Introduction to Functional Programming in Haskell

June 27, 2017
Outline

Haskell Basics

What is functional programming?
  What is a function?
  Equational reasoning
  First-order vs. higher-order functions
  Lazy evaluation

How to functional program
  Functional programming workflow
  Data types
  Type-directed programming
  Haskell style
  Refactoring (bonus section)
Outline

Haskell Basics

What is functional programming?
  What is a function?
  Equational reasoning
  First-order vs. higher-order functions
  Lazy evaluation

How to functional program
  Functional programming workflow
  Data types
  Type-directed programming
  Haskell style
  Refactoring (bonus section)
What is a (pure) function?

A function is **pure** if:

- it *always* returns the same output for the same inputs
- it doesn’t do anything else — no “side effects”

In Haskell: whenever we say “function” we mean **pure function**!
What are (and aren’t) functions?

Always functions:

- mathematical functions $f(x) = x^2 + 2x + 3$
- encryption and compression algorithms

Usually not functions:

- C, Python, JavaScript,... “functions” (procedures)
- Java, C#, Ruby,... methods

Haskell **only** allows you to write (pure) functions!
Why procedures/methods aren’t functions

What is functional programming?
Outline

Haskell Basics

What is functional programming?
  What is a function?
  Equational reasoning
  First-order vs. higher-order functions
  Lazy evaluation

How to functional program
  Functional programming workflow
  Data types
  Type-directed programming
  Haskell style
  Refactoring (bonus section)
Getting into the Haskell mindset

Same symbol, different meaning!
Referential transparency

An expression can be replaced by its value without changing the overall program behavior \((\text{value a.k.a referent})\)

\[
\text{length } [1,2,3] + 4 \\
3 + 4
\]

what if \texttt{length} was a Java method?

Corollary: an expression can be replaced by any expression with the same value without changing program behavior

Supports \textit{equational reasoning}
Equational reasoning

**Computation is just substitution!**

```
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + sum xs
```

```
sum [2,3,4]
⇒ sum (2:(3:(4:[])))
⇒ 2 + sum (3:(4:[]))
⇒ 2 + 3 + sum (4:[])
⇒ 2 + 3 + 4 + sum []
⇒ 2 + 3 + 4 + 0
⇒ 9
```
So then how to I *do anything* in Haskell?

Simple answer... *you don’t!*

Instead you **describe**!
Describing computations

Function definition: a list of equations that relate input to output

Example: reversing a list
- **imperative view**: how do I rearrange the elements in a list?
- **functional view**: how is a list related to its reversal?

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

**Exercise**: use equational reasoning to compute the reverse of the list \([2, 3, 4, 5]\)
Exercise: using equational reasoning

reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]

Pattern matching:
1. conditional
2. bindings

reverse [2,3,4,5] =
reverse [3,4,5] ++ [2] =
reverse [4,5] ++ [3] ++ [2] =

What is functional programming?
Four steps to learning how to program

**Language implementation** — how to evaluate programs

**Output** — how to run programs

**Program** — how to write programs

**Input** — how to define programs
Four steps to learning Haskell

Language implementation — how to evaluate programs
  how to evaluate expressions
Output — how to run programs
  how to apply functions
Program — how to write programs
  how to define functions
Input — how to define programs
  how to define types and values
Outline

Haskell Basics

What is functional programming?
  What is a function?
  Equational reasoning
  First-order vs. higher-order functions
  Lazy evaluation

How to functional program
  Functional programming workflow
  Data types
  Type-directed programming
  Haskell style
  Refactoring (bonus section)
First-order functions

What is functional programming?

Examples
\[
\begin{align*}
\cos & \quad : \text{Float} \rightarrow \text{Float} \\
\text{even} & \quad : \text{Int} \rightarrow \text{Bool} \\
\text{length} & \quad : [\text{a}] \rightarrow \text{Int}
\end{align*}
\]
Higher-order functions

What is functional programming?

Examples

map :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
(.) :: (b -> c) -> (a -> b) -> a -> c
Higher-order functions as control structures

**map:** loop for doing something to each element in a list

- \( \text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \)
- \( \text{map} f \[\] = \[\] \)
- \( \text{map} f (x:xs) = f x : \text{map} f xs \)

\[
\text{map } f [2,3,4,5] = \[f 2,f 3,f 4,f 5\]
\]

\[
\text{map even } [2,3,4,5] \\
= \text{[even 2, even 3, even 4, even 5]} \\
= \text{[True,False,True,False]}
\]

**foldr:** loop for aggregating elements in a list

- \( \text{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \)
- \( \text{foldr} f y \[\] = y \)
- \( \text{foldr} f y (x:xs) = f x (\text{foldr} f y xs) \)

\[
\text{foldr } f y [2,3,4] = f 2(f 3(f 4 y))
\]

\[
\text{foldr } (+) 0 [2,3,4] \\
= (+) 2 ((+) 3 ((+) 4 0)) \\
= 2 + (3 + (4 + 0)) \\
= 9
\]
Function composition

