Haskell
or
functional purity and laziness

Soeren Roerden

Preliminaries

```haskell
{-# LANGUAGE BangPatterns #-}
module Intro where
import Test.QuickCheck
import Test.HUnit
import Text.Show.Functions
import Control.DeepSeq
```

Purity

Purity

Pure computations always result in the same value given the same inputs, meaning they can not perform side effects such as mutation of assignables or I/O.

Purity

```
var impure = function(x, y) {
  console.log("foobar");
  return x + y;
}

var pure = function(x, y) {
  return x + y;
}
```

Syntax and Type Signatures

Syntax and Type Signatures

- Haskell has white space sensitive syntax, just like Python.

Syntax and Type Signatures

- Haskell has white space sensitive syntax, just like Python.
- Type for cons - prepending an element to a list

```
Prelude> :t (:)
(:) :: a -> [a] -> [a]
```

Syntax and Type Signatures

- Haskell has white space sensitive syntax, just like Python.
- Type for cons - prepending an element to a list

```
Prelude> :t (:)
(:) :: a -> [a] -> [a]
```

- Haskell functions are curried by default

```
Prelude> :t (+)
(+) :: Num a => a -> a -> a

Prelude> :t uncurry (+)
uncurry (+) :: Num c => (c, c) -> c
```

Syntax and Type Signatures

- Haskell has white space sensitive syntax, just like Python.
- Type for cons - prepending an element to a list

```
Prelude> :t (:)
(:) :: a -> [a] -> [a]
```

- Haskell functions are curried by default

```
Prelude> :t (+)
(+) :: Num a => a -> a -> a

Prelude> :t uncurry (+)
uncurry (+) :: Num c => (c, c) -> c
```

- Juxtaposition is function application and has highest fixity. Use $ for least fixity.

```
Prelude> (*) 5 $ 3 + 2
25
Prelude> (*) 5 3 + 2
17
```

Syntax and Type Signatures

- Anonymous functions can be defined via the syntax

```
\x -> x + 1
```

Syntax and Type Signatures

- Anonymous functions can be defined via the syntax

  ```
  \x -> x + 1
  ```

- Function definition

  ```
  add1 x = x + 1
  add1' = (+ 1)
  add1'' = \x -> x + 1
  ```

Syntax and Type Signatures

- Anonymous functions can be defined via the syntax

```
\x -> x + 1
```

- Function definition

```
add1 x = x + 1
add1' = (+ 1)
add1'' = \x -> x + 1
```

- Ranges and list comprehensions

```
twoToFour = [2..4]
twoToFour' = [x + 1 | x <- [1..3]]
```

Syntax and Type Signatures

- Anonymous functions can be defined via the syntax

```
\x -> x + 1
```

- Function definition

```
add1 x = x + 1
add1' = (+ 1)
add1'' = \x -> x + 1
```

- Ranges and list comprehensions

```
twoToFour = [2..4]
twoToFour' = [x + 1 | x <- [1..3]]
```

- HUnit tests

```
map_test = map add1 [1,2,3] ~=? twoToFour
map_test' = map add1' [1,2,3] ~=? twoToFour'
```

Syntax and Type Signatures

- \`*fun*\` changes fixity of *fun* from prefix to infix, (*fun*) does the opposite

```
concat' a b = a ++ b
ctest = [concat' "foo" "bar" ~=? "foobar",
         "foo" `concat'` "bar" ~=? "foobar"]
```

Syntax and Type Signatures

- \`*fun*\` changes fixity of *fun* from prefix to infix, (*fun*) does the opposite

```
concat' a b = a ++ b
ctest = [concat' "foo" "bar" ~=? "foobar",
         "foo" `concat'` "bar" ~=? "foobar"]
```

- Pattern matching allows one to define functions by (exhaustive) cases

```
doubleVision :: [a] -> [a]
doubleVision (x:xs) = x : x : doubleVision xs
doubleVision [] = []
vision_test = doubleVision [1,2,3] ~=? [1,1,2,2,3,3]
```

Syntax and Type Signatures

- \`*fun*\` changes fixity of *fun* from prefix to infix, (*fun*) does the opposite

```
concat' a b = a ++ b
ctest = [concat' "foo" "bar" ~=? "foobar",
         "foo" `concat'` "bar" ~=? "foobar"]
```

