a]
```

Note that we no longer need an extra `Leaf` case. Leafs are just `Nodes` where the `[ArbTree a]` list of children is empty.

Exercise: Can you also make `ArbTree` a `Functor` etc.?

Randomness

Random Integers

Getting a random value is obviously not a pure function.

To get a random integer we need interaction with the outside world.

(In UNIX this is usually `/dev/random` or `/dev/urandom`.)

```
getRandomInt :: Int -> IO Int
getRandomInt n = getStdRandom (randomR (0,n))
```

This gives:

```
λ> getRandomInt 20
16
λ> getRandomInt 20
18
```

Random Integer Lists

Generate an integer list with `n` entries in the range `[0..k]`.

```
getInts :: Int -> Int -> IO [Int]
getInts _ 0 = return []
getInts k n =
  getRandomInt k >>= \x ->
    getInts k (n-1) >>= \xs -> do
      return (x:xs)
```

Random Integer Lists

Generate an integer list with `n` entries in the range `[0..k]`.

```
getInts :: Int -> Int -> IO [Int]
getInts _ 0 = return []
getInts k n =
  getRandomInt k >>= \x ->
    getInts k (n-1) >>= \xs -> do
      return (x:xs)
```

We can also write this "point-free":

```
getInts'' :: Int -> Int -> IO [Int]
getInts'' _ 0 = return []
getInts'' k n = (:) <$> getRandomInt k <*> getInts'' k (n-1)
```

Random Lists of Random Integers

Finally, we can also choose the parameters randomly:

```
genIntList :: IO [Int]
genIntList = do
  k <- getRandomInt 20
  n <- getRandomInt 10
  getInts k n
```

This gives, e.g.:

```
λ> genIntList
[0,0,0,0]
λ> genIntList
[-1,-5,-3,-2,-1,6,2,-8]
λ> genIntList
[15,-10,7,-15,5,-13,15,11,13,-11]
```

Project Practicalities

Timeline

- now: find a group (2 or 3 people) and topic

  see https://malv.in/2022/funcproglog/topics.html

  start reading material or explore existing code
- Friday 21st: send one email per group to Malvin:

  Who are you, and what is your topic?
- next week: continue reading, shift to working
- Friday 28th: work-in-progress **presentations**
- Monday 31st: get feedback on current version (optional!)
- Friday 4th: **deadline for report**

Report Template

You should use this template for your project:

https://github.com/funcspec/report-example

(For other larger Haskell projects, use `stack new` to create a new package with lots of boilerplate.)

Project Grading Criteria

You will only get a pass/fail grade (and feedback comments).

Your final report:

- should have a specific topic and a concrete goal
- must be written in literate programming style
- should be well-structured
- must be at most 15 pages

Your program:

- must compile
- must have zero warnings with `-Wall`
- should generate zero hints from `hlint`
- should have tests

Your presentation:

- is about work-in-progress
- should be at most 20 minutes
- can be given by a subgroup

stack and cabal

Bigger projects

- use more than one `.hs` or `.lhs` file
- may depend on other Haskell libraries

To manage projects, we use `stack.yaml` and `package.yaml`.

stack and cabal

Bigger projects

- use more than one `.hs` or `.lhs` file
- may depend on other Haskell libraries

To manage projects, we use `stack.yaml` and `package.yaml`.

The minimal `stack.yaml` is this:

```
resolver: lts-18.21
```

This refers to https://www.stackage.org/lts-18.21 and hopefully ensures that your code still works next year.

Package management

The `package.yaml` describes your project and its **dependencies**:

```
name: report
version: 0.1.0.0
synopsis: My Haskell report project
description: See report.pdf
maintainer: My Name <my.email@example.com>
category: Logic

ghc-options: -Wall

dependencies:
  - base >= 4.14 && < 5
  - random
  - QuickCheck

library:
  source-dirs: lib

executables:
  myprogram:
    main: Main.lhs
    source-dirs: exec
    dependencies:
      - report

tests:
  simpletests:
    main: simpletests.lhs
    source-dirs: test
    dependencies:
      - report
      - QuickCheck
      - hspec
```

Version Control

Please use `git` when working in teams!

It takes half an hour, but you will then prevent the "Dropbox problem" of overwriting each others work.

For a tutorial, click here, for a cheat sheet, click here.

Version Control

Please use `git` when working in teams!

It takes half an hour, but you will then prevent the "Dropbox problem" of overwriting each others work.

For a tutorial, click here, for a cheat sheet, click here.

The most widely used hosted services for *git* are *github* and *gitlab*.

You may submit your report via email, but using git and a service like `github` or `gitlab` is strongly preferred if you want to get help or comments from Malvin during the next weeks.


See you again at 13:00.

Bonus Content: History of the IO Monad

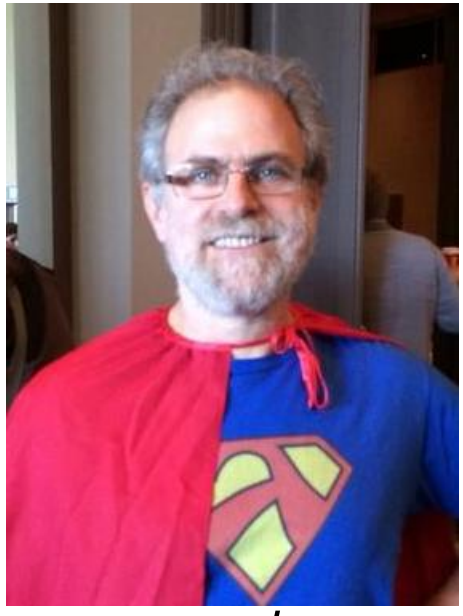
Since 1992 the most famous Monad is `IO`.

As told by Simon Peyton-Jones here, Haskell was useless at first:

https://youtu.be/re96UgMk6GQ?t=31m20s

(watch this from 31:20 until 38:22)

Bonus Content: Invention vs. Discovery



"Most of you use languages that were invented, and you can tell, can't you. This is my invitation to you to use programming languages that are discovered."

Philip Wadler: Propositions as Types

There are many recordings of this talk, but the original paper is this one:

https://homepages.inf.ed.ac.uk/wadler/papers/propositions-as-types/propositions-as-types.pdf