Comp 311
Functional Programming

Nick Vrvilo, Two Sigma Investments
Robert “Corky” Cartwright, Rice University

November 1, 2018

Announcements

- Homework 3 was due before class Today
- Homework 4 will be posted later this evening, and is due on Thursday, November 15th

Streams

- a form of "lazy" sequence
- inspired by signal-processing (e.g. digital circuits)
    - Components accept *streams* of signals as input, transform their input, and produce streams of signals as outputs

Stream Class

```scala
abstract class Stream[+T] {
  def head: T
  def tail: Stream[T]
  def map[S](f: T => S): Stream[S]
  def flatMap[S](f: T => Stream[S]): Stream[S]
  def ++[S >: T](that: Stream[S]): Stream[S]
  def withFilter(f: T => Boolean): Stream[T]
  def nth(n: Int): T
}
```

Streams

```
case object Empty extends Stream[Nothing] {
  def head = throw new NoSuchElementException
  def tail = throw new NoSuchElementException
  def map[S](f: Nothing => S) = Empty
  def flatMap[S](f: Nothing => Stream[S]) = Empty
  def ++[S >: Nothing](that: Stream[S]) = that
  def withFilter(f: Nothing => Boolean) = Empty
  def nth(n: Int) = throw new NoSuchElementException
}
```

Streams

```
case class Cons[+T](head: T, tail: => Stream[T])
extends Stream[T] {
  def map[S](f: T => S): Stream[S] =
    Cons(f(head), tail map f)
  def flatMap[S](f: T => Stream[S]): Stream[S] =
    f(head) ++ tail.flatMap(f)
  def ++[S >: T](that: Stream[S]): Stream[S] =
    Cons (head, tail ++ that)
  …
}
```

You can't actually use by-name parameters with case classes, but pretend this works for now. We'll cover how this would actually be implemented when we talk about companion objects.

Streams

```
case class Cons[+T](head: T, lazyTail: => Stream[T])
extends Stream[T] {
  …
  def withFilter(f: T => Boolean) = {
    if (f(head)) Cons(head, tail.withFilter(f))
    else tail.withFilter(f)
  }
  def nth(n: Int) = {
    require (n >= 0)
    if (n == 0) head
    else tail.nth(n - 1)
  }
}
```

Streams

```
def range(low: Int, high: Int): Stream[Int] =
  if (low > high) Empty
  else Cons(low, range(low + 1, high))
```

Streams

```
def intsFrom(n: Int): Stream[Int] =
  Cons(n, intsFrom(n + 1))
```

Streams

```
val nats = intsFrom(0)
```

Streams

```
def fibGen(a: Int, b: Int): Stream[Int] =
  Cons(a, fibGen(b, a + b))
```

Streams

```
val fibs = fibGen(0, 1)
```

Streams

```
def push(x: Int, ys: Stream[Int]) = {
  Cons(x, ys)
}
```

Streams

```
def isDivisible(m: Int, n: Int) = (m % n == 0)

val noSevens = nats withFilter (isDivisible(_, 7))
```

A Prime Sieve

```
def sieve(stream: Stream[Int]): Stream[Int] =
  Cons(stream.head,
              sieve(stream.tail withFilter
                       (x => !(isDivisible
                                (x, stream.head)))))
```

A Stream of Primes

```
val primes = sieve(intsFrom(2))
```

A Stream of Primes

```
> primes.head
res5: Int = 2
> primes.nth(1)
res6: Int = 3
> primes.nth(2)
res7: Int = 5
> primes.nth(3)
res8: Int = 7
```

Streams

```
def add(xs: Stream[Int],
        ys: Stream[Int]) :Stream[Int] = {

  (xs, ys) match {
    case (Empty, _) => ys
    case (_, Empty) => xs
    case (Cons(x,f), Cons(y,g)) =>
      Cons(x + y, add(f(), g()))
  }
}
```

Streams

```
def ones(): Stream[Int] = Cons(1, ones)
```

Alternative Definition of the Stream of Natural Numbers

```
def nats(): Stream[Int] =
  Cons(0, add(ones, nats))
```

Alternative Definition of the Fibonacci Stream

```
def fibs(): Stream[Int] =
  Cons(0,
           Cons(1,
                     add(fibs.tail, fibs)))
```

Powers of Two

```
def scaleStream(c: Int, stream: Stream[Int]): Stream[Int] =
  stream map (_ * c)

def powersOfTwo(): Stream[Int] =
  Cons(1, scaleStream(2, powersOfTwo))
```

Alternative Definition of the Stream of Primes

```
def primes() =
  Cons(2, intsFrom(3) withFilter isPrime)

def isPrime(n: Int): Boolean = {
  def sieve(next: Stream[Int]): Boolean = {
    if (square(next.head) > n) true
    else if (isDivisible(n, next.head)) false
    else sieve(next.tail)
  }
  sieve(primes)
}
```

Numeric Integration with Streams

$$S_i = c + \sum_{j=1}^{i} x_j dt$$

Numeric Integration with Streams

```
def integral(integrand: Stream[Double],
             init: Double,
             dt: Double) = {

  def inner(): Stream[Double] = {
    Cons(init,
               addStreams(scaleStream(dt,
                                      integrand),
                          inner))
  }
  inner
}
```

Streams and Local State

```
def withdraw(balance: Int, amounts: Stream[Int]):
Stream[Int] = {
  Cons(balance,
             withdraw(balance - amounts.head,
                      amounts.tail))
}
```

Discussion

- Our modeling of a bank account is a purely functional program without state
- Nevertheless:
  - If a user provides the stream of withdrawals, and
  - The stream of balances is displayed as outputs,
- The system will behave from a user's perspective as a stateful system

Discussion

- The key to understanding this paradox is that the “state” is in the world:
  - The user/bank system is stateful and provides the input stream
  - If we could “step outside” our own perspective in time, we could view our withdrawal stream as another stateless stream of transactions