Streams in Scheme*

Vishal Talwar

April 28, 2008

1 Prerequisites

Before embarking on this read, you should be familiar with:

- basic list operations: `car` , `cdr` , `cons` , `list-ref`
- defining and using (higher-order) functions
- recursion

Thats it! Now, we've got some work to do, so put on your sweatbands and read on.

2 Let's define "stream"

A stream is a potentially infinite list. This concept may sound a little out there at first, but we reason about the infinite all the time. For example, say you ask somebody a question. You can consider their response a stream. You don't know how long they might talk for (and let's face it, some people do seem to go on forever). More realistically, consider the positive numbers, a set very convenient for us to represent as an infinite list:

```
1 2 3 4 5 6 7 8 ...
```

That's nice, but this isn't really Scheme yet. How about this:

```
'(1 2 3 4 5 6 7 8 ...)
```

Is our job done? No, because '...' has no meaning in Scheme, although its presence makes a world of difference to us. It implies that you should continue the pattern to generate more elements of the stream. What's the pattern? Well, we know the stream begins with a 1. Given an element in the stream, how do we get the next one?

```
(define (next-integer n)
  _________________)
```

*Material based on §3.5 of SICP

That's simple. We just need to add one to the number to get the next number. This combination of a starting element and a function to get from one element to the next is just one way of thinking of a stream. You'll soon see that this is not the most general way of thinking of them, but it's a very useful one nonetheless. You'll also see that in order to work with streams in a finite amount of computer memory, we'll have to employ a tactic similar to using the '...' to represent "the rest of the stuff".

Wait. Before we go on, there's one stone that has been left unturned in all of this. We said at the very beginning that a stream is *possibly* infinite, so when is a stream ever not, and why would we use a stream instead of a list for that? This question will be brushed aside for the rest of this document, because it actually is a lot more complex than it may seem. In fact, the functions we'll define will blatantly ignore the fact that streams can be finite. Not to worry, a better distinction between streams and lists will be made later on, and we'll see just how cool and useful infinite streams can be.

3 Let's (try to) make a stream

How do you like the Fibonacci numbers? Do they form a stream? Well, we can start by writing them out:

```
'(0 1 1 2 3 5 8 ...)
```

That's nice, but this isn't legal Scheme code either, although it certainly *looks* like it could be a stream. Can we figure out the pattern behind the '...' in this case? You've most likely written a function that generates the *n*th Fibonacci number at some point, so let's do it just once more:

```
(define (fib n)
  (cond ((= n 1) __)
        ((= n 2) __)
        (else (+ (__________)
                 (__________)))))
```

That gives us a way to create more and more Fibonacci numbers, so we must be closing in on a stream now! Let's try to create a stream of Fibonacci numbers, `fibs`, using the function we just defined:

```
(define fibs
  (cons (fib 0)
        (cons (fib 1)
              (cons (fib 2)
                     ...?
```

Hmm, that didn't work. When in the world can we stop writing this function? We ran into this very problem trying to write out the stream of positive integers as a list using `'()`. The issue is that we want `fibs` to be an infinite list of the Fibonacci numbers without having to explicity write out every member like we had started to. No problem. We have a great way of dealing with this all-too-common task, namely, recursion! Are you in the mood for some recursion? If so, read on:

```
1 (define fibs
2   (cons (fib 0) fibs))...?
```

That doesn't seem to be going anywhere either. Even though we got the first element, **`(fib 0)`**, if we **`cons`** that back onto **`fibs`** (line 2), what happens? Let's take a look:

```
   fibs
=> (cons (fib 0) fibs)
=> (cons (fib 0) (cons (fib 0) fibs))
=> (cons (fib 0) (cons (fib 0) (cons (fib 0) fibs)))
   ...
```

We just get a bunch of **`(fib 0)`**'s! The problem might seem obvious: **`fibs`** does not take any arguments, so why should we expect it to do anything different each time we expand it as we did above. In general, no matter how many **`fib`**s we **`cons`** on before the recursive call, we won't get what we want. Let's redefine the function to accept an argument *n* and generate all Fibonacci numbers beginning from the *n*th onwards. This sounds like it could be helpful, so let's try it:

```
(define (fibs-from n)
  (cons (fib n)
        (fibs-from (+ n 1))))
```

Don't pat yourself on the back just yet. Does this do what we wanted? Expand **`(fibs-from 0)`**:

```
(fib-from 0)
(cons (fib 0) (fib-from 1))
(cons (fib 0) (cons (fib 1) (fib-from 2))
(cons (fib 0) (cons (fib 1) (cons (fib 2) (fib-from 3))))
   ...
```

Ok, pat yourself on the back now. This does indeed look like an infinite list of Fibonacci numbers. Have you tried this in a Scheme interpreter? If not, try it now:

```
(fib-list 0)
=> ...
```

In short, you're not likely to see anything except an error message (if you wait long enough). What our recursive definition missing that recursive definitions usually have? A **base case**. **`(fibs-from 0)`** never stops expanding because there's no reason it should! We didn't give it cause to stop recursing, because we absolutely *want* it to be an infinite list.

