Object-Oriented Design	Lecture 12
CS 3500 Fall 2010 (Pucella)	Tuesday, Oct 26, 2010

12 Stream Programming

Today, I want to look at a few more examples of the kind of things you can express with streams and polymorphism.

The idea that I want to propose is to consider that a stream is a way to produce elements (the exact kind of elements produced depending of course on the type of the stream). Putting it another way, a stream is a standardized interface to something that can produce elements. Having a standardized interface means that as soon as you have a gadget that can connect to that standardized interface, you can connect it to anything that implement that standardized interface. If that gadget itself produces elements according to that standardized interface, you have the beginning of a system that lets you connect gadgets together in a standardized way. In the case of streams, they produce elements. You can imagine connecting gadgets to those strams that modify the elements produced before passing them on, or combining the elements from two streams into a single element. Those gadgets for all intents and purpose can be made to look like streams to the outside world.

Recall that for us streams are subtypes of the following trait:

```
trait Stream[A] {
  def hasElement ():Boolean
  def head ():A
  def tail ():Stream[A]
}
```

Recall also the following functions that work on streams, introduced last time:

```
def printAll[T] (st:Stream[T]):Unit =
  if (st.hasElement()) {
    println("  " + st.head());
    printAll(st.tail())
  }

def sum[T <: Addable[T]] (st:Stream[T], u:T):T =
  if (st.hasElement())
    st.head().add(sum(st.tail(),u))
  else
    u
def printN[T] (st:Stream[T], n:Int):Unit =
  if (st.hasElement())
    if (n > 0) {
      println("  " + st.head())
      printN(st.tail(),n-1)
    }
    else
      println("  ...")
```

Functions `printAll()` and `sum()` print all the elements of a stream and sum all the elements of a stream (assuming those elements are `Addable`), respectively. Function `printN()` prints the first n elements of a stream, for a provided n. This is important because as far as the definition of streams is concerned, there is nothing that prevents a stream from producing an infinite sequence of values. This may seem counterintuitive at first, because right now the only streams we have seen are those giving all the elements stored in an aggregate structure, and all our aggregate structures have only held finitely many elements. But as we will see below, there are perfectly natural streams that are infinite in nature, and more importantly, it makes sense to work with them. As long as we don't try to print or work with *all* the elements of such a stream. Thus, we have a `printN()` function to only look at a finite prefix of a stream.

Last time we extended `List` and `BinTree` as subtyped of `Stream`, meaning that we could use stream operations on such types. Here are a few examples to refresh our memory:

```
    val L3:List[Int] =
      List.cons(33,List.cons(66,List.cons(99,List.empty())))

    println("L3 = " + L3)
    println("Elements L3 = ")
    printAll(L3)

    val T = BinTree
    val T1:BinTree[Int] = T.node(33,
                                 T.node(66,
                                        T.node(99,T.empty(),T.empty()),
                                        T.empty()),
                                 T.node(11,
                                        T.node(22,T.empty(),T.empty()),
                                        T.node(44,T.empty(),T.empty())))
    println("T1 = " + T1)
    println("Elements T1 = ")
    printAll(T1)
```

producing output:

```
L3 = 33 66 99
Elements L3 =
  33
  66
  99
T1 = 33[66[99[-,-],-],11[22[-,-],44[-,-]]]
Elements T1 =
  33
  66
  99
  11
  22
  44
```

Let's start by defining some streams independently of any underlying aggregate structure. The simplest kind of stream is an empty stream:

```
def empty[T] ():Stream[T] =
  new Empty[T]

class Empty[T] extends Stream[T] {

  def hasElement ():Boolean = false

  def head ():T = throw new RuntimeException("empty().head()")
  def tail ():Stream[T] = throw new RuntimeException("empty().tail()")
}
```

Equally simple is the (infinite) constant stream that delivers a single value whenever asked for an element.

```
def constant[T] (v:T):Stream[T] =
  new Constant[T](v)

class Constant[T] (v:T) extends Stream[T] {

  def hasElement ():Boolean = true

  def head ():T = v

  def tail ():Stream[T] = this
}
```

That stream is not doing anything especially interesting by itself:

```
    println("Constant(99) = ")
    printN(constant(99),20)
```

outputs:

```
Constant(99) =
  99
  99
  99
  99
  99
  99
  99
  99
  99
  99
  99
  99
  99
  99
  99
  99
  99
  99
  99
  99
  ...
```

Note the `...` at the end indicating that the stream is not empty when we stopped printing.