- Pattern matching allows one to define functions by (exhaustive) cases

```
doubleVision :: [a] -> [a]
doubleVision (x:xs) = x : x : doubleVision xs
doubleVision [] = []
vision_test = doubleVision [1,2,3] ~=? [1,1,2,2,3,3]
```

- Guards enable more flexible checking in pattern matching

```
abs' x
  | x >= 0 = x
  | x < 0 = -x
```

Interlude - QuickCheck

- Quickcheck property tests

```haskell
abs_check :: Integer -> Bool
abs_check x = abs' x >= 0
```

Interlude - QuickCheck

- Quickcheck property tests

```
abs_check :: Integer -> Bool
abs_check x = abs' x >= 0
```

- ```
  Prelude> :load haskell.lhs
  [1 of 1] Compiling Intro            ( haskell.lhs, interpreted )
  Ok, modules loaded: Intro.
  *Intro> quickCheck mymap_check
  +++ OK, passed 100 tests.
  ```
  ```

Syntax and Type Signatures

- Let bindings

```
isPalindrome x = let rev = reverse x in
  rev == x
palindrome_test = isPalindrome "abba" ~=? True
```

Syntax and Type Signatures

- Let bindings

```
isPalindrome x = let rev = reverse x in
  rev == x
palindrome_test = isPalindrome "abba" ~=? True
```

- Where bindings

```
fibonacci n = fibonacci' n 1 0 where
  fibonacci' s c p = if s <= 1
                     then c
                     else fibonacci' (s - 1) (c + p) c
```

Recursion and Higher Order Functions

Syntax and Type Signatures

- What's wrong with this definition?

```
mehFibonacci n
  | n <= 2      = 1
  | otherwise = mehFibonacci (n-1) + mehFibonacci (n-2)
mehFibonacci_test = mehFibonacci 5 ~=? fibonacci 5
```

Syntax and Type Signatures

- What's wrong with this definition?

```
mehFibonacci n
  | n <= 2      = 1
  | otherwise = mehFibonacci (n-1) + mehFibonacci (n-2)
mehFibonacci_test = mehFibonacci 5 ~=? fibonacci 5
```

- ```
  *Intro> mehFibonacci 100000
  *BARF* *stackoverflow* *die*
  ```
  ```

Syntax and Type Signatures

- What's wrong with this definition?

```
mehFibonacci n
  | n <= 2      = 1
  | otherwise = mehFibonacci (n-1) + mehFibonacci (n-2)
mehFibonacci_test = mehFibonacci 5 ~=? fibonacci 5
```

- ```
  *Intro> mehFibonacci 100000
  *BARF* *stackoverflow* *die*
  ```

- Solution: tail calls

```
fibonacci n = fibonacci' n 1 0 where
  fibonacci' s c p = if s <= 1
                     then c
                     else fibonacci' (s - 1) (c + p) c
```
```

Higher order functions

Here's how you implement map

- ```
  myMap :: (a -> b) -> [a] -> [b]
  myMap f (x:xs) = f x : myMap f xs
  myMap f [] = []

  myMap_check :: [Integer] -> Bool
  myMap_check x = myMap (+1) x == map (+1) x
  ```
  ```

Laziness

Laziness

- Counting to infinity

```
infty = [1..]
```

Laziness

- Counting to infinity

  ```
  infty = [1..]
  ```

- Forcing the issue?

  ```
  finiteSlice = take 100 infty
  ```

Laziness

- Counting to infinity

```
infty = [1..]
```

- Forcing the issue?

```
finiteSlice = take 100 infty
```

- Don't try to reduce me to normal form..

```
take' n x = x `deepseq` take n x
loooop :: [Integer]
loooop = take' 10 [1..]
```

Laziness

- Space leaks, or “Oh my god, it’s full of stars..”

  ```
  foldl (+) 0 [1..10000000]
  ```

Laziness

- Space leaks, or "Oh my god, it's full of stars.."

```
foldl (+) 0 [1..10000000]
```

- This is pretty fast

```
import Data.List
foldl' (+) 0 [1..10000000]
```

Laziness

- Space leaks, or "Oh my god, it's full of stars.."

```
foldl (+) 0 [1..10000000]
```

- This is pretty fast

```
import Data.List
foldl' (+) 0 [1..10000000]
```

- Defining a strict(ish) fold using !

```
foldl'' f start xs = run start xs where
  run !acc (x:xs) = run (f acc x) xs
  run !acc [] = acc
```

