

CS61A Notes – Disc 11: Streams

Streaming Along

A stream is an element and a “promise” to evaluate the rest of the stream. You’ve already seen multiple examples of this and its syntax in lecture and in the book, so I will not dwell on that. Suffice it to say, streams is one of the most mysterious topics in CS61A, but it’s also one of the coolest; mysterious, because defining a stream often seems like black magic (and requires MUCH more trust than whatever trust you worked up for recursion); cool, because things like infinite streams allows you to store an INFINITE amount of data in a FINITE amount of space/time!

How is that possible? We’re not going to be too concerned with the below-the-line implementations of streams, but it’s good to have an intuition. Recall that the body of a `lambda` is NOT executed until it is called. For example, typing into STk:

```
(define death (lambda () (/ 5 0)))
```

Scheme says “okay”, happily binding `death` to the `lambda`. But if you try to run it:

```
(death) ;; Scheme blows up
```

The crucial thing to notice is that, when you type the `define` statement, Scheme did NOT try to evaluate `(/ 5 0)` – otherwise, it would’ve died right on the spot. Instead, the evaluation of `(/ 5 0)` is *delayed* until the actual procedure call. Similarly, if we want to represent an infinite amount of information, we don’t have to calculate all of it at once; instead, we can simply calculate ONE piece of information (the `stream-car`), and leave instructions on how to calculate the NEXT piece of information (the “promise” to evaluate the rest).

It’s important to note, however, that Scheme doesn’t quite use plain `lambda` for streams. Instead, Scheme memorizes results of evaluating streams to maximize efficiency. This introduces some complications that we’ll visit later. The `delay` and `force` operators, therefore, are special operators with side effects.

QUESTIONS

1. Define a procedure (`ones`) that, when run with no arguments, returns a cons pair whose `car` is 1, and whose `cdr` is a procedure that, when run, does the same thing.

2. Define a procedure (`integers-starting n`) that takes in a number `n` and, when run, returns a cons pair whose `car` is `n`, and whose `cdr` is a procedure that, when run with no arguments, does the same thing for `n+1`.

Using Stream Operators

Here are some that we’ll be using quite a bit:

`(stream-map proc s ...)` – works just like list `map`; can take in any number of streams

`(stream-filter proc s)` – works just like list `filter`

`(stream-append s1 s2)` – appends two finite streams together (why not infinite streams?)

`(interleave s1 s2)` – interleave two streams into one, with alternating elements from `s1` and `s2`

Constructing Streams

This is the trickiest part of streams. I said that the topic of streams is a black art, and you'll soon see why. The construction of streams is counter-intuitive with a heavy dose of that-can't-possibly-work. So here are some rough guidelines:

- 1. cons-stream is a special form!** `cons-stream` will NOT evaluate its second argument (the `stream-cdr`); obviously, this is desirable, since we'd like to delay that evaluation.
- 2. Trust the, err, stream.** From the first day, we've been chanting "trust the recursion". Well now that you're (slightly more) comfortable with that idea, we need you to do something harder. When you're defining a stream, you **have to think as if that stream is already defined**. It's often very difficult to trace through how a stream is evaluated as you `stream-cdr` down it, so you have to work at the logical level. Therefore, the above definition of `integers` works. However, be careful that you don't trust the stream too much. For example, this won't work:

```
(define integers integers)
```

- 3. Learn how to think about stream-map.** Consider this definition of `integers`, given the stream `ones`, a stream of ones, defined in SICP:

```
(define integers (cons-stream 1 (stream-map + ones integers)))
```

If the above definition of `integers` puzzles you a bit, here's how to think about it:

```
      1                                     <= your stream-car
      1  2  3  4  5  6  ...                <= integers (as taken from the last line)
+     1  1  1  1  1  1  ...                <= ones
=====
      1  2  3  4  5  6  7  ...                <= integers
```

If you're ever confounded by a `stream-map` expression, write it out and all should be clear. For example, let's try a harder one — `partial-sum`, whose *i*th element is the sum of the first *i* integers. It is defined thus:

```
(define partial-sum (cons-stream 0 (stream-map + partial-sum integers)))
```

```
      0                                     <= your stream-car
      0  1  3  6 10 15 ...                <= partial-sum
+     1  2  3  4  5  6  ...                <= ones
=====
      0  1  3  6 10 15 20 ...                <= partial-sum
```

Now, if you find it odd to have `integers` or `partial-sum` as one of the things you're adding up, refer to guideline #2.