Create new functions by **composing** existing functions

- apply the *second* function to the input
- *then* apply the *first* function to output

\[(f \circ g) x = f (g x)\]

**Function composition**

\[
(\cdot) :: (b \to c) \to (a \to b) \to a \to c
\]

\[
f \cdot g = \lambda x \to f (g x)
\]

**Existing functions (types)**

- `not :: Bool \to Bool`
- `succ :: Int \to Int`
- `even :: Int \to Bool`
- `head :: [a] \to a`
- `tail :: [a] \to [a]`

**New function definitions**

- `plus2 = succ . succ`
- `odd = not . even`
- `second = head . tail`
- `drop2 = tail . tail`
Currying/partial application

In Haskell, functions that take multiple arguments are implicitly higher order.

```
plus :: Int -> Int -> Int
```

Curried:
```
plus 2 3
```

Uncurried:
```
plus (2,3)
```

Partial application:
```
increment :: Int -> Int
increment = plus 1
```
Exercises

Is the function \( \text{th} \) well defined?  \( \text{Yes} \)

If so, what does it do and what is its type?  Takes the tail of a list’s head

\[
\begin{align*}
\text{head} & : [a] \rightarrow a \\
\text{tail} & : [a] \rightarrow [a] \\
(\cdot) & : (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c \\
\text{th} & : [[a]] \rightarrow [a] \\
\text{th} & = \text{tail} \cdot \text{head}
\end{align*}
\]
Exercises

Implement revmap using pattern matching

\[
\begin{align*}
\text{map} & :: (a \to b) \to [a] \to [b] \\
\text{map} \ f \ [] & = [] \\
\text{map} \ (x:xs) & = f \ x : \text{map} \ f \ xs \\
\text{reverse} & :: [a] \to [a] \\
\text{reverse} \ [] & = [] \\
\text{reverse} \ (x:xs) & = \text{reverse} \ xs ++ x
\end{align*}
\]

...using function composition

\[
\begin{align*}
(\cdot) & :: (b \to c) \to (a \to b) \to a \to c \\
\text{revmap} & :: (a \to b) \to [a] \to [b] \\
\text{revmap} \ f & = \text{map} \ f \ . \ \text{reverse}
\end{align*}
\]
Outline

Haskell Basics

What is functional programming?
  What is a function?
  Equational reasoning
  First-order vs. higher-order functions
  Lazy evaluation

How to functional program
  Functional programming workflow
  Data types
  Type-directed programming
  Haskell style
  Refactoring (bonus section)
Lazy evaluation

In Haskell expressions are **reduced** (evaluated):

- only when needed
- at most once

```haskell
calculate :: Int -> Int -> Int
calculate a b = if a < 100 then a + a else b
```

**Supports:**

- infinite data structures
- separation of concerns (maybe later)
Outline

Haskell Basics

What is functional programming?
  What is a function?
  Equational reasoning
  First-order vs. higher-order functions
  Lazy evaluation

How to functional program
  Functional programming workflow
  Data types
  Type-directed programming
  Haskell style
  Refactoring (bonus section)
Functional programming workflow

**Warning:** may lead to “obsessive compulsive refactoring disorder”
Functional programming workflow (detailed)

Norman Ramsey, *On Teaching “How to Design Programs”*, ICFP’14
Outline

Haskell Basics

What is functional programming?
  What is a function?
  Equational reasoning
  First-order vs. higher-order functions
  Lazy evaluation

How to functional program
  Functional programming workflow
  Data types
  Type-directed programming
  Haskell style
  Refactoring (bonus section)
Algebraic data types

Data type definition
- introduces a new type of value
- enumerates ways to construct values of this type

Definitions consist of ...
- a type name
- a list of data constructors with argument types

Definition is inductive ...
- the arguments may recursively include the type being defined
- the constructors are the only way to build values of this type

Some data type examples
```haskell
data Bool = True | False

data Nat = Zero | Succ Nat

data Tree = Node Int Tree Tree |
            Leaf Int
```

How to functional program
Anatomy of a data type

```
data Expr = Lit Int  
  | Plus Expr Expr
```

type name

```
Example: 2 + 3 + 4
  Plus (Lit 2) (Plus (Lit 3) (Lit 4))
```

cases

constructor

types of arguments
Type parameters

Specialized lists

type IntList = List Int

| type CharList = List Char

| type RaggedMatrix a = List (List a)

data List a = Nil

| Cons a (List a)

| reference to type parameter

| recursive reference to type

| reference to type parameter
Outline

Haskell Basics

What is functional programming?
  What is a function?
  Equational reasoning
  First-order vs. higher-order functions
  Lazy evaluation

How to functional program
  Functional programming workflow
  Data types
  Type-directed programming
  Haskell style
  Refactoring (bonus section)
Tools for defining functions

Recursion and other functions

\[ \text{sum} :: \text{[Int]} \rightarrow \text{Int} \]
\[ \text{sum} \ \text{xs} = \begin{cases} 0 & \text{if \ null \ xs} \\ \text{head} \ \text{xs} + \text{sum} \ (\text{tail} \ \text{xs}) & \text{else} \end{cases} \]

Pattern matching

\[ \text{sum} :: \text{[Int]} \rightarrow \text{Int} \]
\[ \text{sum} \ [\ ] = 0 \]
\[ \text{sum} \ (x:xs) = x + \text{sum} \ xs \]

1. case analysis

2. decomposition

Higher-order functions

\[ \text{sum} :: \text{[Int]} \rightarrow \text{Int} \]
\[ \text{sum} = \text{foldr} (+) 0 \]

no recursion or variables needed!
What is type-directed programming?

