14 Laziness

Consider the following problem: I'm asking you to take the list of the first 100 natural numbers (starting from 1), square each of them, add the resulting list of integers to itself three times, subtract 1 from every element of the resulting list, and then give me the 5th element of the result. You could implement these operations quite easily using any implementation of lists, and get the result equally easily. But you'll be doing an awful lot of work, considering that all you really want is the 5th element of the resulting list.

If you think about it for a second, all you really need to do is take the 5th element of the initial list (that is, 5), square it to get 25, add it to itself three times to get 75, and then subtract 1 to get 74. No waste work, no need to worry about applying expensive list operations to elements of the list that we'll never need.

So how can we implement lists, or in general data structures, where we avoid doing the work to compute values that we will end up not needing? The general approach for doing this is called *lazy construction*, or just *laziness*, and basically says that we do not do any work when we construct objects, we do all the work when we compute from objects (that is, observe values). The opposite of lazy construction is *eager construction*.

To get lazy construction for an ADT, the basic trick is to make all operations (except for operations that extract data from the ADT) into creators. Creators do not do any work. Then it's just a matter of defining the remaining operations that extract data, and those will do the work, for each of the creators we define. Let's look at an example.

Laziness is very powerful. In particular, it lets us work easily with infinite data. The classic example of infinite data is streams, which you can think of infinite lists. Of course, you cannot simply represent an infinite list directly by listing all its elements. Instead, a stream can be thought of simply as a "promise" to deliver its elements, if you ask for them. If you ask for the first 10 elements of the stream, it will compute them, and then give them to you. If you ask for the 200th element of the stream, it will compute it and give it to you. Until you ask for it, though, it is not explicitly represented. We can also create new streams from old streams, and here again, we do not do any work at creation time, only when we ask for elements of the stream.

We implement the following interface for streams of integers — I'm leaving it as an exercise to define an ADT for polymorphic streams:

```
static Stream constant (int)
static Stream intsFrom (int)
```

```
static Stream cons (int, Stream)
static Stream sum (Stream, Stream)

int head ()
Stream tail ()
```

where `constant(v)` creates a stream delivering always the same value `v`, `intsFrom(v)` creates a stream of all integers starting from a given number `v`, `cons(v,s)` creates a stream that starts with a given value `v` before delievring the elements of stream `s`, and `sum(s1,s2)` creates a stream of elements that are the pointwise sums of the elements of `s1` and `s2`. Here is the specification of the two operations `head` and `tail` that return the first element of a stream, and the stream obtained by removing the first element. Note that all the work is done in those two operations:

```
constant(i).head() = i
intsFrom(i).head() = i
cons(a,S).head() = a
sum(S1,S2).head() = S1.head() + S2.head()

constant(i).tail() = constant(i)
intsFrom(i).tail() = intsFrom(i+1)
cons(a,S).tail() = S
sum(S1,S2).tail() = sum(S1.tail(),S2.tail())
```

It is completely straightforward to implement the above ADT using the design pattern we know:

```
public abstract class Stream {

    public static Stream constant (int i) {
        return new StreamConstant(i);
    }

    public static Stream intsFrom (int i) {
        return new StreamIntsFrom(i);
    }

    static Stream cons (int i, Stream s) {
        return new StreamCons(i,s);
    }

    static Stream sum (Stream s1, Stream s2) {
        return new StreamSum(s1,s2);
```

```
    }

    public abstract int head ();
    public abstract Stream tail ();
}


class StreamConstant extends Stream {

    private int cnst;
    public StreamConstant (int i) {
        cnst = i;
    }

    public int head () {
        return cnst;
    }

    public Stream tail () {
        return Stream.constant(cnst);
    }

}


class StreamIntsFrom extends Stream {

    private int cnst;
    public StreamIntsFrom (int i) {
        cnst = i;
    }

    public int head () {
        return cnst;
    }

    public Stream tail () {
        return Stream.intsFrom(cnst+1);
    }

}
```

```
class StreamCons extends Stream {

    private int first;
    private Stream rest;

    public StreamCons (int i, Stream s) {
        first = i;
        rest = s;
    }

    public int head () {
        return first;
    }

    public Stream tail () {
        return rest;
    }

}


class StreamSum extends Stream {

    private Stream first;
    private Stream second;

    public StreamSum (Stream s1, Stream s2) {
        first = s1;
        second = s2;
    }

    public int head () {
        return first.head() + second.head();
    }

    public Stream tail () {
        return Stream.sum(first.tail(),second.tail());
    }

}
```

Adding a new operation to a lazy ADT corresponds to adding a new creator for that operation, and defining how the observers work on that creator. For instance, suppose we want to have a `filter` operation on streams, that keep only those elements that satisfy a certain predicate. A predicate is a function that takes an element of the stream and that returns a Boolean true or false.

```
  static Stream filter (PredicateFunction, Stream)
```

