Scalaz-Stream Masterclass

Rúnar Bjarnason, Verizon Labs
@runarorama

NEScala 2016, Philadelphia

Scalaz-Stream (FS2)

Functional **S**treams for **S**cala

https://github.com/functional-streams-for-scala/fs2

Disclaimer

This library is changing.

We’ll talk about the *current* version (0.8).

Scalaz 7.1

Scalaz-Stream (FS2)

a purely functional streaming I/O library for Scala

- Streams are essentially “lazy lists” of data and effects.
- Naturally pull-based
- Immutable and referentially transparent

Design goals

- compositional
- expressive
- resource-safe
- comprehensible

Takeaway:
No magic

```scala
import scalaz.stream._
import scalaz.concurrent.Task

val converter: Task[Unit] =
  io.linesR("testdata/fahrenheit.txt")
    .filter(s => !s.trim.isEmpty && !s.startsWith("//"))
    .map(line => fahrenheitToCelsius(line.toDouble).toString)
    .intersperse("\n")
    .pipe(text.utf8Encode)
    .to(io.fileChunkW("testdata/celsius.txt"))
    .run

val u: Unit = converter.run
```

scalaz.concurrent.Task

- Asynchronous
- Compositional
- Purely functional

a **Task** is a first-class program

a **Task** is a list of instructions

Task is a monad

a **Task** doesn't *do* anything until you call `.run`

Constructing Tasks

```
Task.delay(readLine): Task[String]

Task.now(42): Task[Int]

Task.fail(
  new Exception("oops!")
): Task[Nothing]
```

```
fut: scala.concurrent.Future[Int]

Task.async(fut.onComplete): Task[Int]
Task.async {
  k => fut.onComplete {
    case Success(a) => k(\/.right(a))
    case Fail(a) => k(\/.left(e))
  }
}
```

```scala
a: Task[A]
pool: java.util.concurrent.ExecutorService

Task.fork(a)(pool): Task[A]
```

Combining Tasks

```scala
a: Task[A]
b: Task[B]

val c: Task[(A,B)] =
  Nondeterminism[Task].both(a,b)
```

```
a: Task[A]
f: A => Task[B]

val b: Task[B] = a flatMap f
```

```scala
val program: Task[Unit] =
  for {
    _ <- delay(println("What's your name?"))
    n <- delay(scala.io.StdIn.readLine)
    _ <- delay(println(s"Hello $n"))
  } yield ()
```

Running Tasks

```
a: Task[A]

a.run: A
```

```
a: Task[A]
k: (Throwable \/ A) => Unit

a runAsync k: Unit
```

Handling errors

```
Task.delay {
  throw new Exception("oops")
}

Task.fail {
  new Exception("oops")
}
```

```
t: Task[A]

t.attempt: Task[Throwable \/ A]
```

scalaz.stream.Process

```
Process[+F[_],+A]
```

Process[Task,A]

Stream primitives

```
val halt: Process[Nothing,Nothing]

def emit[O](o: O): Process[Nothing,O]

def await[F[_],I,O](
  req: F[I])(
  recv: I => Process[F,O]): Process[F,O]
```

```
foo: F[A]

Process.eval(foo): Process[F,A]
```

```
foo: F[A]

await(foo)(emit): Process[F,A]
```

```
Process.eval(
  Task.delay(readLine)
): Process[Task,String]
```

```
def IO[A](a: => A): Process[Task,A] =
  Process.eval(Task.delay(a))
```

Combining Processes

```
p1: Process[F,A]
p2: Process[F,A]

val p3: Process[F,A] =
  p1 ++ p2
```

```
p1: Process[F,A]
p2: Process[F,A]

val p3: Process[F,A] =
  p1 append p2
```

```
val twoLines: Process[Task,String] =
  IO(readLine) ++ IO(readLine)
```

```
val stdIn: Process[Task,String] =
  IO(readLine) ++ stdIn
```

```scala
val stdIn: Process[Task,String] =
  IO(readLine).repeat
```

```
val cat: Process[Task,Unit] =
  stdIn flatMap { s =>
    IO(println(s))
  }
```

```
val cat: Process[Task,Unit] =
  for {
    s <- stdIn
    _ <- IO(println(s))
  } yield ()
```

```
def grep(r: Regex): Process[Task,Unit] = {
  val p = r.pattern.asPredicate.test _
  def out(s: String) = IO(println(s))

  stdIn filter p flatMap out
}
```

Running Processes

```
F: Monad

p: Process[F,A]

p.run: F[Unit]
```

```
p: Process[F,A]

p.runLog: F[List[A]]
```

```
p: Process[F,A]

B: Monoid

f: A => B

p runFoldMap f: F[B]
```

