Introduction to Functional Programming in Haskell
Outline

Why learn functional programming?

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

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

Type inference
Outline

Why learn functional programming?

The essence of functional programming

How to functional program

Type inference
Why learn (pure) functional programming?

1. This course: strong correspondence of core concepts to PL theory
   - **abstract syntax** can be represented by **algebraic data types**
   - **denotational semantics** can be represented by **functions**

2. It will make you a better (imperative) programmer
   - forces you to think **recursively** and **compositionally**
   - forces you to **minimize use of state**
     
     ...essential skills for solving **big** problems

3. It is the future!
   - more scalable and parallelizable (MapReduce)
   - functional features have been added to most mainstream languages
   - many cool new libraries built around functional paradigm
Outline

Why learn functional programming?

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

How to functional program

Type inference
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

Why learn functional programming?

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

How to functional program

Type inference
Getting into the Haskell mindset

In Haskell, “=” means is not change to!

**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;
}
```
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);
    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!

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

\[
\text{sum \[2,3,4\]}
\]
\[
\Rightarrow \text{sum (2:(3:(4:[])))}
\]
\[
\Rightarrow 2 + \text{sum (3:(4:[]))}
\]
\[
\Rightarrow 2 + 3 + \text{sum (4:[])}
\]
\[
\Rightarrow 2 + 3 + 4 + \text{sum []}
\]
\[
\Rightarrow 2 + 3 + 4 + 0
\]
\[
\Rightarrow 9
\]
Describing computations

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

- matched top-to-bottom
- applied left-to-right

Example: reversing a list

**Imperative view**: how do I rearrange the elements in the list? ✗

**Functional view**: how is a list related to its reversal? ✓

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

Why learn functional programming?

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

How to functional program

Type inference
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

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

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

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

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

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

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

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

Function composition

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

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

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

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

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

Haskell Curry

a pair of ints
Outline

Why learn functional programming?

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

How to functional program

Type inference
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

Why learn functional programming?

The essence of functional programming

How to functional program

Haskell style

Functional programming workflow
Data types
Type classes
Type-directed programming
Refactoring (bonus section)

Type inference
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

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

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

Why learn functional programming?

The essence of functional programming

How to functional program

Haskell style

Functional programming workflow

Data types

Type classes

Type-directed programming

Refactoring (bonus section)

Type inference
FP workflow (simple)

“obsessive compulsive refactoring disorder”
FP workflow (detailed)

1. Data Description
   - names used in signature
   - validated by

2. Function Description
   - Signature/Purpose/Heading
   - signature guides template

3. Functional Examples
   - overlooked cases
   - inputs

4. Function Template
   - demands more
   - guide writing
   - write body

5. Code
   - are also

6. Tests

7. Review & Refactor

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

Why learn functional programming?

The essence of functional programming

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

Type inference
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

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

Example: $2 + 3 + 4 \quad \text{Plus (Lit 2) (Plus (Lit 3) (Lit 4))}$
FP data types vs. OO classes

Haskell
\[
data \text{ Tree } = \text{ Node } \text{ Int } \text{ Tree } \text{ Tree} \\
| \text{ Leaf }
\]

- separation of type- and value-level
- set of cases closed
- set of operations open

Java
\[
\text{abstract class Tree } \{ \ldots \}
\text{class Node extends Tree } \{
\text{int label;}
\text{Tree left, right;}
\ldots
\}
\text{class Leaf extends Tree } \{ \ldots \}
\]

- merger of type- and value-level
- set of cases open
- set of operations closed

Extensibility of cases vs. operations = the “expression problem”
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

Why learn functional programming?

The essence of functional programming

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

Type inference
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

```haskell
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

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

List elements must support equality!

```haskell
elem :: Eq a => a -> [a] -> Bool
elem _    []    = False
elem y (x:xs) = x == y || elem y xs
```

use method ⇒ add type class constraint

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

Why learn functional programming?

The essence of functional programming

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

Type inference
Tools for defining functions

Recursion and other functions

\[
\text{sum} :: [\text{Int}] \to \text{Int} \\
\text{sum}\,\text{xs} = \begin{cases} 
0 & \text{null}\,\text{xs} \\
\text{head}\,\text{xs} + \text{sum}\,\text{tail}\,\text{xs} & \text{otherwise}
\end{cases}
\]

Pattern matching

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

Higher-order functions

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

(1) case analysis

(2) decomposition

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><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 type</td>
</tr>
<tr>
<td>• decompose into parts</td>
<td>• function composition or partial application</td>
</tr>
<tr>
<td><strong>function type</strong></td>
<td>• build with lambda abstraction</td>
</tr>
<tr>
<td>• apply it to something</td>
<td></td>
</tr>
</tbody>
</table>
Outline

Why learn functional programming?

