CSC 347 - Concepts of Programming Languages

More Functional Folds

Instructor: James Riely

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?*
- *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`?

```scala
val ints: List[Int] = (0 to 10).toList

ints.fold(0): (x, y) =>
  throw new Exception()  // Show me the call stack involved in executing this function!
```

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

Scala Folds

❓ How does Scala implement `fold`?

```scala
package scala.collection

trait IterableOnceOps[+A, +CC[_], +C] extends Any { this: IterableOnce[A] =>

  // . . .

  def fold[A1 >: A](z: A1)(op: (A1, A1) => A1): A1 = foldLeft(z)(op)

  // . . .
}
```

Collections: Fold Left vs. Fold Right

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

```scala
package scala.collection

trait LinearSeqOps[+A, +CC[X] <: LinearSeq[X],
                   +C <: LinearSeq[A] with LinearSeqOps[A, CC, C]]
  extends Any with SeqOps[A, CC, C] {

  // . . .

  override def foldLeft[B](z: B)(op: (B, A) => B): B = {
    var acc = z
    var these: LinearSeq[A] = coll
    while (!these.isEmpty) {        // <-- Ed.: Lipstick on a pig.
      acc = op(acc, these.head)
      these = these.tail
    }
    acc
  }
}
```

- `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!**

```scala
package scala.collection.immutable.List

sealed abstract class List[+A] extends /* . . . */ {
  // . . .
  final override def foldRight[B](z: B)(op: (A, B) => B): B = {
    var acc = z
    var these: List[A] = reverse   // <-- Ed.: Lipstick on a pig.
    while (!these.isEmpty) {       // <------  Airbrushed!
      acc = op(these.head, acc)
      these = these.tail
    }
    acc
  }
  // . . .
  final override def reverse: List[A] = {
    var result: List[A] = Nil
    var these = this
    while (!these.isEmpty) {
      result = these.head :: result
      these = these.tail
    }
    result
  }
  // . . .
}
```

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

Scala Iteration: Find

Find an Element (Linear Find)

```scala
// Setup
val intArr: Array[Int] = (0 to 100).toArray   // Need random access!

// Find: Old school iteration
val key = 7
var idx: Int = -1
var i = 0
while i < intArr.length && idx == -1 do
  if intArr(i) == key then
    idx = i
  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`?

```scala
val ints: List[Int] = (0 to 99_999).toList

val key = 42

ints.foldLeft[(Int, Int)](0, -1):
  case ((i, k), _) if k != -1 => (i + 1, k)
  case ((i, k), n) if n == key => (i + 1, i)
  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?

```scala
val ints: List[Int] = (0 to 99_999).toList

val key = 42

import scala.annotation.tailrec

def find(key: Int, intList: List[Int]): Int =
  @tailrec
  def search(i: Int, remaining: List[Int]): Int =
    remaining match
      case Nil => -1
      case (x :: xs) if x == key => i // Yay, I can stop!
      case (_ :: xs) => search(i + 1, xs)
  search(0, intList)

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.prependedAll(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

```scala
val lazy1 = LazyList.from(1)
lazy1.tail.head
lazy1
val lazyListOf5 = lazy1.take(5)
val listOf5 = lazyListOf5.toList
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

```scala
val lazyInts: LazyList[Int] = LazyList.from(0)
val key = 7

opaque type Index = Int
object Index:
  def apply(i: Int): Index = i
val NotFound = Index(-1)

val indexedInts = lazyInts.zipWithIndex.map((n, i) => n -> Index(i))

indexedInts.foldLeft(NotFound):
  case (k, _) if k != NotFound => k
  case (_, (n, i)) if n == key => Index(i)
  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

```scala
val lazyInts: LazyList[Int] = LazyList.from(0)
val key = 7

opaque type Index = Int
object Index:
  def apply(i: Int): Index = i
val NotFound = Index(-1)

val indexedInts = lazyInts.zipWithIndex.map((n, i) => n -> Index(i))

indexedInts.foldRight(NotFound):
  case (_, k) if k != NotFound => k
  case ((n, i), _) if n == key => Index(i)
  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

```scala
// foldLeft is defined in class
//   scala.collection.immutable.LazyList

@tailrec
override def foldLeft[B](z: B)(op: (B, A) => B): B =
  if (isEmpty) z
  else tail.foldLeft(op(z, head))(op)
```

