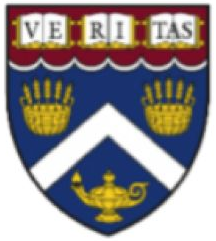
CSCI E-88A Introduction to Functional and Stream Processing for Big Data Systems

Harvard University Extension, Spring 2020
Marina Popova, Edward Sumitra

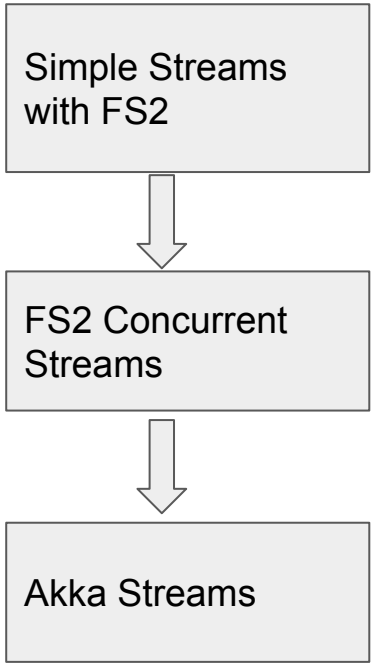


Lecture 07 - Scala Streams

Agenda

- Admin info: Lab
- Scala streams lectures
- Effects in Programs
- FS2
- Creating Streams
- Stream operators
- Effectful Streams
- Streaming Examples

Scala Streams Lectures



Processing large datasets lazily

Processing large datasets concurrently

Processing large datasets across multiple machines

Techniques for building Reactive Systems

Built on foundations of functional programming

Effects in programs

A pure function returns the same output and has no side effects

All three functions have side effects

Side effects make it difficult to reason about programs

```
object FunctionUtils {

  var globalCounter: Int = 5

  def increment(i: Int): Int = {
    globalCounter += i
    globalCounter
  }


  def incrementCounter(i: Int): Unit =
    globalCounter += i


  def myLogger(message: String): Unit =
    println(s"[${message}]")

}
```

Effects in programs

Side effects makes it difficult to reason about programs.
A technique for dealing with side effects is to separate program descriptions from program execution.
One way is to push side effects to the program edges or “the end of the universe”

How?
Move the side effect into the function signature
  effectively declaring the side effect instead of evaluating the side effect

```
def myLogger2(message: String): () => Unit =
  () => println(s"[${message}]")
```

Effects in Programs

Examples of effects in functions and programs

IO: functions that read from console or database; write to console or database

Option: functions that may or may not have a value e.g., division by zero

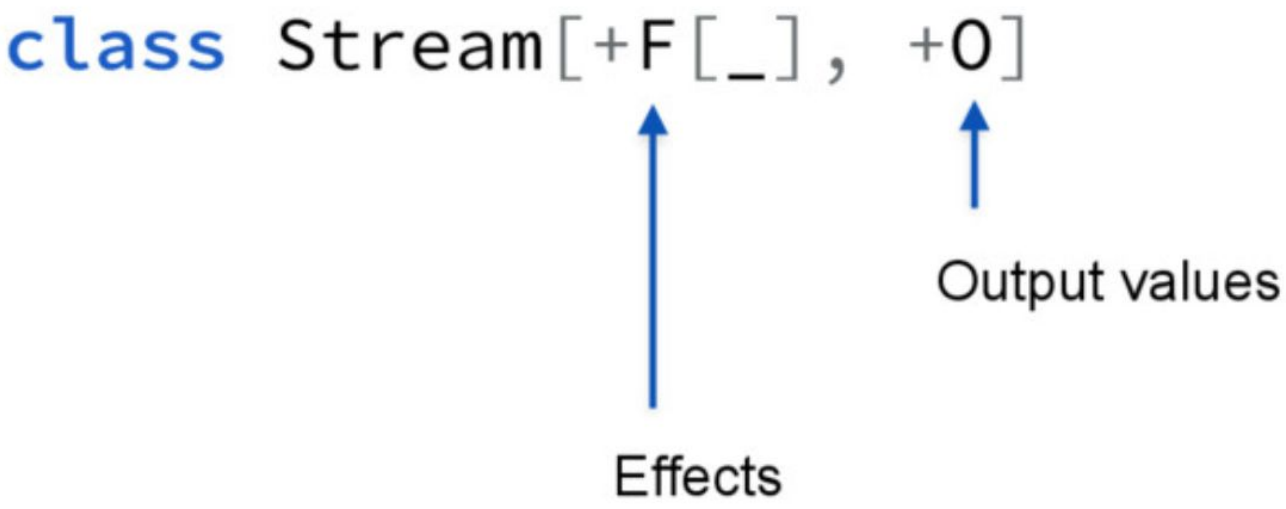
Future: functions that may eventually have a value e.g, async API calls

