Datatypes and Higher-order functions

Owen Arden
UC Santa Cruz

Based on course materials developed by Nadia Polikarpova

Representing complex data

- We've seen:
  - base types: `Bool`, `Int`, `Integer`, `Float`
  - some ways to build up types: given types `T1`, `T2`
    - functions: `T1` -> `T2`
    - tuples: `(T1, T2)`
    - lists: `[T1]`
- Algebraic Data Types: a single, powerful technique for building up types to represent complex data
  - lets you define your own data types
  - subsumes tuples and lists!

Product types

- Tuples can do the job but there are two problems...

```plaintext
deadlineDate :: (Int, Int, Int)
deadlineDate = (2, 4, 2019)

deadlineTime :: (Int, Int, Int)
deadlineTime = (11, 59, 59)

-- | Deadline date extended by one day
extension :: (Int, Int, Int) -> (Int, Int, Int)
estension = ...
```
- Can you spot them?
1. Verbose and unreadable

```haskell
type Date = (Int, Int, Int)
type Time = (Int, Int, Int)
deadlineDate :: Date
deadlineDate = (2, 4, 2019)
deadlineTime :: Time
deadlineTime = (11, 59, 59)
```

-- | Deadline date extended by one day
extension :: Date -> Date
extension = ...

2. Unsafe

- We want this to fail at compile time!!!
  extension deadlineTime

- Solution: construct two different datatypes
  ```haskell
data Date = Date Int Int Int
data Time = Time Int Int Int
```

```haskell
-- constructor" "parameter types

deadlineDate :: Date
deadlineDate = Date 2 4 2019
deadlineTime :: Time
deadlineTime = Time 11 59 59
```

Record Syntax

- Haskell’s record syntax allows you to name the constructor parameters:
  ```haskell
  data Date = Date Int Int Int
  ```

  ```haskell
  You can write:
  ```
  ```haskell
data Date = Date {
    month :: Int,
    day :: Int,
    year :: Int
  }
  ```
  ```haskell
deadlineDate = Date 2 4 2019
deadlineMonth = month deadlineDate
  ```
Building data types

- Three key ways to build complex types/values:
  1. **Product types** *(each-of)*: a value of T contains a value of T1 and a value of T2 [done]
  2. **Sum types** *(one-of)*: a value of T contains a value of T1 or a value of T2
  3. **Recursive types**: a value of T contains a sub-value of the same type Ts

Example: NanoMD

- Suppose I want to represent a text document with simple markup. Each paragraph is either:
  - plain text (String)
  - heading: level and text (Int and String)
  - list: ordered? and items (Bool and [String])
- I want to store all paragraphs in a list

```haskell
doc = [(1, "Notes from 130") -- Lvl 1 heading
      , "There are two types of languages:" -- Plain text
      , (1, ["purely functional", "purely evil"])
      ] -- But this doesn't type check!!
```

Sum Types

- Solution: construct a new type for paragraphs that is a sum (one-of) the three options!
  - plain text (String)
  - heading: level and text (Int and String)
  - list: ordered? and items (Bool and [String])
- I want to store all paragraphs in a list

```haskell
data Paragraph =
  Text String  -- 3 constructors,
  Heading Int String  -- each with different
  List Bool [String]  -- parameters
```
Constructing datatypes

```haskell
data T =
  C1 T11 .. T1k
| C2 T21 .. T2l
| ..
| Cn Tn1 .. Tnm
```

T is the new datatype

C1 .. Cn are the constructors of T

A value of type T is

- either C1 v1 .. vk with vi :: T1i
- or C2 v1 .. vl with vi :: T2l
- or ...
- or Cn v1 .. vm with vi :: Tnm

Apply a constructor = pack some values into a box (and label it)

- Text "Hey there!"
  - put "Hey there!" in a box labeled Text
- Heading 1 "Introduction"
  - put 1 and "Introduction" in a box labeled Heading
- Boxes have different labels but same type (Paragraph)

You can think of a T value as a box:

