Comp 311
Functional Programming

Eric Allen, PhD
Vice President, Engineering
Two Sigma Investments, LLC
Announcements

• Homework 2 Available from Piazza (Due October 1)
• Two Sigma Info Session at Huff House, 4pm Today
Traversing Multiple Recursive Datatypes
Taking the First Few Elements

def take(n: Nat, xs: List): List = {
    // require n <= size(xs)
    (n,xs) match {
        case (Zero, xs) => Empty
        case (Next(m), Cons(y, ys)) => Cons(y, take(m, ys))
    }
}
def take(n: Int, xs: List): List = {
    require ((n >= 0) && (n <= size(xs)))
    (n,xs) match {
        case (0, xs) => Empty
        case (n, Cons(y, ys)) => Cons(y, take(n-1, ys))
    }
}
Dropping the First Few Elements

```scala
def drop(n: Int, xs: List): List = {
  require (n <= size(xs))
  (n, xs) match {
    case (0, xs) => xs
    case (n, Cons(y, ys)) => drop(n-1, ys)
  }
}
```
Functional Update of a List

```scala
def update(xs: List, i: Nat, y: Int): List = {
  require (xs != Empty)  // && i < size(xs)

  (xs, i) match {
    case (Cons(z, zs), Zero) => Cons(y, zs)
    case (Cons(z, zs), Next(j)) => Cons(z, update(zs,j,y))
  }
}
```
def update(xs: List, i: Int, y: Int): List = {
  require ((i >= 0) && (i < size(xs)))
  assert (xs != Empty)

  (xs, i) match {
    case (Cons(z, zs), 0) => Cons(y, zs)
    case (Cons(z, zs), _) => Cons(z, update(zs, i-1, y))
  }
}
Design Abstraction
Our Function Templates Reveal Common Structure

def containsZero(xs: List): Boolean = {
  xs match {
    case Empty => false
    case Cons(n, ys) => (n == 0) || containsZero(ys)
  }
}

def containsOne(xs: List): Boolean = {
  xs match {
    case Empty => false
    case Cons(n, ys) => (n == 1) || containsOne(ys)
  }
}
def contains(m: Int, xs: List): Boolean = {
    xs match {
        case Empty => false
        case Cons(n, ys) => (n == m) || contains(m, ys)
    }
}
But Sometimes the Part We Want to Abstract Is a Function

def below(m: Int, xs: List): List = {
  xs match {
    case Empty => Empty
    case Cons(n, ys) => {
      if (n < m) Cons(n, below(m, ys))
      else below(m, ys)
    }
  }
}
But Sometimes the Part We Want to Abstract Is a Function

```scala
def above(m: Int, xs: List): List = {
  xs match {
    case Empty => Empty
    case Cons(n, ys) => {
      if (n > m) Cons(n, above(m, ys))
      else above(m, ys)
    }
  }
}
```
Taking Functions As Parameters

def filter(f: (Int) => Boolean, xs: List): List = {
  xs match {
    case Empty => Empty
    case Cons(n, ys) => {
      if (f(n)) Cons(n, filter(f, ys))
      else filter(f, ys)
    }
  }
}
Passing Functions as Arguments

\[
\text{val } xs = \text{Cons}(1, \text{Cons}(2, \text{Cons}(3, \text{Cons}(4, \text{Cons}(5, \text{Cons}(6, \text{Empty})))))
\]

\[
\text{filter}((\text{n: Int}) \Rightarrow (\text{n} > 0)), \, xs \mapsto*
\text{Cons}(1, \text{Cons}(2, \text{Cons}(3, \text{Cons}(4, \text{Cons}(5, \text{Cons}(6, \text{Empty})))))
\]

\[
\text{filter}((\text{n: Int}) \Rightarrow (\text{n} < 0)), \, xs \mapsto*
\text{Empty}
\]

\[
\text{filter}((\text{n: Int}) \Rightarrow (\text{n} < 3)), \, xs \mapsto*
\text{Cons}(1, \text{Cons}(2, \text{Empty}))
\]
Passing Functions as Arguments

val xs = Cons(1,Cons(2,Cons(3,Cons(4,Cons(5,Cons(6,Empty)))))))

filter(((n: Int) => (n > 0)), xs) ↦ *
Cons(1,Cons(2,Cons(3,Cons(4,Cons(5,Cons(6,Empty)))))))

filter(((n: Int) => (n < 0)), xs) ↦ *
Empty

filter(((n: Int) => (n < 3)), xs) ↦ *
Cons(1,Cons(2,Empty))

These are function literals
First-Class Functions

• Function literals are expressions with static arrow types that reduce to function values

• The value type of a function value is also an arrow type

• Function values are first-class values:
  • They are allowed to be passed as arguments
  • They are allowed to be returned as results
Simplifying Function Literals

• Parameter types on function literals are allowed to be elided whenever the types are clear from context

\[
\text{filter}(((n: \text{Int}) \Rightarrow (n > 0)), \text{xs})
\]

can be written as

\[
\text{filter}(((n) \Rightarrow (n > 0)), \text{xs})
\]
Simplifying Function Literals

- Parentheses around a single parameter is allowed to be omitted

\[
\text{filter}(((n) => (n > 0)), \text{xs})
\]

can be written as

\[
\text{filter}(n => (n > 0), \text{xs})
\]
Simplifying Function Literals

• When a single parameter is used only once in the body of a function literal:
  • We can drop the parameter list
  • We simply write the body with an _ at the place where the parameter is used

