Getting Started with Haskell

Munich Lambda - Paul Koerbitz

January 20, 2014

Getting started

- Clone:
  `git clone git@github.com:defworkshop/haskell-workshop.git`

Getting started

- Clone:
  `git clone`
  `git@github.com:defworkshop/haskell-workshop.git`
- Get dependencies:
  `$ cabal install --dependencies-only`

Getting started

- Clone:
  `git clone git@github.com:defworkshop/haskell-workshop.git`
- Get dependencies:
  `$ cabal install --dependencies-only`
- use `GHCi`:
  `$ cabal repl`

Basic Functions

Functions

```
abs :: Int -> Int
abs x = if x < 0 then (-x) else x
```

Functions - Pattern Matching, Recursion, Precedence

```haskell
add :: Int -> Int -> Int
add x 0 = x
add x y = if y > 0
          then add (succ x) (pred y)
          else add (pred x) (succ y)
```

where Bindings

```haskell
add :: Int -> Int -> Int
add x 0 = x
add x y = if y > 0
          then add succ_x pred_y
          else add pred_x succ_y
    where
        pred_x = pred x
        succ_x = succ x
        pred_y = pred y
        succ_y = succ y
```

let Bindings

```haskell
add :: Int -> Int -> Int
add x 0 = x
add x y = if y > 0
          then let succ_x = succ x
                   pred_y = pred y
               in add succ_x pred_y
          else let pred_x = pred x
                   succ_y = succ y
               in add pred_x succ_y
```

Datatypes

Datatypes - Basics

- Some datatypes are built in, for example `Int`, `Integer`, `Char`, `Double`, ...

Datatypes - Basics

- Some datatypes are built in, for example `Int`, `Integer`, `Char`, `Double`, . . .
- Datatypes can be defined with the `data` keyword:

```haskell
-- This is like a `struct` in other languages
data IntPair = IntPair Int Int

-- This is like an `enum` in other languages
data Color = Red
           | Green
           | Blue
```

Datatypes - Product Types and Records

- *Product types* are much like `struct`s in C, C++, etc.

```
data Car = Car String String Int Int
```

Datatypes - Product Types and Records

- *Product types* are much like `structs` in C, C++, etc.

```
data Car = Car String String Int Int
```

- *Records* give the fields names.

```
data Car = Car { carMake :: String,
                 carModel :: String,
                 carYear :: Int,
                 carHorsepower :: Int }
```

Datatypes - Product Types and Records

- *Product types* are much like `structs` in C, C++, etc.

```
data Car = Car String String Int Int
```

- *Records* give the fields names.

```
data Car = Car { carMake :: String,
                 carModel :: String,
                 carYear :: Int,
                 carHorsepower :: Int }
```

  - Name of records fields live in the same namespace as other bindings, so they must be unique in a module.

Datatypes - Sum Types aka Disjoint Unions

- *Sum types* can have several representations

```
data Tx = InitialTx Timestamp Amount
        | FailedTx  Timestamp Amount String
        | SuccessfulTx Timestamp Amount String
```

Datatypes - Sum Types aka Disjoint Unions

- *Sum types* can have several representations

```
data Tx = InitialTx Timestamp Amount
        | FailedTx  Timestamp Amount String
        | SuccessfulTx Timestamp Amount String
```

- We can work with sum types by pattern matching their constructors:

```
getFailedTimestamp :: Tx -> Maybe Timestamp
getFailedTimestamp (FailedTx ts _ _) = Just ts
getFailedTimestamp _                 = Nothing
```

Datatypes - Recursive Datatypes

- Datatype definitions can refer to themselves:

```
data IntList = ILNil
             | ILCons Int IntList
```

Datatypes - Recursive Datatypes

- Datatype definitions can refer to themselves:

```
data IntList = ILNil
             | ILCons Int IntList
```

- ... and can be processed by recursion

```
length :: IntList -> Int
length ILNil          = 0
length (ILCons _ xs) = length xs + 1
```

Type Parameters

- We don't want to define lists for every single datatype! Type parameters allow this:

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

Type Parameters

- We don't want to define lists for every single datatype! Type parameters allow this:

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

- Type parameters can also be used in functions:

```
length :: List a -> Int
length Nil          = 0
length (Cons _ xs) = length xs + 1
```

Type Parameters

- We don't want to define lists for every single datatype! Type parameters allow this:

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

- Type parameters can also be used in functions:

```
length :: List a -> Int
length Nil           = 0
length (Cons _ xs) = length xs + 1
```

- Haskell has syntactic sugar for lists: `[]` is `Nil`, `x:xs` is `Cons x xs` and `[a]` is `List a`:

```
length :: [a] -> Int
length []     = 0
length (_:xs) = length xs + 1
```