One step higher on the complexity scale is a stream that counts upwards from a given integer:

```
  def intsFrom (v:Int):Stream[Int] =
    new IntsFrom(v)

  class IntsFrom (v:Int) extends Stream[Int] {

    def hasElement ():Boolean = true

    def head ():Int = v
    def tail ():Stream[Int] = new IntsFrom(v+1)
  }
```

The interesting to note here is that `tail()` produce a new instance of `intsFrom()` that naturally enough, starts counting one higher.

```
    println("intsFrom(10) = ")
    printN(intsFrom(10),20)
```

produces output

```
intsFrom(10) =
  10
  11
  12
  13
  14
  15
  16
  17
  18
  19
  20
  21
  22
  23
  24
  25
  26
  27
  28
  29
  ...
```

The only thing we've done until now has been to produce streams. Let's start worrying about transforming streams. The following gadget connects to a incoming stream and produces one out of every two element of that incoming stream, dropping all other elements on the floor.

```
  def oneOutOfTwo[T] (st:Stream[T]):Stream[T] =
    new OneOutOfTwo[T](st)

  class OneOutOfTwo[T] (st:Stream[T]) extends Stream[T] {

    def hasElement ():Boolean = st.hasElement()
    def head ():T = st.head()

    def tail ():Stream[T] =
      if (st.tail().hasElement())
        new OneOutOfTwo(st.tail().tail())
      else
        empty()
  }
```

Note that `tail()` method is set up so that it always returns a new instance of the gadget that is ready to deliver the next element from the incoming stream to be delivered. If there are no more elements to be delivered, then it returns an empty stream. We can now produce the stream of odd numbers and the stream of even numbers by skipping one out of two elements from the appropriate stream of integers:

```
    println("Odds = ")
    printN(oneOutOfTwo(intsFrom(1)),20)
    println("Evens = ")
    printN(oneOutOfTwo(intsFrom(2)),20)
```

which produces:

```
Odds =
  1
  3
  5
  7
  9
  11
  13
  15
  17
  19
  21
  23
  25
  27
  29
  31
  33
  35
  37
  39
  ...
Evens =
  2
  4
  6
  8
  10
  12
  14
  16
  18
  20
  22
  24
  26
  28
  30
  32
  34
  36
  38
  40
  ...
```

Let's look at a slightly more complex gadget. Suppose we have two streams. We can imagine a gadget that connects those two streams in sequence: it first delivers all the elements from the first streams, and when the elements of that stream are exhausted, it delivers the elements of the second stream.

```
def sequence[T] (st1:Stream[T], st2:Stream[T]):Stream[T] =
  new Sequence[T](st1,st2)

class Sequence[T] (st1:Stream[T], st2:Stream[T]) extends Stream[T] {

  def hasElement ():Boolean = {
    st1.hasElement() || st2.hasElement()
  }

  def head ():T =
    if (st1.hasElement())
      st1.head()
    else
```

```
        st2.head()

    def tail ():Stream[T] =
      if (st1.hasElement())
        new Sequence[T](st1.tail(),st2)
      else
        st2.tail()
  }
```

When we create a `sequence()`, we give it two streams (that produce the same type of elements). The `Sequence` class records the current first stream and the current second stream. When we ask for an element, it returns an element from the first stream if one is available, otherwise, from the second. When taking the tail of the produced stream, it tries to take the tail of the first stream. If the first stream is exhausted, then it takes the tail of the second stream. Think about it some, read the code, meditate on it. Try drawing pictures to understand what is happening. This is a common pattern when working with streams.

```
    println("Sequence(T1,L3) = ")
    printAll(sequence(T1,L3))
```

produces output:

```
Sequence(T1,L3) =
  33
  66
  99
  11
  22
  44
  33
  66
  99
```