For example,

\(((x: \text{Int}) \Rightarrow (x < 0))\)

becomes

\(_ < 0\)
Passing Functions as Arguments

\[ \text{val } xs = \text{Cons}(1,\text{Cons}(2,\text{Cons}(3,\text{Cons}(4,\text{Cons}(5,\text{Cons}(6,\text{Empty})))))) \]

\[ \text{filter}(_ < 3, xs) \mapsto* \text{Cons}(1,\text{Cons}(2,\text{Empty})) \]
Mapping a Computation Over a List

def double(xs: List) = {
  xs match {
    case Empty => Empty
    case Cons(y, ys) => Cons(y * y, double(ys))
  }
}
We Might Express a Similar Computation Mathematically as a Comprehension

\{2x \mid x \in xs\}
Mapping a Computation Over a List

def negate(xs: List) = {
  xs match {
    case Empty => Empty
    case Cons(y, ys) => (-y, negate(ys))
  }
}
Negation as a Comprehension

\[\{-x \mid x \in xs\}\]
Mapping a Computation Over a List

def map(f: Int => Int, xs: List) = {
  xs match {
    case Empty => Empty
    case Cons(y,ys) => Cons(f(y), map(f,ys))
  }
}
Mapping a Computation Over a List

\begin{verbatim}
val xs = Cons(1,Cons(2,Cons(3,Cons(4,Cons(5,Cons(6,Empty))))))
negate(xs) ↦*
Cons(-1,Cons(-2,Cons(-3,Cons(-4,Cons(-5,Cons(-6,Empty))))))

double(xs) ↦*
Cons(1,Cons(4,Cons(9,Cons(16,Cons(25,Cons(36,Empty))))))
\end{verbatim}
Mapping a Computation Over a List

```
val xs = Cons(1,Cons(2,Cons(3,Cons(4,Cons(5,Cons(6,Empty))))))

map(_ , xs) →*
Cons(-1,Cons(-2,Cons(-3,Cons(-4,Cons(-5,Cons(-6,Empty))))))

map(x => x * x , xs) →*
Cons(1,Cons(4,Cons(9,Cons(16,Cons(25,Cons(36,Empty))))))
```
Recall Our Sum Function Over Lists

def sum(xs: List): Int = {
    xs match {
        case Empty => 0
        case Cons(y, ys) => y + sum(ys)
    }
}
In Mathematics, We Might Write a Summation

\[ \sum_{n \in xs} n \]
And Our Product Function Over Lists

def product(xs: List): Int = {
    xs match {
        case Empty => 1
        case Cons(y, ys) => y * sum(ys)
    }
}
In Mathematics, We Might Write a Summation

$$\prod_{n \in xs} n$$
We Abstract to a Reduction Function Over Lists

def reduce(base: Int, f: (Int, Int) => Int, xs: List): Int = {
    xs match {
        case Empty => base
        case Cons(y, ys) => f(y, reduce(base, f, ys))
    }
}
Example Reductions

\[
\text{val } \text{xs }= \text{Cons}(1,\text{Cons}(2,\text{Cons}(3,\text{Cons}(4,\text{Cons}(5,\text{Cons}(6,\text{Empty}))))))
\]

\[
\begin{align*}
\text{reduce}(0, (x,y) \Rightarrow x + y, \text{xs}) &\Rightarrow* 21 \\
\text{reduce}(1, (x,y) \Rightarrow x \times y, \text{xs}) &\Rightarrow* 720
\end{align*}
\]
Simplifying Function Literals

• When *each* parameter is used only once in the body of a function literal, and in the order in which they are passed:
  
  • We can drop the parameter list
  
  • We simply write the body with an _ at the place where each parameter is used

  For example,

  $$(((x: \text{Int}, y: \text{Int}) \Rightarrow (x + y)))$$

  becomes

  _  +  _
Example Reductions

val xs = Cons(1,Cons(2,Cons(3,Cons(4,Cons(5,Cons(6,Empty))))))

\[
\begin{align*}
\text{reduce}(0, \_+\_, \text{x}s) & \mapsto 21 \\
\text{reduce}(1, \_*\_, \text{x}s) & \mapsto 720
\end{align*}
\]

Note the multiple parameters
Combinations of Maps and Reductions

\[ \sum_{n \in xs} n^2 + 1 \]
Combinations of Maps and Reductions

\[
\text{reduce}(0, \_+\_, \text{map}(x\mapsto x^2 + 1, \text{xs})) \mapsto 97
\]
def square(x: Int) = x * x

def summation(f: Int => Int, xs: List) = reduce(0, _+_, map(f, xs))
Summation

\[\text{summation}(\text{square}(\_)+1, \text{xs}) \mapsto 97\]
More Syntactic Sugar

• Functions defined with `def` can be passed as arguments whenever an expression of a compatible function type is expected.

• What constitutes a compatible function type?
Partially Applied Functions

- If we want to pass a function as an argument, but supply some of the arguments to the function overselves, we can wrap an application to the function in a function literal:

```
map(x=>x + 1, xs)
```
Partially Applied Functions

- If we want to pass a function as an argument, but supply some of the arguments to the function overselves, we can wrap an application to the function in a function literal:

  \[
  \text{map}(x \mapsto x + 1, \text{x}s)
  \]

  which is equivalent to

  \[
  \text{map}(\_ + 1, \text{x}s)
  \]
Partially Applied Functions

• **Eta Expansion:** Wrapping a function in function literal that takes all of the arguments of f and immediately calls f with those arguments

\[(x: \text{Int}) \Rightarrow \text{square}(x)\]

is equivalent to

\[\text{square} \]
Mapping a Computation Over a List

We can use eta expansion to pass operators as arguments:

\[ \text{map}(x \mapsto -x, \ xs) \]
Mapping a Computation Over a List

We can use eta expansion to pass operators as arguments:

```
map(_, xs)
```
Recommended Viewing

• Guy L. Steele Jr., “Growing a Language”:

  https://www.youtube.com/watch?v=_ahvzDzKdB0