Informatics 1

Functional Programming Lecture 7

Function properties

Philip Wadler

University of Edinburgh

Part I

Fold, right and left

Fold right

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f u []        =  u
foldr f u (x:xs)    =  x `f` (foldr f v xs)

  foldr (++) "" ["abc","def","ghi","jkl"]
=
  "abc" ++ foldr (++) "" ["def","ghi","jkl"]
=
  "abc" ++ ("def" ++ foldr (++) "" ["ghi","jkl"])
=
  "abc" ++ ("def" ++ ("ghi" ++ foldr (++) "" ["jkl"]))
=
  "abc" ++ ("def" ++ ("ghi" ++ ("jkl" ++ foldr (++) "" [])))
=
  "abc" ++ ("def" ++ ("ghi" ++ ("jkl" ++ "")))
```

Fold left

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f u []       =  u
foldl f u (x:xs)   =  foldl f (u `f` x) xs

  foldl (++) "" ["abc","def","ghi","jkl"]
=
  foldl (++) ("" ++ "abc") ["def","ghi","jkl"]
=
  foldl (++) (("" ++ "abc") ++ "def") ["ghi","jkl"]
=
  foldl (++) ((("" ++ "abc") ++ "def") ++ "ghi") ["jkl"]
=
  foldl (++) (((("" ++ "abc") ++ "def") ++ "ghi") ++ "jkl") []
=
  ((("" ++ "abc") ++ "def") ++ "ghi") ++ "jkl"
```

Fold right, non-empty list

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f [x]      =  x
foldr1 f (x:xs)   =  x `f` (foldr1 f xs)

  foldr1 (`max`) [3, 1, 4, 2]
=
  foldr1 (`max`) (3 : (1 : (4 : (2 : []))))
=
  3 `max` foldr1 (`max`) (1 : (4 : (2 : [])))
=
  3 `max` (1 `max` foldr1 (`max`) (4 : (2 : [])))
=
  3 `max` (1 `max` (4 `max` foldr1 (`max`) (2 : [])))
=
  3 `max` (1 `max` (4 `max` 2))
```

Fold left, non-empty list

```
foldl1 :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs)  =  foldl f x xs

  foldl1 (`max`) [3, 1, 4, 2]
=
  foldl1 (`max`) (3 : (1 : (4 : (2 : []))))
=
  foldl (`max`) 3 (1 : (4 : (2 : [])))
=
  foldl (`max`) (3 `max` 1) (4 : (2 : []))
=
  foldl (`max`) ((3 `max` 1) `max` 4) (2 : [])
=
  foldl (`max`) (((3 `max` 1) `max` 4) `max` 2)[]
=
  (((3 `max` 1) `max` 4) `max` 2)
```

Part II

Append

Append

```
(++) :: [a] -> [a] -> [a]
[] ++ ys       =  ys
(x:xs) ++ ys   =  x : (xs ++ ys)

  "abc" ++ "de"
=
  ('a' : ('b' : ('c' : []))) ++ ('d' : ('e' : []))
=
  'a' : (('b' : ('c' : [])) ++ ('d' : ('e' : [])))
=
  'a' : ('b' : (('c' : []) ++ ('d' : ('e' : []))))
=
  'a' : ('b' : ('c' : ([] ++ ('d' : ('e' : [])))))
=
  'a' : ('b' : ('c' : ('d' : ('e' : []))))
=
  "abcde"
```

Append

```
(++) :: [a] -> [a] -> [a]
[] ++ ys        =  ys
(x:xs) ++ ys    =  x : (xs ++ ys)

   "abc" ++ "de"
=
   'a' : ("bc" ++ "de")
=
   'a' : ('b' : ("c" ++ "de"))
=
   'a' : ('b' : ('c' : ("" ++ "de")))
=
   'a' : ('b' : ('c' : "de"))
=
   "abcde"
```

Properties of operators

- There are a few key properties about operators: *associativity*, *identity*, *commutativity*, *distributivity*, *zero*, *idempotence*. You should know and understand these properties.
- When you meet a new operator, the first question you should ask is "Is it associative?" The second is "Does it have an identity?"
- Associativity is our friend, because it means we don't need to worry about parentheses. The program is easier to read.
- Associativity is our friend, because it is key to writing programs that run twice as fast on dual-core machines, and a thousand times as fast on machines with a thousand cores.

Properties of append

```
prop_append_assoc :: [Int] -> [Int] -> [Int] -> Bool
prop_append_assoc xs ys zs  =
  xs ++ (ys ++ zs)  ==  (xs ++ ys) ++ zs

prop_append_ident :: [Int] -> Bool
prop_append_ident xs  =
  xs ++ [] == xs  &&  xs == [] ++ xs

prop_append_cons :: Int -> [Int] -> Bool
prop_append_cons x xs  =
  [x] ++ xs  ==  x : xs
```

Infix vs prefix notation