- either a box labeled C1 with values of types T11 .. T1k inside
- or a box labeled C2 with values of types T21 .. T2l inside
- or ...
- or a box labeled Cn with values of types Tn1 .. Tnm inside

You can think of a T value as a box:

- either a box labeled C1 with values of types T11 .. T1k inside
- or a box labeled C2 with values of types T21 .. T2l inside
- or ...
- or a box labeled Cn with values of types Tn1 .. Tnm inside

You can think of a T value as a box:

- either a box labeled C1 with values of types T11 .. T1k inside
- or a box labeled C2 with values of types T21 .. T2l inside
- or ...
- or a box labeled Cn with values of types Tn1 .. Tnm inside

Constructing datatypes

```haskell
data Paragraph =
  Text String | Heading Int String | List Bool [String]
```

What would GHCi say to

```haskell
> T Text "Hey there!"
```

- A. Syntax error
- B. Type error
- C. Paragraph
- D. [Paragraph]
- E. [String]
Example: NanoMD

```haskell
data Paragraph =
    Text String |
    Heading Int String |
    List Bool [String]
```

Now I can create a document like so:

```haskell
doc :: [Paragraph]
doc = [Heading 1 "Notes from 130",
      Text "There are two types of languages:"
      , List True ["purely functional", "purely evil"]
]
```

---

Example: NanoMD

Now I want to convert documents into HTML.

I need to write a function:

```haskell
html :: Paragraph -> String
html p = ???
```

-- depends on the kind of paragraph!

How to tell what’s in the box?
- Look at the label!

---

Pattern Matching

Pattern matching = looking at the label and extracting values from the box
- we’ve seen it before
- but now for arbitrary datatypes

```haskell
html :: Paragraph -> String
html (Text str) = ...
-- It's a plain text! Get string
html (Heading lvl str) = ...
-- It's a heading! Get level and string
html (List ord items) = ...
-- It's a list? Get ordered and items
```
Dangers of pattern matching (1)

```haskell
html :: Paragraph -> String
html (Text str) = ...
html (List ord items) = ...
```

What would GHCi say to:

```haskell
html (Heading 1 "Introduction")
```

Answer: Runtime error (no matching pattern)

---

Beware of missing and overlapped patterns

- GHC warns you about overlapped patterns
- GHC warns you about missing patterns when called with `-W` (use `:set -W` in GHCi)

---

Pattern matching expression

We've seen: pattern matching in equations

You can also pattern-match inside your program using the `case` expression:

```haskell
html :: Paragraph -> String
html p =
    case p of
        Text str -> unlines [open "p", str, close "p"]
        Heading lvl str -> ...
        List ord items -> ...
```
**Quiz**

What is the type of:

```haskell
let p = Text "Hey there!"
in case p of
  Text str -> str
  Heading lvl_1 -> lvl
  List ord_ _ -> ord
```

- A. Syntax error
- B. Type error
- C. String
- D. Paragraph
- E. Paragraph -> String

---

**Pattern matching expression: typing**

The `case` expression

```haskell
case e of
  pattern1 -> e1
  pattern2 -> e2
  ...
  patternN -> eN
```

has type `T` if

- each `e1...eN` has type `T`
- `e` has some type `D`
- each `pattern1...patternN` is a valid pattern for `D`

  - I.e. a variable or a constructor of `D` applied to other patterns

The expression `e` is called the `match scrutinee`.

---

**Building data types**

- Three key ways to build complex types/values:
  1. **Product types (each-of):** a value of `T` contains a value of `T1 and a value of T2** [done]
  2. **Sum types (one-of):** a value of `T` contains a value of `T1 or a value of T2** [done]
  3. **Recursive types:** a value of `T` contains a sub-value of the same type `Ts`
Recursive types

Let’s define natural numbers from scratch:

\[
\text{data } \text{Nat} = ???
\]

Recursive types

\[
\text{data } \text{Nat} = \text{Zero} \mid \text{Succ } \text{Nat}
\]

A Nat value is:

- either an empty box labeled \text{Zero}
- or a box labeled \text{Succ} with another \text{Nat} in it!

