SICP sections 3.5.1 - 3.5.2

November 05, 2007 at 20:29 Tags SICP

I'm going to use Common Lisp for this section, since I much prefer CL's macros to Scheme's and macro usage is required to implement the `delay` and `cons-stream` functions.

Why are macros required here ? In the book, the authors, after defining some stream primitives vaguely note that how `cons-stream` works, without showing its actual implementation:

```
(cons-stream <a> <b>)
```

Is equivalent to:

```
(cons <a> (delay <b>))
```

But this can be misleading. What we really want here is *normal order evaluation*, where the arguments to a function aren't evaluated prior to calling it. We **don't** want the second argument of `cons-stream` to be evaluated, because it is *delayed*. If we translate the `show` function[1] to CL:

```
(defun show (x)
  (format t "~a~%" x)
  x)
```

We can understand how the call to `cons-stream` works. First, let's define `cons-stream` naively as:

```
(defun cons-stream (a b)
  (cons a (delay b)))
```

Now consider[2]:

```
(deflex ss (cons-stream 1 (show (+ 2 1))))
=> 3
(stream-cdr ss)
```

What printed 3 is the call to `cons-stream`, because the evaluation rules of Lisp are *applicative order*. What we really want is for the call to `cons-stream` to print nothing, and then the call to `stream-cdr` to print the value:

```
(deflex ss (cons-stream 1 (show (+ 2 1))))
(stream-cdr ss)
=> 3
```

How can delayed order be implemented in Common Lisp ? Using*macros* ! When a macro is called, the Lisp evaluator doesn't evaluate any of its arguments automatically, it leaves the choice of which arguments to evaluate to the macro itself[3]. Here's the correct `cons-stream`:

```
(defmacro cons-stream (a b)
  `(cons ,a (delay ,b)))
```

`delay` must be implemented as a macro as well, for the same reason:

```
(defmacro delay (expr)`(memo-proc (lambda () ,expr)))
```

`memo-proc` was defined by the authors. Here's its translation to CL:

```
(defun memo-proc (proc)
  (let ((already-run? nil) (result nil))
    (lambda ()
      (if (not already-run?)
        (progn
          (setf result (funcall proc))
          (setf already-run? t)
          result)
        result))))
```

And here are the rest of the primitives:

```
(defvar the-empty-stream '())
(defun stream-null? (s)
  (null s))

(defun force (delayed-object)
  (funcall delayed-object))

(defun stream-car (s)
  (car s))

(defun stream-cdr (s)
  (force (cdr s)))
```

Using these primitives we can implement some of the basic stream operations:

```
(defun stream-ref (s n)
  (if (= n 0)
    (stream-car s)
    (stream-ref (stream-cdr s) (- n 1))))

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

(defun stream-filter (pred s)
  (cond ((stream-null? s) the-empty-stream)
        ((funcall pred (stream-car s))
          (cons-stream
            (stream-car s)
            (stream-filter pred (stream-cdr s))))
        (t (stream-filter pred (stream-cdr s)))))
```

And use them:

```
(deflex ss
  (stream-filter
    #'prime?
    (stream-enumerate-interval 10 100)))

(stream-car ss)
=> 11
(stream-car (stream-cdr ss))
=> 13
(stream-car (stream-cdr (stream-cdr ss)))
=> 17
```

Exercise 3.50

```
(defun stream-map (proc &rest argstreams)
  (if (stream-null? (car argstreams))
    the-empty-stream
    (cons-stream
      (apply proc (mapcar #'stream-car argstreams))
      (apply #'stream-map
        (cons proc (mapcar #'stream-cdr argstreams))))))
```

Usage:

```
(deflex s1 (stream-enumerate-interval 10 100))
(deflex s2 (stream-enumerate-interval 20 200))
(deflex s3 (stream-enumerate-interval 30 300))

(deflex ss (stream-map #'+ s1 s2 s3))

(stream-ref ss 0)
=> 60
(stream-ref ss 1)
=> 63
(stream-ref ss 2)
=> 66
```

Exercise 3.51

```
(deflex x (stream-map #'show (stream-enumerate-interval 0 10)))
=> 0
(stream-ref x 5)
=>
1
2
3
4
5
(stream-ref x 7)
=>
6
7
```

Note that the second `stream-ref` prints only two numbers because of `memo-proc` and the previous call to `stream-ref`.

