Curry time - Learn you a Haskell

Cameron Reuschel - Vincent Truchseß


STAND BACK
λ
I'M GOING TO TRY
HASKELL

A pure functional Programming Language

- Everything is immutable
- Everything is lazy
- Everything is a function
- Everything is awesome

Subsection 1

Getting started

History - The Inspiration

Figure 1: James Haskell - 2010

History - The Creator

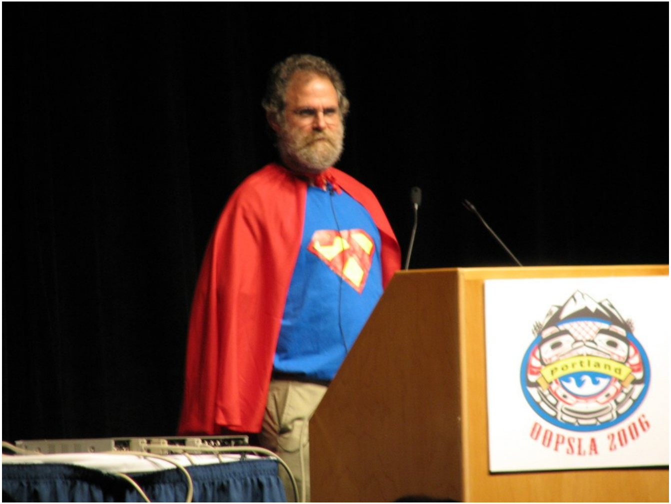

Figure 2: Philip Wadler aka Lambda Man

Section 1

Functional Concepts

Subsection 1

Purity

What is a Side Effect?

Any operation which modifies the state of the computer or which interacts with the outside world

- variable assignment
- displaying something
- printing to console
- writing to disk
- accessing a database

Figure 3: XKCD on Side Effects

Purity: No Side Effects

- Haskell is **pure** - no side effects
- = is mathematical equality
- Purity leads to **referential transparency**: for every `x = expr` you can replace `x` with `expr` without changing semantics
- An expression `f x` is **pure** if it is referentially transparent for every referentially transparent `x`

Referential Transparency - Example

Not referentially transparent:
Successive calls to `count()` return different values.

```
int counter = 0;
int count() { return ++counter; }

int x = count();

int a, b;
a = x; b = x; // a == b == 1
a = count(); b = count(); // a == 2, b == 3
```

Pure functions do not modify any state.
They always return the same result given the same input.

Subsection 2

Lazyness

Lazyness

. . . not today

Lazyness

- Eager evaluation: expression is evaluated as soon as it is used
- Lazy evaluation: expression is only evaluated when it is needed

```
int counter = 0;
private int count() { return ++counter; }

// Eager: foo == 1337; counter == 1;
int foo = Optional.of(1337).orElse(count());

// Lazy: foo == 1337; counter == 0;
int foo = Optional.of(1337).orElseGet(() -> count());
```

Everything in Haskell is evaluated lazily.

Section 2

Functions

Basic Syntax

```
sum :: Num a => a -> a -> a
sum x y = x + y

-- type declarations can be omitted
times2 a = a `sum` a

abs :: (Num a, Ord a) => a -> a
abs x = if x < 0 then -x else x

compareTo :: (Num a, Ord a1) => a1 -> a1 -> a
compareTo x y
  | x > y = 1
  | x < y = -1
  | otherwise = 0
```

Currying

All functions take a single argument and return a single value

```
sum :: Num a => a -> a -> a
sum x y = x + y

addTwo :: Num a => a -> a
addTwo = sum 2
```

`sum` is a **curried** function: it takes an `x` and returns a function that takes a `y` that returns the sum of `x` and `y`

```
-- (x +) :: a -> a
sum' :: Num a => a -> a -> a
sum' x = (x +)
```

