CS 450

Lecture 10: Streams

Carl D. Offner

1 Introduction

Streams are abstract sequences. They are potentially infinite—in fact, we will see that the most interesting and powerful uses of streams come in handling infinite sequences. To start, however, we will think of them as finite in length. Finite streams are entirely equivalent to lists. Nevertheless, they have their own initializers and access routines:

```
the-empty-stream   ; a data object -- a stream with no elements
(stream-null? x)
(stream-car x)
(stream-cdr x)
(cons-stream a x)
```

To start out, we can think of streams as lists. Later, we will see why this is not a good idea in general, even for finite lists.

2 Implementing standard processes using streams

Here are two computations we might want to perform:

1. Given a binary tree whose leaves are integers, find the sum of the squares of the leaves that are odd.

```
(define (sum-odd-squares tree)
  (if (not (pair? tree))
      (if (odd? tree)
        (square tree)
        0)
      (+ (sum-odd-squares (left-branch tree))
       (sum-odd-squares (right-branch tree)) )))
```

2. Construct a list of all the odd Fibonacci numbers `fib(k)` with $k \leq n$

```
;;; Assume we have already defined the procedure (fib k) which evaluates
;;; to the k'th Fibonacci number -- we have seen previously how to do this.

(define (odd-fibs n)
  (define (next k)
    (if (> k n)
      '()
      (let ((f (fib k)))
        (if (odd? f)
            (cons f (next (+ k 1)))
            (next (+ k 1)) ))))
  (next 1) )
```

Conceptually, what is going on in these two processes is this:

$$\begin{tikzcd}
\boxed{\begin{array}{c}\texttt{enumerate}\\\texttt{tree leaves}\end{array}} \arrow[r] & \boxed{\begin{array}{c}\texttt{filter}\\\texttt{odd?}\end{array}} \arrow[r] & \boxed{\begin{array}{c}\texttt{map}\\\texttt{square}\end{array}} \arrow[r] & \boxed{\begin{array}{c}\texttt{accumulate}\\\texttt{+, 0}\end{array}} \arrow[r] & {} \\
\boxed{\begin{array}{c}\texttt{enumerate}\\\texttt{integers}\end{array}} \arrow[r] & \boxed{\begin{array}{c}\texttt{map}\\\texttt{fib}\end{array}} \arrow[r] & \boxed{\begin{array}{c}\texttt{filter}\\\texttt{odd?}\end{array}} \arrow[r] & \boxed{\begin{array}{c}\texttt{accumulate}\\\texttt{cons, '()}\end{array}} \arrow[r] & {}
\end{tikzcd}$$

We would like to write our procedures so that these processes become explicit. So first let's do (1). First we need a procedure to take a tree and create a stream consisting of the leaves of the tree:

```
(define (stream-enumerate-tree tree)
  (if (not (pair? tree))
      (cons-stream tree the-empty-stream)
      (stream-append (stream-enumerate-tree (left-branch tree))
                     (stream-enumerate-tree (right-branch tree)) )))
```

Next, we need some general-purpose higher-order procedures that act on streams:

```
(define (stream-append s1 s2)
  (if (stream-null? s1)
      s2
      (cons-stream (stream-car s1)
                   (stream-append (stream-cdr s1) s2) )))

(define (stream-filter pred stream)
  (cond ((stream-null? stream) the-empty-stream)
        ((pred (stream-car stream))
         (cons-stream (stream-car stream)
                      (stream-filter pred (stream-cdr stream))))
        (else (stream-filter pred (stream-cdr stream))) ))
```

```
(define (stream-map proc stream)
  (if (stream-null? stream)
      the-empty-stream
      (cons-stream (proc (stream-car stream))
                   (stream-map proc (stream-cdr stream)) )))

(define (stream-accumulate proc init stream)
  (if (stream-null? stream)
      init
      (proc (stream-car stream)
          (stream-accumulate proc init (stream-cdr stream)) )))
```

Now in terms of these definitions, we have simply

```
(define (sum-odd-squares tree)
  (stream-accumulate + 0
                     (stream-map square
                                 (stream-filter odd?
                                                (stream-enumerate-tree
                                                 tree) ))))
```

In our original code, the set of leaves of the tree was implicit in the code. Here, however, we have made it an explicit object—a "stream". And doing that makes it possible to write our code much more clearly, in terms of procedures that produce or consume such streams.