```
(++) :: [a] -> [a] -> [a]
[] ++ ys       =  ys
(x:xs) ++ ys   =  x : (xs ++ ys)

prop_append_assoc :: [Int] -> [Int] -> [Int] -> Bool
prop_append_assoc xs ys zs  =
  xs ++ (ys ++ zs)  ==  (xs ++ ys) ++ zs
```

vs

```
append :: [a] -> [a] -> [a]
append [] ys       =  ys
append (x:xs) ys   =  x : append xs ys

prop_append_assoc :: [Int] -> [Int] -> [Int] -> Bool
prop_append_assoc xs ys zs  =
  append xs (append ys zs)  ==  append (append xs ys) zs
```

Efficiency

```
(++) :: [a] -> [a] -> [a]
[] ++ ys        =  ys
(x:xs) ++ ys    =  x : (xs ++ ys)

  "abc" ++ "de"
=
  'a' : ("bc" ++ "de")
=
  'a' : ('b' : ("c" ++ "de"))
=
  'a' : ('b' : ('c' : ("" ++ "de")))
=
  'a' : ('b' : ('c' : "de"))
=
  "abcde"
```

Computing `xs ++ ys` takes about n steps, where n is the length of `xs`.

A useful fact

```
-- prop_sum.hs
import Test.QuickCheck

prop_sum :: Int -> Property
prop_sum n  =  n >= 0 ==>  sum [1..n]  ==  n * (n+1) `div` 2
```

```
[melchior]dts: ghci prop_sum.hs
GHCi, version 6.8.3: http://www.haskell.org/ghc/ :? for help
*Main> quickCheck prop_sum
+++ OK, passed 100 tests.
*Main>
```

Associate to the left: counting the cost

```
    foldl (++) "" ["abc", "def", "ghi", "jkl"]
=
    ((("" ++ "abc") ++ "def") ++ "ghi") ++ "jkl"
=   -- 0 steps
    (("abc" ++ "def") ++ "ghi") ++ "jkl"
=   -- 3 steps
    ("abcdef" ++ "ghi") ++ "jkl"
=   -- 6 steps
    "abcdefghi" ++ "jkl"
=   -- 9 steps
    "abcdefghijkl"
```

$$0 + 3 + 6 + 9 = 18$$

Associate to the right: counting the cost