Some Nat values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{Zero}</td>
<td>0</td>
</tr>
<tr>
<td>\text{Succ } \text{Zero}</td>
<td>1</td>
</tr>
<tr>
<td>\text{Succ } (\text{Succ } \text{Zero})</td>
<td>2</td>
</tr>
<tr>
<td>\text{Succ } (\text{Succ } (\text{Succ } \text{Zero}))</td>
<td>3</td>
</tr>
</tbody>
</table>

Functions on recursive types

Principle: Recursive code mirrors recursive data
1. Recursive type as a parameter

```haskell
data Nat = Zero  -- base constructor
         | Succ Nat -- inductive constructor
```

Step 1: add a pattern per constructor

```haskell
toInt :: Nat -> Int
toInt Zero = ...  -- base case
toInt (Succ n) = ... -- inductive case
                -- (recursive call goes here)
```

Step 2: fill in base case

```haskell
toInt :: Nat -> Int
toInt Zero = 0  -- base case
toInt (Succ n) = ... -- inductive case
                -- (recursive call goes here)
```

Step 3: fill in inductive case using a recursive call:

```haskell
toInt :: Nat -> Int
toInt Zero = 0  -- base case
toInt (Succ n) = 1 + toItem n -- inductive case
```
QUIZ

What does this evaluate to?

```haskell
let foo i = if i <= 0 then Zero else Succ (foo (i - 1))
in foo 2
```

- A. Syntax error
- B. Type error
- C. 2
- D. Succ Zero
- E. Succ (Succ Zero)

2. Recursive type as a result

```haskell
data Nat = Zero -- base constructor | Succ Nat -- inductive constructor

fromInt :: Int -> Nat
fromInt n
  | n <= 0 = Zero -- base case
  | otherwise = Succ (fromInt (n - 1)) -- inductive case
```

2. Putting the two together

```haskell
data Nat = Zero -- base constructor | Succ Nat -- inductive constructor

add :: Nat -> Nat -> Nat
add Zero m = m -- base case
add (Succ n) m = Succ (add n m) -- inductive case

sub :: Nat -> Nat -> Nat
sub n Zero = n -- base case 1
sub Zero _ = Zero -- base case 2
sub (Succ n) (Succ m) = sub n (m) -- inductive case
```
2. Putting the two together

data Nat = Zero | Succ Nat

add :: Nat -> Nat -> Nat
add Zero m = m
add (Succ n) m = Succ (add n m)

sub :: Nat -> Nat -> Nat
sub Zero _ = Zero
sub Zero _ = Zero
sub (Succ n) (Succ m) = sub n m

Lessons learned:

- Recursive code mirrors recursive data
- With multiple arguments of a recursive type, which one should I recurse on?
- The name of the game is to pick the right inductive strategy!

Lists

Lists aren’t built-in! They are an algebraic data type like any other:

data List = Nil | Cons Int List

- List [1, 2, 3] is represented as Cons 1 (Cons 2 (Cons 3 Nil))
- Built-in list constructors [] and : are just fancy syntax for Nil and Cons

Functions on lists follow the same general strategy:

length :: List -> Int
length Nil = 0
length (Cons _ xs) = 1 + length xs

Lists

What is the right inductive strategy for appending two lists?

append :: List -> List -> List
append ??? ??? = ???
Lists

What is the right inductive strategy for appending two lists?

\[
\text{append} :: \text{List} \rightarrow \text{List} \rightarrow \text{List}
\]

append \(\text{Nil} \ y\) = \(y\)

append \(\text{Cons} \ x \ \text{xs} \ y\) = \(\text{Cons} \ x \ (\text{append} \ \text{xs} \ \text{ys})\)

Recursion is…

Building solutions for big problems from solutions for sub-problems

- **Base case**: what is the simplest version of this problem and how do I solve it?
- **Inductive strategy**: how do I break down this problem into sub-problems?
- **Inductive case**: how do I solve the problem given the solutions for subproblems?
- But it can get kinda repetitive!
Example: evens

