Computing with Infinite Sequences

Infinite Streams

You saw that all elements of a stream except the first one are computed only when they are needed to produce a result.

This opens up the possibility to define infinite streams!

For instance, here is the stream of all integers starting from a given number:

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

The stream of all natural numbers:

Infinite Streams

You saw that all elements of a stream except the first one are computed only when they are needed to produce a result.

This opens up the possibility to define infinite streams!

For instance, here is the stream of all integers starting from a given number:

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

The stream of all natural numbers:

```
val nats = from(0)
```

The stream of all multiples of 4:

Infinite Streams

You saw that all elements of a stream except the first one are computed only when they are needed to produce a result.

This opens up the possibility to define infinite streams!

For instance, here is the stream of all integers starting from a given number:

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

The stream of all natural numbers:

```
val nats = from(0)
```

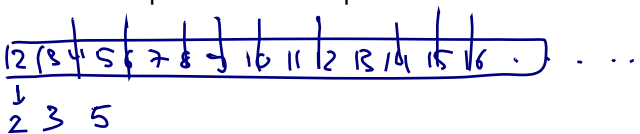
The stream of all multiples of 4:

```
nats map (_ * 4)
```

The Sieve of Eratosthenes

The Sieve of Eratosthenes is an ancient technique to calculate prime numbers.

The idea is as follows:



- Start with all integers from 2, the first prime number.
- Eliminate all multiples of 2.
- The first element of the resulting list is 3, a prime number.
- Eliminate all multiples of 3.
- Iterate forever. At each step, the first number in the list is a prime number and we eliminate all its multiples.

The Sieve of Eratosthenes in Code

Here's a function that implements this principle:

```
def sieve(s: Stream[Int]): Stream[Int] =
  s.head #:: sieve(s.tail filter (_ % s.head != 0))

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

To see the list of the first `N` prime numbers, you can write

```
(primes take N).toList
```

Back to Square Roots

Our previous algorithm for square roots always used a `isGoodEnough` test to tell when to terminate the iteration.

With streams we can now express the concept of a converging sequence without having to worry about when to terminate it:

```
def sqrtStream(x: Double): Stream[Double] = {
  def improve(guess: Double) = (guess + x / guess) / 2
  lazy val guesses: Stream[Double] = 1 #:: (guesses map improve)
  guesses
}
```

Termination

We can add `isGoodEnough` later.

```
def isGoodEnough(guess: Double, x: Double) =
  math.abs((guess * guess - x) / x) < 0.0001

sqrtStream(4) filter (isGoodEnough(_, 4))
```

Exercise:

Consider two ways to express the infinite stream of multiples of a given number `N`:

```
val xs = from(1) map (_ * N)

val ys = from(1) filter (_ % N == 0)
```

Which of the two streams generates its results faster?

- `from(1) map (_ * N)`
- `from(1) filter (_ % N == 0)`

Exercise:

Consider two ways to express the infinite stream of multiples of a given number N:

```
val xs = from(1) map (_ * N)

val ys = from(1) filter (_ % N == 0)
```

N = 3

1 2 3 4 5
3 6 9 12 ....

1 2 3 4 5 6 7 8 9 10 11 12
3 6

Which of the two streams generates its results faster?

- ● `from(1) map (_ * N)`
- ○ `from(1) filter (_ % N == 0)`