Typeclasses 101

- Many operations should work for values of many, but not all types. This can be achieved with *typeclasses* in Haskell.

```
qsort :: Ord a => [a] -> [a]
qsort []     = []
qsort (x:xs) = lessOrEqual ++ [x] ++ greater
  where
    lessOrEqual = filter (<= x) xs
    greater     = filter (> x) xs
```

Typeclasses 101

- Many operations should work for values of many, but not all types. This can be achieved with *typeclasses* in Haskell.

```
qsort :: Ord a => [a] -> [a]
qsort []     = []
qsort (x:xs) = lessOrEqual ++ [x] ++ greater
  where
    lessOrEqual = filter (<= x) xs
    greater     = filter (> x) xs
```

- Useful typeclasses include `Eq`, `Ord`, `Show`, `Num`, `Enum`

Typeclasses 101

- Many operations should work for values of many, but not all types. This can be achieved with *typeclasses* in Haskell.

```
qsort :: Ord a => [a] -> [a]
qsort []     = []
qsort (x:xs) = lessOrEqual ++ [x] ++ greater
  where
    lessOrEqual = filter (<= x) xs
    greater     = filter (> x) xs
```

- Useful typeclasses include `Eq`, `Ord`, `Show`, `Num`, `Enum`
- New datatypes can sometimes be given instances in typeclasses with the deriving keyword:

```
data Pair a b = Pair a b
                deriving (Eq, Ord, Show)
```

Higher Order Functions

Higher Order Functions

- Higher order functions take functions as arguments.

Higher Order Functions

- Higher order functions take functions as arguments.
- Example: Apply a function to every element in a list:

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

Higher Order Functions

- Higher order functions take functions as arguments.
- Example: Apply a function to every element in a list:

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

- Example: *Fold* a list to a singe element:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl _ acc []       = acc
foldl f acc (x:xs) = foldl f (f acc x) xs
```

Where is my `for` loop?

- There are no `for`, `while`, or similar loops in Haskell.

Where is my `for` loop?

- There are no `for`, `while`, or similar loops in Haskell.
- Many specific iteration patterns are factored into higher-order-functions such as `map` and `foldl`.

Where is my `for` loop?

- There are no `for`, `while`, or similar loops in Haskell.
- Many specific iteration patterns are factored into higher-order-functions such as `map` and `foldl`.
- You can write your own loops via recursion:

```
countIf :: (a -> Bool) -> [a] -> Int
countIf p xs = go 0 p xs
  where
    go cnt _ []       = cnt
    go cnt p (x:xs) = if p x
                         then go (cnt+1) p xs
                         else go cnt p xs
```

Lambdas

- Higher order functions become even more useful if we can create functions in place.

Lambdas

- Higher order functions become even more useful if we can create functions in place.
- Lambdas can be created with the `\ ->` syntax:

```
countIf :: (a -> Bool) -> [a] -> Int
countIf p xs = foldl (\cnt x -> if p x
                                then (cnt+1)
                                else cnt) 0 xs
```

Additional Points

Laziness

- Lazy evaluation is the default in Haskell, so arguments are not evaluated until they have to be.

Laziness

- Lazy evaluation is the default in Haskell, so arguments are not evaluated until they have to be.
- You can see this in *ghci*:

```
ghci> let a = sum [1..10*1000*1000]
-- This is very fast
ghci> show a
-- This takes some time
"50000005000000"
```

Laziness

- Lazy evaluation is the default in Haskell, so arguments are not evaluated until they have to be.
- You can see this in *ghci*:

```
ghci> let a = sum [1..10*1000*1000]
-- This is very fast
ghci> show a
-- This takes some time
"50000005000000"
```

- The great thing about laziness is that it decouples production from consumption.

When Laziness bites

- Unfortunately laziness can sometimes have unexpected consequences:

```
length :: Int -> [a] -> Int
length acc []     = acc
length acc (x:xs) = length (1+acc) xs
```

```
ghci> length [1..10*1000*1000]
-- Takes forever, needs huge amounts of memory
```

When Laziness bites

- Unfortunately laziness can sometimes have unexpected consequences:

```
length :: Int -> [a] -> Int
length acc []     = acc
length acc (x:xs) = length (1+acc) xs
```

```
ghci> length [1..10*1000*1000]
-- Takes forever, needs huge amounts of memory
```

- The problem is that `(+)` is lazy, so we build up a huge *thunk* `(1+(1+(1+(1+(1+ ...)))))`

When Laziness bites

- Unfortunately laziness can sometimes have unexpected consequences:

```
length :: Int -> [a] -> Int
length acc []     = acc
length acc (x:xs) = length (1+acc) xs
```

```
ghci> length [1..10*1000*1000]
-- Takes forever, needs huge amounts of memory
```

- The problem is that `(+)` is lazy, so we build up a huge *thunk* `(1+(1+(1+(1+(1+ ...)))))`
- We can avoid this by forcing the evaluation of `acc` with `seq`:

