I was recently introduced to lazy computation in the context of streams—pretty cool stuff. It’s probably helpful to first understand streams.

Streams

A stream is an infinite data type—a “stream” of values that are generated by some computation, similar to a queue except that it is by definition infinite. I found it helpful to think of a stream as a function: the nth element in the stream can be seen as f(n).

**A stream can really be any set of values**. You could have a stream of 1s (i.e., in which f(n) = 1 for all n) or a stream of the natural numbers (in which the first element is 1 and every element n is equal to the previous element plus 1) or a stream of primes or a stream of…

But it’s essential that you can always get the *next* element in a stream, i.e., it is infinite.

Lazy computation

But there’s a problem: if I just define an infinite stream of natural numbers, won’t my program run till infinity, computing every natural number (an unbounded set) and placing it in the stream? In other words: **how can I abstract an infinite stream without actually computing every value?**

The answer: **lazy (a.k.a. deferred) computation**. The motivation here is that we don’t really need to compute the nth natural number until we need to use it—so why not just wait and be lazy about it?

Lazy computation tells the compiler to put off the evaluation of some expression (or set of expressions) until it is absolutely essential that the computation be performed, at which point the computation is **forced**.

In this way, we can define a stream via a function for each element and avoid actually computing every single element; but, when the element is requested, we can *then* perform the necessary computation. **Thus, we avoid infinite loops and successfully implement the desired abstraction.**

An Example: powers of two

Consider a stream of numbers that represent all the powers of two, i.e. < 1, 2, 4, 8, 16…>.

We can actually define this stream recursively: each new element of the stream is just the current stream, with every element multiplied by 2.

Visually:

* **< 1, 2, 4, 8, 16…>** = < 1, 2 • **<1, 2, 4, 8…>**>
* < 1, 2, 4, 8, 16…> = < 1, 2 • <1, 2 • **<1, 2, 4…>**>>
* Etc., etc.

To be clear: we start with a stream with 1 as the first element; the next element is just the current stream (at this point, with a single element of 1) multiplied by 2, i.e., our new stream is <1, 2, …>. We repeat this recursive procedure, multiplying every element of the stream by 2 for each new element generated.

Streams, in code

Lets view the above example (powers-of-two stream) in code. Assume:

1. We have some map expression that applies a function t every element in the stream. For example, if each element of our stream s is generated by f(n), then map g s results in a steam s’ with each element generated by g(f(n)).
2. ‘a streams are represented as pairs of (‘a, ‘a stream).

*let rec s = (1, map (fun x -> x \* 2) s);*

Oops—this code will run us into an infinite loop, as s will be recursively recomputed with no termination.

This is exactly the problem we described above; lets fix it with lazy computation.

*let rec s = (1, lazy (map (fun x -> x \* 2) s));*

Now, we don’t have to perform any computation at all! The stream is simply initialized with no further actions. Yet, if we request the nth element, the stream will perform all the necessary computation and return the element as desired.