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 (MapReduce)
   - functional features have been added to most mainstream languages
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
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!

**Haskell**

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

**Java**

```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);
  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.

```
length [1, 2, 3] + 4
⇒ 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
```

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

Evaluate:

1. `double (succ (double 3))`

2. `(double . succ) 3`

3. `(succ . double) 3`
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

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

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

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

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

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

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

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

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

Curried
\[
\text{plus} \quad 2 \quad 3
\]
\[ \text{plus} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]

Uncurried
\[
\text{plus} \quad (2,3)
\]
\[ \text{plus} :: (\text{Int},\text{Int}) \rightarrow \text{Int} \]

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

\[ \text{a pair of ints} \]
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

```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-directed programming
      Haskell style

Refactoring and reuse

Type inference
FP workflow (simple)

“obsessive compulsive refactoring disorder”
FP workflow (detailed)

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

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

```haskell
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**

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

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

**Java**

```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

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

Refactoring and reuse

Type inference
Tools for defining functions

Recursion and other functions

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

(1) case analysis

Pattern matching

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

(2) decomposition

Higher-order functions

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

no recursion or variables needed!

How to functional program
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
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

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:
  \( \lambda x \rightarrow f \ x \equiv f \)

- 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
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

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>class Eq a where</code></td>
<td><code>==</code> :: <code>a</code> -&gt; <code>a</code> -&gt; <code>Bool</code> types whose values can be compared for equality</td>
</tr>
<tr>
<td><code>class Show a where</code></td>
<td><code>show</code> :: <code>a</code> -&gt; <code>String</code> types whose values can be shown as strings</td>
</tr>
<tr>
<td><code>class Num a where</code></td>
<td><code>+</code> :: <code>a</code> -&gt; <code>a</code> -&gt; <code>a</code> types whose values can be manipulated like numbers</td>
</tr>
<tr>
<td></td>
<td><code>*</code> :: <code>a</code> -&gt; <code>a</code> -&gt; <code>a</code></td>
</tr>
<tr>
<td></td>
<td><code>negate</code> :: <code>a</code> -&gt; <code>a</code></td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Refactoring and reuse 43 / 47
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*
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 $T_1 = T_{\text{arg}} \rightarrow T_{\text{res}}$
4. unify $T_{\text{arg}} =? T_2$, yielding type variable assignment $\sigma$
5. return $e_1 \ e_2 : \sigma T_{\text{res}}$ (\textit{T}_{\text{res}} with type variables substituted)

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

Example: \texttt{map even}
Exercises

Given

data Maybe a = Nothing | Just a

\gt \quad :: \quad \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}

\text{not} \quad :: \quad \text{Bool} \rightarrow \text{Bool}

\text{map} \quad :: \quad (\text{a} \rightarrow \text{b}) \rightarrow \left[ \text{a} \right] \rightarrow \left[ \text{b} \right]

\text{even} \quad :: \quad \text{Int} \rightarrow \text{Bool}

(.) \quad :: \quad (\text{b} \rightarrow \text{c}) \rightarrow (\text{a} \rightarrow \text{b}) \rightarrow \text{a} \rightarrow \text{c}

1. \quad \text{Just}

2. \quad \text{not even 3}

3. \quad \text{not (even 3)}

4. \quad \text{not . even}

5. \quad \text{even . not}

6. \quad \text{map (Just . even)}