```
length :: Int -> [a] -> Int
length acc []     = acc
length acc (x:xs) = acc `seq` length (1+acc) xs
```

Currying and Partial Function Application

- *Currying*: Take a function of n parameters into a function of n-k parameters which returns a function of k parameters.

Currying and Partial Function Application

- *Currying*: Take a function of n parameters into a function of n-k parameters which returns a function of k parameters.
- Currying is the default modus operandi in Haskell:

```
add :: Int -> Int -> Bool
add x y = x + y
```

  add is a function that takes an `Int` and returns a function of type `Int -> Int`.

Currying and Partial Function Application

- *Currying*: Take a function of n parameters into a function of n-k parameters which returns a function of k parameters.
- Currying is the default modus operandi in Haskell:

  ```haskell
  add :: Int -> Int -> Bool
  add x y = x + y
  ```

  add is a function that takes an `Int` and returns a function of type `Int -> Int`.
- Since functions are *curried* by default, partial function application is very natural in Haskell:

  ```haskell
  map (add 3) [1..5] -- [4, 5, 6, 7, 8]
  ```

Operators are just functions

- Haskell may seem like it is full of operators, but operators are just functions:

```
(!?) -> [a] -> Int -> Maybe a
(!?) []       _ = Nothing
(!?) (x:xs) 0 = Just x
(!?) (x:xs) n = (!?) xs (n-1)
```

Operators are just functions

- Haskell may seem like it is full of operators, but operators are just functions:

```
(!?) -> [a] -> Int -> Maybe a
(!?) []      _ = Nothing
(!?) (x:xs) 0 = Just x
(!?) (x:xs) n = (!?) xs (n-1)
```

- Operators are written inline by default, but don't have to be:

```
ghci> [0,1,2,3,4] !? 3
Just 3
ghci> (!?) [0,1,2,3,4] 3
Just 3
```

Operators are just functions

- Haskell may seem like it is full of operators, but operators are just functions:

```
(!?) -> [a] -> Int -> Maybe a
(!?) []     _ = Nothing
(!?) (x:xs) 0 = Just x
(!?) (x:xs) n = (!?) xs (n-1)
```

- Operators are written inline by default, but don't have to be:

```
ghci> [0,1,2,3,4] !? 3
Just 3
ghci> (!?) [0,1,2,3,4] 3
Just 3
```

- We can write regular functions *inline* by surrounding them with backticks:

```
ghci> 6 `mod` 3
0
```

Pointfree vs. Pointful Style

- So far we have written our Haskell in what is called *pointful* style:

```
countIf :: (a -> Bool) -> [a] -> Int
countIf p xs = foldl (\cnt x -> if p x
                                then (cnt+1)
                                else cnt) 0 xs
```

Pointfree vs. Pointful Style

- So far we have written our Haskell in what is called *pointful* style:

```haskell
countIf :: (a -> Bool) -> [a] -> Int
countIf p xs = foldl (\cnt x -> if p x
                                then (cnt+1)
                                else cnt) 0 xs
```

- An alternative is *point-free* style:

```haskell
countIf :: (a -> Bool) -> [a] -> Int
countIf p = foldl (\cnt x -> if p x
                             then (cnt+1)
                             else cnt) 0
```

Pointfree vs. Pointful Style

- So far we have written our Haskell in what is called *pointful* style:

```
countIf :: (a -> Bool) -> [a] -> Int
countIf p xs = foldl (\cnt x -> if p x
                                then (cnt+1)
                                else cnt) 0 xs
```

- An alternative is *point-free* style:

```
countIf :: (a -> Bool) -> [a] -> Int
countIf p = foldl (\cnt x -> if p x
                             then (cnt+1)
                             else cnt) 0
```

- Pointfree style focuses on how functions can be defined in terms of other functions.

(.) and ($)

- Functions can be composed with the function composition operator (.):

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g x = f (g x)
```

(.) and ($)

- Functions can be composed with the function composition operator (.):

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g x = f (g x)
```

- The ($) operator is *function application*, but has very low precedence and binds to the right. This can be convenient to avoid writing to many parenthesis:

```
concat . map show . take 10 $ [1..]
```

Hack, Hack, Hurra!

Need a project?

- Write a Sudoku solver

Hack, Hack, Hurra!

Need a project?

- Write a Sudoku solver
- Auf wie viele Arten kann man das 'Haus-vom-Nikolaus' zeichnen? `http://de.wikipedia.org/wiki/Haus_vom_Nikolaus`

Hack, Hack, Hurra!

Need a project?

- Write a Sudoku solver
- Auf wie viele Arten kann man das ‘Haus-vom-Nikolaus’ zeichnen? `http://de.wikipedia.org/wiki/Haus_vom_Nikolaus`
- Hashlife: `http://www.drdobbs.com/jvm/an-algorithm-for-compressing-space-and-t/184406478`