Let’s write a function evens:

```haskell
-- evens [] ==> []
-- evens [1,2,3,4] ==> [2,4]
evens :: [Int] -> [Int]
evens [] = ...
evens (x:xs) = ...
```

Example: four-letter words

Let’s write a function fourChars:

```haskell
-- fourChars [] ==> []
-- fourChars ["i","must","do","work"] ==> ["must","work"]
fourChars :: [String] -> [String]
fourChars [] = ...
fourChars (x:xs) = ...
```

Yikes, Most Code is the Same!

```haskell
foo [] = []
foo (x:xs) |
  x mod 2 == 0 = x : foo xs
  otherwise = foo xs

foo [] = []
foo (x:xs) |
  length x == 4 = x : foo xs
  otherwise = foo xs
```

Only difference is condition

* x mod 2 == 0 vs length x == 4
Moral of the day

D.R.Y. Don’t Repeat Yourself!

Can we
• reuse the general pattern and
• substitute in the custom condition?

HOFs to the rescue!

General Pattern
• expressed as a higher-order function
• takes customizable operations as arguments

Specific Operation
• passed in as an argument to the HOF

The “filter” pattern

Use the filter pattern to avoid duplicating code!
The “filter” pattern

General Pattern
- HOF filter
- Recursively traverse list and pick out elements that satisfy a predicate

Specific Operation
- Predicates isEven and isFour

```
filter f [] = []
filter f (x:xs) = f x : filter f xs
```

```
evens = filter isEven
where
  isEven = x `mod` 2 == 0

fourChars = filter isFour
where
  isFour = length x == 4
```

Let’s talk about types

```
evens :: [1,2,3,4] ==> [2,4]
evens :: [Int] -> [Int]
evens xs = filter isEven xs
  where
    isEven :: Int -> Bool
    isEven x = x `mod` 2 == 0

filter :: ???
```
Let’s talk about types

-- fourChars ["i","must","do","work"] ==> ["must","work"]
fourChars :: [String] -> [String]
fourChars xs = filter isFour xs
  where
    isFour :: String -> Bool
    isFour x = length x == 4
filter :: ???

Let’s talk about types

Uh oh! So what’s the type of filter?

filter :: (Int -> Bool) -> [Int] -> [Int] -- ???
filter :: (String -> Bool) -> [String] -> [String] -- ???

- It does not care what the list elements are
- as long as the predicate can handle them
- It’s type is polymorphic (generic) in the type of list elements
  -- For any type ‘a’
  -- if you give me a predicate on ‘a’s
  -- and a list of ‘a’s,
  -- I’ll give you back a list of ‘a’s
filter :: (a -> Bool) -> [a] -> [a]

Example: all caps

Let’s write a function shout:

-- shout [] ==> []
-- shout ['h','e','l','l','o'] ==> ['H','E','L','L','O']
shout :: [Char] -> [Char]
shout [] = ...
shout (x:xs) = ...
Example: squares

Let's write a function squares:

```haskell
-- squares [] ==> []
-- squares [1,2,3,4] ==> [1,4,9,16]
squares :: [Int] -> [Int]
squares [] = ... squares (x:xs) = ...
```

Yikes, Most Code is the Same!

Let's rename the functions to foo:

```haskell
-- shout
foo [] = []
foo (x:xs) = toUpper x : foo xs

-- squares
foo [] = []
foo (x:xs) = (x * x) : foo xs

Let's refactor into the common pattern

pattern = ...
```

The “map” pattern

| shout [] = [] | squares [] = [] |
| ---=--- | ---=--- |
| shout (x:xs) = toUpper x : shout xs | squares (x:xs) = (x*x) : squares xs |

The map Pattern

General Pattern
- HOF map
- Apply a transformation f to each element of a list

Specific Operations
- Transformations toUpper and \x \rightarrow x * x
The “map” pattern

map f [] = []
map f (x:xs) = f x : map f xs

Let's refactor shout and squares

shout = map ... 
squares = map ...