It's quite useful to note that these higher-order procedures are pretty general, and so they can be reused. For instance, let us now handle (2). We only need one new procedure:

```
(define (stream-enumerate-interval low high)
  (if (> low high)
      the-empty-stream
      (cons-stream low
                   (stream-enumerate-interval (+ low 1) high) )))
```

And now we can represent (2):

```
(define (odd-fibs n)
  (stream-accumulate cons '()
                     (stream-filter odd?
                            (stream-map fib
                                        (stream-enumerate-interval 1 n) ))))
```

Here in our original code the set of numbers from 1 to n was implicit. In our new code, we have made it an explicit stream and as before, our procedures either produce or consume such streams.

We can put these stream tools together in different ways. For instance, suppose we want to construct a list of the squares of the first `n` Fibonacci numbers. Here's how we do it:

```
(define (list-square-fibs n)
  (stream-accumulate cons '()
                     (stream-map square
                                 (stream-map fib
                                             (stream-enumerate-interval
                                              1 n) ))))
```

Here's another example:

```
(define (product-of-squares-of-odd-elements stream)
  (stream-accumulate * 1
              (stream-map square
                          (stream-filter odd? stream) )))
```

Here's another example: suppose we have a stream of records containing information about employees. Say we have a selector `salary` which extracts the employee's salary from that employee's record—`(salary record)` evaluates to the employee's salary. Suppose we want to find the salary of the highest-paid employee who is a programmer. Here's how we do it:

```
(define (salary-of-highest-paid-programmer record-stream)
  (stream-accumulate max 0
                     (stream-map salary
                                 (stream-filter programmer? record-stream) )))
```

Here's another higher-order procedure:

```
(define (stream-for-each proc stream)
  (if (stream-null? stream)
      'done
      (begin (proc (stream-car stream))
             (stream-for-each proc (stream-cdr stream)) )))
```

This is useful, for instance, for viewing a stream:

```
(define (display-stream stream)
  (stream-for-each display-line stream) )

(define (display-line x)
  (newline)
  (display x) )
```

3 Delayed evaluation

Up to now, we have been regarding streams as the same as lists. However, thinking of streams—even finite streams—as lists leads to severe inefficiencies. For instance, suppose we want to compute the sum of the primes from `a` to `b`. Here is a straightforward way to do this:

```
(define (sum-primes a b)
  (define (iter count accum)
    (cond ((> count b) accum)
          ((prime? count) (iter (+ count 1) (+ accum count)))
          (else (iter (+ count 1) accum)) ))
  (iter a 0) )
```

It would be nicer to write it like this:

```
(define (sum-primes a b)
  (stream-accumulate + 0
                     (stream-filter prime?
                                    (stream-enumerate-interval a b) )))
```

But here we have to first create the whole list of integers from a to b, then create the whole list of primes from a to b, and then sum them.

Even worse: suppose we want to find the second prime in the interval $[10,000 \ldots 100,000]$. We could write this:

```
(stream-car (stream-cdr (stream-filter prime?
                                       (stream-enumerate-interval 10000
                                                                  100000))))
```

But this would first create a list of 90,000 numbers, checking each of them for primality, and then throwing away all but the first two.

The solution is to only create elements of a stream on demand. Specifically,

- We make `cons-stream` a special form which does not evaluate its second argument.
- `stream-cdr` performs the actual evaluation.

In order to implement this, we use a new special form `delay` and and a new primitive procedure `force`. Both of these are built into Scheme.