Let's complicate things even further. Right now, all our gadgets have returned elements of the same type as the elements of the streams they are connected to. Let's write a gadget that returns something different. In particular, let's write a gadget that connects to two streams and that returns *pairs* of elements, one taken from each stream. We need to figure out what happens when one of the streams the gadget is connected to exhausts before the other does. For simplicity, we make the gadget exhaust itself when one of the streams it is connected to has no more elements.

```
  def zip[T,U] (st1:Stream[T],st2:Stream[U]):Stream[Pair[T,U]] =
    new Zip[T,U](st1,st2)
```

```
class Zip[T,U] (st1:Stream[T],st2:Stream[U]) extends Stream[Pair[T,U]]
 {

   def hasElement ():Boolean = {
     st1.hasElement() && st2.hasElement()
   }

   def head ():Pair[T,U] = Pair.create(st1.head(),st2.head())

   def tail ():Stream[Pair[T,U]] = new Zip[T,U](st1.tail(),st2.tail())
}
```

This code uses the polymorphic class `Pair` I gave you in the last lecture. Again, we create a `zip()` gadget by passing in two streams to connect to. When calling `head()`, we read the head of both streams we're connected to, and create a pair that we return. When calling `tail()`, we simply take the tail of the underlying streams, creating a new `Zip` gadget connected to the newly created streams. There's actually less bookkeeping in `Zip` than in `Sequence`.

```
    println("Zip(Constant(99),IntsFrom(3)) = ")
    printN(zip(constant(99),intsFrom(3)),20)
    println("Zip(Constant(99),T1) = ")
    printN(zip(constant(99),T1),20)
```

procudes output:

```
Zip(Constant(99),IntsFrom(3)) =
  (99,3)
  (99,4)
  (99,5)
  (99,6)
  (99,7)
  (99,8)
  (99,9)
  (99,10)
  (99,11)
  (99,12)
  (99,13)
  (99,14)
  (99,15)
  (99,16)
  (99,17)
  (99,18)
  (99,19)
  (99,20)
  (99,21)
  (99,22)
  ...
Zip(Constant(99),T1) =
  (99,33)
  (99,66)
  (99,99)
  (99,11)
  (99,22)
  (99,44)
```

The above gadgets are all fairly straightforward, in the sense that they do not require tricks of any kinds, or any fields beyond the obvious ones, or any feature we have not seen yet. The following few examples require a bit more cleverness.

First, let's write a variant of the `zip()` gadget that connects to two streams and returns pairs of elements from those two streams. Instead of pairing the first elements together, then the second elements together, then the third elements together and so, however, we want to produce pairs of all the possible combinations of an element from the first stream and an element of the second stream. Call it a `cartesian()` gadget (inspired by the cartesian product of two sets in set theory).

```
def cartesian[T,U] (st1:Stream[T],st2:Stream[U]):Stream[Pair[T,U]] =
  zip(new Duplicates[T](st1,1,1),new Prefixes[U](st2,st2,1,1))


class Duplicates[T] (st:Stream[T],init:Int,curr:Int)
                                    extends Stream[T] {

  def hasElement ():Boolean = st.hasElement()

  def head ():T = st.head()

  def tail ():Stream[T] =
    if (curr > 1)
      new Duplicates(st,init,curr-1)
    else
      new Duplicates(st.tail(),init+1,init+1)
}
```

```
class Prefixes[T] (st:Stream[T], ost:Stream[T],
                   init:Int, curr:Int) extends Stream[T] {

  def hasElement ():Boolean = st.hasElement()

  def head ():T = st.head ()

  def tail ():Stream[T] =
    if (curr > 1)
      new Prefixes(st.tail(),ost,init,curr-1)
    else
      new Prefixes(ost,ost,init+1,init+1)
}
```

The trick here is to figure how to code the gadget in such a way that we are not stuck trying to pair everything in the second stream with the first element of the first stream, because then if the second stream is infinite then we never get to the point where we can pair the second element of the first stream with anything in the second stream, let along the third element of the first stream, or the fourth. Study the above code. Intuitively, `Duplicates` produces, from the stream `a b c d e ...`, the stream `a b b c c c d d d d e e e e e ...`, while `Prefixes` produces, from the stream `a b c d e`, the stream `a a b a b c a b c d a b c d e`. The final result works, though:

```
    println("Cartesian(T1,IntsFrom(1)) = ")
    printN(cartesian(T1,intsFrom(1)),20)
```

produces output:

```
Cartesian(T1,IntsFrom(1)) =
  (33,1)
  (66,1)
  (66,2)
  (99,1)
  (99,2)
  (99,3)
  (11,1)
  (11,2)
  (11,3)
  (11,4)
  (22,1)
  (22,2)
  (22,3)
  (22,4)
  (22,5)
  (44,1)
  (44,2)
  (44,3)
  (44,4)
  (44,5)
  ...
```

