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
   Functional programming workflow
   Data types
   Type-directed programming
   Haskell style

Refactoring and reuse
   Refactoring
   Type classes

Type inference
Outline

Why learn functional programming?

The essence of functional programming

How to functional program

Refactoring and reuse

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
   - functional features have been added to most mainstream languages
Quotes from pragmatists

Programming is not about data. It's about transforming data … That's why I think functional programming is a natural successor to object-oriented programming. In OOP, we're constantly concerned about the state of our data. In functional programming, our focus is on the transformation of data. And transformation is where the value is added.

—Dave Thomas, The Pragmatic Programmer
Quotes from pragmatists

"Without understanding functional programming, you can’t invent MapReduce, the algorithm that makes Google so massively scalable."

—Joel Spolsky, Joel on Software
So that’s the big deal about functional languages; and it is one big fricking deal. There is a freight train barreling down the tracks towards us, with multi-core emblazoned on it; and you’d better be ready by the time it gets here.

—Robert C. Martin, a.k.a. “Uncle Bob”
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

Refactoring and reuse

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

Refactoring and reuse

Type inference
Getting into the Haskell mindset

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

Quicksort in Haskell

```haskell
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort (filter (\leq 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!

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

\[
\begin{align*}
\text{sum} \ [2,3,4] & \Rightarrow \text{sum} \ (2:(3:(4:[]))) \\
& \Rightarrow 2 + \text{sum} \ (3:(4:[])) \\
& \Rightarrow 2 + 3 + \text{sum} \ (4:[]) \\
& \Rightarrow 2 + 3 + 4 + \text{sum} \ [] \\
& \Rightarrow 2 + 3 + 4 + 0 \\
& \Rightarrow 9
\end{align*}
\]
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]
```
Exercise

1. Evaluate: \( \text{double (succ (double 3))} \)
   
   - A: 12
   - B: 14
   - C: 16

2. Prove, up to associativity of (+), using equational reasoning:

   \[
   \text{double (succ x)} = \text{succ (succ (double x))}
   \]
Exercise

1. Evaluate: $\text{foo (foo 0 0) 0}$
   
   - A: 0
   - B: 2
   - C: 3

2. Evaluate: $\text{foo 0 (foo 0 0)}$
   
   - A: 0
   - B: 2
   - C: 3

foo :: Int -> Int -> Int
foo 0 x = x + 1
foo x 0 = x + 2
foo x y = x + y
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

Refactoring and reuse

Type inference
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

\[
\text{map} :: (a \to b) \to [a] \to [b]
\]

- \(\text{map} f \; [] = []\)
- \(\text{map} f \; (x:xs) = f \; x : \text{map} \; f \; xs\)

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

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

\[
\text{foldr} :: (a \to b \to b) \to b \to [a] \to b
\]

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

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

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

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

Function composition

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

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

**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

\[
\text{plus :: Int -> Int -> Int}
\]

\[
\text{increment :: Int -> Int}
\]

\[
\text{increment = plus 1}
\]

Haskell Curry

Curried

\[
\text{plus 2 3}
\]

\[
\text{plus :: Int -> Int -> Int}
\]

Uncurried

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

\[
\text{plus :: (Int,Int) -> Int}
\]

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

Refactoring and reuse

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

Functional programming workflow

Data types
Type-directed programming
Haskell style

Refactoring and reuse

Type inference
FP workflow (simple)

“obsessive compulsive refactoring disorder”
FP workflow (detailed)

1A. Data Description
1B. Data Examples

2. Function Description (Signature/Purpose/Header)

3. Functional Examples

4. Function Template

5. Code

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
   Functional programming workflow
      Data types
         Type-directed programming
      Haskell style

Refactoring and reuse

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

```haskell
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:** $2 + 3 + 4 \rightarrow \text{Plus (Lit 2) (Plus (Lit 3) (Lit 4))}$
FP data types vs. OO classes

Haskell

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

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

Java

```
abstract class Tree {
    ...
}
class Node extends Tree {
    int label;
    Tree left, right;
    ...
}
class Leaf extends Tree {
    ...
}
```

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

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

(type parameters)

\[
data \text{ List } a = \text{ Nil} \\
| \text{ Cons } a \ (\text{List } a)
\]

(Like generics in Java)

reference to type parameter

recursive reference to type

Specialized lists

type \text{IntList} = \text{List Int} \\
type \text{CharList} = \text{List Char} \\
type \text{RaggedMatrix } a = \text{List } (\text{List } a)

How to functional program
Outline

Why learn functional programming?

The essence of functional programming

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

Refactoring and reuse

Type inference
Tools for defining functions

Recursion and other functions

\[
\text{sum} :: [\text{Int}] \rightarrow \text{Int} \\
\text{sum} \hspace{1em} \text{xs} = \begin{cases} 
0 & \text{if } \text{null} \hspace{1em} \text{xs} \\
\text{head} \hspace{1em} \text{xs} + \text{sum} \hspace{1em} (\text{tail} \hspace{1em} \text{xs}) & \text{else}
\end{cases}
\]

Pattern matching

\[
\text{sum} :: [\text{Int}] \rightarrow \text{Int} \\
\text{sum} \hspace{1em} [] = 0 \\
\text{sum} \hspace{1em} (x:xs) = x + \text{sum} \hspace{1em} xs
\]

(1) case analysis

(2) decomposition

Higher-order functions

\[
\text{sum} :: [\text{Int}] \rightarrow \text{Int} \\
\text{sum} = \text{foldr} \hspace{1em} (+) \hspace{1em} 0
\]

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 another function

- **algebraic data type**
  - build with data constructor

- **function type**
  - function composition or partial application
  - build with lambda abstraction
Outline

Why learn functional programming?

The essence of functional programming

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

Refactoring and reuse

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
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

Why learn functional programming?

The essence of functional programming

How to functional program

Refactoring and reuse
  Refactoring
  Type classes

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 prove with equational reasoning and/or induction

- Eta reduction:
  \[ \lambda x \to f \ x \equiv f \]

- 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

intersperse = commas
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]
varys (Var x) = [x]
varys (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

Why learn functional programming?

The essence of functional programming

How to functional program

Refactoring and reuse
  Refactoring
    Type classes

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

Refactoring and reuse

Type inference
**How to perform type inference**

If a literal, data constructor, or named function: write down the type – you’re done!

Otherwise:

1. identify the top-level application $e_1 \, e_2$
2. recursively infer their types $e_1 : T_1$ and $e_2 : T_2$
3. $T_1$ should be a function type $T_1 = T_{\text{arg}} \to T_{\text{res}}$
4. unify $T_{\text{arg}} = ? T_2$, yielding type variable assignment $\sigma$
5. return $e_1 \, e_2 : \sigma T_{\text{res}}$  ($T_{\text{res}}$ with type variables substituted)

If any of these steps fails, it is a **type error**!

Example: `map even`
Exercises

Given

```haskell
data Maybe a = Nothing | Just a

gt :: Int -> Int -> Bool  not :: Bool -> Bool
map :: (a -> b) -> [a] -> [b]  even :: Int -> Bool
(.) :: (b -> c) -> (a -> b) -> a -> c
```

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