Lazy Lists

Principles of Functional Programming

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 then throw Error("no prime")
  else if isPrime(from) then
    if n == 1 then 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 elements of a sequence until they are needed for the evaluation result (which might be never)*

This idea is implemented in a new class, the `LazyList`.

Lazy lists are similar to lists, but their elements are evaluated only *on demand*.

Defining Lazy Lists

Lazy lists are defined from a constant `LazyList.empty` and a constructor `LazyList.cons`.

For instance,

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

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

```
LazyList(1, 2, 3)
```

The `to(LazyList)` method on a collection will turn the collection into a lazy list:

```
(1 to 1000).to(LazyList)      > res0: LazyList[Int] = LazyList(<not computed>)
```

LazyList Ranges

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

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

Compare to the same function that produces a list:

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

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.
- `lazyRange(start, end)` returns a single object of type `LazyList`.
- The elements are only computed when they are needed, where "needed" means that someone calls `head` or `tail` on the lazy list.

Methods on Lazy Lists

`LazyList` supports almost all methods of `List`.

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

```
LazyList.range(1000, 10000).filter(isPrime)(1)
```

LazyList Cons Operator

The one major exception is `::`.

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

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

```
x #:: xs  ==  LazyList.cons(x, xs)
```

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

Implementation of Lazy Lists

The implementation of lazy lists is quite subtle.

As a simplification, we consider for now that lazy lists are only lazy in their `tail`. `head` and `isEmpty` are computed when the lazy list is created.

This is not the actual behavior of lazy lists, but makes the implementation simpler to understand.

Here's the trait `TailLazyList`:

```
trait TailLazyList[+A] extends Seq[A]:
  def isEmpty: Boolean
  def head: A
  def tail: TailLazyList[A]
  ...
```

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

Implementation of Lazy Lists (2)

Concrete implementations of lazy lists are defined in the `TailLazyList` companion object. Here's a first draft:

```
object TailLazyList:
  def cons[T](hd: T, tl: => TailLazyList[T]) = new TailLazyList[T]:
    def isEmpty = false
    def head = hd
    def tail = tl
    override def toString = "LazyList(" + hd + ", ?)"

  val empty = new TailLazyList[Nothing]:
    def isEmpty = true
    def head = throw NoSuchElementException("empty.head")
    def tail = throw NoSuchElementException("empty.tail")
    override def toString = "LazyList()"
```

Difference to List

The only important difference between the implementations of `List` and (simplified) `LazyList` concern `tl`, the second parameter of `TailLazyList.cons`.

For lazy lists, this is a by-name parameter.

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

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

Other LazyList Methods

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

For instance, here's `filter`:

```
extension [T](xs: TailLazyList[T])
  def filter(p: T => Boolean): TailLazyList[T] =
    if isEmpty then xs
    else if p(xs.head) then cons(xs.head, xs.tail.filter(p))
    else xs.tail.filter(p)
```

Exercise

Consider this modification of `lazyRange`.

```
def lazyRange(lo: Int, hi: Int): TailLazyList[Int] =
  print(lo+" ")
  if lo >= hi then TailLazyList.empty
  else TailLazyList.cons(lo, lazyRange(lo + 1, hi))
```

When you write `lazyRange(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 `lazyRange`.

```
def lazyRange(lo: Int, hi: Int): TailLazyList[Int] =
  print(lo+" ")
  if lo >= hi then TailLazyList.empty
  else TailLazyList.cons(lo, lazyRange(lo + 1, hi))
```

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

O	Nothing
O	1
X	1 2 3
O	1 2 3 4
O	1 2 3 4 5 6 7 8 9