Typeclasses and (Inductive) Data Types

Typeclasses

- Problem - Ad hoc polymorphism:

```haskell
isEqual :: ? a -> a -> Bool
isEqual x y = ?
```

Typeclasses

- Problem - Ad hoc polymorphism:

```
isEqual :: ? a -> a -> Bool
isEqual x y = ?
```

- Solution:

```
class Equality a where
  eq :: a -> a -> Bool

isEqual :: (Equality a) => a -> a -> Bool
isEqual x y = x `eq` y
```

Algebraic Data Types

- Defining new data types as sums of products

```
data Boolean = BTrue | BFalse deriving (Show)
data Option a = Some a | None deriving (Eq, Show)
data ConsList a = CNil | Cons a (ConsList a) deriving (Eq, Show)
data BinTree a = Empty | Node (BinTree a) a (BinTree a)
               deriving (Show)
```

Algebraic Data Types

- Defining new data types as sums of products

```
data Boolean = BTrue | BFalse deriving (Show)
data Option a = Some a | None deriving (Eq, Show)
data ConsList a = CNil | Cons a (ConsList a) deriving (Eq, Show)
data BinTree a = Empty | Node (BinTree a) a (BinTree a)
               deriving (Show)
```

- Using them

```
mapConsList :: (a -> b) -> ConsList a -> ConsList b
mapConsList f (Cons x xs) = Cons (f x) (mapConsList f xs)
mapConsList f CNil = CNil

mapConsList_test = mapConsList (* 2)
                   (Cons 1 $ Cons 2 $ CNil) ~=?
                   (Cons 2 $ Cons 4 $ CNil)
```

Defining instances

- Custom instances

```
instance Equality Boolean where
  eq BTrue BTrue = True
  eq BFalse BFalse = True
  eq _ _ = False

instance_tests = [BTrue `eq` BTrue ~=? True,
                  BTrue `eq` BFalse ~=? False]
```

Defining instances

- Custom instances

```
instance Equality Boolean where
  eq BTrue BTrue = True
  eq BFalse BFalse = True
  eq _ _ = False

instance_tests = [BTrue `eq` BTrue ~=? True,
                  BTrue `eq` BFalse ~=? False]
```

- Nesting

```
instance Equality a => Equality (ConsList a) where
  eq x y = isEq True x y where
    isEq True CNil CNil = True
    isEq True (Cons x xs) (Cons y ys) = isEq (x `eq` y) xs ys
    isEq _ _ _ = False

instance_tests' = [Cons BTrue CNil `eq` Cons BTrue CNil ~=? True,
                   Cons BTrue CNil `eq` CNil ~=? False]
```

Records

- Defining a carrot

```
data Carrot = Carrot { len :: Integer
                     , color :: String
                     , taste :: String
                     } deriving (Show, Eq)
some_carrot = Carrot { len = 8,
                       color = "reddish",
                       taste = "hints of carrot" }
newtype Color = Color { getColor :: String }
```

Records

- Defining a carrot

```haskell
data Carrot = Carrot { len :: Integer
                     , color :: String
                     , taste :: String
                     } deriving (Show, Eq)
some_carrot = Carrot { len = 8,
                       color = "reddish",
                       taste = "hints of carrot" }
newtype Color = Color { getColor :: String }
```

- Accessors are auto generated

```
*Intro> :t color
color :: Carrot -> String
```

Records

- Defining a carrot

```
data Carrot = Carrot { len :: Integer
                     , color :: String
                     , taste :: String
                     } deriving (Show, Eq)
some_carrot = Carrot { len = 8,
                       color = "reddish",
                       taste = "hints of carrot" }
newtype Color = Color { getColor :: String }
```

- Accessors are auto generated

```
*Intro> :t color
color :: Carrot -> String
```

- Most obnoxious thing in all of Haskell: record names must be unique within a given module

Functors, Idioms, Monads .. also Monoids

Monoids

- Motivation: You want to combine some things, like strings

  ```
  combineStrings xs = foldl (++) "" xs
  ```

  But suddenly you have to combine (Option)al strings and would prefer not to repeat yourself

Monoids

- Motivation: You want to combine some things, like strings

  ```
  combineStrings xs = foldl (++) "" xs
  ```

  But suddenly you have to combine (Option)al strings and would prefer not to repeat yourself
- Monoids to the rescue

  ```
  class Monoid a where
    mempty :: a
    mappend :: a -> a -> a
    mconcat :: [a] -> a
    mconcat = foldr mappend mempty
  ```

