Computing with Infinite Sequences

Principles of Functional Programming

Infinite Lists

You saw that the elements of a lazy list are computed only when they are needed to produce a result.

This opens up the possibility to define infinite lists!

For instance, here is the (lazy) list of all integers starting from a given number:

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

The list of all natural numbers:

Infinite Lists

You saw that the elements of a lazy list are computed only when they are needed to produce a result.

This opens up the possibility to define infinite lists!

For instance, here is the (lazy) list of all integers starting from a given number:

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

The list of all natural numbers:

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

The list of all multiples of 4:

Infinite Lists

You saw that the elements of a lazy list are computed only when they are needed to produce a result.

This opens up the possibility to define infinite lists!

For instance, here is the (lazy) list of all integers starting from a given number:

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

The list of all natural numbers:

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

The list 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: LazyList[Int]): LazyList[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 lazy lists we can now express the concept of a converging sequence without having to worry about when to terminate it:

```
def sqrtSeq(x: Double): LazyList[Double] =
  def improve(guess: Double) = (guess + x / guess) / 2
  lazy val guesses: LazyList[Double] = 1 #:: guesses.map(improve)
  guesses
```

Termination

We can add `isGoodEnough` later.

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

sqrtSeq(2).filter(isGoodEnough(_, 2))
```

Exercise:

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

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

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

Which of the two lazy lists generates its results faster?

- O `from(1).map(_ * N)`
- O `from(1).filter(_ % N == 0)`
- O `there's no difference`

Exercise:

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

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

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

Which of the two lazy lists generates its results faster?

```
X         from(1).map(_ * N)
O         from(1).filter(_ % N == 0)
O         there's no difference
```