

Streams

Collections and Combinatorial Search

We've seen a number of immutable collections that provide powerful operations, in particular for combinatorial search.

For instance, to find the second prime number between 1000 and 10000:

```
((1000 to 10000) filter isPrime)(1)
```

This is *much* shorter than the recursive alternative:

```
def secondPrime(from: Int, to: Int) = nthPrime(from, to, 2)
def nthPrime(from: Int, to: Int, n: Int): Int =
  if (from >= to) throw new Error("no prime")
  else if (isPrime(from))
    if (n == 1) from else nthPrime(from + 1, to, n - 1)
  else nthPrime(from + 1, to, n)
```

Performance Problem

But from a standpoint of performance,

```
((1000 to 10000) filter isPrime)(1)
```

is pretty bad; it constructs *all* prime numbers between 1000 and 10000 in a list, but only ever looks at the first two elements of that list.

Reducing the upper bound would speed things up, but risks that we miss the second prime number all together.

Delayed Evaluation

However, we can make the short-code efficient by using a trick:

Avoid computing the tail of a sequence until it is needed for the evaluation result (which might be never)

This idea is implemented in a new class, the Stream.

Streams are similar to lists, but their tail is evaluated only *on demand*.

Defining Streams

Streams are defined from a constant `Stream.empty` and a constructor `Stream.cons`.

For instance,

```
val xs = Stream.cons(1, Stream.cons(2, Stream.empty))
```

They can also be defined like the other collections by using the object `Stream` as a factory.

```
Stream(1, 2, 3)
```

The `toStream` method on a collection will turn the collection into a stream:

```
(1 to 1000).toStream > res0: Stream[Int] = Stream(1, ?)
```

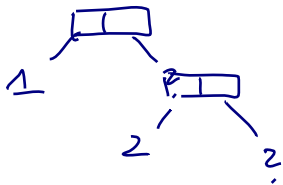


Stream Ranges

Let's try to write a function that returns `(lo until hi).toStream` directly:

StreamRange

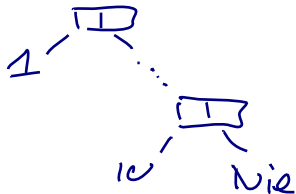
```
def streamRange(lo: Int, hi: Int): Stream[Int] =  
  if (lo >= hi) Stream.empty  
  else Stream.cons(lo, streamRange(lo + 1, hi))
```



Compare to the same function that produces a list:

```
def listRange(lo: Int, hi: Int): List[Int] =  
  if (lo >= hi) Nil  
  else lo :: listRange(lo + 1, hi)
```

listRange(1, 10)



Comparing the Two Range Functions

The functions have almost identical structure yet they evaluate quite differently.

- ▶ `listRange(start, end)` will produce a list with `end - start` elements and return it.
- ▶ `streamRange(start, end)` returns a single object of type `Stream` with `start` as head element.
- ▶ The other elements are only computed when they are needed, where “needed” means that someone calls `tail` on the stream.

Methods on Streams

Stream supports almost all methods of List.

For instance, to find the second prime number between 1000 and 10000:

```
((1000 to 10000).toStream filter isPrime)(1)
```


Stream Cons Operator

The one major exception is `::`.

`x :: xs` always produces a list, never a stream.

There is however an alternative operator `#::` which produces a stream.

$$x \#:: xs == \text{Stream.cons}(x, xs)$$

`#::` can be used in expressions as well as patterns.

Implementation of Streams

The implementation of streams is quite close to the one of lists.

Here's the trait Stream:

```
trait Stream[+A] extends Seq[A] {  
  def isEmpty: Boolean  
  def head: A  
  def tail: Stream[A]  
  ...  
}
```

As for lists, all other methods can be defined in terms of these three.

Implementation of Streams(2)

Concrete implementations of streams are defined in the Stream companion object. Here's a first draft:

```
object Stream {  
  def cons[T](hd: T, tl: => Stream[T]) = new Stream[T] {  
    def isEmpty = false  
    def head = hd  
    def tail = tl  
  }  
  val empty = new Stream[Nothing] {  
    def isEmpty = true  
    def head = throw new NoSuchElementException("empty.head")  
    def tail = throw new NoSuchElementException("empty.tail")  
  }  
}
```

Stream.empty ≈ Nil
Stream.cons ≈ ::

Difference to List

The only important difference between the implementations of `List` and `Stream` concern `tl`, the second parameter of `Stream.cons`.

For streams, this is a by-name parameter.

That's why the second argument to `Stream.cons` is not evaluated at the point of call.

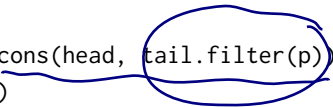
Instead, it will be evaluated each time someone calls `tail` on a `Stream` object.

Other Stream Methods

The other stream methods are implemented analogously to their list counterparts.

For instance, here's filter:

```
class Stream[+T] {  
  ...  
  def filter(p: T => Boolean): Stream[T] =  
    if (isEmpty) this  
    else if (p(head)) cons(head, tail.filter(p))  
    else tail.filter(p)  
}
```

Hand-drawn blue annotations highlighting the recursive call in the filter method. A blue circle is drawn around the expression `tail.filter(p)` in the `else if` branch. A blue line is drawn under the `cons(head, tail.filter(p))` expression, extending from the `cons` function to the end of the line.

Exercise

Consider this modification of `streamRange`.

```
def streamRange(lo: Int, hi: Int): Stream[Int] = {  
  print(lo+" ")  
  if (lo >= hi) Stream.empty  
  else Stream.cons(lo, streamRange(lo + 1, hi))  
}
```

When you write `streamRange(1, 10).take(3).toList`
what gets printed?

- 0 Nothing
- 0 1
- 0 1 2 3
- 0 1 2 3 4
- 0 1 2 3 4 5 6 7 8 9

Exercise

Consider this modification of `streamRange`.

```
def streamRange(lo: Int, hi: Int): Stream[Int] = {  
  print(lo+" ")  
  if (lo >= hi) Stream.empty  
  else Stream.cons(lo, streamRange(lo + 1, hi))  
}
```

When you write `streamRange(1, 10).take(3).toList`
what gets printed?

- 0 Nothing
- 0 1
- 1 2 3
- 0 1 2 3 4
- 0 1 2 3 4 5 6 7 8 9