As far as implementing predicates, we use the same technique we used when working with map and reduce operations, and define an interface for predicates:

```
public interface PredicateFunction {
  public boolean apply(int v);
}
```

Thus, the only thing we can do with a predicate is apply it to a value. Here is an example of a predicate over integers, that returns true if and only if an integer is even:

```
public class Even implements PredicateFunction {
  private Even () {}

  public static Even function() {
    return new Even();
  }

  public boolean apply (int v) {
    return (v % 2 == 0);
  }
}
```

As a slightly different example, consider the following predicate, that checks if a number is not divisible by a given number (supplied when we construct the predicate).

```
public class NotDivisibleBy implements PredicateFunction {
  private int divisor;
  private NotDivisibleBy (int i) {
    divisor = i;
  }

  public static NotDivisibleBy function (int i) {
    return new NotDivisibleBy(i);
  }
```

```
  public boolean apply (int v) {
    return (!(v % divisor == 0));
  }
}
```

The specification for `filter` is as follows:

```
  filter(pf,s).head()
     = s.head()                      if pf.apply(s.head())=true
     = filter(pf,s.tail()).head()    otherwise
  filter(pf,s).tail()
     = filter(pf,s.tail())           if pf.apply(s.head())=true
     = filter(pf,s.tail()).tail()    otherwise
```

Think about this for a bit. Note also that this specification tells us that something like `filter(false,s).head()`, where `false` is a predicate that always returns false, is not equal to anything — in practice, this means that the implementation is free to do whatever it wants, and in the implementation below, it will get stuck in an infinite loop. We add the creator to the `Stream` class:

```
public class Stream {
  ...

  static Stream filter (PredicateFunction pf, Stream s) {
    return new StreamFilter(pf,s);
  }

  ...
}
```

We easily implement the concrete class corresponding to the `filter` creator:

```
class StreamFilter extends Stream {

    private Stream underlying;
    private PredicateFunction predicate;

    public StreamFilter (PredicateFunction pf, Stream s) {
        underlying = s;
        predicate = pf;
    }
```

```
    public int head () {
        if (predicate.apply(underlying.head()))
            return underlying.head();
        else
            return Stream.filter(predicate,underlying.tail()).head();
    }

    public Stream tail () {
        if (predicate.apply(underlying.head()))
            return Stream.filter(predicate,underlying.tail());
        else
        return Stream.filter(predicate,underlying.tail()).tail();
    }

}
```

Let's try out some sample streams. Let's define a function to print the first n elements of a stream of integers:

```
  public static void printFirstN (Stream s, int n) {
    Stream temp = s;
    for (int i = 0; i < n; i++) {
      System.out.print(temp.head() + " ");
      temp = temp.tail();
    }
    System.out.println();
  }
```

We can now try out something such as:

```
  Stream s1 = Stream.intsFrom(1);
  Stream s2 = Stream.filter(Even.function(),s1);

  printFirstN(Stream.sum(s1,s2),20);
```

This prints the first 20 elements of the stream obtained by summing the stream starting from 1 (i.e., `1 2 3 4 5 ...`) and the stream starting from 1 where we've removed all the non-even elements (i.e., `2 4 6 8 10 ...`). This yields:

```
3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57 60
```

Make sure you understand what is happening.

As a cute example, we can try to compute, using (a variant of) the Sieve of Eratosthenes, the stream of all prime numbers. The sieve computes the list of prime numbers by essentially starting with all integers from 2 on, and then keeping 2 and removing all multiples of 2, then moving to the next unremoved integer (3), keeping it and removing all multiples of 3, moving to the next unremoved integer (5), keeping it and removing all multiples of 5, and so on. You can convince yourself that what you are left with is the stream of all prime numbres.

Here is the operation `sieve`, expressed as a constructor:

```
  static Stream sieve (Stream)
```

with specification:

```
  sieve(s).head() = s.head()
  sieve(s).tail() = sieve(filter(NotDivisibleBy.function(s.head()),s.tail()))
```

Again, we add the creator to the `Stream` class:

```
public class Stream {
  ...

  static Stream sieve (Stream s) {
    return new StreamSieve(s);
  }

  ...
}
```

and implement the corresponding concrete subclass:

```
class StreamSieve extends Stream {

    private Stream stream;

    public StreamSieve (Stream s) {
        stream = s;
    }

    public int head () {
        return stream.head();
    }
```

```
    public Stream tail () {
        Stream multRemoved =
            Stream.filter(NotDivisibleBy.function(stream.head()),
                          stream.tail());
        return Stream.sieve(multRemoved);
    }
}
```

We can now compute the stream of prime numbers:

```
Stream primes = Stream.sieve(Stream.intsFrom(2));
```

By looking at the first N elements of the stream, we can get all the primes we want, such as:

```
printFirstN(primes,20);
```

with output:

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71
```

Note, however, that this is far from being an efficient way for computing prime numbers, as you can tell immediately by trying to execute `printFirstN(primes,100);`.