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 classes
  Type-directed programming
  Haskell style
  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!
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.

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

**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**

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

```
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

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

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

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

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

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

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

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

- \(\text{not} :: \text{Bool} \rightarrow \text{Bool}\)
- \(\text{succ} :: \text{Int} \rightarrow \text{Int}\)
- \(\text{even} :: \text{Int} \rightarrow \text{Bool}\)
- \(\text{head} :: [a] \rightarrow a\)
- \(\text{tail} :: [a] \rightarrow [a]\)

Definitions of new functions

- \(\text{plus2} = \text{succ} \circ \text{succ}\)
- \(\text{odd} = \text{not} \circ \text{even}\)
- \(\text{second} = \text{head} \circ \text{tail}\)
- \(\text{drop2} = \text{tail} \circ \text{tail}\)
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

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

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

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

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

```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?
Outline

Why learn functional programming?

The essence of functional programming

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

Type inference
FP workflow (simple)

Refactor → Define functions → Identify/define types

“obsessive compulsive refactoring disorder”
FP workflow (detailed)

1A. Data Description

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 classes
    Type-directed programming
    Haskell style
    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

\[
\begin{align*}
\text{data } & \text{Bool } = \text{True } \mid \text{False} \\
\text{data } & \text{Nat } = \text{Zero } \mid \text{Succ Nat} \\
\text{data } & \text{Tree } = \text{Node Int Tree Tree} \\
& \quad \mid \text{Leaf Int}
\end{align*}
\]

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: \(2 + 3 + 4\) \(\text{Plus} \ (\text{Lit} \ 2) \ (\text{Plus} \ (\text{Lit} \ 3) \ (\text{Lit} \ 4))\)
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

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

(Like generics in Java)

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

  Functional programming workflow
  Data types
    Type classes
  Type-directed programming
  Haskell style
  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

```
class Eq a where
  (==) :: a -> a -> Bool
  types whose values can be compared for equality

class Show a where
  show :: a -> String
  types whose values can be shown as strings

class Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  types whose values can be manipulated like numbers
```
Type constraints

List elements can be of any type

\[
\text{length :: } [a] \rightarrow \text{Int}  \\
\text{length } [] = 0  \\
\text{length } (_:xs) = 1 + \text{length } xs
\]

List elements must support equality!

\[
\text{elem :: } \text{Eq } a \Rightarrow a \rightarrow [a] \rightarrow \text{Bool}  \\
\text{elem } _ [] = \text{False}  \\
\text{elem } y (x:xs) = x == y || \text{elem } y xs
\]

use method ⇒ add type class constraint
Outline

Why learn functional programming?

The essence of functional programming

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

Type inference
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)
```

(1) case analysis

Pattern matching

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

(2) decomposition

Higher-order functions

```haskell
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**
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 classes
  Type-directed programming
  Haskell style
  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
Function application:

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

```
(f x)  \→  f \ x
(f \ x) \ y  \→  f \ x \ y
(f \ x) + (g \ y)  \→  f \ x + g \ y
```

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
  Functional programming workflow
  Data types
  Type classes
  Type-directed programming
  Haskell style
  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 prove with equational reasoning and/or induction

- Eta reduction:
  \( \lambda x \rightarrow f \; x \equiv f \)

- 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]**

```haskell
commas [] = []
commas [x] = [x]
commas (x:xs) = x : ", " : commas xs
```

**Introduce parameters for constants**

**seps :: String -> [String] -> [String]**

```haskell
seps _ [] = []
seps _ [x] = [x]
seps s (x:xs) = x : s : seps s xs
```

**Broaden the types**

**intersperse :: a -> [a] -> [a]**

```haskell
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]
varys (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

defvars :: Expr -> [Name]
defvars (Var x) = [x]
defvars (BinOp _ l r) = defvars l ++ defvars r

defeval :: Env -> Expr -> Int
defeval m (Var x) = get x m
defeval m (BinOp o l r) = op o (defeval m l) (defeval m r)
where
doop Add = (+)
doop Sub = (-)
doop Mul = (*)
Outline

Why learn functional programming?

The essence of functional programming

How to functional program

Type inference
How to perform type inference

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

Otherwise:

1. pick an 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 $T_1 = T_{\text{arg}} \rightarrow T_{\text{res}}$
4. unify $T_{\text{arg}} = T_2$, yielding type variable assignment $\sigma$
5. return $e_1 \ e_2 : \sigma T_{\text{res}}$ (with type variables substituted)

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

Given

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)