Introduction to Functional Programming in Haskell

June 21, 2015
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
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
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
Getting into the Haskell mindset

What is functional programming?

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

Java
int sum (List < Int > xs) {
    int s = 0;
    for (int x : xs) {
        s += x;
    }
    return s;
}
Referential transparency

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

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

what if length was a Java method?

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

Supports equational reasoning
Equational reasoning

Computation is just substitution!

\[
\begin{align*}
\text{sum} :: [\text{Int}] & \rightarrow \text{Int} \\
\text{sum} [\,] & = 0 \\
\text{sum} (x:xs) & = x + \text{sum} \; xs
\end{align*}
\]

\[
\begin{align*}
\text{sum} \; [2,3,4] & \\
\Rightarrow & \quad \text{sum} \; (2:(3:(4:[]))) \\
\Rightarrow & \quad 2 + \text{sum} \; (3:(4:[])) \\
\Rightarrow & \quad 2 + 3 + \text{sum} \; (4[:]) \\
\Rightarrow & \quad 2 + 3 + 4 + \text{sum} \; [] \\
\Rightarrow & \quad 2 + 3 + 4 + 0 \\
\Rightarrow & \quad 9
\end{align*}
\]
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] =
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
First-order functions

What is functional programming?

Examples

\[
\begin{align*}
\cos & : \text{Float} \rightarrow \text{Float} \\
even & : \text{Int} \rightarrow \text{Bool} \\
\text{length} & : [\text{a}] \rightarrow \text{Int}
\end{align*}
\]
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

```haskell
class :: (a -> b) -> [a] -> [b]
c map f [] = []
c map f (x:xs) = f x : map f xs

map f [2,3,4,5] = [f 2, f 3, f 4, f 5]
c map even [2,3,4,5] = [even 2, even 3, even 4, even 5]
  = [True, False, True, False]
```

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

```haskell
c foldr :: (a -> b -> b) -> b -> [a] -> b
c foldr f y [] = y
c foldr f y (x:xs) = f x (foldr f y xs)

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 -> Bool`
- `succ :: Int -> Int`
- `even :: Int -> Bool`
- `head :: [a] -> a`
- `tail :: [a] -> [a]`

**New function definitions**

- `plus2 = succ . succ`
- `odd = not . even`
- `second = head . tail`
- `drop2 = tail . tail`
In Haskell, functions that take multiple arguments are **implicitly higher order**.
Exercises

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

*If so, what does it do and what is its type?*  \( \text{Takes the tail of a list’s head} \)

\[
\begin{align*}
\text{th} &:: \text{??} \\
\text{th} &:: [a] \rightarrow [a] \\
\text{th} &:: [a] \rightarrow [a]
\end{align*}
\]

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

Implement \texttt{revmap} using \textit{pattern matching}

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

\[
\text{reverse} : [a] \rightarrow [a] \\
\text{reverse } [\ ] = [\ ] \\
\text{reverse } (x:xs) = \text{reverse } xs \ ++ \ x
\]

...using \textit{function composition}

\[
(\.) : (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c
\]

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

\[
\text{revmap} : (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
\text{revmap } f = \text{map } f \ . \ \text{reverse}
\]
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
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

Refactor

Define functions

Identify/define types

Warning: may lead to “obsessive compulsive refactoring disorder”
Anatomy of a data type

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

Example: 2 + 3 + 4

```
Plus (Lit 2) (Plus (Lit 3) (Lit 4))
```
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)
Tools for defining functions

Recursion and other functions

```haskell
sum :: [Int] -> Int
sum xs = if null xs then 0
          else head xs + sum (tail xs)
```

Pattern matching

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

1. case analysis

2. decomposition

Higher-order functions

```haskell
sum :: [Int] -> Int
sum = foldr (+) 0
```

no recursion or variables needed!