This should worry you. We were so close to calling the output of **`(fibs-from n)`** a stream, but the barrier we just hit seems to be insurmountable. The good news is that it isn't. We will get to the magic shortly, but first, be sure to take a break or go over the examples again if it is unclear what a **stream** is or how **`fibs-from`** is like a stream, but not *quite*.

4 Let's pretend we have streams

4.1 Infinite recursion

Let's recap what went wrong when we tried to define a stream for the Fibonacci sequence. Try to say this in your own words before reading on.

We'd really like to say that `(fibs-from n)` is `(fib n)` consed onto `(fibs-from (+ n 1))`. This way, we can think of streams almost the same way we think of lists. However, this definition is *infinitely* recursive, because `(fibs-from (+ n 1))` will call `(fibs-from (+ n 2))` and that will, in turn, call `(fibs-from (+ n 3))`, and so on ad infinitum. For now, let's ignore this problem and pretend as though streams **just work**.

4.2 Diving into streams

So, now what? What can we do with these imaginary streams? Well, as we just said, it would be nice if we could treat them like lists. We'd like to `car` and `cdr` them in order to pull elements out of them, for starters. Let's imagine `s-car` and `s-cdr` functions that operate on streams the same way `car` and `cdr` do on lists, and test them on a couple of predefined streams. We'll call the stream of positive numbers `ints` and the stream of Fibonacci numbers `fibs`.

```
;; ints first
(s-car ints)
=> 1
(s-car (s-cdr ints))
=> 2
(s-car (s-cdr (s-cdr ints)))
=> 3

;; on to fibs
(s-car fibs)
=> 0
(s-car (s-cdr fibs))
=> 0
(s-car (s-cdr (s-cdr (s-cdr fibs))))
=> 2
(s-car (s-cdr (s-cdr (s-cdr (s-cdr (s-cdr fibs))))))
=> 5
```

Notice that we refrained from looking at the output of `(s-cdr` *stream*`)` directly. Don't worry about it for now. What we see is that we can `s-cdr` down a stream and pull elements out with `s-car` . This is a breakthrough! Convince yourself why before continuing. Hint: can we now work with infinite lists? It's (finally) time for some real exercises. We can build up some more stream operators out of the two we already have. Let's start simple.

1. Define a function `s-cadr`, which does to streams what `cadr` does to lists:

```
(define (s-cadr s)
  (______ (______ s)))
```

That was easy, right? Don't worry, there's more blanks in the next one.

2. Define a function `s-generate` that takes a stream and a number *n* and returns a **list** of *n* elements from that stream. This is an extremely useful function, so let's see it in action:

```
(s-generate ints 0)
=> ()
(s-generate ints 10)
=> (1 2 3 4 5 6 7 8 9 10)
(s-generate fibs 10)
=> (0 1 1 2 3 5 8 13 21 34)
```

```
(define (s-generate s n)
   (if (______) __
       (cons (_____ s)
             (s-generate ______ ______))))
```

We're all set now to turn the imagined into the real.

5 Let's touch base with reality

Recall the infinite recursion that was the root of all our problems. What if there was a way to tell Scheme:

> Hold on! Don't be so eager to compute the next element in the stream (i.e. the next Fibonacci number)! When I need it, I'll ask you for it, but you must promise me you'll compute it when I *do* ask.

If Scheme agreed to this, would we be able to define streams? Yes! To make this happen, let's define a new type of `cons` called `s-cons` [1] . This is a very special `cons` because when we say

```
(s-cons thing1 thing2)
```

Scheme promises that it will ignore `thing2` until we ask for it, and no sooner. What we are essentially saying is that

> Infinite recursion allows us to model an infinite stream, *provided that* we have some way of delaying the recursion until we ask for it to happen.

Another way to look at it is that

[1] `s-cons` is defined in `streams.scm`, so be sure to load the file before attempting to use it.

A list exists all at once. A stream is uncovered element by element as we ask for them.

