

CSC 447 - Concepts of Programming Languages

More Functional Folds

Instructor: Eric J. Fredericks



Learning Objectives

- ❓ How can we apply the concepts of strictness and non-strictness to folds?
 - Motivate the concept of lazy lists and infinite collections (streams)
 - Develop more purely functional folds (as in Haskell, etc.)
 - Delegate execution control to a fold without sacrificing efficiency or program termination



Scala Iteration

? Sum a list of Ints: Old-School Iteration

```
1 // Setup
2 val ints: List[Int] = (0 to 10_000).toList
3
4 // Sum: Old school iteration
5 var sumIt: Int = 0
6 for i <- ints do
7   sumIt += i
```

```
1 var sumIt: Int = 50005000
```



Scala Folds

💡 Sum a list of Ints: Scala Collections Fold

```
1 val ints: List[Int] = (0 to 10_000).toList
2
3 val sumFold = ints.fold(0)(_ + _)
```

```
1 val sumFold: Int = 50005000
```



Collections: Fold Left vs. Fold Right

Fold Left

💡 *from the left*

```
1 val ints: List[Int] = (0 to 10).toList
2
3 val sumL = ints.foldLeft(0)(_ + _)
```

```
1 val sumL: Int = 55
```

Fold Right

💡 *from the right*

```
1 val ints: List[Int] = (0 to 10).toList
2
3 val sumR = ints.foldRight(0)(_ + _)
```

```
1 val sumR: Int = 55
```

-
- Which argument in the function passed is the accumulator ("z" or "zero")?
 - How is `fold` implemented?



Scala Folds

? How does Scala implement `fold` ?

```
1 val ints: List[Int] = (0 to 10).toList
2
3 ints.fold(0): (x, y) =>
4   ???
```

```
1
2
3
4
5
6
7
8
9
10
```



Scala Folds

? How does Scala implement `fold` ?

```
1 val ints: List[Int] = (0 to 10).toList
2
3 ints.fold(0): (x, y) =>
4   throw new Exception() // Show me the call stack involved in executing this function!
```

```
1 java.lang.Exception
2   at rs$line$27$.init$$$anonfun$1(rs$line$27:2)
3   at scala.runtime.java8.JFunction2$mcIII$sp.apply(JFunction2$mcIII$sp.scala:17)
4   at scala.collection.LinearSeqOps.foldLeft(LinearSeq.scala:183)
5   at scala.collection.LinearSeqOps.foldLeft$(LinearSeq.scala:179)
6   at scala.collection.immutable.List.foldLeft(List.scala:79)
7   at scala.collection.IterableOnceOps.fold(IterableOnce.scala:792)
8   at scala.collection.IterableOnceOps.fold$(IterableOnce.scala:792)
9   at scala.collection.AbstractIterable.fold(Iterable.scala:935)
10  ... 33 elided
```



Scala Folds

? How does Scala implement `fold` ?

```
1 package scala.collection
2
3 trait IterableOnceOps[+A, +CC[_], +C] extends Any { this: IterableOnce[A] =>
4
5   // . . .
6
7   def fold[A1 >: A](z: A1)(op: (A1, A1) => A1): A1 = foldLeft(z)(op)
8
9   // . . .
10 }
```




Collections: Fold Left vs. Fold Right

? How does Scala implement `foldLeft` and `foldRight` ?

```
1 package scala.collection
2
3 trait LinearSeqOps[+A, +CC[X] <: LinearSeq[X],
4                   +C <: LinearSeq[A] with LinearSeqOps[A, CC, C]]
5   extends Any with SeqOps[A, CC, C] {
6
7   // . . .
8
9   override def foldLeft[B](z: B)(op: (B, A) => B): B = {
10     var acc = z
11     var these: LinearSeq[A] = coll
12     while (!these.isEmpty) {           // <-- Ed.: Lipstick on a pig.
13       acc = op(acc, these.head)
14       these = these.tail
15     }
16     acc
17   }
18 }
```