State: functions that model the effect of state

FS2

A purely functional, effectful stream processing library

https://fs2.io/



Lists vs. Streams

List	Stream
Eager Evaluates all items in memory	Lazy Only evaluates limited number of items, typically a single item, in memory
Will always terminate	Need not terminate i.e, can be infinite
Item values always available	Streams with effects need to be realized and the end of the universe

FS2 Creating Streams

From values and sequences

```
// empty Stream
val emptyIntStream: Stream[Pure, Int] = Stream.empty

// from values
val stream1 = Stream.emit(1)

val stream2 = Stream(1)

val stream3 = Stream(1,2,3,4,5)

// from sequences
val stream4 = Stream.emits(List(1))

val stream5 = Stream.emits(List(1,2,3,4,5))

val stream6 = Stream.emits( os = 1 to 1000)
```

FS2 Creating Streams

Stream reification and sampling

Streams can be infinite, need to sample data for testing

```
// stream reification
val list5 = stream5.toList

val vec6 = stream6.toVector

// sampling stream values
val subset1 = stream5.take(2).toList

val subset2 = stream5.drop(1).take(3).toList
```

FS2 Creating Streams

DEMO

FS2 Creating Streams

Creating infinite streams

```
// creating infinite streams
val constantStream = Stream.constant("a")

val rep1 = Stream(1,2,3).repeat

val wholeNumbers = Stream.iterate( start = 1)(_ + 1)
```

```
def iterate[F[x] >: Pure[x], A](start: A)(f: (A) ⇒ A): Stream[F, A]
```

An infinite `Stream` that repeatedly applies a given function to a start value. `start` is the first value emitted, followed by `f(start)`, then `f(f(start))`, and so on.

DEMO

FS2 Stream Operators

adding, map, filter, reduce

```
// stream combinators 1
val seqStreams = Stream(1,2,3) ++ Stream(3,4)

val tens = wholeNumbers.map(_ * 10)

val byFives = wholeNumbers.filter(_ % 5 == 0)

val sumStream = Stream.emits( os = 1 to 100).fold(0)(_ + _).toList

val onesAndTens = wholeNumbers.take(3) ++ wholeNumbers.map(_ * 10).take(3)
```