`(cons-stream a b)` is equivalent to `(cons a (delay b))`

```
(define (stream-car stream) (car stream))
(define (stream-cdr stream) (force (cdr stream)))
```

So now let's see how to evaluate

```
(stream-car (stream-cdr (stream-filter prime?
                                       (stream-enumerate-interval 10000
                                                                  100000))))
```

where

```
(define (stream-enumerate-interval low high)
  (if (> low high)
      the-empty-stream
      (cons-stream low
                   (stream-enumerate-interval (+ low 1) high) )))
```

We do this in four steps:

Step 1. Produce `(stream-enumerate-interval 10000 100000)`

Step 2. Pass this to `(stream-filter prime? ...)`

Step 3. Pass this to `(stream-cdr ...)`

Step 4. Pass this to `(stream-car ...)`

So here we go:

Step 1. We first produce `(stream-enumerate-interval 10000 100000)`; this is

```
(cons 10000 (delay (stream-enumerate-interval 10001 100000)))
```

Step 2. We next want to evaluate

```
(stream-filter prime? (stream-enumerate-interval 10000
                                                 100000))
```

That is, we want to evaluate

```
(stream-filter prime? (cons 10000
                            (delay (stream-enumerate-interval 10001
                                                              100000))))
```

Now remembering

```
(define (stream-filter pred stream)
  (cond ((stream-null? stream) the-empty-stream)
        ((pred (stream-car stream))
         (cons-stream (stream-car stream)
                      (stream-filter pred (stream-cdr stream))))
        (else (stream-filter pred (stream-cdr stream))) ))
```

we have, since `(prime? 10000)` is `#f`,

```
(stream-filter prime? (stream-cdr stream))
```

where `stream` is

```
                     (cons 10000
                           (delay (stream-enumerate-interval 10001
                                                             100000)))
```

Now `stream-cdr` forces the `delay`, like this:

```
(stream-filter prime? (force (delay (stream-enumerate-interval 10001
                                                                100000))))
```

so we get

```
(stream-filter prime? (stream-enumerate-interval 10001
                                                  1000000))
```

which is

```
(stream-filter prime? (cons 10001
                            (delay (stream-enumerate-interval 10002
                                                               100000))))
```

and this continues until we get to

```
(stream-filter prime? (cons 10007
                            (delay (stream-enumerate-interval 10008
                                                               100000))))
```

Since 10007 *is* prime, this becomes

```
(cons 10007
      (delay (stream-filter prime?
                            (stream-cdr (cons 10007
                                              (delay (stream-enumerate-interval
                                                      10008
                                                      100000)))))))
```

Step 3. This result is now passed to `stream-cdr` in our original expression. This forces the first `delay`, and evaluates to

```
(stream-filter prime?
               (stream-cdr (cons 10007
                                 (delay (stream-enumerate-interval 10008
                                                                    100000))))))
```

Both arguments to stream-filter have to be evaluated. Evaluating the second argument causes stream-filter to force the `delay` in the `cdr` of its argument, yielding

```
(stream-filter prime? (cons 10008
                            (delay (stream-enumerate-interval 10009
                                                               100000))))
```

which in turn keeps going until it finds the next prime, which is 10009. That is, we get

```
(cons 10009
      (delay (stream-filter prime?
                            (stream-cdr (cons 10009
                                              (delay (stream-enumerate-interval
                                                      10010
                                                      100000)))))))
```

Step 4. This is now passed to the `stream-car` in our original expression. This is just `car`, and so we get 10009.

4 Implementing `delay` and `force`

Here is one way that one could implement `delay` and `force`:

`delay` is a special form, such that

`(delay <exp>)` is equivalent to `(lambda () <exp>)`

`force` is not a special form: it is a procedure which just evaluates its argument by calling it as a procedure:

```
(define (force delayed-object)
  (delayed-object))
```

Now actually, it's a little more complicated, because delayed objects are also tagged so that they print out as a PROMISE:

```
==> (define a (delay b))

a
==> a

(PROMISE
  b)
```

But that's a minor point.

4.1 Be careful about this!

Note that we could `define` the variable `force`, because it is the name of a procedure, and procedures can be defined.

However, we could not use `define` to specify what we mean by `delay`, because `delay` is a special form. Suppose we tried to do it, like this:

```
    (define (pseudo-delay exp)                ;;; WRONG!!!
      (lambda () exp))                        ;;; WRONG!!!
```

Other than me yelling at you, what would go wrong if you did this?

Therefore `delay` has to be created in Scheme by some other technique. It can be specified as a macro—this is pretty typical, in fact. But we're not covering macros in this class, so just don't worry about it. It will be provided for you.