- `var`
- `while`
- *This is glorified iteration, implemented directly in Scala.*
- **No way to avoid iterating over every element!**
- **Linear in the length of the list!**

```
1 package scala.collection.immutable.List
2
3 sealed abstract class List[+A] extends /* . . . */ {
4   // . . .
5   final override def foldRight[B](z: B)(op: (A, B) => B): B = {
6     var acc = z
7     var these: List[A] = reverse      // <-- Ed.: Lipstick on a pig.
8     while (!these.isEmpty) {          // <----- Airbrushed!
9       acc = op(these.head, acc)
10      these = these.tail
11    }
12    acc
13  }
14  // . . .
15  final override def reverse: List[A] = {
16    var result: List[A] = Nil
17    var these = this
18    while (!these.isEmpty) {
19      result = these.head :: result
20      these = these.tail
21    }
22    result
23  }
24  // . . .
25 }
```

- `var`
- `while` X 2 (`reverse` call)
- **More glorified iteration over every element.**



Scala Iteration: Find

? Find an Element (Linear Find)

```
1 // Setup
2 val intArr: Array[Int] = (0 to 100).toArray // Need random access!
3
4 // Find: Old school iteration
5 val key = 7
6 var idx: Int = -1
7 var i = 0
8 while i < intArr.length && idx == -1 do
9   if intArr(i) == key then
10     idx = i
11   i += 1
```

```
val intArr: Array[Int] = // . . .
val key: Int = 7
var idx: Int = 7
var i: Int = 8
```

? Scala Iteration: Find

? Why can't I just `break`, as in Java and C?

```
1 val intArr: Array[Int] = (0 to 100).toArray // Need random access!
2 val key = 7
3 var idx: Int = -1
4
5 var i = 0
6 while i < intArr.length do
7   if intArr(i) == key then
8     idx = i
9     // Gee, wouldn't it be nice if I could break here?
10  i += 1
```

```
val intArr: Array[Int] = // . . .
val key: Int = 7
var idx: Int = 7
var i: Int = 101
```



Scala Iteration: Find

? Why can't I just `break`, as in Java and C?

```
1 import scala.util.control.*
2
3 val intArr: Array[Int] = (0 to 100).toArray // Need random access!
4
5 val loop = new Breaks
6 val key = 7
7 var idx: Int = -1
8 var i = 0
9 loop.breakable:
10   while(i < intArr.length) do
11     if intArr(i) == key then
12       idx = i
13       loop.break
14     i += 1
15
16 i // Tell me the value of i.
```

```
val intArr: Array[Int] = // . . .
val loop: scala.util.control.Breaks = scala.util.control.Breaks@14e83c9d
val key: Int = 7
var idx: Int = 7
```

- ! `Breaks` is implemented using exceptions to control execution!
- ! Using exceptions for flow control is bad!
- ! Please do not do this!



Scala Fold Left: Find

? How do I find with a foldLeft ?

```
1 val ints: List[Int] = (0 to 99_999).toList
2
3 val key = 42
4
5 ints.foldLeft[(Int, Int)](0, -1):
6   case ((i, k), _) if k != -1 => (i + 1, k)
7   case ((i, k), n) if n == key => (i + 1, i)
8   case ((i, _), _) => (i + 1, -1)
```

```
val ints: List[Int] = // . . .
val key: Int = 42
val res5: (Int, Int) = (100000, 42)
```

- ! Loop control is delegated to List's foldLeft function.
- ! I found the element early, but had to go through all 10,000 elements!
- ? Can we do better?



Recursion: Find

? How about recursion?

```
1 val ints: List[Int] = (0 to 99_999).toList
2
3 val key = 42
4
5 import scala.annotation.tailrec
6
7 def find(key: Int, intList: List[Int]): Int =
8   @tailrec
9   def search(i: Int, remaining: List[Int]): Int =
10     remaining match
11       case Nil => -1
12       case (x :: xs) if x == key => i // Yay, I can stop!
13       case (_ :: xs) => search(i + 1, xs)
14   search(0, intList)
15
16 find(key, ints)
```

```
val ints: List[Int] = // . . .
val key: Int = 42
val res6: Int = 42
```

- ! Now, I can abort the search when I'm done!
- ! The compiler can optimize to a loop *outside* of the Scala language (don't care).
- ! But I'm still responsible for execution control . . .
- ? Is there a way get the best of both (delegated execution control and efficiency)?
- ? **Not with folds from standard Scala collections!**



