SCHEMES AND STREAMS 13

COMPUTER SCIENCE 61A

August 5, 2014

Throughout this course, we've learned of a few neat ways to optimize a program's use of time and space. Last week, we learned about iterators and generators, which used lazy computation to store a (possibly infinite) sequence in finite space. This week, we'll be learning about a new data structure in Scheme called streams which use this same type of lazy computation, but with a syntax more similar to the lists.

0.1 Warmup

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 with no arguments, does the same thing as `ones`.

2. Define a procedure `integers-starting` 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 with `n`+1.

1 Introduction to Streams

A stream is an element and a "promise" to evaluate the rest of the stream. Streams are 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 use the same idea that we used in the warm up! 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`, Scheme did *not* try to evaluate `(/ 5 0)`. Instead, its evaluation 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; instead, we can simply calculate **one** piece of information (the first element), and leave instructions on how to calculate the **next** piece of information (the "promise").

We've seen this type of lazy computation with iterators and generators – the idea is similar! Its simply the data structure and its corresponding operations that are different.

2 Scheme's Stream Syntax

Note that as long as we give special consideration to the fact that the rest of a stream is not evaluated until it is needed, we can just treat the stream **as if it were a regular list**. Thus, we have the following operators built-in to Scheme. As an exercise, write down the regular list operators that correspond to these stream operators and what each should return.

- `(cons-stream element s)`
- `(stream-car s)`,`(stream-cdr s)`
- `the-empty-stream`
- `(stream-null? s)`
- `(stream-map func s ...)`,`(stream-filter func s)`
- `(stream-append s1 s2)`
- `(interleave s1 s2)`

Recursion is a natural way to construct streams, since it mirrors recursion so much. For example, we could rewrite the warm up the following way:

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

Notice what is happening here. We start out with a stream whose first element is `n`, and whose promise creates another stream. So when we do compute the promise, we get another stream whose first element is `n+1`, and whose promise is yet another incremented `integers-starting` stream. Hence, we effectively get an infinite stream of integers, computed one at a time. This is almost like an infinite recursion, but one which can be viewed one step at a time, and so does not cause an infinite loop.

2.1 Questions

1. Convert the following lists into streams:

   ```
   (cons 1 (cons 2 (cons 24 (cons 23 '()))))
   ```

   ```
   (cons 1 (cons (cons 1 (cons 3 '())) (cons 3 '())))
   ```

2. Write the procedure `list-to-stream` which takes in a Scheme list `lst` and outputs a stream of all the same elements. You don't have to deal with deep lists.

   ```
   (define (list-to-stream lst)
   ```

3. Write the procedure `add-streams` which takes in two streams, `s1` and `s2`, and outputs a stream of the elements of the stream `s1` added to the corresponding elements of the stream `s2`.

```
(define (add-streams s1 s2)
```

4. Write the procedure `interleave` which takes in two streams, `s1` and `s2`, and outputs a stream that interleaves the elements of a stream `s1` with the elements of the stream `s2`, alternating between the elements of the two.

```
STk> (ss (interleave (integers-starting 4)
            (cons-stream 2 (cons-stream 0 the-empty-stream))))
(4 2 5 0 6 7 8 9 10 11 ...)

(define (interleave s1 s2)
```

5. Define a procedure `chocolate` that takes in a `name` and returns a stream like so:
`(chocolate 'andrew) ==> (andrew really likes chocolate andrew really really likes chocolate andrew really really really likes chocolate ...)`

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

2.2 Higher Order Magic with Streams

In addition to those mentioned in the previous section, the following are also built-in procedures you'll be using quite a bit for manipulating streams:

- `(stream-map <proc> <stream> ...)` - maps the procedure `<proc>` over a single stream, or multiple if the inputted procedure requires multiple arguments.
- `(stream-filter <proc> <stream>)` - filters a stream `<stream>` based on the condition specified by `<proc>`.
- `(stream-append <s1> <s2>)` - appends two finite streams together (why not infinite streams?)

With these new procedures, we can do much more advanced and elegant manipulation of streams. However, this means that things can get more complicated as well.

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.

2.3 Questions

1. Using filter, write the procedure `remove-ele` that takes in a stream `s` and the element `el` and removes el from s.

   ```
   (define (remove-ele s el)
   ```

2.4 Thinking about Stream-map

Mapping with streams can be a bit tough to start thinking about. But drawing out what `stream-map` does can really help you get a hang of it. Consider this definition of the stream `integers`, given the stream `ones`, a stream of ones.

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

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

```
    1                              <== your stream-car
       1   2   3   4   5   6    ...    <== integers
+      1   1   1   1   1   1    ...    <== ones
============================
    1  2   3   4   5   6   7    ...    <== integers
```

If you're ever confounded by an `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    ...    <== integers
==============================
    0  1   3   6   10  15   21    ...    <== partial-sum
```

Practice makes perfect, so don't worry if you don't get the hang of it just yet. Ironic how it's so easy to get lost with `stream-map`!

2.5 Questions

1. Define `facts`, a stream of factorials.

   ```
   STk> (ss facts)
   (1 2 6 24 120 720 5040 40320 362880 3628800 ...)

   (define facts
   ```

2. Define a procedure `lists-starting` 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 (list-starting n)
   ```

2.6 Infinite Loops

While it's important to trust streams like you trust recursion, sometimes you can trust it too much, which can lead to infinite loops. One of the most important rules for streams is to **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, you want to avoid 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 `fibs` 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 if you think about it, by the definition of Fibonacci numbers, you really can't just start with a single number). Additionally, 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. This goes on forever, leading to an infinite loop, with no `cons-stream` to save us by not evaluating the rest.

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    ...    <== fibs
+           1   1   2   3   5    ...    <== (stream-cdr fibs)
=============================
    0   1   1   2   3   5   8     ...   <== Hooray!
```

So then, let's try understanding a few of these:

2.7 Questions

For the following questions, describe what the code is doing and write down the first four elements (unless the code loops infinitely or errors). If the code loops infinitely or errors, explain why. It might be helpful to write out what's happening logically for the `stream-map` problems, as we did above.

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

4.
```
(define s4
    (cons-stream 1
        (cons-stream 2
            (stream-filter (lambda (x) (not (= x 1))) s4))))
```

5.
```
(define s5
    (cons-stream 1
        (add-streams s5 integers)))
```

3 Challenge Question

1. Define `powers`, whose `i`th element (counting from 1) is i^i. You may not use the built-in `expt` procedure - you may only use addition and multiplication and any stream procedures you already have.

```
STk> (ss powers 7)
(1 4 27 256 3125 46656 823543 ...)
```