Figure 4: James Haskell Eating Curry

Higher order Functions & Lambdas

- A **higher order function** is a function that takes another function as an argument
- A **lambda expression** is an anonymous closure with syntax `\arg arg2 ... -> expression`

```haskell
flip :: (a -> b -> c) -> (b -> a -> c)
flip f = \x y -> f y x

negate :: (a -> Bool) -> (a -> Bool)
negate p = not . p
```

Section 3

Working with Types

Basic Types

Besides the usual Number types (Integers, Floats, Fractions, . . .) Haskell also includes:

```
Chars:   'a', 'b', 'c', ...
Strings: "hello" = ['h', 'e', 'l', 'l', 'o']
Tuples:  (1, "hello", (\a -> a * 42))
```

Subsection 1

Lists

Creating Lists

```
favoritePrimes :: [Int]
favoritePrimes = [3,7,9,11]

evenNumbers = [x | x <- [0..50], x `mod` 2 == 0]
evenNumbers' = [0,2..50]
evenNumbersAndOne = 1 : evenNumbers

alphabet = ['a'..'z'] ++ ['A'..'Z']
```

Basic list functions

```
head [1, 2, 3]            -- > 1
tail [1, 2, 3]            -- > [2, 3]
init [1, 2, 3]            -- > [1, 2]
last [1, 2, 3]            -- > 3

take 2 [1, 2, 3]          -- > [1, 2]
takeWhile (< 3) [1, 2, 3] -- > [1, 2]

drop 2 [1, 2, 3]          -- > [3]
dropWhile (< 3) [1, 2, 3] -- > [3]
```

More on Lists

```
zip ['a', 'b'] [1..]                -- > [('a',1), ('b', 2)]
zipWith (+) [1, 2, 3] [4, 5, 6] -- > [5, 7, 9]

map abs [-1, -2, 3]                 -- > [1, 2, 3]
filter even [1, 2, 3, 4]            -- > [2, 4]
any even [3, 5, 7]                  -- > False

cycle [1, 2, 3] -- > [1, 2, 3, 1, 2, 3, ...]
repeat 'g'      -- > "gggggggggggggggggg..."
```

Due to lazy evaluation we can have infinite lists.
Don't run `length` on this. It takes forever.

Folds - Formally known as Reducers

`foldl` accumulates a *sequence* into a value *left to right*

```
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
foldl (+) 0 [1..5]

foldl (+)     (0 + 1)                         [2..5]
foldl (+)    ((0 + 1) + 2)                    [3..5]
foldl (+)   (((0 + 1) + 2) + 3)               [4, 5]
foldl (+)  ((((0 + 1) + 2) + 3) + 4)          [5]
foldl (+) (((((0 + 1) + 2) + 3) + 4) + 5)     []
```

Folds - Formally known as Reducers

`foldr` accumulates a *sequence* into a value *right to left*

```
foldr :: Foldable t => (b -> b -> a) -> b -> t a -> b
foldr (+) 0 [1..5]
```

```
(1 +                     (foldr (+) 0 [2..5]))
(1 + (2 +                (foldr (+) 0 [3..5])))
(1 + (2 + (3 +           (foldr (+) 0 [4, 5]))))
(1 + (2 + (3 + (4 +      (foldr (+) 0 [5]   )))))
(1 + (2 + (3 + (4 + (5 + (foldr (+) 0 []    ))))))
```

Subsection 2

Custom Data Types

Sum Types

Sum types are essentially represented as `enums` in C-like languages

```
data BracketValidationResult
   = TooManyOpen
   | TooManyClosed
   | Fine
   | NoCode
```

Product Types

Product types are essentially `structs` in C

```
data Tape = Tape [Int] Int [Int]
tape = Tape [1, 2] 3 [4]
left  (Tape l _ _) = l
right (Tape _ _ r) = r
curr  (Tape _ c _) = c

-- record syntax
data Tape = Tape
  { left :: [Int], curr :: Int, right :: [Int] }
tape  = Tape [1, 2] 3 [4]
tape' = Tape {left = [1, 2], curr = 3, right = [4]}
```

