Lecture: Streams

A *stream* is a data structure that is very similar to a list. We write `Stream.Empty` instead of `Nil` to denote the empty stream, and we construct streams using `#::` instead of `::`. For example, the following expression is a stream of numbers 1, 2, 3:

```
1 #:: 2 #:: 3 #:: Stream.Empty
```

However, the crucial different between streams and lists is that the tail of a stream (i.e., the right-hand side of `#::` is only evaluated when it is *needed*. For example, the function in fig. 29.1a produces a list a numbers and prints to the screen on each recursive call. If we evaluate `fromToL(0, 100)`, we get a list of numbers and 100 lines of output since the entire list is created immediately.

In contrast, the function fig. 29.1b produces a stream of numbers. The expression `val astream = fromToS(0, 100)` produces only one line of output. Instead, each time we evaluate `astream.tail`, `astream.tail.tail`, and so on, we get one additional line of output. However, evaluating `astream.tail` a second time doesn't produce any output: i.e., the tail is not re-evaluated.

Infinite Streams

We can exploit the fact that the tail is only evaluated when needed to create an infinite stream of numbers:

```
def from(n: Int): Stream[Int] = n #:: from(n + 1)

val positives = from(1)
```

This cannot be done with lists! Morever, the Scala console shows exactly what has been evaluated:

```
scala> positives.tail.tail.tail.tail
res2: scala.collection.immutable.Stream[Int] = Stream(5, ?)
scala> positives
res3: Stream[Int] = Stream(1, 2, 3, 4, 5, ?)
```

You can work with infinite streams freely, so long as you don't try to access all the values (your program won't halt–there are inifinitely many values). E.g., `positives.length` does not terminate.

Stream Transformations

Streams have methods such as `.map`, `.filter`, `.flatMap`, etc. just like lists. However, they also produce streams. For example, the following expression produces a stream of even numbers:

```
val evens = from(1).filter(n => n % 2 == 0)
```

However, we have to careful when we use filter. If we filter an infinite stream with a predicate that matches none of the values in the stream, we will get an infinite loop. E.g., the following expression is an infinite loop:

```
from(1).filter(n => n < 0)
```

```
def fromToL(m: Int, n: Int): List[Int] = {
  println(s"within fromToL($m, $n)")
  if (m > n)  Nil
  else m :: fromToL(m + 1, n)
}
```

(a) Generating a list of numbers.

```
def fromToS(m: Int, n: Int): Stream[Int] = {
  println(s"within fromToS($m, $n)")
  if (m > n) Stream.Empty
  else m #:: fromTo(m + 1, n)
}
```

(b) Generating a stream of numbers.

Fibonnaci Numbers

The following code produces an infinite stream of fibonnaci numbers:

```
def nthFib(nPredPred: Stream[Int], nPred: Stream[Int]): Stream[Int] = {
  (nPredPred.head + nPred.head) #:: nthFib(nPredPred.tail, nPred.tail)
}

val fibs: Stream[Int] = 0 #:: 1 #:: nthFib(fibs, fibs.tail)
```