- 4. Specify the first element(s).** Recall that a stream is one element and a promise to evaluate more. Well, often, you have to specify that one element so that there's a starting point. Therefore, unsurprisingly, when you define streams, it often looks like

```
(cons-stream [first element] [a stream of black magic]).
```

But there are many traps in this. In general, what you're avoiding is an infinite loop when you try to look at some element of a stream. `stream-cdr` is usually the dangerous one here, as it may force evaluations that you want to delay. Note that **Scheme stops evaluating a stream once it finds one element**. So simply make sure that it'll always find one element immediately. For example, consider this definition of `fib`s that produces a stream of Fibonacci numbers:

```
(define fibs (cons-stream 0 (stream-map + fibs (stream-cdr fibs))))
```

Its intentions are admirable enough; to construct the next `fib` number, we add the current one to the previous one. But let's take a look at how it logically stacks up:

```

      0                                     <= your stream-car
      0 1 1 2 3 5 ... <= fibs?
+     1 1 2 3 5 8 ... <= (stream-cdr fibs)
=====
0 1 2 3 5 8 13 ... <= not quite fibs...

```

Close, but no cigar (and by the definition of Fibonacci numbers you really can't just start with a single number). Actually, it's even worse than that; if you type in the above definition of `fibs`, and call `(stream-cdr fibs)`, you'll send STk into a most unfortunate infinite loop. Why? Well, `stream-cdr` forces the evaluation of `(stream-map + fibs (stream-cdr fibs))`. **stream-map is not a special form**, so it's going to evaluate both its arguments, `fibs` and `(stream-cdr fibs)`. What's `fibs`? Well, `fibs` is a stream starting with 0, so that's fine. What's `(stream-cdr fibs)`? Well, `stream-cdr` forces the evaluation of `(stream-map + fibs (stream-cdr fibs))`. **stream-map is not a special form**, so it's going to evaluate both its arguments, `fibs` and `(stream-cdr fibs)`. What's `fibs`? Well, `fibs` is a stream starting with 0, so that's fine. What's `(stream-cdr fibs)`? You get the point.

How do we stop that horrid infinite loop? Well, it was asking for `(stream-cdr fibs)` that was giving us trouble – whenever we try to evaluate `(stream-cdr fibs)`, it goes into an infinite loop. Thus, why don't we just specify the `stream-cdr`?

```

(define fibs
  (cons-stream 0
    (cons-stream 1 (add-streams fibs (stream-cdr fibs)))))

```

So, then, let's try it again. What's `(stream-cdr fibs)`? Well, `(stream-cdr fibs)` is a stream starting with 1. There! Done! See? Now, it's pretty magical that adding one more element fixes the `stream-cdr` problem for the whole stream. Convince yourself of this. As a general rule of thumb, **if in the body of your definition you use the `stream-cdr` of what you're defining, you probably need to specify two elements**. Let's check that it logically works out as well:

```

      0 1                                     <= your stream-car
      0 1 1 2 3 5 ... <= fibs
+     1 1 2 3 5 8 ... <= (stream-cdr fibs)
=====
0 1 1 2 3 5 8 13 ... <= win!

```

QUESTIONS: Describe what the following expressions define.