This might be clear as daylight or it might be very confusing. Let's assume it's the latter. What does it even mean to "ask" Scheme for an element? Let's take a shot in the dark and try to create an extramely simple stream, the stream of ones, and "ask" for its elements:

```
(define ones
  (s-cons __ _____))
```

Wow. Is that all? Does this even work? Let's try some of the operations we're familiar with.

```
(s-car ones)
=> 1
(s-car (s-cdr ones))
=> 1
(s-generate ones 10)
=> (1 1 1 1 1 1 1 1 1 1)
```

Ah, so that's what we mean by "asking"! We ask for elements when we manipulate the stream to pull them out like we just did. But where did the infinite recursion disappear to? Did Scheme suddenly get lazy and take a break from our infinite antics? Not likely. It's merely upholding the request we made just a minute ago and waiting until we ask for the rest of the stream with an `s-cdr` before attempting to compute any more of it. Now, with the help of `s-cons` , we can make our own streams. Our example with `ones` was so simple we only used a single `s-cons` , but are we restricted to using a single `s-cons` to make streams? Certainly not. Let's try to come up with a useful stream that uses more than one of them, like the stream of alternating 0's and 1's:

```
(define zeroes-ones
  (s-cons __
          (s-cons __
                  zeroes-ones)))
```

Nice, that one proves we can use `s-cons` in any situation we use `cons` . The same way a `cons` creates a new list, `s-cons` creates a new **stream**.

Remember our troubles with the Fibonacci sequence above? The last thing we tried was something like this:

```
(define (fibs-from n)
  (cons (fib n)
        (fibs-from (+ n 1))))
```

That's right, we were using `cons` because we didn't know any better. We have a better tool now: `s-cons` . This should be easy to fix. How about the following:

```
(define (fibs-from n)
  (_______ (fib n)
        (fibs-from (+ n 1))))
```

That most definitely looks right, but how can we get the entire Fibonacci sequence (`fibs` from before) without that pesky argument *n* getting in the way? Give it a go:

```
(define fibs _______________________)
```

Treat yourself to an expensive meal at your favourite restaurant. You just created your first *real* stream, so swap out your sweatbands for some fresh ones and read on.

5.1 Round 1: Simple Streams

Let's define a few streams from scratch, just like we did with `fibs` and `ones`.

1. Define the stream `(ints-from n)` which is the stream of all integers starting from *n* on. Then, using this stream, define a new stream `ints` of *all* positive integers (1,2,3,...).

   ```
   (s-car (ints-from 5))
   => 5
   (s-car (s-cdr (ints-from 5)))
   => 6
   (s-generate (ints-from 5) 10)
   => (5 6 7 8 9 10 11 12 13 14)
   ```

   ```
   (define (ints-from n)
     (s-cons __
             __________________))

   (define ints _______________)
   ```

2. Good so far? That couldn't have been any harder than `fibs`, because they're nearly the same! So is this one. Define a pair of streams, `evens-from` and `evens`, that define a stream of even numbers in a similar fashion to the last problem.

   ```
   (define (evens-from n)
     (s-cons __
             ___________________))

   (define evens _______________)
   ```

Excellent. If you haven't yet worked up a sweat, move on to the next round. Otherwise, take a shower and call it a day.

5.2 Round 2: Simple Stream Operations

1. This is a warmup that will come in handy later. Define the following functions:

   - `(id n)`: returns n (does nothing to input)
   - `(double n)`: returns $2n$
   - `(square n)`: returns n^2
   - `(pow2 n)`: returns 2^n
   - `(multiple?  m n)`: returns `#t` if m is a multiple of n, `#f` otherwise.
   - `(even?  n)`: returns `#t` if n is even, `#f`.

```
(define (double n)
                                                    )
(define (square n)
                                                    )
(define (pow2 n)
                                                    )
(define (multiple? m n)

                                                    )
;; Version 1: doesn't use multiple
(define (even? n)

                                                    )
;; Version 2: uses multiple
(define (even? n)
                                                    )
```

2. We've already seen that `s-generate` is a rather handy function. In fact, we can think of it as converting part of a stream into a list. Can we think of other think of other things to do with streams? How about the stream equivalent of `list-ref` ? Define away!

```
(s-ref ints 0)     ;; 0 1 2 3 4 5 6 7 ...
=> 1
(s-ref ints 4)     ;; 1 2 3 4 5 6 7 ...
=> 5
(s-ref fibs 0)     ;; 0 1 1 2 3 5 8 ...
=> 0
(s-ref fibs 4)     ;; 0 1 1 2 3 5 8 ...
=> 3
```

```
;; Version 1: Don't use s-generate
(define (s-ref s n)



                                          )

;; Version 2: Use  s-generate and  list-ref (mind your indices!)
(define (s-ref s n)


                                          )
```

That's useful. We're starting to develop a Scheme vocabulary to reason with streams. There's still a whole lot more we can do with them, though. Recall one definition of streams which involved a **starting element** and a **function** to generate each new element from previous one. This is a bit of a generalization of our usual pattern of making streams. Let's look once again at `ints-from` and `evens-from`:

```
(define (ints-from n)
  (s-cons n (ints-from (+ n 1)))
(define (evens-from n)
  (s-cons n (evens-from (+ n 2))))
```

Let's look again with some pieces changed:

```
(define (stream n)
  (s-cons n (stream (function n))))
```

