Compositional I/O Streams in Scala

Rúnar Bjarnason, Verizon Labs
@runarorama

QCon London, March 2016

Scalaz-Stream (FS2)

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

github.com/functional-streams-for-scala

Scala

Functional Programming

Paul Chiusano
Rúnar Bjarnason

Foreword by Martin Odersky

MANNING

Disclaimer

This library is changing.

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

Funnel – oncue.github.io/funnel

http4s – http4s.org

streamz – github.com/krasserm/streamz

Scalaz-Stream (FS2)

a purely functional streaming I/O library for Scala

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

Design goals

- compositional
- expressive
- resource-safe
- comprehensible

```
import scalaz.stream._

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"))
```

Process[Task,A]

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

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

Task.now(42): Task[Int]

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

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

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

Combining Tasks

```
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
```

scalaz.stream.Process

Process[F[_],A]

Process[Task,A]

Stream primitives

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

def emit[A](a: A): Process[Nothing,A]

def eval[F[_],A](eff: F[A]): 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 Streams

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

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

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

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

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

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

```
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

```
p: Process[Task,A]
```

```
p.run: Task[Unit]
```

```
p: Process[Task,A]
```

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

```
p: Process[F,A]

B: Monoid

f: A => B

p runFoldMap f: F[B]
```

```
F: Monad

p: Process[F,A]

p.run: F[Unit]
```

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

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

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

Pipes

```
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]]
```

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

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

```scala
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)
}
```

Multiple sources

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

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

```
tee.zip: Tee[A,B,(A,B)]
```

```
tee.interleave: Tee[A,A,A]
```

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

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

```
as: Process[Task,A]
bs: Process[Task,B]

y:  Wye[A,B,C]

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

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

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

scalaz.stream.async

Queues & Signals

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

```scala
import scalaz.stream.async._

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

def unboundedQueue[A]: Queue[A]

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

```
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

github.com/runarorama/ircz

Server: 38 lines of Scala

Client: 14 lines of Scala

Uses scalaz-netty

github.com/runarorama/ircz

```
def serve(address: InetSocketAddress) =
  merge.mergeN {
    Netty serve address map { client =>
      for {
        c <- client
        _ <- IO(clients += c.sink)
        _ <- c.source to messageQueue.enqueue
      } yield ()
    }
  }
```

github.com/runarorama/ircz

```
val relay = for {
  message <- messageQueue.dequeue
  client <- emitAll(clients)
  _ <- emit(message) to client
} yield ()
```

github.com/runarorama/ircz

```
val main = (serve wye relay)(wye.merge)
```

github.com/runarorama/ircz

```
client = for {
  c <- Netty connect Server.address
  in = c.source
       .pipe(text.utf8Decode)
       .to(io.stdOutLines)
  out = io.stdInLines
        .pipe(text.utf8Encode)
        .to(c.sink)
  _ <- (in wye out)(wye.merge)
} yield ()
```

github.com/functional-streams-for-scala

github.com/runarorama/ircz

oncue.github.io/funnel