Introduction to Functional Programming in Haskell

April 2, 2015
Outline

Haskell basics

The essence of 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

The essence of 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 a **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

- output depends on environment
- may perform arbitrary side effects
Outline

Haskell basics

The essence of 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

```
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 = s + x;
    }
    return s;
}
```

In Haskell, “=” means is not change to!
Getting into the Haskell mindset

Quicksort in Haskell

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

Quicksort in C

```
void qsort(int low, int high) {
    int i = low, j = high;
    int pivot = numbers[low + (high-low)/2];
    while (i <= j) {
        while (numbers[i] < pivot) {
            i++;
        }
        while (numbers[j] > pivot) {
            j--;
        }
        if (i <= j) {
            swap(i, j);
            i++;
            j--;
        }
    }
    if (low < j)
        qsort(low, j);
    if (i < high)
        qsort(i, high);
}
```

void swap(int i, int j) {
    int temp = numbers[i];
    numbers[i] = numbers[j];
    numbers[j] = temp;
}
Referential transparency

An expression can be replaced by its value without changing the overall program behavior.

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

what if \text{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!

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

Equations:

```
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
```
Describing computations

**Function definition**: a list of **equations** that relate inputs to output

### Example: reversing a list

<table>
<thead>
<tr>
<th><strong>imperative view</strong></th>
<th>how do I rearrange the elements in the list?</th>
<th>❌</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>functional view</strong></td>
<td>how is a list related to its reversal?</td>
<td>✓</td>
</tr>
</tbody>
</table>

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

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

Haskell basics

The essence of 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

Examples

- \( \text{cos} :: \text{Float} \rightarrow \text{Float} \)
- \( \text{even} :: \text{Int} \rightarrow \text{Bool} \)
- \( \text{length} :: [\text{a}] \rightarrow \text{Int} \)
Higher-order functions

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

\[
\text{map} :: (a \rightarrow b) -> [a] -> [b] \\
\text{map} f \; [] = [] \\
\text{map} f \; (x:xs) = f \; x : \text{map} \; f \; xs \\
\text{map} f \; [2,3,4,5] = [f \; 2, f \; 3, f \; 4, f \; 5] \\
\text{map even} \; [2,3,4,5] = [\text{even} \; 2, \text{even} \; 3, \text{even} \; 4, \text{even} \; 5] \\
\text{map even} \; [2,3,4,5] = [\text{True}, \text{False}, \text{True}, \text{False}]
\]

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

\[
\text{foldr} :: (a->b->b) -> b -> [a] -> b \\
\text{foldr} f \; y \; [] = y \\
\text{foldr} f \; y \; (x:xs) = f \; x \; (\text{foldr} \; f \; y \; xs) \\
\text{foldr} (+) \; 0 \; [2,3,4] = 2 + (3 + (4 + 0)) = 9
\]
Function composition

Can create new functions by **composing** existing functions

- *apply the second function, then apply the first*

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

Types of existing functions

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

Definitions of new functions

- `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

\[ \text{plus} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]

Curried
\[ \text{plus} \ 2 \ 3 \]

Uncurried
\[ \text{plus} \ (2,3) \]

\[ \text{increment} :: \text{Int} \rightarrow \text{Int} \]
\[ \text{increment} = \text{plus} \ 1 \]
Outline

Haskell basics

The essence of 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:

- only when needed
- at most once

Supports:

- infinite data structures
- separation of concerns

```
nats :: [Int]
nats = 1 : map (+1) nats

fact :: Int -> Int
fact n = product (take n nats)

min3 :: [Int] -> [Int]
min3 = take 3 . sort
```

What is the running time of this function?
Outline

Haskell basics

The essence of 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

Refactor

Define functions

Identify/define types

“obsessive compulsive refactoring disorder”
FP workflow (detailed)

7. Review & Refactor

6. Tests

5. Code

4. Function Template

3. Functional Examples

2. Function Description (Signature/Purpose/Header)

1A. Data Description

1B. Data Examples

write body

demands more

guide writing

are also

signature guides template

names used in signature

overlooked cases

inputs

validated by

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

Haskell basics

The essence of 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 new type of value
- enumerates ways to construct values of this type

Some example data types
```
data Bool = True | False
data Nat = Zero | Succ Nat
data Tree = Node Int Tree Tree | Leaf Int
```

Definitions consists 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
Anatomy of a data type definition

data Expr = Lit Int
      | Plus Expr Expr

Example: 2 + 3 + 4   Plus (Lit 2) (Plus (Lit 3) (Lit 4))
Type parameters

Like generics in Java

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

reference to type parameter
recursive reference to type

Specialized lists

type IntList = List Int
type CharList = List Char
type RaggedMatrix a = List (List a)
Outline

Haskell basics

The essence of 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

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

(1) case analysis

Pattern matching

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

(2) decomposition

Higher-order functions

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

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

Use the **type** of a function to help write its **body**
Basic goal: transform values of **argument types** into **result type**

<table>
<thead>
<tr>
<th>If argument type is …</th>
<th>If result type is …</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>atomic type</strong> (e.g. Int, Char)</td>
<td><strong>atomic type</strong></td>
</tr>
<tr>
<td>- apply functions to it</td>
<td>- output of another function</td>
</tr>
<tr>
<td><strong>algebraic data type</strong></td>
<td><strong>algebraic data type</strong></td>
</tr>
<tr>
<td>- use pattern matching</td>
<td>- build with data constructor</td>
</tr>
<tr>
<td>- case analysis</td>
<td>- function composition or partial application</td>
</tr>
<tr>
<td>- decompose into parts</td>
<td>- build with lambda abstraction</td>
</tr>
<tr>
<td><strong>function type</strong></td>
<td><strong>function type</strong></td>
</tr>
<tr>
<td>- apply it to something</td>
<td>- function composition or partial application</td>
</tr>
<tr>
<td></td>
<td>- build with lambda abstraction</td>
</tr>
</tbody>
</table>
Outline

Haskell basics

The essence of 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 layout errors)
- align patterns and guards

See style guides on course web page
Function application:
- is *just a space*
- associates to the left
- binds most strongly

\[
\begin{align*}
  f(x) & \quad \text{f x} \\
  (f \ x) \ y & \quad f \ x \ y \\
  (f \ x) + (g \ y) & \quad f \ x + g \ y
\end{align*}
\]

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

Haskell basics

The essence of 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)
Refactoring in the FP workflow

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:
  \[ \text{map } f \cdot \text{map } g \equiv \text{map } (f \cdot g) \]

- Fold–map fusion:
  \[ \text{foldr } f \ b \cdot \text{map } g \equiv \text{foldr } (f \cdot g) \ b \]

“Algebra of computer programs”

John Backus, *Can Programming be Liberated from the von Neumann Style?*, ACM Turing Award Lecture, 1978
Strategy: systematic generalization

commas :: [String] -> [String]
commas [] = []
commas [x] = [x]
commas (x:xs) = x : ", " : commas xs

Introduce parameters for constants
seps :: String -> [String] -> [String]
seps _ [] = []
seps _ [x] = [x]
seps s (x:xs) = x : s : seps s xs

Broaden the types
intersperse :: a -> [a] -> [a]intersperse _ [] = []intersperse _ [x] = [x]
intersperse s (x:xs) = x : s : intersperse s xs

intersperse = commas
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: abstract repeated templates

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
Refactoring data types

Factor out shared structure

data Expr = Var Name
        \| BinOp Op Expr Expr

data Op = Add \| Sub \| Mul

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

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