DEMO

FS2 Stream Operators

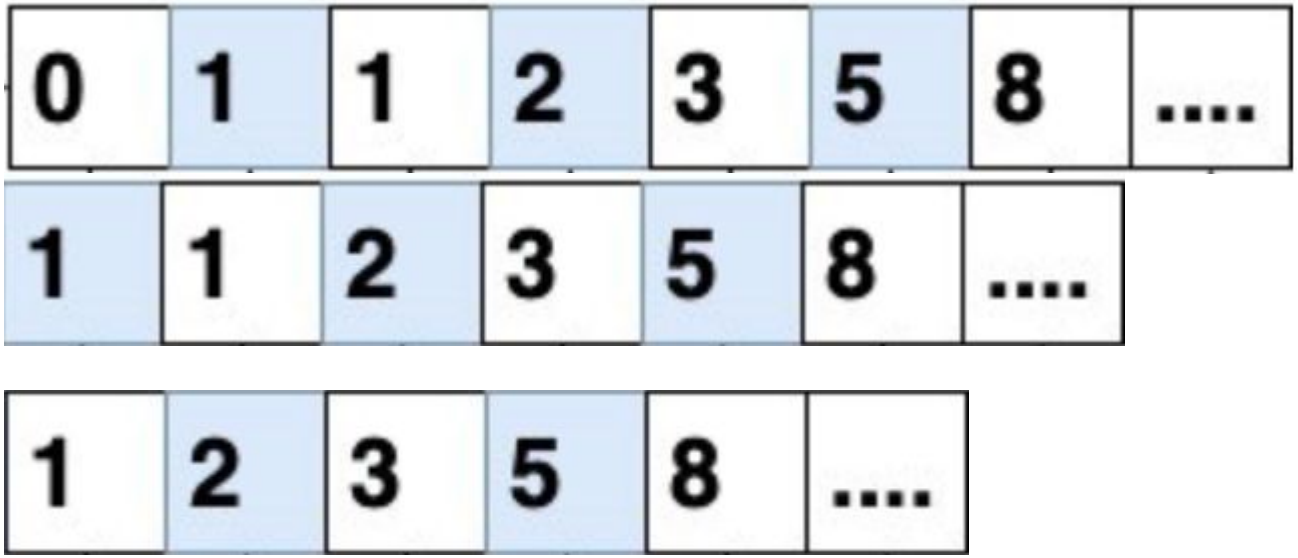
Zip, zipWith,intersperse, interleave, flatMap, for comprehensions

```
// stream combinators 2
val csvItems = Stream(1,2,3,4,5).map(_.toString).intersperse( separator = ",")

val mixedItems = Stream(1,2,3,4,5).interleave(Stream(10,20,30,40,50))

val charInts = Stream.emits( os = 'a' to 'e').zip(Stream.emits( os = 1 to 5))

val charInts2 =
  Stream.emits( os = 'a' to 'e')
    .zipWith(Stream.emits( os = 1 to 5)) { (c, i) =>
      s"[${c}, ${i}]"
    }

val charInts3 = for {
  c <- Stream.emits( os = 'a' to 'e')
  i <- Stream.emits( os = 1 to 5)
} yield s"[${c},${i}]"
```

DEMO

Examples

Fibonacci series (pure streams)




```
val fibs: Stream[Pure, Int] =
  Stream(0,1) ++ fibs.zipWith(fibs.tail)(_ + _)
```

Examples

Print numbers at a given rate

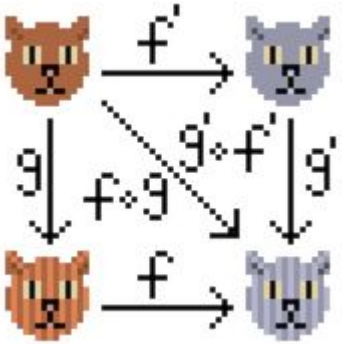
```
// example
def putNum(i: Int) = IO(println(i))

val printRange = Stream.emits( os = 1 to 10).evalMap(putNum)

val rate = Stream.awakeEvery[IO](2 seconds)

val printWithRate = rate.zip(printRange)

printWithRate.compile.drain.unsafeRunSync
```

Summary

- Streams are lazy and are good for processing large data sets that do not fit in memory
- Streams can be infinite i.e, do not terminate
- Streams can be composed just like functions
- Complex streams and data pipelines can be built from simpler streams
- The same Functional Programming Techniques like pure functions, immutable data, composition through higher-order-functions are used to assemble complex streams
- Separating processing programs into a descriptive step and execution step allows one to apply the advantages of functional programming for streams with effects.