The “map” pattern

-- For any types `a` and `b`
-- if you give me a transformation from `a` to `b`
-- and a list of `a`'s,
-- I'll give you back a list of `b`'s
map :: (a -> b) -> [a] -> [b]

Type says it all!

- The only meaningful thing a function of this type can do is apply its first argument to elements of the list (Hoogle it!)

Things to try at home:

- can you write a function map' :: (a -> b) -> [a] -> [b] whose behavior is different from map?
- can you write a function map'' :: (a -> b) -> [a] -> [b] such that map'' f xs returns a list whose elements are not in map f xs?

QUIZ

What is the type of map?

map f [] = []
map f (x:xs) = f x : map f xs

(A) (Char -> Char) -> [Char] -> [Char]
(B) (Int -> Int) -> [Int] -> [Int]
(C) (a -> a) -> [a] -> [a]
(D) (a -> b) -> [a] -> [b]
(E) (a -> b) -> [c] -> [d]
**Don’t Repeat Yourself**

Benefits of factoring code with HOFs:

- Reuse iteration pattern
  - think in terms of standard patterns
  - less to write
  - easier to communicate

- Avoid bugs due to repetition

---

**Recall: length of a list**

```haskell
-- len [] ==> 0
-- len ['carne', 'asada'] ==> 2
len :: [a] -> Int
len [] = 0
len (x:xs) = 1 + len xs
```

---

**Recall: summing a list**

```haskell
-- sum [] ==> 0
-- sum [1,2,3] ==> 6
sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + sum xs
```
Example: string concatenation

Let's write a function `cat`:

```haskell
-- cat [] ==> ""
-- cat ["carne","asada","torta"] ==> "carneasadatorta"
cat :: [String] -> String
cat [] = ...
cat (x:xs) = ...
```

Can you spot the pattern?

```haskell
-- len
len [] = 0
len (x:xs) = 1 + len xs

-- sum
sum [] = 0
sum (x:xs) = x + sum xs

-- cat
cat [] = ""
cat (x:xs) = x ++ cat xs

pattern = ...
```

The “fold-right” pattern

<table>
<thead>
<tr>
<th>len</th>
<th>sum</th>
<th>cat</th>
</tr>
</thead>
</table>
| len [] = 0 | sum [] = 0 | cat [] = ""
| len (x:xs) = 1 + len xs | sum (x:xs) = x + sum xs | cat (x:xs) = x ++ sum xs

`foldr f b [] = b`
`foldr f b (x:xs) = f x (foldr f b xs)`

The `foldr` Pattern

- Recurse on tail
- Combine result with the head using some binary operation
The “fold-right” pattern

foldr f b [] = b
foldr f b (x:xs) = f x (foldr f b xs)

Let’s refactor sum, len and cat:

sep = foldr ... ...
len = foldr ... ...

Factor the recursion out!

You can write it more clearly as:

sum = foldr (+) 0
cat = foldr (++) ""

You can write it more clearly as:

len = foldr (\x n -> 1 + n) 0
sum = foldr (\x n -> x + n) 0
cat = foldr (\x s -> x ++ n) ""
QUIZ
What does this evaluate to?*

foldr f b [] = b
foldr f b (x:xs) = f x (foldr f b xs)

quiz = foldr (:) [] [1,2,3]

- (A) Type error
- (B) 1,2,3
- (C) 3,2,1
- (D) 1,3,2
- (E) 1,2,3

The “fold-right” pattern

foldr f b [] = b
foldr f b (x:xs) = f x (foldr f b xs)

foldr (:) [] [1,2,3]

=> (:) 2 (foldr (:) [] [2,3])
=> (:) 1 (:) 2 (foldr (:) [] [3])
=> (:) 1 (:) 2 (:) 3 (foldr (:) [] []))
=> (:) 1 (:) 2 (:) 3 []
=> 1 : (2 : (3 : []))
=> [1,2,3]

