A VERY STRANGE SIEVE

or: How I Learned to Stop Worrying and Love the Stream

Fundamentals of Programming

Mini-Project #1

Brian Scott Bohme

Table of Contents

Introduction **3**
Understanding the Stream **3**
The Strikeout Function **4**
Partial Sums and Compositions of Functions **5**
The Wonderful Sieve **6**
Generating Tests **7**
Playing with Inputs **8**
Parting Thoughts **10**

Introduction

When I first started reading about this mini-project, I was honestly mildly enthused. I had done filter-like operations on other data types such as lists before, so what would make this "sieve" so interesting? I thought the result would be painfully arbitrary, so I'm humbled to say that I was very, very wrong. I have now come to witness an amazing phenomenon with the mutation of streams that I still have trouble wrapping my mind around as I write this paper. Who knew that by filtering elements and taking the partial sums you could obtain… Well, I don't want to spoil the surprise now. Let's start from the beginning, shall we?

Understanding the Stream

The first part was fully understanding what a stream is, and what is happening when I construct a stream. A stream is an ordered data structure that is lazy in it's construction. Think of a list: you have the head of the list, then the tail, which is the rest of the elements. Imagine if you knew the head, but didn't compute the tail elements until forced to (i.e. accessing the second, third, forth, etc. element of the list). This is what's considered a lazy list, while an infinite lazy list is a stream!

```
type 'a xstream = Xcons of 'a * (unit -> 'a xstream);;
```

A stream uses a function with a unit as a parameter to delay the computation of the rest of the stream. When the force function is called upon, it will calculate the next element the stream.

```
let force thunk= thunk ();;
```

With that knowledge of streams established, it was pretty easy to create various streams to test in the functions I was going to write. I first wrote a function to generate the stream of natural numbers.

```
let rec stream n =
    Xcons (n, fun ( ) -> stream (n + 1));;
```

The tail function, parameterized by a unit, has a recursive call on the original function (with an incremented starting value input) as an output. Thus, every time the tail is thunked, you get the proceeding natural number in the stream. I used this same logic to construct other stream-generating functions, just changing the unit function's output.

The Strikeout Function

After constructing an arsenal of stream-generating functions, I had to write the strike-out function. I knew immediately that the plan was to iterate through the original stream, essentially reconstructing it identically, except every *n+2* element deleted. To remove an element, I knew I just had to not include it in the recursive call after being thunked. Therefore, the resulting stream would be the previously iterated stream pointing to the tail of the current stream, instead of the head element. As with lists, I knew the easiest and most efficient way to accomplish this would be with the help of a counter. The counter should start at the input value, and decrement with every recursive call if not equal to -1. If equal to -1, then it is the *n+2* element, so I'd want to "delete" it. In this case, you just reset the counter to the original, and perform the cursive call on only the tail of the stream. Easy-peazy! The implementation of this was very straightforward after understanding the structural recursion of the strike out function:

```
let striker xs_init f =
    let rec filter xs f_counter =
        match xs with
            | Xcons (hd, tl) ->
            if f_counter = -1
```

```
            then filter (force tl) f
        else Xcons (hd, fun () -> filter (force tl) (f_counter - 1))
    in if f < 0
    then raise (Negative_int "striker")
    else filter xs_init f;;
```

By matching the stream, I was able to easily access the head and the tail of the stream, which is imperative in successfully implementing the recursion.

Partial Sums and Compositions of Functions

The structure of the following functions followed closely from the strikeout function. For example, the partial sums function followed a similar pattern with the recursion: do something with the head, and then recurse on the tail by thunking. This time however there was no conditional statements. All we needed was to add each head together, and use that accumulation for the elements of the resulting stream. This was easily done with an accumulator starting at zero. Adding the head to the accumulator, making that the new head, and repeating with the tail is the simple structural recursion that this function follows:

```
let partial_sums xs_init =
    let rec visit xs acc =
        match xs with
        | Xcons (hd, tl) ->
            let partial_sum = acc + hd
            in Xcons (partial_sum, fun () -> visit (force tl)
partial_sum)
```

```
    in visit xs_init 0;;
```

With sieve and partial sums function defined, the final two questions were simply a composition of the preceding functions. Stike and partial sum does exactly what the function name says: it first sieves every *n+2* element, which is then put into the partial sum function:

```
let strike_and_partial_sum xs_init n =
    partial_sums (striker xs_init n);;
```

No mysteries there. The final sieve function is almost the same as this, expect you are calling the sieve and partial sum *k* times. Adding a counter easily accomplishes this task:

```
let sieve xs_init k =
    let rec visit xs n =
        let xs_new = strike_and_partial_sum xs n in
        if n = 0
        then xs_new
        else visit xs_new (n - 1)
    in visit xs_init k;;
```

The Wonderful Sieve

And here we are, at the heart of the problem. When I first ran the sieve code, I was shocked at the result. With no multiplication, the stream of the *k* powers of the natural numbers was obtained. It's quite beautiful actually, because the intermediate results seem so arbitrary, but the final result is *iconic*. I wrestled with how to explain this phenomenon for quite some time. I tried to visualize it happening. First with the *k = 2* case:

1 ~~0~~ 0 ~~0~~ 0 ~~0~~ 0 ~~0~~ 0 ~~0~~ 0… (starting with this stream…)

1 ~~1~~ 1 ~~1~~ 1 ~~1~~ 1 ~~1~~ 1 ~~1~~ 1… (became this, which is self explanatory…)

1 ~~2~~ 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11… (no surprises here…)

1 4 9 16 25 36 49 64 81 100 121… (and suddenly magic is produced!!!)