Scala Collections: List

```
1 val ints: List[Int] = (1 to Int.MaxValue).toList // N.B. Range is lazy!
```

```
java.lang.OutOfMemoryError: Java heap space: failed reallocation of scalar replaced objects
  at java.base/java.lang.Integer.valueOf(Integer.java:1005)
  at scala.runtime.BoxesRunTime.boxToInteger(BoxesRunTime.java:63)
  at scala.collection.immutable.RangeIterator.next(Range.scala:641)
  at scala.collection.immutable.List.prependAll(List.scala:156)
  at scala.collection.IterableOnceOps.toList(IterableOnce.scala:1446)
  at scala.collection.IterableOnceOps.toList$(IterableOnce.scala:1446)
  at scala.collection.AbstractIterable.toList(Iterable.scala:935)
```

- ❗ Scala (standard) collections are strict!
- ❗ All elements of the collection are evaluated - their values exist *in memory*.
- ❓ What if I made the the `List` non-strict, a.k.a. "lazy?"
- ❓ Scala collections defines a `LazyList` ! (inspired by other FP languages)



Scala Collections: LazyList

```
1 val lazy1 = LazyList.from(1)
2 lazy1.tail.head
3 lazy1
4 val lazyListOf5 = lazy1.take(5)
5 val listOf5 = lazyListOf5.toList
6 lazyListOf5
```

```
val lazy1: LazyList[Int] = LazyList(<not computed>)
val res2: Int = 2
val res3: LazyList[Int] = LazyList(1, 2, <not computed>) // 1 and 2 are memoized!
val lazyListOf5: LazyList[Int] = LazyList(<not computed>)
val listOf5: List[Int] = List(1, 2, 3, 4, 5)
val res4: LazyList[Int] = LazyList(1, 2, 3, 4, 5) // 1, 2, 3, 4, 5 are memoized!
```

- ❗ `LazyList` may be infinite!
- ❗ Related to the concept of a **stream**.



Scala LazyList: Find with FoldLeft

```
1 val lazyInts: LazyList[Int] = LazyList.from(0)
2 val key = 7
3
4 opaque type Index = Int
5 object Index:
6   def apply(i: Int): Index = i
7   val NotFound = Index(-1)
8
9 val indexedInts = lazyInts.zipWithIndex.map((n, i) => n -> Index(i))
10
11 indexedInts.foldLeft(NotFound):
12   case (k, _) if k != NotFound => k
13   case (_, (n, i)) if n == key => Index(i)
14   case _ => NotFound
```

```
val lazyInts: LazyList[Int] = LazyList(<not computed>)
val key: Int = 7
// defined object Index
val NotFound: Index = -1
val indexedInts: LazyList[(Int, Int)] = LazyList(<not computed>)
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
```

! Still trying to go through the entire LazyList !?



Scala LazyList: Find with FoldRight

```
1 val lazyInts: LazyList[Int] = LazyList.from(0)
2 val key = 7
3
4 opaque type Index = Int
5 object Index:
6   def apply(i: Int): Index = i
7   val NotFound = Index(-1)
8
9 val indexedInts = lazyInts.zipWithIndex.map((n, i) => n -> Index(i))
10
11 indexedInts.foldRight(NotFound):
12   case (_, k) if k != NotFound => k
13   case ((n, i), _) if n == key => Index(i)
14   case _ => NotFound
```

```
val lazyInts: LazyList[Int] = LazyList(<not computed>)
val key: Int = 7
// defined object Index
val NotFound: Index = -1
val indexedInts: LazyList[(Int, Int)] = LazyList(<not computed>)
java.lang.OutOfMemoryError: Java heap space
```

! Still trying to go through the entire LazyList !?



Scala LazyList: Fold Definitions

❓ How are LazyList's fold s defined?