Use the type of a function to help write its body.
Type-directed programming

Basic goal: transform values of **argument types** into **result type**

If argument type is ...
- **atomic type** (e.g. Int, Char)
  - apply functions to it
- **algebraic data type**
  - use pattern matching
    - case analysis
    - decompose into parts
- **function type**
  - apply it to something

If result type is ...
- **atomic type**
  - output of function
- **algebraic data type**
  - build with data constructor
- **function type**
  - function composition or partial application
  - build with lambda abstraction
Outline

Haskell Basics

What is functional programming?
  What is a function?
  Equational reasoning
  First-order vs. higher-order functions
  Lazy evaluation

How to functional program
  Functional programming workflow
  Data types
  Type-directed programming
  Haskell style
  Refactoring (bonus section)
Good Haskell style

Why it matters:
- layout is significant!
- eliminate misconceptions
- we care about elegance

Easy stuff:
- use spaces! (tabs cause errors)
- align patterns and guards
Formatting function applications

Function application...
- is *just a space*
- associates to the left
- binds most strongly

Use parentheses only to *override* this behavior
- \( f \ (g \ x) \)
- \( f \ (x + y) \)
Outline

Haskell Basics

What is functional programming?
  What is a function?
  Equational reasoning
  First-order vs. higher-order functions
  Lazy evaluation

How to functional program
  Functional programming workflow
  Data types
  Type-directed programming
  Haskell style
  Refactoring (bonus section)
Motivations:
- separate concerns
- promote reuse
- promote understandability
- gain insights

“obsessive compulsive refactoring disorder”
Refactoring relations

Semantics-preserving laws prove with equational reasoning and/or induction

- Eta reduction:
  \( x \rightarrow f \ x \equiv f \)
- Map-map fusion:
  \( \text{map } f \ . \ \text{map } g \equiv \text{map } (f \ . \ g) \)
- Fold-map fusion:
  \( \text{foldr } f \ b \ . \ \text{map } g \equiv \text{foldr } (f \ . \ g) \ b \)
Strategy: systematic generalization

Strategy:

** commas :: [String] -> [String] 
commas [] = [] 
commas [x] = [x] 
commas (h:t) = h : "", " : commas t 

Introduce parameters for constants

** seps :: String -> [String] -> [String] 
seps _ [] = [] 
seps _ [x] = [x] 
seps s (h:t) = h : s : seps s t 

Broaden the types

** intersperse :: a -> [a] -> [a] 
intersperse _ [] = [] 
intersperse _ [x] = [x] 
isintersperse s (h:t) = h : s intersperse s t 

Strategy: abstract repeated templates

**abstract** \((v)\): extract and make reusable (as a function)

```haskell
showResult :: Maybe Float -> String
showResult Nothing  = "ERROR"
showResult (Just v) = show v

moveCommand -> Maybe Dir -> Command
moveCommand Nothing  = Stay
moveCommand (Just d) = Move d

safeAdd :: Int -> Maybe Int -> Int
safeAdd x Nothing  = x
safeAdd x (Just y) = x + y
```

Repeated structure:
- pattern match
- default value if *Nothing*
- apply function to contents if *Just*
Strategy: systematic generalization

Describe repeated structure in function

```haskell
maybe :: b -> (a -> b) -> Maybe a -> b
maybe b _ Nothing  = b
maybe _ f (Just a) = f a
```

Reuse in implementations

```haskell
showResult  = maybe “ERROR” show
moveCommand = maybe Stay Move
safeAdd x   = maybe x (x+)
```
Refactoring data types

data Expr = Var Name
  | Add Expr Expr
  | Sub Expr Expr
  | Mul Expr Expr

vars :: Expr -> [Name]
vars (Var x)   = [x]
vars (Add l r) = vars l ++ vars r
vars (Sub l r) = vars l ++ vars r
vars (Mul l r) = vars l ++ vars r

eval :: Env -> Expr -> Int
eval m (Var x)   = get x m
eval m (Add l r) = eval m l + eval m r
eval m (Sub l r) = eval m l - eval m r
eval m (Mul l r) = eval m l * eval m r
Describe repeated structure in function

```haskell
data Expr = Var Name
    | BinOp Op Expr Expr

data Op = Add | Sub | Mul

vars :: Expr -> [Name]
vars (Var x)       = [x]
vars (BinOp _ l r) = vars l ++ vars r

eval :: Env -> Expr -> Int
eval m (Var x)   = get x m
eval m (BinOp o l r) = op o (eval ml) (eval m r)
    where
        op Add = (+)
        op Sub = (-)
        op Mul = (*)
```