Topics in Functional Programming

Joseph Winder

What I studied this semester

- Concepts in Functional Programming
- Some Functional Data Structures
- A little Category Theory
- The Zipper Data Structure + Category Theory
- Monadic Programming + Category Theory
- A little Computability Theory
- Lambda Calculus

Signpost

- Side-Effects & Referential Transparency
- Data Types
- Type Classes
- Monads

(Examples presented in Haskell)

Functional programs do NOT

- Have mutable state
- Have side-effects... kinda!
- Have sequences of statements effecting global state
- Have objects with state and behavior

Functional programs do

- Execute by evaluating expressions
- Emphasize types of data
- Treat functions exactly like values
- Have referential transparency

Side-effects

- A side-effect is an action that has an affect on another part of the domain
  - Inserting data into a database
  - Getting data from an input device

Referential Transparency

- A named value is referentially transparent if it can be replaced by its value at any point in the program
- Is side-effecting code referentially transparent?
    - How is this problem solved?

A named integer

- ```
  x :: Int
  x = 5
  ```

- `::` means "has type"
  ```

A function

- ```
  double :: Int -> Int
  double x = 2*x
  ```

- `double` has type `Int -> Int`

- Input is an integer, output is an integer
  ```

Lists

- A list is a recursive data structure that is either
  - empty
  - a value and another list

Lists (cont.)

- ```
  data List a = Nil
              | Cons a (List a)
  ```

- Example:

  ```
  xs :: List Int
  xs = Cons 3 (Cons 5 (Cons 6 Nil))
  ```
  ```

Lists (cont.)

- ```
  data [a] = []
            | a : [a]
  ```

- Same example:

  ```
  xs :: [Int]
  xs = 3:5:6:[]
  ```
  ```

Lists (cont.)

- Shorter shortcut:

  `[3,5,6]` is equivalent to `3:5:6:[]`

  `3:[5,6]` is equivalent to `3:5:6:[]`

Functions over lists

- `map`
  - Takes a function and a list
  - Returns a new list with the values of the original list having the function is applied to them
  - Leaves the original inputs unaffected
    - Every function does this!

Functions over lists (cont.)

- `map double [1,2,3]`
  - returns `[2,4,6]`

Functions over lists (cont.)

- `filter`
  - Takes a predicate and a list
  - Returns a new list whose elements satisfy the predicate
- `filter (>0) [-2,-1,3,4]`
  - returns `[3,4]`

Functions over lists (cont.)

- `zip`
  - Takes two lists
  - Returns a list of pairs containing the respective elements of the original lists
- `zip [1,2,3] "abc"`
  - returns `[(1,'a'), (2,'b'), (3,'c')]`

Examples

- ```
  quickSort [] = []
  quickSort (x:xs) = filter (<=x) xs
                  ++ [x]
                  ++ filter (>x) xs
  ```

- ```
  insertSort [] = []
  insertSort (x:xs) = ins x (insertSort xs)

  ins x [] = [x]
  ins x (y:ys) = if x <= y
                  then x:y:y
                  else y:(ins x ys)
  ```

Type classes

- A collection of types that respond to the type class's members

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

Type classes (cont.)

- ```
  class Eq a where
     (==), (/=) :: a -> a -> Bool
     (==) = not (/=)
     (/=) = not (==)
  ```

- ```
  instance Eq Bool where
     True == True = True
     False == False = True
  ```

Monads

- A monad is a neat type class that offers a binding between (potentially complex) computations
- It offers a way to separate pure functional code from side-effecting code
- It also respects referential transparency

Monads (cont.)

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

- `m` is not just a type, but a type constructor

- Example:

  `Int` is a type

  `List, Nil, Cons` are type constructors for the type `List a`
  ```

Monads (cont.)

- `class Monad m where`
```
   return :: a -> m a
   (>>=) :: m a -> (a -> m b) -> m b
```
- `return` type-constructs a value
  - Example with lists: `return 1 -- [1]`
- `(>>=)` binds the output of a computation to the input of another computation
  - This needs multiple examples

Monads (cont.)

- `data Identity a = Id a`

- ```
  instance Monad Identity where
    return = Id
    Id x >>= f = f x
  ```

- ```
  Id 2 >>= (2*)    -- 4

  Id [1] >>= (concat [2,3])    -- [1,2,3]
  ```

Monads (cont.)

- `data Maybe a = Nothing | Just a`

- ```
  safeDiv :: Int       ->
             Maybe Int ->
             Maybe Int

  safeDiv _ Nothing = Nothing
  safeDiv x (Just y) = Just (x/y)
  ```
  ```

Monads (cont.)

- ```
  instance Monad Maybe where
    return = Just

    Nothing >>= _ = Nothing
    Just x >>= f = Just (f x)
  ```

- `safeDiv x y = y >>= (x/)`
  ```

Monads (cont.)

- ```
  instance Monad [] where
    return x = [x]
    xs >>= f = concat (map f xs)
  ```

- ```
  getPowers n = [n^m | m <- [1,2..10]]
  powers ns = ns >>= getPowers
  ```

The I/O Monad

- Why doesn't `putStr getLine` work?
- ```
  getLine :: IO String
  putStr :: String -> IO ()
  ```
  ```

The I/O Monad (cont.)

- `getLine :: IO String`
  `putStr :: String -> IO ()`
- `(>>=) :: m a -> (a -> m b) -> m b`
- Use the bind function to move the string from the side-effecting `getLine` into the input of `putStr`
- `getLine >>= putStr`

The I/O Monad (cont.)

- `getLine >>= putStr`
- Do these both have side-effects?
- Is this referentially transparent?
  - `getLine :: IO String`
  - `putStr :: String -> IO ()`
  -
    ```
    (>>=) :: IO String              ->
             (String -> IO ()) ->
             IO ()
    ```

The I/O Monad (cont.)

- ```
  instance Monad IO where
    return a = IO a
    g >>= f = do { x <- g ; f x }
  ```
  ```

The State Monad

- A state is a value that is needed to perform a computation
- A state is not a parameter to a computation
- `data State st res = st -> (res,st)`
- This is just a data type that consists of functions that return a result, given a particular state type

The State Monad (cont.)

- A state monad is able to attach state information to any type of calculation
- `data State st res = st -> (res,st)`
- `instance Monad (State st_type) where`

```
return r = \s -> (r,s)

process >>= f = \s ->
  let (res,st) = process s
  in (f res) st
```

Monads

- Monads are rooted in Category Theory

Some uses of functional programming

- Honest about side-effects & referential transparency
- Lazy Evaluation, Partial Evaluation, Memoization
- Recursion, Tail-Call Optimization
- Safe Multi-Threading, Large scale data processing

Questions?