Fold Left

```
1 // foldLeft is defined in class
2 //   scala.collection.immutable.LazyList
3
4 @tailrec
5 override def foldLeft[B](z: B)(op: (B, A) => B): B =
6   if (isEmpty) z
7   else tail.foldLeft(op(z, head))(op)
```

Fold

```
1 // fold is defined in trait
2 //   scala.collection.IterableOnceOps
3 def fold[A1 >: A](z: A1)(op: (A1, A1) => A1): A1 =
4   foldLeft(z)(op) // dynamic dispatch to LazyList!
```

Fold Right

```
1 // foldRight is defined in trait
2 //   scala.collection.IterableOnceOps
3
4 def foldRight[B](z: B)(op: (A, B) => B): B =
5   reversed.foldLeft(z)((b, a) => op(a, b))
6
7 protected def reversed: Iterable[A] = {
8   var xs: immutable.List[A] = immutable.Nil
9   val it = iterator
10  while (it.hasNext) xs = it.next() :: xs
11  xs
12 }
```

- ❗ var
- ❗ iteration / while
- ❗ Continual evaluation of tail while retaining each "cons cell" in memory
- ❗ Not quite what we want ...
- 💡 Parameters are strictly evaluated by default in most languages!
- 💡 Parameters are not strictly defined in Haskell!
- 💡 Let's try making parameters non-strict!



FP Fold Left

💡 Haskell-style Fold Left

```
1 import scala.annotation.tailrec
2
3 opaque type Index = Int
4 object Index:
5   def apply(i: Int): Index = i
6   val NotFound = Index(-1)
7
8 // Translation from Haskell.
9 @tailrec
10 def foldl[A, B](f: (=> B) => A => B)(z: => B)(a: LazyList[A]): B =
11   a match
12     case LazyList() => z           // LazyList() analagous to Nil (sort of)
13     case x#::xs => foldl(f)(f(z)(x))(xs) // #:: is how we pattern match with LazyList
```

```
1 def findLeft(key: Int)(indexedInts: LazyList[(Int, Index)]): Index =
2   def check(findRest: => Index)(next: (Int, Index)): Index =
3     next match
4       case (n, i) if key == n => i
5       case _ => findRest
6   foldl(check)(NotFound)(indexedInts)
7
8 val lazyInts: LazyList[Int] = LazyList.from(0)
9 val indexedInts = lazyInts.zipWithIndex.map((n, i) => n -> Index(i))
10 findLeft(7)(indexedInts)
```

```
// defined object Index
val NotFound: Index = -1
def foldl[A, B](f: (=> B) => A => B)(z: => B)(a: LazyList[A]): B
def findLeft(key: Int)(indexedInts: LazyList[(Int, Index)]): Index
val lazyInts: LazyList[Int] = LazyList(<not computed>)
val indexedInts: LazyList[(Int, Index)] = LazyList(<not computed>)
java.lang.OutOfMemoryError: Java heap space
```

❗ So far, this is no better.



FP Fold Right

💡 Haskell-style Fold Right

```
1 import scala.annotation.tailrec
2
3 opaque type Index = Int
4 object Index:
5   def apply(i: Int): Index = i
6   val NotFound = Index(-1)
7
8 // Translation from Haskell.
9 def foldr[A, B](f: A => (=> B) => B)(z: => B)(a: LazyList[A]): B =
10   a match
11     case LazyList() => z           // LazyList() analagous to Nil (sort of)
12     case x#::xs => f(x)(foldr(f)(z)(xs)) // #:: is how we pattern match with LazyList
```

```
1 def findRight(key: Int)(indexedInts: LazyList[(Int, Index)]): Index =
2   def check(next: (Int, Index))(findRest: => Index): Index =
3     next match
4       case (n, i) if key == n => i
5       case _ => findRest
6   foldr(check)(NotFound)(indexedInts)
7
8 val lazyInts: LazyList[Int] = LazyList.from(0)
9 val indexedInts = lazyInts.zipWithIndex.map((n, i) => n -> Index(i))
10 findRight(7)(indexedInts)
```

```
// defined object Index
val NotFound: Index = -1
def foldr[A, B](f: A => (=> B) => B)(z: => B)(a: LazyList[A]): B
def findRight(key: Int)(indexedInts: LazyList[(Int, Index)]): Index
val lazyInts: LazyList[Int] = LazyList(<not computed>)
val indexedInts: LazyList[(Int, Index)] = LazyList(<not computed>)
val res3: Index = 7
```

❗ No elements visited unnecessarily.

❗ Much better!