Monoids

- Instantiating..

```
instance Monoid a => Monoid (Option a) where
  mempty = None
  mappend (Some x) (Some y) = Some (x `mappend` y)
  mappend None x = x
  mappend x None = x

-- strings are lists of characters
instance Monoid [a] where
  mempty = []
  mappend = (++)
```

Monoids

- ```
  combineThings :: Monoid a => [a] -> a
  combineThings xs = mconcat xs

  combining_tests = [
    combineThings ["foo", "bar", "baz"] ~=?
    "foobarbaz",
    combineThings [Some "foo", None, Some "bar"] ~=?
    Some "foobar"]
  ```
  ```

Functors

- Classic problem:

```
getFromMaps x y =
  case lookup "foo" x of
    Just val -> case lookup "bar" y of
      Just val2 -> Just (val ++ val2)
      Nothing -> Nothing
    Nothing -> Nothing
-- ARGH
```

Functors

- Classic problem:

```
getFromMaps x y =
  case lookup "foo" x of
    Just val -> case lookup "bar" y of
      Just val2 -> Just (val ++ val2)
      Nothing -> Nothing
    Nothing -> Nothing
-- ARGH
```

- Or even just:

```
doSomethingMaybe x = case x of
  Just a -> Just $ a ++ "bar"
  Nothing -> Nothing
```

Functors

- Enter functors

```
doSomethingMaybe' x = fmap (++ "bar") x
```

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

```
instance Functor Option where
  fmap f (Some x) = Some (f x)
  fmap f None = None
```

Functors

- Enter functors

  ```
  doSomethingMaybe' x = fmap (++ "bar") x
  ```

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

  ```
  instance Functor Option where
    fmap f (Some x) = Some (f x)
    fmap f None = None
  ```

- Option has *kind* $* \rightarrow *$, which means it forms a concrete type of kind $*$ only when applied to a type argment.

Idioms - Applicative Functors

- Queue Strauss' Sunrise - What if I could just write this

```
getFromMaps' x y = (++) <$> lookup "foo" x <*> lookup "bar" y
```

Idioms - Applicative Functors

- Queue Strauss' Sunrise - What if I could just write this

```
getFromMaps' x y = (++) <$> lookup "foo" x <*> lookup "bar" y
```

- Applicative functors

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

instance Applicative Option where
  pure = Some
  None <*> _ = None
  Some f <*> v = fmap f v

(<$>) :: (Functor f) => (a -> b) -> f a -> f b
f <$> x = fmap f x
```

Monads

- Even though they have many, many other uses, let's look at I/O. Idea:
  `type IO a = RealWorld -> (a, RealWorld)`

Monads

- Even though they have many, many other uses, let's look at I/O. Idea:

  ```
  type IO a = RealWorld -> (a, RealWorld)
  ```

- Applicatives can't guarantee ordering of effects, we have to make sure that things are evaluated and bound in the right sequence.

Monads

- Even though they have many, many other uses, let's look at I/O. Idea:

  ```
  type IO a = RealWorld -> (a, RealWorld)
  ```

- Applicatives can't guarantee ordering of effects, we have to make sure that things are evaluated and bound in the right sequence.
- The IO Monad allows one to write

  ```
  getInputAndPrint :: IO ()
  getInputAndPrint = do
    putStrLn "Give me some input"
    input <- getLine
    putStrLn input
  ```

  making sure that the actions are performed in the specified sequence

Monads

- So how does this work? Let's start with a definition:

```haskell
class (Applicative m) => Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
-- alternatively join :: m (m a) -> m a
```

Monads

- So how does this work? Let's start with a definition:

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

- Option instance

```
instance Monad Option where
  return = Some
  Some x >>= f = f x
  None >>= f = None
```

Monads

- How to use it?

```
getFromMaps'' x y = lookup "foo" x >>=
                    (\val1 -> (lookup "bar" y >>=
                              (\val2 -> return $ val1 ++ val2)))
```

Monads

- How to use it?

```
getFromMaps'' x y = lookup "foo" x >>=
                    (\val1 -> (lookup "bar" y >>=
                               (\val2 -> return $ val1 ++ val2)))
```