```
    foldr (++) "" ["abc", "def", "ghi", "jkl"]
=
    "abc" ++ ("def" ++ ("ghi" ++ ("jkl" ++ "")))
=   -- 3 steps
    'a':'b':'c':("def" ++ ("ghi" ++ ("jkl" ++ "")))
=   -- 3 steps
    'a':'b':'c':'d':'e':'f' ++ ("ghi" ++ ("jkl" ++ ""))
=   -- 3 steps
    'a':'b':'c':'d':'e':'f':'g':'h':'i' ++ ("jkl" ++ "")
=   -- 3 steps
    'a':'b':'c':'d':'e':'f':'g':'h':'i':'j':'k':'l':""
```

$$3 + 3 + 3 + 3 = 12$$

Associativity and Efficiency: Left vs. Right

Consider m lists, $xs_1, \ldots, xs_n$, each of length n.

Associated to the left

$$(((\,[] \mathbin{+\!\!+} xs_1) \mathbin{+\!\!+} xs_2) \mathbin{+\!\!+} xs_3) \cdots \mathbin{+\!\!+} xs_m$$

computing takes

$$\underbrace{0 + n + 2n + 3n + ... + (m-1)n}_{m \text{ times}}$$

steps. If we have m lists of length n, it takes about $m^2 n$ steps.

Associated to the right

$$xs_1 \mathbin{+\!\!+} \cdots (xs_{m-2} \mathbin{+\!\!+} (xs_{m-1} \mathbin{+\!\!+} (xs_m \mathbin{+\!\!+} [\,])))$$

computing takes

$$\underbrace{n + n + n + \cdots + n}_{m \text{ times}}$$

steps. If we have m lists of length n, it takes about mn steps. When $m = 1000$, the first is a thousand times slower than the second!

Associativity and Efficiency: Sequential vs. Parallel

Sequential:

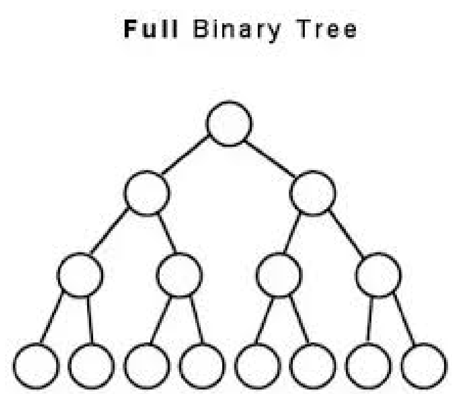
$$((((((x_1 + x_2) + x_3) + x_4) + x_5) + x_6) + x_7) + x_8$$

Summing 8 numbers takes 7 steps. If we have m numbers it takes $m - 1$ steps.

Parallel:

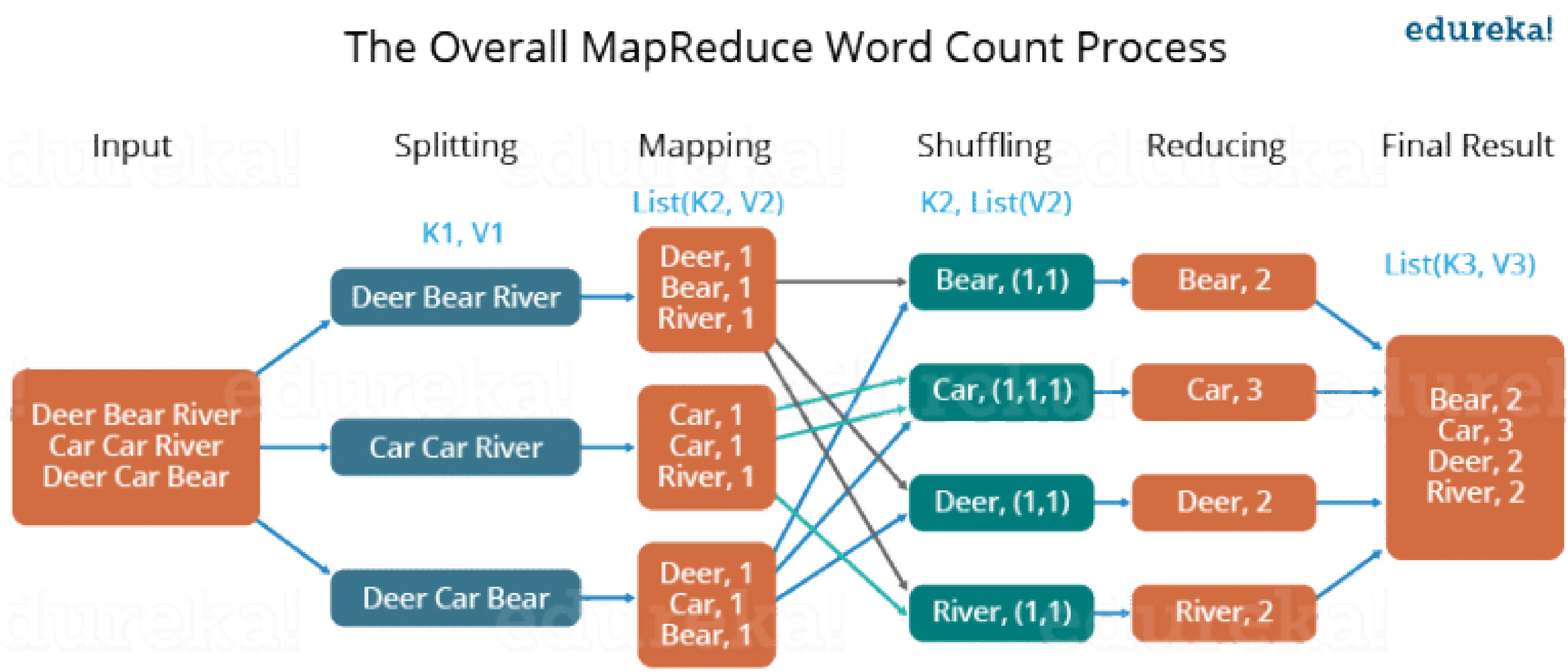
$$((x_1 + x_2) + (x_3 + x_4)) + ((x_5 + x_6) + (x_7 + x_8))$$

Summing 8 numbers takes 3 steps.



If we have m numbers it takes $\log_2 m$ steps. When $m = 1000$, the first is a hundred times slower than the second!

Map-Reduce


The Overall MapReduce Word Count Process
edureka!
Input
Splitting
Mapping
Shuffling
Reducing
Final Result
K1, V1
List(K2, V2)
K2, List(V2)
List(K3, V3)
Deer Bear River
Car Car River
Deer Car Bear
Deer Bear River
Car Car River
Deer Car Bear
Deer, 1
Bear, 1
River, 1
Car, 1
Car, 1
River, 1
Deer, 1
Car, 1
Bear, 1
Bear, (1,1)
Car, (1,1,1)
Deer, (1,1)
River, (1,1)
Bear, 2
Car, 3
Deer, 2
River, 2
Bear, 2
Car, 3
Deer, 2
River, 2