Fold Right

```scala
// foldRight is defined in trait
//   scala.collection.IterableOnceOps

def foldRight[B](z: B)(op: (A, B) => B): B =
  reversed.foldLeft(z)((b, a) => op(a, b))

protected def reversed: Iterable[A] = {
  var xs: immutable.List[A] = immutable.Nil
  val it = iterator
  while (it.hasNext) xs = it.next() :: xs
  xs
}
```

Fold

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

- `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

```scala
import scala.annotation.tailrec

opaque type Index = Int
object Index:
  def apply(i: Int): Index = i
val NotFound = Index(-1)

// Translation from Haskell.
@tailrec
def foldl[A, B](f: (=> B) => A => B)(z: => B)(a: LazyList[A]): B =
a match
  case LazyList() => z                    // LazyList() analagous to Nil (sort of)
  case x#::xs => foldl(f)(f(z)(x))(xs)    // #:: is how we pattern match with LazyList
```

```scala
def findLeft(key: Int)(indexedInts: LazyList[(Int, Index)]): Index =
  def check(findRest: => Index)(next: (Int, Index)): Index =
    next match
      case (n, i) if key == n => i
      case _ => findRest
  foldl(check)(NotFound)(indexedInts)

val lazyInts: LazyList[Int] = LazyList.from(0)
val indexedInts = lazyInts.zipWithIndex.map((n, i) => n -> Index(i))
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

```scala
import scala.annotation.tailrec

opaque type Index = Int
object Index:
  def apply(i: Int): Index = i
val NotFound = Index(-1)

// Translation from Haskell.
def foldr[A, B](f: A => (=> B) => B)(z: => B)(a: LazyList[A]): B =
  a match
    case LazyList() => z                    // LazyList() analagous to Nil (sort of)
    case x#::xs => f(x)(foldr(f)(z)(xs))    // #:: is how we pattern match with LazyList
```

```scala
def findRight(key: Int)(indexedInts: LazyList[(Int, Index)]): Index =
  def check(next: (Int, Index))(findRest: => Index): Index =
    next match
      case (n, i) if key == n => i
      case _ => findRest
  foldr(check)(NotFound)(indexedInts)

val lazyInts: LazyList[Int] = LazyList.from(0)
val indexedInts = lazyInts.zipWithIndex.map((n, i) => n -> Index(i))
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

```scala
opaque type Index = Int
object Index:
  def apply(i: Int): Index = i
val NotFound = Index(-1)

// Translation from Haskell.
def foldr[A, B](f: A => (=> B) => B)(z: => B)(a: LazyList[A]): B =
  a match
    case LazyList() => z                      // LazyList() analagous to Nil (sort of)
    case x#::xs => f(x)(foldr(f)(z)(xs))      // #:: is how we pattern match with LazyList
```

```scala
def findRight(key: Int)(indexedInts: LazyList[(Int, Index)]): Index =
  def check(next: (Int, Index))(findRest: => Index): Index =
    next match
      case (n, i) if key == n => i
      case _ => findRest
  foldr(check)(NotFound)(indexedInts)

val lazyInts: LazyList[Int] = LazyList.from(0)
val indexedInts = lazyInts.zipWithIndex.map((n, i) => n -> Index(i))
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

```scala
import scala.annotation.tailrec

opaque type Index = Int
object Index:
  def apply(i: Int): Index = i
val NotFound = Index(-1)

@tailrec
def foldl[A, B](f: (=> B) => A => B)(z: => B)(a: LazyList[A]): B =
  a match
    case LazyList() => z                    // LazyList() analagous to Nil (sort of)
    case x#::xs => foldl(f)(f(z)(x))(xs)    // #:: is how we pattern match with LazyList
```

```scala
def findLeft(key: Int)(indexedInts: LazyList[(Int, Index)]): Index =
  def check(findRest: => Index)(next: (Int, Index)): Index =
    next match
      case (n, i) if key == n => i
      case _ => findRest
  foldl(check)(NotFound)(indexedInts)

val lazyInts: LazyList[Int] = LazyList.from(0)
val indexedInts = lazyInts.zipWithIndex.map((n, i) => n -> Index(i))
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.