- Since this doesn't look so nice, introduce some sugar

```
getFromMaps''' x y = do
  val1 <- lookup "foo" x
  val2 <- lookup "bar" y
  return $ val1 ++ val2
```

Monads

- How to use it?

```haskell
getFromMaps'' x y = lookup "foo" x >>=
                    (\val1 -> (lookup "bar" y >>=
                               (\val2 -> return $ val1 ++ val2)))
```

- Since this doesn't look so nice, introduce some sugar

```haskell
getFromMaps''' x y = do
  val1 <- lookup "foo" x
  val2 <- lookup "bar" y
  return $ val1 ++ val2
```

- List is also a monad

```haskell
do
  x <- [1,2,3]
  fs <- [(+1), (+2)]
  return $ fs x
[2,3,3,4,4,5]
```

Monads

- That was easy, so let's move on to State..

```
newtype State' s a = State' { runState :: s -> (a, s) }
instance Monad (State' a) where
  return x = State' $ \s -> (x, s)
  oldState >>= f = State' $ \s ->
    let (intermediateVal, intermediateState) = runState oldState s
    in runState (f intermediateVal) intermediateState
```

Monads

- That was easy, so let's move on to State..

```
newtype State' s a = State' { runState :: s -> (a, s) }
instance Monad (State' a) where
  return x = State' $ \s -> (x, s)
  oldState >>= f = State' $ \s ->
    let (intermediateVal, intermediateState) = runState oldState s
    in runState (f intermediateVal) intermediateState
```

- Utilities

```
put :: a -> State' a ()
put s = State' $ \_ -> ((), s)

get = State' $ \s -> (s, s)
```

Codata

Codata

- [1..] rewritten

```haskell
infty' = go 1 where
  go n = n : go (n + 1)
```

Codata

- [1..] rewritten

```haskell
infty' = go 1 where
  go n = n : go (n + 1)
```

- An infinite list of fibonacci numbers

```haskell
fibs = 1 : 1 : rec fibs where
  rec (x:y:xs) = (x + y) : rec (y : xs)
cofibonacci n = head $ drop (n - 1) $ take n fibs
```

Codata

- [1..] rewritten

```haskell
infty' = go 1 where
  go n = n : go (n + 1)
```

- An infinite list of fibonacci numbers

```haskell
fibs = 1 : 1 : rec fibs where
  rec (x:y:xs) = (x + y) : rec (y : xs)
cofibonacci n = head $ drop (n - 1) $ take n fibs
```

- The blurred distinction between data and codata

```haskell
Prelude> head $ 1 : undefined
1
```

Existential Types

Existential Types

```haskell
data Color' = Color' { red :: Integer, green :: Integer,
                       blue :: Integer, alpha :: Maybe Integer }
              deriving (Show, Eq)

class Colorizable a where
  colorize :: a -> Maybe Color'
-- I have omitted the instance declarations, insert here

data Colorlike = forall a. Colorizable a => Colorish

colorlist :: [Colorlike]
colorlist = [Colorish "255/50/0/20", Colorish [255, 50, 0, 20]]
```

Epilogue

Join the Haskell meetup!: `http://www.meetup.com/NY-Haskell/`

Epilogue

- Get the Haskell Platform at `http://www.haskell.org/platform/`

Epilogue

- Get the Haskell Platform at `http://www.haskell.org/platform/`
- Install QuickCheck and HUnit using

  `cabal install HUnit QuickCheck`

Epilogue

- Get the Haskell Platform at `http://www.haskell.org/platform/`
- Install QuickCheck and HUnit using

  ```
  cabal install HUnit QuickCheck
  ```
- Grab the slides and run

  ```
  make extract
  ```

  to produce plain Haskell code or

  ```
  runhaskell haskell.lhs
  ```

  to just run it

Epilogue

- Get the Haskell Platform at `http://www.haskell.org/platform/`
- Install QuickCheck and HUnit using

  `cabal install HUnit QuickCheck`
- Grab the slides and run

  `make extract`

  to produce plain Haskell code or

  `runhaskell haskell.lhs`

  to just run it
- To play around with it in ghci use

  `Prelude>:load haskell.lhs`

Epilogue

```
tests = test $ [map_test, map_test', vision_test,
                palindrome_test, mehFibonacci_test,
                mapConsList_test] ++
        ctest ++ instance_tests ++ instance_tests' ++
        combining_tests
main = do
  runTestTT tests
  quickCheck myMap_check
  quickCheck abs_check
```