There is a great fluidity to the code and the recursion that I really appreciate, especially when writing it out. It actually reminds me—oddly enough—of ocean waves: coming up the shore, sweeping some sand and particles away, while pushing the rest of the debris together on the shoreline.

Generating Tests

Just to really make sure that the series of powers was being produced, I built a function to rigorously test the hypothesis. The function first generates the stream of natural numbers, and takes a random prefix from the stream (generated by Random.int 1000). The general power function is then mapped over the list, which is a painless way to generate a list of successive powers. This is compared to the same length prefix of our sieve function:

```
let k_powers_test k =
      let n = Random.int 1000 in
      prefix (sieve zeros k) n
      =
      List.map (power k) (prefix naturals1 n);;
```

Sure enough, all the unit tests 1 through 10 pass. Knowing that this is truly what is happening with the sieve function gave me some liberty to play around with the inputs. What if I put inputted the stream of all 1's? Well, I knew from doing the recursion by hand above that the *k-1* output of the stream is the stream of all 1's. Therefore, I hypothesized that running the sieve function with the stream of 1's will produce the stream of the *k+1* powers of natural numbers.

And following this logic, I knew that if I put in the stream of natural numbers, the result would be the stream of the *k+2* powers of the natural numbers. These results were intuitive for me, but what if we changed the first number of the original stream, such that the input was 2, 0, 0, 0… or 10, 0, 0, 0…? I started with the case of the two-then-zeros stream. When I first saw the results, I didn't recognize a pattern…

```
# prefix (sieve two_then_zeros 2) 15;;
- : int list = [2; 8; 18; 32; 50; 72; 98; 128; 162; 200; 242; 288;
338; 392; 450]
# prefix (sieve two_then_zeros 3) 15;;
- : int list = [2; 16; 54; 128; 250; 432; 686; 1024; 1458; 2000; 2662;
3456; 4394; 5488; 6750]
```

"2, 8, 18" I thought, "what's so special about '2, 8, 18'…". Then it dawned on me: it's the *k*-powers of the natural numbers again, but all multiplied by 2! A-ha! I whipped a test for this (similar to the previous one but mapped with a doubling function as well), and it corroborated my hypothesis. Surely, the ten-then-two series should do the same but with each element multiplied by 10. A simple test affirmed this—now we're on a roll.

Playing with Inputs

I wanted to observe what happens when more *interesting* streams were put as inputs. So I tested the stream of odd numbers (1, 3, 5, 7…). These were the first couple of results I got:

```
# let test = striker naturals1 0;;
val test : int xstream = Xcons (1, <fun>)
# prefix (sieve test 0) 10;;
- : int list = [1; 6; 15; 28; 45; 66; 91; 120; 153; 190]
```

```
# prefix (sieve test 1) 10;;
- : int list = [1; 12; 45; 112; 225; 396; 637; 960; 1377; 1900]
# prefix (sieve test 2) 10;;
- : int list = [1; 24; 135; 448; 1125; 2376; 4459; 7680; 12393; 19000]
# prefix (sieve test 3) 10;;
- : int list = [1; 48; 405; 1792; 5625; 14256; 31213; 61440; 111537;
190000]
```

The *k = 0* stream is interesting, because it is the stream of odd numbers, but each element is multiplied by its index. So 1 is multiplied by 1, 3 is doubled, 5 is tripled, 7 is quadrupled, etc. This pattern continues with the rest of streams! To test this, I programmed a simple function that uses an accumulator to multiple an element in a list by its index in the list:

```
let rec index_multiplier lst_init =
      let rec visit lst k =
            match lst with
            | [] -> lst
            | h::t -> (k * h) :: (visit t (k + 1))
      in visit lst_init 1;;
```

Afterwards, I used a similar test function that I've been using beforehand, but mapping the new function over it *n* times:

```
let rec index_multiplier lst_init =
      let rec visit lst k =
            match lst with
            | [] -> lst
            | h::t -> (k * h) :: (visit t (k + 1))
```

```
    in visit lst_init 1;;


let rec repeated_map lst f n =
    if n = -1 then lst
    else repeated_map (f lst) f (n - 1);;


let odd_stream_sieve k =
    let n = Random.int 1000 in
    prefix (sieve odds k) n
    =
    repeated_map (prefix odds n) index_multiplier k;;
```

Crossed my fingers… and it passed! And trying the same with an even stream passed as well. Lucky day!

Parting Thoughts

At the end of the day, it was extremely satisfying seeing these results come about. We obtained results that usually are only associated with multiplication with a matter of two functions. It is hard to wrap my mind around, but watching it happen in the terminal is tremendously rewarding. I'm glad I kept testing different cases, because I initially just stopped with the 1, 0, 0… sieve case, which I thought was interesting on its own. But seeing how different inputs produce such recognizable and intuitive results is even more thrilling. At the end of the project, I feel two principals have emerged. Firstly, my understanding and visualization of recursion has been strengthened, which I credit to the fluidity of the sieve function. However, a

deeper curiosity for pattern making and recognition has also emerged. How does this result relate to something like a Fibonacci Series? What other seemingly arbitrary mutations to a starting stream can produce such wonderful and intelligent results? These are questions I look forward to exploring while I, in the words of Dr. Strangelove, continue to "stop worrying and love the stream" even more.