❓ Why does `foldr` work better?



FP Fold Right Evaluation

💡 findRight with foldr evaluation

```
1 opaque type Index = Int
2 object Index:
3   def apply(i: Int): Index = i
4   val NotFound = Index(-1)
5
6 // Translation from Haskell.
7 def foldr[A, B](f: A => (=> B) => B)(z: => B)(a: LazyList[A]): B =
8   a match
9     case LazyList() => z           // LazyList() analogous to Nil (sort of)
10    case x#::xs => f(x)(foldr(f)(z)(xs)) // #:: is how we pattern match with LazyList
```

```
1 def findRight(key: Int)(indexedInts: LazyList[(Int, Index)]): Index =
2   def check(next: (Int, Index))(findRest: => Index): Index =
3     next match
4       case (n, i) if key == n => i
5       case _ => findRest
6   foldr(check)(NotFound)(indexedInts)
7
8 val lazyInts: LazyList[Int] = LazyList.from(0)
9 val indexedInts = lazyInts.zipWithIndex.map((n, i) => n -> Index(i))
10 findRight(7)(indexedInts)
```

```
findRight(7)(indexedInts)
foldr(check)(NotFound)(0..)
check((0, Index(0)))(foldr(check)(NotFound)(..))
foldr(check)(NotFound)(1..)
check((1, Index(1)))(foldr(check)(NotFound)(..))
foldr(check)(NotFound)(2..)
check((2, Index(2)))(foldr(check)(NotFound)(..))
foldr(check)(NotFound)(3..)
check((3, Index(3)))(foldr(check)(NotFound)(..))
foldr(check)(NotFound)(4..)
check((4, Index(4)))(foldr(check)(NotFound)(..))
foldr(check)(NotFound)(5..)
check((5, Index(5)))(foldr(check)(NotFound)(..))
foldr(check)(NotFound)(6..)
check((6, Index(6)))(foldr(check)(NotFound)(..))
foldr(check)(NotFound)(7..)
check((7, Index(7)))(foldr(check)(NotFound)(..))
Index(7)
```



FP Fold Left Evaluation



findLeft with foldl evaluation

```
1 import scala.annotation.tailrec
2
3 opaque type Index = Int
4 object Index:
5   def apply(i: Int): Index = i
6   val NotFound = Index(-1)
7
8 @tailrec
9 def foldl[A, B](f: (=> B) => A => B)(z: => B)(a: LazyList[A]): B =
10   a match
11     case LazyList() => z           // LazyList() analagous to Nil (sort of)
12     case x#::xs => foldl(f)(f(z)(x))(xs) // #:: is how we pattern match with LazyList
```

```
1 def findLeft(key: Int)(indexedInts: LazyList[(Int, Index)]): Index =
2   def check(findRest: => Index)(next: (Int, Index)): Index =
3     next match
4       case (n, i) if key == n => i
5       case _ => findRest
6   foldl(check)(NotFound)(indexedInts)
7
8 val lazyInts: LazyList[Int] = LazyList.from(0)
9 val indexedInts = lazyInts.zipWithIndex.map((n, i) => n -> Index(i))
10 findLeft(3)(indexedInts)
```

```
findLeft(3)(0..)
foldl(check)(NotFound)(0..)
foldl(check)(check(NotFound)((0, Index(0))))(1..)
foldl(check)(check(check(NotFound)((0, Index(0))))((1, Index(1))))(2..)
foldl(check)(check(check(check(NotFound)((0, Index(0))))((1, Index(1))))((2, Index(2))))(3..)
foldl(check)(check(check(check(check(NotFound)((0, Index(0))))((1, Index(1))))((2, Index(2))))((3, Index(3))))(4..)
...
```

- ! Huge thunk!
- ! We never call `check` !
- ! Program never terminates!



Summary

- We can leverage non-strict semantics for collections processing.
- Expressions can have a value if some of their subexpressions do not.
- Call-by-name is useful!
- Non-strictness is what allows programs to work with infinite data structures.
- Non-strict folds: `foldr` is often the "right fold" - works great with non-strict functions on infinite lists!
- Non-strict folds: `foldl` is rarely useful!
- We can delegate execution control to folds without sacrificing efficiency.