Introduction to Functional Programming in Haskell

September 29, 2015
Outline

The essence of functional programming
  What is a function?
  Equational reasoning
  First-order vs. higher-order functions

How to functional program
  Functional programming workflow
  Data types
  Type-directed programming
  Haskell style

Modularity and reuse
  Refactoring
  Type classes
  Lazy evaluation
Outline

The essence of functional programming
  What is a function?
    Equational reasoning
    First-order vs. higher-order functions

How to functional program
  Functional programming workflow
  Data types
  Type-directed programming
  Haskell style

Modularity and reuse
  Refactoring
  Type classes
  Lazy evaluation
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

The essence of functional programming
  What is a function?
  Equational reasoning
    First-order vs. higher-order functions

How to functional program
  Functional programming workflow
  Data types
  Type-directed programming
  Haskell style

Modularity and reuse
  Refactoring
  Type classes
  Lazy evaluation
Getting into the Haskell mindset

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

Quicksort in Haskell

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

Quicksort in C

```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 `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

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>Imperative view</th>
<th>how do I rearrange the elements in the list?</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional view</td>
<td>how is a list related to its reversal?</td>
<td>✓</td>
</tr>
</tbody>
</table>

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

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

The essence of functional programming
- What is a function?
- Equational reasoning
- First-order vs. higher-order functions

How to functional program
- Functional programming workflow
- Data types
- Type-directed programming
- Haskell style

Modularity and reuse
- Refactoring
- Type classes
- Lazy evaluation
First-order functions

Examples
- `cos :: Float -> Float`
- `even :: Int -> Bool`
- `length :: [a] -> 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

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

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

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

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

```haskell
foldr f y [2,3,4] = \(f 2 \ (f 3 \ (f 4 \ y))\)
foldr (+) 0 [2,3,4] = (+) 2 ((+) 3 ((+) 4 0))
= 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 \circ succ`
- `odd = not \circ even`
- `second = head \circ tail`
- `drop2 = tail \circ tail`
Currying / partial application

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

```
plus :: Int -> Int -> Int
increment :: Int -> Int
increment = plus 1
```

Curried

```
plus :: Int -> Int -> Int
plus 2 3
```

Uncurried

```
plus :: (Int,Int) -> Int
plus (2,3)
```

A pair of ints
Outline

The essence of functional programming
  What is a function?
  Equational reasoning
  First-order vs. higher-order functions

How to functional program
  Functional programming workflow
  Data types
  Type-directed programming
  Haskell style

Modularity and reuse
  Refactoring
  Type classes
  Lazy evaluation
FP workflow (simple)

Refactor

Define functions

Identify/define types

“obsessive compulsive refactoring disorder”
FP workflow (detailed)

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

The essence of functional programming
  What is a function?
  Equational reasoning
  First-order vs. higher-order functions

How to functional program
  Functional programming workflow
  Data types
  Type-directed programming
  Haskell style

Modularity and reuse
  Refactoring
  Type classes
  Lazy evaluation
Algebraic data types

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

Definitions consists of …
- a type name
- a list of data constructors with argument types

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

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

Example: $2 + 3 + 4 \Rightarrow \text{Plus (Lit 2) (Plus (Lit 3) (Lit 4))}$
Type parameters

Like generics in Java

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

Specialized lists

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

The essence of functional programming
  What is a function?
  Equational reasoning
  First-order vs. higher-order functions

How to functional program
  Functional programming workflow
  Data types
  Type-directed programming
  Haskell style

Modularity and reuse
  Refactoring
  Type classes
  Lazy evaluation
Tools for defining functions

Recursion and other functions

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

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**

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

The essence of functional programming
   What is a function?
   Equational reasoning
   First-order vs. higher-order functions

How to functional program
   Functional programming workflow
   Data types
   Type-directed programming
      Haskell style

Modularity and reuse
   Refactoring
   Type classes
   Lazy evaluation
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
Formatting function applications

Function application:

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

\[ f(x) \]
\[ (f \ x) \ y \]
\[ (f \ x) + (g \ y) \]

Use parentheses only to *override* this behavior:

- \( f (g \ x) \)
- \( f (x + y) \)
Outline

The essence of functional programming
  What is a function?
  Equational reasoning
  First-order vs. higher-order functions

How to functional program
  Functional programming workflow
  Data types
  Type-directed programming
  Haskell style

Modularity and reuse
  Refactoring
  Type classes
  Lazy evaluation
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:
  \[ \text{map } f \ . \ \text{map } g \equiv \text{map } (f \ . \ g) \]

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

“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

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

**intersperse** :: a -> [a] -> [a]
- intersperse _ [] = []
- intersperse _ [x] = [x]
- intersperse s (x:xs) = x : s : intersperse s xs

Introduce parameters for constants

Broaden the types

Modularity and reuse
Strategy: abstract repeated templates

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

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

```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 m l) (eval m r)
  where
    op Add = (+)
    op Sub = (-)
    op Mul = (*)
```
Outline

The essence of functional programming
  What is a function?
  Equational reasoning
  First-order vs. higher-order functions

How to functional program
  Functional programming workflow
  Data types
  Type-directed programming
  Haskell style

Modularity and reuse
  Refactoring
  Type classes
  Lazy evaluation
What is a type class?

1. an **interface** that is supported by many different types
2. a **set of types** that have a common behavior

```
class Eq a where
    (==) :: a -> a -> Bool

class Show a where
    show :: a -> String

class Num a where
    (+) :: a -> a -> a
    (*) :: a -> a -> a
    negate :: a -> a
    ...
```

types whose values can be compared for equality

types whose values can be shown as strings

types whose values can be manipulated like numbers
Type constraints

- List elements can be of any type
  
  \[
  \text{length} :: [a] \to \text{Int} \\
  \text{length} \; [] = 0 \\
  \text{length} \; (_:xs) = 1 + \text{length} \; xs
  \]

- List elements must support equality!
  
  \[
  \text{elem} :: \text{Eq} \; a \Rightarrow a \to [a] \to \text{Bool} \\
  \text{elem} \; _ \; [] = \text{False} \\
  \text{elem} \; y \; (x:xs) = x == y \; \text{||} \; \text{elem} \; y \; xs
  \]

- Use method ⇒ add type class constraint

\[
\text{class Eq} \; a \text{ where} \\
(==) :: a \to a \to \text{Bool}
\]
Outline

The essence of functional programming
  What is a function?
  Equational reasoning
  First-order vs. higher-order functions

How to functional program
  Functional programming workflow
  Data types
  Type-directed programming
  Haskell style

Modularity and reuse
  Refactoring
  Type classes
  Lazy evaluation
Lazy evaluation

In Haskell, expressions are reduced:

- only when needed
- at most once

Supports:

- infinite data structures
- separation of concerns

```haskell
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?