In Class:

1. `(define s1 (add-stream (stream-map (lambda (x) (* x 2)) s1) s1))`

2. `(define s2
 (cons-stream 1
 (add-stream (stream-map (lambda (x) (* x 2)) s2) s2)))`

3. `(define s4
 (cons-stream 1
 (cons-stream 2
 (stream-filter (lambda (x) (not (= x 1))) s4))))`

4. Define facts without defining any procedures; the stream should be a stream of $1!, 2!, 3!, 4!, \dots$. More specifically, it returns a stream with elements `(1 2 6 24 ...)`

```
(define facts
  (cons-stream
```

Extra Practice:

5.

```
(define s5
  (cons-stream 1
    (add-streams s5 integers)))
```

6.

```
(define s3
  (cons-stream 1
    (stream-filter (lambda (x) (not (= x 1))) s3)))
```

7. **(HARD!)** Define powers; the stream should be $1^1, 2^2, 3^3, \dots$, or, `(1 4 16 64 ...)`. Of course, you cannot use the `exponents` procedure. I've given you a start, but you don't have to use it.

```
(define powers (helper integers integers))
(define (helper s t)
  (cons-stream
```

Constructing Streams Through Procedures

You'll find this the most natural way to construct streams, since it mirrors recursion so much. For example, to use a trite example,

```
(define (integers-starting n) (cons-stream n (integers-starting (+ n 1))))
```

So `(integers-starting 1)` is a stream whose first element is 1, with a promise to evaluate `(integers-starting 2)`. The rules are similar to above; you still specify a first element, etc. Pretty simple? Let's try a few.

QUESTIONS

1. Define a procedure (`lists-starting n`) that takes in `n` and returns a stream containing (n) , $(n\ n+1)$, $(n\ n+1\ n+2)$, etc. For example, (`lists-starting 1`) returns a stream containing `(1) (1 2) (1 2 3) (1 2 3 4)...`

```
(define (lists-starting n)
```

2. Define a procedure (`chocolate name`) that takes in a name and returns a stream like so:

```
(chocolate 'chung) =>
```

```
(chung really likes chocolate chung really really likes chocolate chung really really  
really likes chocolate ...)
```

You'll want to use helper procedures.

```
(define (chocolate name)
```

Stream Processing

Sometimes you'll be asked to write procedures that convert one given stream into another exerting a certain property.

QUESTIONS:

1. Define a procedure, (`stream-censor s replacements`) that takes in a stream `s` and a table `replacements` and returns a stream with all the instances of all the `car` of entries in `replacements` replaced with the `cadr` of entries in `replacements`:

```
(stream-censor (hello you weirdo ...) ((you I-am) (weirdo an-idiot))) =>
```

```
(hello I-am an-idiot ...)
```

```
(define (stream-censor s replacements)
```

2. Define a procedure (`make-alternating s`) that takes in a stream of positive numbers and alternates their signs. So (`make-alternating ones`) => `(1 -1 1 -1 ...)` and (`make-alternating integers`) => `(1 -2 3 -4 ...)`. Assume `s` is an infinite stream.

```
(define (make-alternating s)
```

Extra Practice:

My Body's Floating Down the Muddy Stream

Now, more fun with streams!

MORE QUESTIONS (extra practice)

1. Given streams `ones`, `twos`, `threes`, and `fours`, write down the first ten elements of:

```
(interleave ones (interleave twos (interleave threes fours)))
```

2. Construct a stream `all-integers` that includes 0 and both the negative and positive integers.

```
(define all-integers
```

3. Suppose we were foolish enough to try to implement `stream-accumulate`:

```
(define (stream-accumulate combiner null-value s)
  (cond ((stream-null? s) null-value)
        (else (combiner
                  (stream-car s)
                  (stream-accumulate combiner null-value (stream-cdr s))))))
```

What happens when we do:

a. `(define foo (stream-accumulate + 0 integers))`

b. `(define bar (cons-stream 1 (stream-accumulate + 0 integers)))`

c. `(define baz (stream-accumulate
 (lambda (x y) (cons-stream x y))
 the-empty-stream integers))`

4. SICP ex. 3.68, page 341. If you understand this, you'll be fine.

5. Define a procedure, `(list->stream ls)` that takes in a list and converts it into a stream.

```
(define (list->stream ls)
```