Mix and Match

```
data Point = Point Float Float

data Shape
  = Circle Point Float
  | Rectangle
    { upperLeft :: Point
    , lowerRight :: Point }
```

Figure 5: James Haskell is in shape

Subsection 3

Type Classes

Type Clases 1

Type classes are used to ‘implement’ an interface for a type:

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

Implementing `Eq` for a type `T` makes the type magically work for every function that expects an instance of `Eq`

```
instance Eq Tape where
  x == y =
    left x == left y
    && curr x == curr y
    && right x == right y
```

Type Classes 2

Type class instances can be derived from a type:

```
data Tape = Tape [Int] Int [Int] deriving (Eq, Show)
```

Type classes itself can derive from other type classes:

```
class (Eq a) => Num a where ...
```

Builtin useful type classes:

```
Eq, Show, Read, Ord, Bounded, Enum
Num, Integral, Real, Fractional
Foldable, Functor, Monad
```

Overview - Type Class Hierarchy

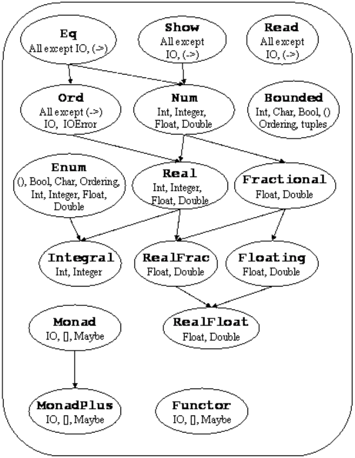

Figure 6: Standard Haskell Classes
https://www.haskell.org/onlinereport/basic.html

Subsection 4

Pattern matching

Pattern matching: Simple case

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

fib n = case n of
  0 -> 1
  1 -> 1
  n -> fib (n-1) + fib (n-2)
```

Pattern Matching: Deconstruction

```
quicksort [] = []
quicksort (p:xs) = (quicksort lesser)
    ++ [p] ++ (quicksort greater)
    where (lesser, greater) = partition (< p) xs

partition :: (a -> Bool) -> [a] -> ([a], [a])
```

Pattern Matching: Deconstruction

```
-- with overflow handling
increment :: Tape -> Tape

data Tape = Tape [Int] Int [Int]
increment (Tape left curr right) =
    Tape left ((curr + 1) `mod` 256) right

data Tape =
  Tape { left :: [Int], curr :: Int, right :: [Int] }
increment Tape
  { left = l
  , curr = c
  , right = r
  } = Tape l ((c + 1) `mod` 256) r
```

Section 4

Examples

An Example - FizzBuzz

```
fizzBuzz = zipWith stringify [1..] fizzBuzzes
  where
    stringify num "" = show num
    stringify _ str = str
    -- > stringify [(1, ""), (2, ""), (3, "Fizz")]
    -- > ["1", "2", "Fizz"]
    fizzBuzzes = zipWith (++) fizzes buzzes
    -- > ["", "", "Fizz", "", "Buzz", "Fizz",...]
    fizzes = cycle ["", "", "Fizz"]
    buzzes = cycle ["", "", "", "", "Buzz"]
```

```
["1","2","Fizz","4","Buzz","Fizz","7","8","Fizz"...]
```

Another Example - The Fibonnacci Sequence

A naive implementation

```
fib 0 = 1
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

A less naive implementation

```
fib = 1:1:(zipWith (+) fib (tail fib))

1:1:(      zipWith (+) 1:1:[...] 1:[...])
1:1:2:(    zipWith (+) 1:2:[...] 2:[...])
1:1:2:3:(  zipWith (+) 2:3:[...] 3:[...])
1:1:2:3:5:(zipWith (+) 3:5:[...] 5:[...])
```

Another Example - Prime Numbers

An implementation of the ***Sieve of Eratosthenes***

```
indexIsPrime = go 1 False : repeat True
  where
    go i (True : xs) = True : go (i + 1) sieve
      where
        mask = replicate (i - 1) True ++ [False]
        sieve = zipWith (&&) xs (cycle mask)
    go i (False : xs) = False : go (i + 1) xs

primes = map fst $ filter snd $ zip [1..] indexIsPrime
```

Section 5

Brainfuck

What is Brainfuck?