The essence of functional programming

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

Type inference
Refactoring in the FP workflow

Motivations:
- separate concerns
- promote reuse
- promote understandability
- gain insights

“obsessive compulsive refactoring disorder”
Refactoring relations

Semantics-preserving laws can prove with equational reasoning + 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`

**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`
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

```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 = (*)
```

How to functional program
Outline

Why learn functional programming?

The essence of functional programming

How to functional program

Type inference
How to perform type inference (bottom-up strategy)

For each literal, data constructor, and named function: write down the type.

Repeat until you know the type of the whole expression:

1. find an application $e_1 \ e_2$ where you know $e_1 :: T_1$ and $e_2 :: T_2$

2. $T_1$ should be a function type $T_1 = T_{arg} \rightarrow T_{res}$

3. check that the argument type is what the function expects: $T_{arg} \equiv T_2 \rightsquigarrow \sigma$
   - this step is called **type unification**
   - $\sigma$ is an assignment of the type variables to make the two sides equal

4. write down $e_1 \ e_2 :: \sigma T_{res}$ \quad ($\sigma T_{res} = T_{res}$ with type variables substituted)

If any of these steps fails, it is a **type error**!
Find a type variable assignment ($\sigma$) that makes the two sides equal

Unification by example

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$\equiv$</th>
<th>$T_2$</th>
<th>$\rightsquigarrow$</th>
<th>$\sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>$\equiv$</td>
<td>Int</td>
<td>$\rightsquigarrow$</td>
<td>${a = \text{Int}}$</td>
</tr>
<tr>
<td>Bool</td>
<td>$\equiv$</td>
<td>a</td>
<td>$\rightsquigarrow$</td>
<td>${a = \text{Bool}}$</td>
</tr>
<tr>
<td>a</td>
<td>$\equiv$</td>
<td>b</td>
<td>$\rightsquigarrow$</td>
<td>${a = b}$</td>
</tr>
<tr>
<td>Int</td>
<td>$\equiv$</td>
<td>Bool</td>
<td>$\rightsquigarrow$</td>
<td>Fail!</td>
</tr>
<tr>
<td>a</td>
<td>$\equiv$</td>
<td>Int -&gt; Bool</td>
<td>$\rightsquigarrow$</td>
<td>${a = \text{Int} \to \text{Bool}}$</td>
</tr>
<tr>
<td>a -&gt; Bool</td>
<td>$\equiv$</td>
<td>Int -&gt; Bool</td>
<td>$\rightsquigarrow$</td>
<td>${a = \text{Int}}$</td>
</tr>
</tbody>
</table>
Find a **type variable assignment** ($\sigma$) that makes the two sides **equal**

### Unification by example

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$\equiv$</th>
<th>$T_2$</th>
<th>$\leadsto$</th>
<th>$\sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a \to b$</td>
<td>$\equiv$</td>
<td>$\text{Int} \to \text{Bool}$</td>
<td>$\leadsto$</td>
<td>${a = \text{Int}, b = \text{Bool}}$</td>
</tr>
<tr>
<td>$\text{Int} \to a$</td>
<td>$\equiv$</td>
<td>$b \to \text{Bool}$</td>
<td>$\leadsto$</td>
<td>${a = \text{Bool}, b = \text{Int}}$</td>
</tr>
<tr>
<td>$a \to b$</td>
<td>$\equiv$</td>
<td>$\text{Int}$</td>
<td>$\leadsto$</td>
<td>Fail!</td>
</tr>
<tr>
<td>$a \to b$</td>
<td>$\equiv$</td>
<td>$\text{Int} \to \text{Bool} \to \text{Char}$</td>
<td>$\leadsto$</td>
<td>${a = \text{Int}, b = \text{Bool} \to \text{Char}}$</td>
</tr>
<tr>
<td>$a \to b$</td>
<td>$\equiv$</td>
<td>$(\text{Int} \to \text{Bool}) \to \text{Char}$</td>
<td>$\leadsto$</td>
<td>${a = \text{Int} \to \text{Bool}, b = \text{Char}}$</td>
</tr>
<tr>
<td>$(a \to b) \to c$</td>
<td>$\equiv$</td>
<td>$\text{Int} \to \text{Bool} \to \text{Char}$</td>
<td>$\leadsto$</td>
<td>Fail!</td>
</tr>
</tbody>
</table>
Exercises

Given

data Maybe a = Nothing | Just a

\(\text{gt} :: \text{Int} \to \text{Int} \to \text{Bool}\)
\(\text{not} :: \text{Bool} \to \text{Bool}\)
\(\text{map} :: (a \to b) \to [a] \to [b]\)
\(\text{even} :: \text{Int} \to \text{Bool}\)
\(\text{(.)} :: (b \to c) \to (a \to b) \to a \to c\)

1. Just
2. not even 3
3. not (even 3)
4. not . even
5. even . not
6. map (Just . even)