The “fold-right” pattern

fold f b [x1, x2, x3, x4]

=> f x1 (foldr f b [x2, x3, x4])
=> f x1 (f x2 (foldr f b [x3, x4]))
=> f x1 (f x2 (f x3 (foldr f b [x4])))
=> f x1 (f x2 (f x3 (f x4 (foldr f b []))))
=> f x1 (f x2 (f x3 (f x4 b)))

Accumulate the values from the right

For example:

foldr (+) 0 [1, 2, 3, 4]

=> 1 + (foldr (+) 0 [2, 3, 4])
=> 1 + (2 + (foldr (+) 0 [3, 4]))
=> 1 + (2 + (3 + (foldr (+) 0 [])))
=> 1 + (2 + (3 + (4 + 0)))
Tail recursion

Recursive call is the top-most sub-expression in the function body
- i.e. no computations allowed on recursively returned value
- i.e. value returned by the recursive call == value returned by function

The “fold-right” pattern

Is foldr tail recursive?
Answer: No! It calls the binary operations on the results of the recursive call

What about tail-recursive versions?

Let's write tail-recursive sum!

```haskell
sumTR :: [Int] -> Int
sumTR = ...
```
What about tail-recursive versions?

Let's write tail-recursive `sum`!

```haskell
sumTR :: [Int] -> Int
sumTR xs = helper @ xs
  where
    helper acc [] = acc
    helper acc (x:xs) = helper (acc + x) xs
```

What about tail-recursive versions?

Let's run `sumTR` to see how it works.

```
sumTR [1,2,3] => helper @ [1,2,3] => helper 1 [2,3] => 1 + 1 => 2
sumTR [1,2,3] => helper 2 [3] => 3 + 2 => 5
sumTR [1,2,3] => helper 3 [3] => 3 + 3 => 6
sumTR [1,2,3] => helper 6 [] => 6
```

Note: `helper` directly returns the result of recursive call!

What about tail-recursive versions?

Let's write tail-recursive `cat`!

```haskell
catTR :: [String] -> String
catTR = ...
```
What about tail-recursive versions?

Let's write tail-recursive cat!

\[\text{catTR} :: [\text{String}] \rightarrow \text{String} \]
\[
\text{catTR } x s = \text{helper } "" x s \\
\text{where} \\
\quad \text{helper acc } [] = \text{acc} \\
\quad \text{helper acc } (x:xs) = \text{helper } (\text{acc } ++ x) x s
\]

Let's run \text{catTR} to see how it works.

\[
\text{catTR } "[\text{carne}, \text{asada}, \text{torta}]" \rightarrow "\text{carneasadatorta}"
\]

Note: \text{helper} directly returns the result of recursive call!

Can you spot the pattern?

\[-- \text{sumTR} \]
\[
\text{foo xs} = \text{helper } @ x s \\
\text{where} \\
\quad \text{helper acc } [] = \text{acc} \\
\quad \text{helper acc } (x:xs) = \text{helper } (\text{acc } + x) x s
\]

\[-- \text{catTR} \]
\[
\text{foo xs} = \text{helper } "" x s \\
\text{where} \\
\quad \text{helper acc } [] = \text{acc} \\
\quad \text{helper acc } (x:xs) = \text{helper } (\text{acc } ++ x) x s
\]

pattern = ...
The “fold-left” pattern

General Pattern
- Use a helper function with an extra accumulator argument
- To compute new accumulator, combine current accumulator with the head using some binary operation

The foldl Pattern

```
foldl f b xs = helper b xs
where
  helper acc [] = acc
  helper acc (x:xs) = helper (f acc x) xs
```

Let's refactor sumTR and catTR:

```
sumTR = foldl ...

catTR = foldl ...
```

Factor the tail-recursion out!