Scala Functional Libraries

Cats and Cats-effect



Cats IO

FS2

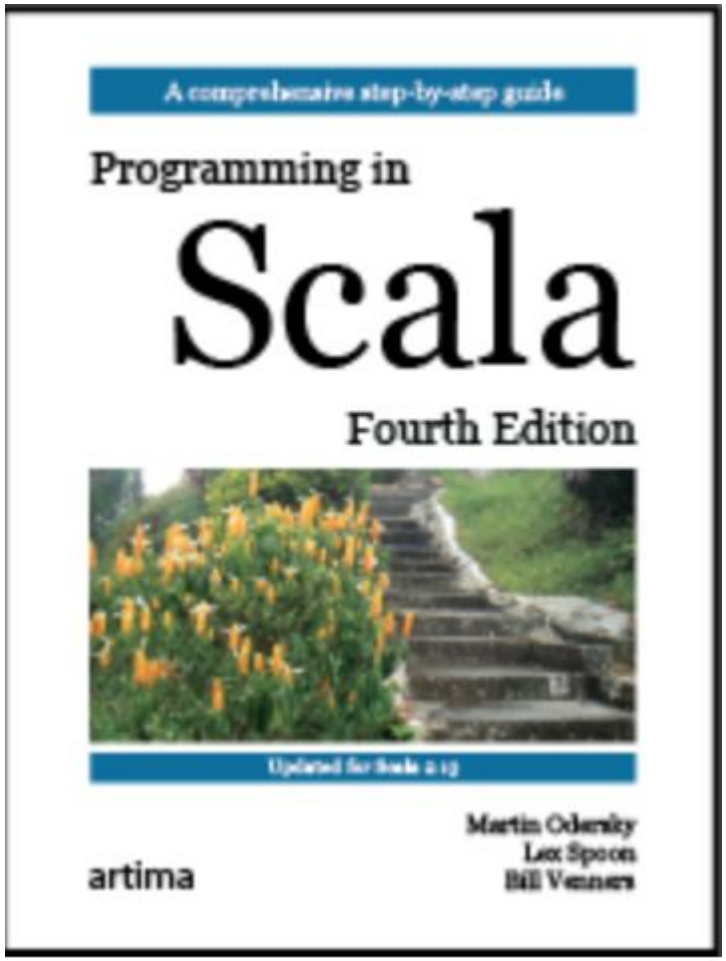
https://fs2.io/

Stream API docs:
Stream Class
Stream Companion Object

Scala Reference

Programming in Scala

[Programming in Scala, First Edition]

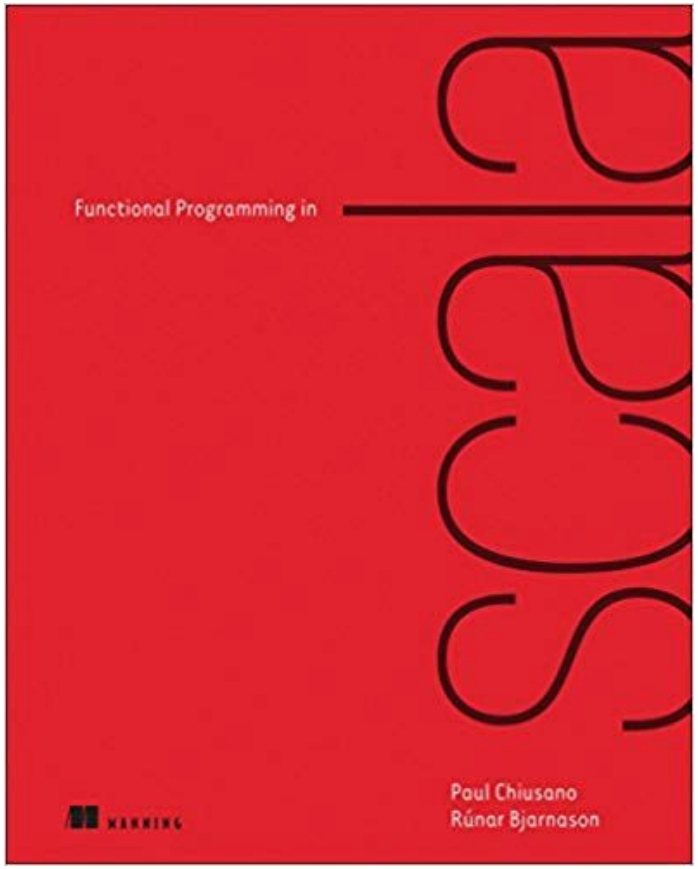


@Marina Popova, Edward Sumitra

Scala Function Programming Reference

Functional Programming in Scala
(Red Book)

Paul Chiusano and
Runar Bjarnason



https://www.amazon.com/Functional-Programming-Scala-Paul-Chiusano/dp/1617290653/