Pipes

```
Process.await1[A]: Process1[A,A]
```

```scala
def take[I](n: Int): Process1[I,I] =
  if (n <= 0) halt
  else await1[I] ++ take(n - 1)
```

```
as: Process[F,A]

p: Process1[A,B]

as pipe p: Process[F,B]
```

```
as: Process[F,A]

val p = process1.chunk(10)

as pipe p: Process[F,Vector[A]]
```

```
as: Process[F,A]

as.chunk(10): Process[F,Vector[A]]
```

```
def distinct[A]: Process1[A,A] = {
  def go(seen: Set[A]): Process1[A,A] =
    Process.await1[A].flatMap { a =>
      if (seen(a)) go(seen)
      else Process.emit(a) ++ go(seen + a)
    }
  go(Set.empty)
}
```

```
Process1[A,B] ~=
  Process[(A=>?),O]
```

Multiple sources

scalaz.stream.tee

```
val f1 = scalaz.stream.io.linesR("/tmp/foo.txt")
val f2 = scalaz.stream.io.linesR("/tmp/bar.txt")

type Source[A] = Process[Task,A]

f1 zip f2: Source[(String,String)]
f1 interleave f2: Source[String]
f1 until f2.map(_ == "stop"): Source[String]
```

```
f1 zip f2
f1 interleave f2
f1 until f2.map(_ == "stop")
```

```
f1.tee(f2)(tee.zip)
f1.tee(f2)(tee.interleave)
f1.map(_ == "stop").tee(f2)(tee.until)
```

```
as: Process[F,A]
bs: Process[F,B]
t:  Tee[A,B,C]

(as tee bs)(t): Process[F,C]
```

```
val add: Tee[Int,Int,Int] = {
  for {
    x <- awaitL[Int]
    y <- awaitR[Int]
  } yield x + y
}.repeat

val sumEach = (p1 tee p2)(add)
```

```
Tee[A,B,O] ~=

Process[λ[x] =
   (A=>x) ∨ (B=>x), O]
```

scalaz.stream.wye

```scala
val f1 = IO(System.in.read).repeat
val f2 = io.linesR("/tmp/foo.txt")

type Source[A] = Process[Task,A]

f1 either f2: Source[Int \/ String]
f1.map(_.toChar.toString) merge f2: Source[String]

f1.map(_ => true))(f2)(wye.interrupt): Source[String]
```

```
as: Process[F,A]
bs: Process[F,B]
y:  Wye[A,B,C]

(as wye bs)(y): Process[F,C]
```

```
Wye[A,B,O] ≅

Process[λ[x] =
   (A=>x, B=>x, (A,B)=>x), O]
```

scalaz.stream.merge

```
ps: Process[F,Process[F,A]]

merge.mergeN(ps): Process[F,A]
```

```
nondeterminism.njoin(maxOpen,
                     maxQueued)(ps)
```

Sinks

```
x : Process[F,A]

y : Sink[F,A]

x to y : Process[F,Unit]
```

```
import scalaz.stream.io

io.stdInLines: Process[Task,String]

io.stdOutLines: Sink[Task,String]

val cat =
  io.stdInLines to io.stdOutLines
```

A sink is just a stream of functions

```scala
type Sink[F[_],A] =
  Process[F, A => Task[Unit]]
```

```scala
val stdOut: Sink[Task,String] =
  IO { s =>
    Task.delay(println(s))
  }.repeat
```

Channels

```
x : Process[F,A]

y : Channel[F,A,B]

x through y : Process[F,B]
```

A channel is just a stream of functions

```
type Channel[F[_],A,B] =
  Process[F, A => F[B]]
```

```
type Sink[F[_],A] =
  Channel[F,A,Unit]
```

```
s: java.io.InputStream

io.chunkR(s): Channel[Task,Int,ByteVector]
```

scalaz.stream.async

Queues & Signals

```
trait Queue[A] {
  ...
  def enqueue: Sink[Task,A]
  def dequeue: Process[Task,A]
  ...
}
```

```
import scalaz.stream.async._

def boundedQueue[A](n: Int): Queue[A]

def unboundedQueue[A]: Queue[A]

def circularBuffer[A](n: Int): Queue[A]
```

```
val pool =
  java.util.concurrent.Executors.newFixedThreadPool(16)

implicit val S =
  scalaz.concurrent.Strategy.Executor(pool)
```

```
trait Signal[A] {
  ...
  def get: Task[A]
  def set(a: A) Task[Unit]
  ...
}
```

```
trait Signal[A] {
  ...
  def discrete: Process[Task,A]
  def continuous: Process[Task,A]
  ...
}
```

Demo:

Internet Relay Chat

https://github.com/runarorama/ircz