- Tape with cells holding a single byte each
- A pointer to a cell can be moved left and right
- The value of the cell can be incremented and decremented

Comment	Description
>	Move the pointer to the right
<	Move the pointer to the left
+	Increment the memory cell under the pointer
–	Decrement the memory cell under the pointer
.	Output the character signified by the cell at the pointer
,	Input a character and store it in the cell at the pointer
[	Jump past the matching] if the cell is 0
]	Jump back to the matching [if the cell is nonzero

The Idea

- Build an interpreter for Brainfuck in Haskell
- Code and input through `stdin` separated by `!`
- Do not use any side effects

Find the whole program including tests at https://github.com/XDracam/brainfuck-haskell

Subsection 1

Getting started

Defining the Tape

```
data Tape = Tape
  { left :: [Int]
  , curr :: Int
  , right :: [Int]
  } deriving (Eq)

emptyTape :: Tape
emptyTape = Tape [] 0 []
```

Printing the Tape

```
import Data.List (intercalate, intersperse)

instance Show Tape where
  show (Tape l c r) =
    show $ "[" ++ l'
        ++ "|>>" ++ show c ++ "<<|"
        ++ r' ++ "]"
    where
      l' = intersperse '|'
           $ intercalate ""
           $ show <$> reverse l
      r' = intersperse '|'
           $ intercalate ""
           $ show <$> r
```

Moving the tape

```
moveLeft :: Tape -> Tape
moveLeft Tape [] rh r = Tape [] 0 (rh : r)
moveLeft Tape (c:l) rh r = Tape l c (rh : r)

moveRight :: Tape -> Tape
moveRight Tape l lh [] = Tape (lh : l) 0 []
moveRight Tape l lh (c:r) = Tape (lh : l) c r
```

Incrementing and Decrementing

```
increment :: Tape -> Tape
increment t = t {curr = (curr t + 1) `mod` 256}

decrement :: Tape -> Tape
decrement t = t {curr = (curr t - 1) `mod` 256}
```

Reading and Writing

```
readChar :: Tape -> Char
readChar Tape {curr = c} = chr c

writeChar :: Tape -> Char -> Tape
writeChar t c = t {curr = ord c}
```

Note: `writeChar` returns a function that yields a new tape after taking a char to write. The actual IO is performed in the *IO layer*.

Subsection 2

Dealing with Input

Handle the Raw Input

```
extractCode :: String -> String
extractCode =
  filter (`elem` validChars) . takeWhile (/= '!')
  where
    validChars = "<>[],.+-"

parseInput :: [String] -> (String, String)
parseInput codeLines = (extractCode code, tail input)
  where
    codeWithLines = intercalate "\n" codeLines
    (code, input) = span (/= '!') codeWithLines
```

Validate Brackets

```haskell
data ValidationResult
  = TooManyOpen | TooManyClosed | Fine | NoCode
  deriving (Eq, Show)

validateBrackets :: String -> ValidationResult
validateBrackets code
  | null code = NoCode
  | count > 0 = TooManyOpen
  | count < 0 = TooManyClosed
  | otherwise = Fine
  where
    count sum '[' = sum + 1
    count sum ']' = sum - 1
    count sum _ = sum
    count = foldl count 0 code
```

Subsection 3

Interpreting the Code

Defining the Basics

```
handleChar :: Char -> Tape -> Tape
handleChar '>' = moveRight
handleChar '<' = moveLeft
handleChar '+' = increment
handleChar '-' = decrement
handleChar other = error $ "Unexpected char: " ++ [other]

data InterpreterState = InterpreterState
  { code :: String
  , seen :: String
  , input :: String
  , output :: String
  , tape :: Tape
  }
```

Running the code

```
interpretCode :: String -> String -> (Tape, String)
interpretCode code input =
  go (InterpreterState code "" input "" emptyTape)
  where
    go :: InterpreterState -> (Tape, String)
    go (InterpreterState "" _ _ out t) = (t, reverse out)
    go s@(InterpreterState (c:code) seen inp out t) =
```

...