QUIZ

What does this evaluate to?*

```
foldl f b xs = helper b xs
where
  helper acc [] = acc
  helper acc (x:xs) = helper (f acc x) xs

quiz = foldl (
x : -> x : xs [] [] 1,2,3
```

- (A) Type error
- (B) [1,2,3]
- (C) [2,3,4]
- (D) [1,2,3,4,5]
- (E) [1,3,5]
The “fold-left” pattern

\[
\text{foldl } f \ b \ [x_1, x_2, x_3, x_4] \\
\Rightarrow \text{helper } b \ [x_2, x_3, x_4] \\
\Rightarrow \text{helper } (f \ b) \ x_2 \ [x_3, x_4] \\
\Rightarrow \text{helper } (f \ (f \ b \ x_2)) \ x_3 \ [x_4] \\
\Rightarrow \text{helper } (f \ (f \ (f \ b \ x_2)) \ x_3) \ x_4 \ [] \\
\Rightarrow (f \ (f \ (f \ (f \ b \ x_2)) \ x_2) \ x_3) \ x_4)
\]

Accumulate the values from the left.

For example:

\[
\text{foldl } (+) \ 0 \ [1, 2, 3, 4] \\
\Rightarrow \text{helper } 0 \ [2, 3, 4] \\
\Rightarrow \text{helper } ((0 + 1) + 2) \ [3, 4] \\
\Rightarrow \text{helper } (((0 + 1) + 2) + 3) \ [4] \\
\Rightarrow (((0 + 1) + 2) + 3) + 4 \ []
\]

Left vs. Right

\[
\text{foldl } f \ b \ [x_1, x_2, x_3] \Rightarrow f \ (f \ (f \ b \ x_2) \ x_3) \ x_4 \ -- \ \text{left}
\]
\[
\text{foldr } f \ b \ [x_1, x_2, x_3] \Rightarrow f \ x_1 \ (f \ x_2 \ (f \ x_3 \ b)) \ -- \ \text{Right}
\]

For example:

\[
\text{foldl } (+) \ 0 \ [1, 2, 3] \Rightarrow ((0 + 1) + 2) + 3 \ -- \ \text{left}
\]
\[
\text{foldr } (+) \ 0 \ [1, 2, 3] \Rightarrow 1 + (2 + (3 + 0)) \ -- \ \text{Right}
\]

Different types!

\[
\text{foldl} :: (b \rightarrow a \rightarrow b) \rightarrow a \rightarrow [b] \rightarrow b \ -- \ \text{Left}
\]
\[
\text{foldr} :: (a \rightarrow b \rightarrow a) \rightarrow b \rightarrow [a] \rightarrow b \ -- \ \text{Right}
\]

Useful HOF: flip

-- you can write
\[
\text{foldl } (\backslash x s x \rightarrow x : x s) [] [1,2,3]
\]

-- more concisely like so:
\[
\text{foldl } \text{flip } (\backslash []) [1,2,3]
\]

What is the type of flip?

\[
\text{flip} :: (a \rightarrow b \rightarrow c) \rightarrow b \rightarrow a \rightarrow c
\]
Useful HOF: compose

```haskell
-- you can write
map \(\lambda x \to f \circ g \, x\) \, ys

-- more concisely like so:
map (f . g) \, ys

What is the type of \((.\)\)?

\((.\) :: (b \to c) \to (a \to b) \to a \to c
```

Higher Order Functions

Iteration patterns over collections:
- **Filter** values in a collection given a *predicate*
- **Map** (iterate) a given *transformation* over a collection
- **Fold** (reduce) a collection into a value, given a *binary operation* to combine results

Useful helper HOFs:
- **Flip** the order of function's (first two) arguments
- **Compose** two functions

Higher Order Functions

HOFs can be put into libraries to enable modularity
- Data structure library implements map, filter, fold for its collections
  - generic efficient implementation
  - generic optimizations: map \(f \circ (g \, x\s) \to \map (f . g) \, x\s
- Data structure clients use HOFs with specific operations
  - no need to know the implementation of the collection

Enabled the “big data” revolution e.g. *MapReduce, Spark*
That's all folks!