See the basic structure we've been using to create a stream? We take a number *n* as our starting line, and apply *function* to any element to get the next element. In the first case, *function* have been `add-one` and in the second it might have been `add-two`. Let's write a function `s-make` that will **create** a stream for us given these two parts of our definition. This is a bit complicated, so we'll go back to using _ temporarily.

```
(define (s-make f n)
  (define (stream n)
    (________ __ (stream _____)))
  (stream __))
```

The blanks should be straightforward to fill in, but more importantly, let's use this function to define some new streams, including the ones we just discussed. Feel free to use `add-one` and `add-two`. You might find some of the simple functions you defined earlier to be useful.

```
(define ones          (s-make __________ __))
(define twos          (s-make __________ __))
(define ints          (s-make __________ __))
(define evens         (s-make __________ __))
(define powers-of-two (s-make __________ __))
```

An important thing to notice is that you just defined `ints` and `evens` **without** needing to define `ints-from` and `evens-from`. This function lets us make some streams very conveniently! (Can we make `fibs` with `s-make`? What is the general pattern missing that our earlier definition of `fibs-from` had.)

5.3 Final Round: Complex Stream Operations

3. There's yet more we can do to operate on streams. Can you think of anything? Can we do something (apply a function) to every element in a stream? Well, we don't know how long the stream goes on for, and everything we've worked with so far is an infinite stream. How in the world can we then do something to every element when there are an infinite number of elements? Hmm, what about making a new stream? What goes in the new stream? How about this: every element in the new stream is some function of an element in the old stream. That sounds like something we could possibly work with. So basically, the new stream is simply "a function away" from the old stream. Can we get an example? Ok:

   ```
   ones = 1,1,1,1,1,...
   twos = ones * 2 = 1*2,1*2,1*2,1*2,... = 2,2,2,2,2,...
   ```

   That makes sense. When we say `ones * 2`, we're really saying take every element of `ones` and double it. The result is the stream of twos. Let's assume we have a function `s-map` that takes a function f and a stream of elements $(e_1, e_2, e_3, \ldots)$ and returns a stream of elements $(f(e_1), f(e_2), f(e_3), \ldots)$. That's a mouthful, but here's how we'd use it to do the above in Scheme:

   ```
   (define twos
     (s-map double ones))
   ```

   That's all! Looks almost too simple, so let's actually *write* `s-map` to see why.

   ```
   (define (s-map f s)
     (s-cons _____________             ;; apply function to first element
             (s-map __ __________)))   ;; recurse on rest of stream
   ```

Nice, we just defined another extremely useful function. You know what comes next.

4. Feel free to use any of the simple functions you've already defined in combination with `s-map` to define the following familiar streams:

```
(define twos-new
  (s-map __________ twos))
(define threes
  (s-map add-one __________))
(define evens
  (s-map __________ ints))
(define powers-of-two
  (s-map __________ __________))
(define true-falses
  (s-map even? __________))
```

5. We just increased our repertoire yet again with `s-map` , which creates a new stream that's a function of another stream. What about making a new stream that's a function of **two** other streams. Let's call it `s-merge` ; it'll take a function f and two streams, $(n_1, n_2, n_3, \ldots)$ and $(m_1, m_2, m_3, \ldots)$, and create a new stream $(f(n_1, m_1), f(n_2, m_2), f(n_3, m_3), \ldots)$. Once again, let's see an example before actually writing the function.

```
(define twos
  (s-merge + ones ones))
```

```
(s-generate twos 10)
=> (2 2 2 2 2 2 2 2 2 2)
```

Looks good, on to the definition:

```
(define (s-merge f s1 s2)
  (s-cons _________________________      ;; apply f to first element of both streams
    (s-merge __ _________ _________))))  ;; recurse on rest of both streams
```

6. Ok, you're probably bored of looking at our old friends again, so let's look at them in a new light. Define `ints`, `odds`, and `fibs` as merges of other streams. Feel free to use any stream or function we've defined so far. `ints` and `fibs` are slightly tricky, so don't feel bad it they take some time to work out. Hint: they're both recursively defined.

```
;; use ones and evens
(define odds
  (s-merge

                       ))
(define ints
  (s-cons 1
          (s-merge

                               )))
(define fibs
  (s-cons 0
          (s-cons 1
                  (s-merge

                                         ))))
```

7. Let's define our last two stream operators. `s-filter` takes two arguments, a predicate function of one argument (like `even?`) and a stream and returns the stream of elements that "pass" the predicate function. That is, if the predicate function returns `#t` for an element in the input stream, that element should be part of the output stream.

```
(define (s-filter f s)
  (if
      (s-cons
                                                        )
      (s-filter                                         )))
(define evens
  (s-filter                                             ))
```

8. This is it, our last stream operation. `s-interleave`