Handling Read and Write

```
go s@(InterpreterState (c:code) seen inp out t) =
  case c of
    '.' -> go s { code = code, seen = '.' : seen
                , output = readChar t : out}
    ',' ->
      if null inp
        then error "Error: No input left."
        else go s {code = code, seen = seen'
                  , input = inp', tape = tape'}
      where ci:inp' = inp
            tape' = writeChar t ci
            seen' = ',' : seen
    -- LOOP HANDLING GOES HERE --
    c -> go s {code = code, seen = c : seen
              , tape = handleChar c t}
```

Find Corresponding Brackets

```
partitionByFinding :: Char -> String -> (String, String)
partitionByFinding c toView = go c toView "" 0
  where
    go :: Char -> String -> String -> Int -> (String, Stri
    go c [] found _ =
      error $
      "Unexpected error: Failure to find a " ++
      [c] ++ " after finding " ++ found
    go c (h:toView) found 0
      | c == h = (c : found, toView)
    go c (h:toView) found open =
      case h of
        '[' -> go c toView ('[' : found) (open + 1)
        ']' -> go c toView (']' : found) (open - 1)
        other -> go c toView (other : found) open
```

Handling Loops

```
go s@(InterpreterState (c:code) seen inp out t) =
-- READ/WRITE HANDLING GOES HERE --
'[' ->
  if curr t == 0 -- skip loop?
    then go s {code = todo, seen = loop ++ ('[' : seen)}
    else go s {code = code, seen = '[' : seen}
  where (loop, todo) = partitionByFinding ']' code
']' ->
  if curr t == 0 -- exit loop?
    then go s {code = code, seen = ']' : seen}
    else go s {code = loop ++ (']' : code), seen = rem}
  where (loop, rem) = partitionByFinding '[' seen
c -> go s {code = code, seen = c : seen
          , tape = handleChar c t}
```

Subsection 4

Dealing with IO and Side Effects

Dealing with Side Effects

- Haskell is **pure**: There are no side effects
- But every program interacts with its environment in some way
- The `IO` monad *describes* an interaction with the environment
- Descriptions can be *composed* through the *bind* operator >>=
- The `main` function in Haskell returns an `IO ()` which describes the sum of all side effects to be executed by the Haskell runtime

Simulating imperative programming

```
putStrLn :: String -> IO ()
getLine :: IO String
```

```
getLine >>= (\firstLine ->
  getLine >>= (\secondLine ->
    putStrLine (firstLine ++ secondLine)
      >> putStrLine "Done."))
```

is equivalent to:

```
do
  firstLine <- getLine
  secondLine <- getLine
  putStrLine $ firstLine ++ secondLine
  putStrLine "Done."
```

IO - Example

`getLine` yields an `IO String` which describes how to *later* yield a string by executing controlled side effects:

```haskell
takeLinesUntil :: (String -> Bool) -> IO [String]
takeLinesUntil predicate = go predicate []
  where
    go predicate lines = do
      line <- getLine
      if predicate line
        then return $ reverse lines
        else go predicate $ line : lines
```

```haskell
main :: IO ()
main = do
  args <- getArgs
  putStrLn "\nEnter code and input:\n"
  codeLines <- takeLinesUntil null
  let (code, input) = parseInput codeLines
  case validateBrackets code of
    TooManyOpen -> putStrLn tooManyOpenError
    TooManyClosed -> putStrLn tooManyClosedError
    NoCode -> putStrLn noCodeError
    Fine -> do
      let (out, _) = interpretCode code input
      putStrLn "Output:\n"
      putStrLn out
```