The next two gadgets operate on a stream by modifying it using a *function*, the way you've seen functions acting on lists in Fundies 1. The first gadget, `map()` takes a stream and a function that can transform the elements of that stream, and produces a new stream where each element from the connected stream has been transformed by the function.

```
def map[T,U] (st:Stream[T],f:(T)=>U):Stream[U] =
  new Map[T,U](st,f)

class Map[T,U] (st:Stream[T], f:(T)=>U) extends Stream[U] {

  def hasElement ():Boolean = st.hasElement()

  def head ():U = f(st.head())

  def tail ():Stream[U] = new Map(st.tail(),f)
}
```

The code is simple, once one knows how to specify that an argument to a function is itself a function. In Scala, the type of "function taking an A to a B" is written `(A)=>B`; the type of "function taking an A and a B to a C" is written `(A,B)=>C`, and so on. Thus, the type of `map()` says it expects a stream of `T`s and a function `(T)=>U` transforming `T`s to `U`s, and constructs a stream of `U`s. To call `map()`, we need to give it a function of the right type, like this:

```
    def sq (x:Int):Int = x*x
    println("Squares(IntsFrom(1)) = ")
    val squares:Stream[Int] = map(intsFrom(1),sq)
    printN(squares,20)
```

which produces output:

```
Squares(IntsFrom1) =
  1
  4
```

```
9
16
25
36
49
64
81
100
121
144
169
196
225
256
289
324
361
400
...
```

A related gadget is one that takes a predicate (that is, a function that returns a `Boolean`) and filters a stream, keeping only the elements for which the predicate returns `true`.

```
def filter[T] (st:Stream[T],p:(T)=>Boolean):Stream[T] =
  new Filter[T](st,p)

class Filter[T] (st:Stream[T], p:(T)=>Boolean) extends Stream[T] {

  private def findNext (s:Stream[T]):Stream[T] =
    if (s.hasElement()) {
      if (p(s.head()))
        s
      else
        findNext(s.tail())
    } else
      s

  def hasElement ():Boolean = findNext(st).hasElement()

  def head ():T = findNext(st).head()

  def tail ():Stream[T] = new Filter(findNext(st).tail(),p)
}
```

Note that here, the next element to return is not the first element of the connected stream (i.e., we do not try to maintain the invariant that the next element on the connected stream satisfies the predicate). Trying to do that leads to problems when the connected stream runs out of elements satisfying the predicate. Note that in the above implementation, if the connected stream has no more elements satisfying the predicate, and we ask the `filter()` gadget for the next element on the stream satisfying the predicate, the code will enter an infinite loop. This is unavoidable.

```
def digitTest (x:Int):Boolean = (x % 10 <= 5)
println("Squares with last digit <= 5) = ")
printN(filter(squares, digitTest, 20)
```

We now have enough ingredients to compute the stream of all prime numbers, using (a variant of) the Sieve of Eratosthenes. The sieve computes the list of prime numbers by essentially starting with all integers from 2 on, and then keeping 2 and removing all multiples of 2, then moving to the next unremoved integer (3), keeping it and removing all multiples of 3, moving to the next unremoved integer (5), keeping it and removing all multiples of 5, and so on. You can convince yourself that what you are left with is the stream of all prime numbers.

```
def sieve (st:Stream[Int]):Stream[Int] =
  new Sieve(st)

class Sieve (st:Stream[Int]) extends Stream[Int] {

  def hasElement ():Boolean = st.hasElement()

  def head ():Int = st.head()

  def tail ():Stream[Int] = {
    def notDivisibleBy (x:Int):Boolean = !(x % st.head() == 0)
    val multRemoved:Stream[Int] = filter(st.tail(),notDivisibleBy)
    new Sieve(multRemoved)
  }
}
```

We can now compute the stream of prime numbers by taking

```
val primes:Stream[Int] = sieve(intsFrom(2))
```

and indeed:

```
println("Primes = ")
printN(primes,12)
```

producing output:

```
Primes =
  2
  3
  5
  7
  11
  13
  17
  19
  23
  29
  31
  37
  ...
```

Note, however, that this is far from being an efficient way for computing prime numbers, as you can tell immediately by trying to print the first 100 elements of `primes`.