Exercise 3.52

```
(deflex sum 0)
(defun accum (x)
  (setf sum (+ x sum))
  sum)
=> sum is 0

(deflex seq (stream-map #'accum (stream-enumerate-interval 1 20)))
=> sum is 1

(deflex y (stream-filter #'evenp seq))
=> sum is 6

(deflex z (stream-filter (lambda (x) (= (rem x 5) 0)) seq))
=> sum is 10

(stream-ref y 7)
=> 136
=> sum is 136

(display-stream z)
=>
10
15
45
55
105
120
190
210
```

Exercise 3.53

```
(deflex s (cons-stream 1 (add-streams s s)))
=> 1, 2, 4, 8, 16 ...
```

What is `s` ? A stream beginning with 1 and the next element being the promise to compute the sum of `s` with itself. So, when asked for the second element, `s` returns the first element of the sum, which is double the first element of `s` itself. When asked for the third element, `s` returns the second element of the sum, which is the double of the second element, and so on.

Exercise 3.54

```
(defun mul-streams (s1 s2)
  (stream-map #'* s1 s2))

(deflex factorials
  (cons-stream 1 (mul-streams
                   factorials
                   (integers-starting-from 2))))
```

Exercise 3.55

The first element of this stream should be the first element of the original stream. The next element is a promise to compute the sum of partial sums itself (recursively), and the rest of the original stream:

```
(defun partial-sums (s)
  (cons-stream
    (stream-car s)
    (add-streams
      (stream-cdr s)
      (partial-sums s))))
```

Exercise 3.56

```
(defun merge (s1 s2)
  (cond ((stream-null? s1) s2)
        ((stream-null? s2) s1)
        (t
          (let ((s1car (stream-car s1))
                (s2car (stream-car s2)))
            (cond ((< s1car s2car)
                   (cons-stream s1car (merge (stream-cdr s1) s2)))
                  ((> s1car s2car)
                   (cons-stream s2car (merge s1 (stream-cdr s2))))
                  (t
                    (cons-stream
                      s1car
                      (merge (stream-cdr s1) (stream-cdr s2)))))))))


(deflex s
  (cons-stream
    1
    (merge
      (scale-stream S 2)
      (merge
        (scale-stream S 3)
        (scale-stream S 5)))))
```

Exercise 3.57

With memoization, `n-1` additions are performed for computing the `n@th fibonacci number, since each call of @force` on the stream returned by `add-streams` recomputes the `fibs` stream only once.

Without memoization, the growth is exponential because in the call to`add-streams`, `(stream-cdr fibs)` will do all the work `fibs` does, but that is repeated.

Exercise 3.58

This stream performs long division of `num` by `den` in base `radix`, returning subsequent numbers after the floating point with each call.

```
(expand 1 7 10)
=> 1 4 2 8 5 7 1 4 2 8

(expand 3 8 10)
=> 3 7 5 0 0 0 0 0 0 0
```

Exercise 3.59

a.

```
(defun integrate-series (s)
  (labels (
      (integrate-aux (s n)
        (cons-stream
          (/ (stream-car s) n)
          (integrate-aux
            (stream-cdr s)
            (+ n 1)))))
    (integrate-aux s 1)))
```

b.

```
(deflex sine-series
  (cons-stream 0 (integrate-series cosine-series)))

(deflex cosine-series
  (cons-stream 1
    (scale-stream
      (integrate-series sine-series)
      -1)))
```

Exercise 3.60

We can reach the solution if we think recursively. The first element of the multiplication of two power series is the product of the series' first elements. So far simple. Now, the rest of it can be computed by taking the first element of `s1`, multiplying it by all the elements of `s2` and then adding the multiplication of the rest of the elements of `s1` by `s2`.

```
(defun mul-series (s1 s2)
  (cons-stream
    (* (stream-car s1) (stream-car s2))
    (add-streams
      (scale-stream (stream-cdr s2) (stream-car s1))
      (mul-series (stream-cdr s1) s2))))
```

And the proposed test:

```
(deflex la
  (add-streams
    (mul-series sine-series sine-series)
    (mul-series cosine-series cosine-series)))
=> 1 0 0 0 ...
```

Exercise 3.61

```
(defun invert-unit-series (sr)
  (cons-stream
    1
    (scale-stream
      (mul-series
        (invert-unit-series sr)
        (stream-cdr sr))
      -1)))
```

Exercise 3.62

```
(defun div-series (num denom)
  (let ((denom-const (stream-car denom)))
    (if (zerop denom-const)
      (error "denom constant term is zero")
      (mul-series
        (invert-unit-series
          (scale-stream denom denom-const))
        num))))
```

And the tangent is:

```
(deflex tangent-series
  (div-series sine-series cosine-series))
```

[1] Defined by the authors in exercise 3.51

[2] `deflex` is explained here

[3] This is a simplified view of a complex reality. However, I don't intend to teach macros here – there are enough Lisp resources online for that. Macros are a tricky business – I must confess I don't fully master their use myself yet.

For comments, please send me an email.

Back to top