“Functional Programming in Scala”
Chapter 5
Laziness, Strictness

Nathaniel Osgood

Important Topics

- Strict vs. non-strict constructs
- Incremental processing (key for big data)
- Non-strictness as facilitating reuse
- Scala mechanisms for laziness
  - By name parameters
  - Lazy values
  - Streams
- Memory demands: Incremental handling, caching

Defining the Perfect Squares as a Stream

- import scala.math._
- val positiveInts = Stream.iterate(0L)((i:Long) => i+1)

- def isSquare(n: Long) = { val root = round(floor(sqrt(n))); root*root == n }

- val nonSquares = positiveInts.filter(x => !isSquare(x))

Result?

- lazy val aCached = { println("foo"); 3.14159 }
- aCached
- aCached

What do the Following Yield?

- def printAndReturnN(n: Int) = { println(n); n }
- printAndReturnN(4)

- def eagerPrintFn4(v : Int) = v + v
- eagerPrintFn4(printAndReturnN(4))

- def deferredPrintFn5(fnGetV : () => Int) = fnGetV() + fnGetV()
- deferredPrintFn5(() => printAndReturnN(4))

- def deferredPrintFn4(v : =>Int) = v + v
- deferredPrintFn4(printAndReturnN(4))

Defining the Fibonacci Sequence as a Stream

- def fibonacci(a:Long, b:Long) : Stream[Long] = a #:: fibonacci(b, a+b)
- val fibs = fibonacci(0, 1)

Stream of square Fibonacci numbers:

- val squareFibs = fibs.filter(isSquare)

Stream of successive Fibonacci numbers:

- def successiveRatios(s : Stream[Long]) : Stream[Double] = (s.head / s.tail.head.toDouble) #:: successiveRatios(s.tail)
- successiveRatios(fibs).take(20).force

Compare the Following

- Source.fromFile("document.txt")("ISO-8859-1").getLines.toStream.flatMap(_.split(raw"[^a-zA-Z]+")).filter(word => word.length > 0).map(word => word.toLowerCase).groupBy(word => word.size).mapValues(groupedWords => groupedWords.size).toArray.sorted

- Same thing as above, but “toArray” rather than “toStream”

Co-recursion

- Here, we generate data elements (e.g., emit a stream) while we maintain state