SICP section 3.5.3

November 10, 2007 at 08:41 Tags SICP

The code for this section is in Common Lisp.

Exercise 3.63

It is less efficient because it involves a recursive function call, which isn't optimized by `memo-proc`. This call recurses back on `sqrt-stream` for each new element forced.

Exercise 3.64

I'll define a helper procedure first. It should be familiar from conventional list processing:

```
(defun stream-cadr (s)
  (stream-car (stream-cdr s)))
```

Now:

```
(defun stream-limit (s tolerance)
  (if (<
       (abs (- (stream-car s) (stream-cadr s)))
       tolerance)
    (stream-cadr s)
    (stream-limit (stream-cdr s) tolerance)))
```

Exercise 3.65

First, the approxumation function:

```
(defun ln2-summands (n)
  (cons-stream
    (/ 1.0 n)
    (stream-map #'- (ln2-summands (1+ n)))))

(deflex ln2-stream
  (partial-sums (ln2-summands 1)))
```

This converges slowly:

```
(display-n-stream-elements ln2-stream 10)
=>
1.0
0.5
0.8333334
0.5833334
0.78333336
0.6166667
0.7595238
0.6345238
0.7456349
0.6456349
```

Now, with Euler's acceleration:

```
(display-n-stream-elements
  (euler-transform ln2-stream)
  10)
=>
0.70000005
0.69047624
0.6944445
0.6924243
0.69358975
0.69285715
0.69334733
0.6930033
0.69325393
0.69306576
```

Indeed, much quicker (the real value of `ln2` is 0.693147181056). When accelerated using the tableau method, however, the values become accurate very rapidly:

```
(display-n-stream-elements
  (accelerated-sequence
    #'euler-transform ln2-stream) 7)
=>
1.0
0.70000005
0.69327736
0.6931489
0.6931472
0.6931472
0.6931471
```

Exercise 3.66

`pairs` gives the first pair, and then interleaves the elements of the second stream with the head of the first, with a recursive call on itself with the tails of both streams. So, in examining the sequence we can "peel off" sub-sequences. Here are the first 10 pairs:

```
(1 1)
(1 2)
(2 2)
(1 3)
(2 3)
(1 4)
(3 3)
(1 5)
(2 4)
(1 6)
```

Notice that after the initial `(1 1)`, `(1 n)` appears every second element. This is because the nature of the `interleave` function. So we can already say that `(1 100)` will take 198 elements to show up, because `(1 n)` appears on position `2n-2`. The next task is more tricky. It will help to look at the stream with all the `(1 n)` pairs peeled off:

```
(2 2)
(2 3)
(3 3)
(2 4)
(3 4)
(2 5)
(4 4)
(2 6)
(3 5)
(2 7)
```

Looks familiar, doesn't it ? Again, `(2 n)` now appears every second element. But recall that we took `(1 n)` away, and two of these squeeze between each pair of `(2 n)`. Therefore, `(2 n)` come in steps of 4 in the `pairs` stream. Similarly, `(3 n)` come in steps of 8. Overall, `(m n)` is the n-th element of `2^m` apart, or approximately `n*2^m`.

Exercise 3.67

```
(defun all-pairs (s1 s2)
  (cons-stream
    (list (stream-car s1) (stream-car s2))
    (interleave
      (stream-map
        (lambda (x) (list (stream-car s1) x))
        (stream-cdr s2))
      (interleave
        (stream-map
          (lambda (x) (list x (stream-car s2)))
          (stream-cdr s1))
        (all-pairs (stream-cdr s1) (stream-cdr s2))))))
```

Exercise 3.68

Calling Louis's `pairs` will cause an infinite loop, because there's no delayed definition like we're used to have with streams. `interleave` evaluates its second argument, which is a recursive call to

`pairs`, which in turn calls itself again and again. In the original version, `interleave` is called through `cons-stream` that doesn't actually evaluate it until asked for the next element.

Exercise 3.69

Conceptually, we can think of triplets recursively in terms of pairs. If we take the first element of the first stream, and enumerate all the pairs taken from the other two streams with this element prepended, and repeat this procedure for every element in the first stream, we will eventually enumerate all triplets. Perhaps the code will be clearer than the explanation :-)

```
(defun triplets (s1 s2 s3)
  (cons-stream
    (list
      (stream-car s1)
      (stream-car s2)
      (stream-car s3))
    (interleave
      (stream-map
        (lambda (x) (append (list (stream-car s1)) x))
        (stream-cdr (pairs s2 s3)))
      (triplets
        (stream-cdr s1)
        (stream-cdr s2)
        (stream-cdr s3)))))
```

Note that I'm taking the `stream-cdr` or `pairs`, to skip the first pair which is already present in the first element given to `cons-stream`.

To generate the Pythagorean triplets:

```
(deflex ppp (triplets integers integers integers))

(deflex pythagorean
  (stream-filter
    (lambda (triplet)
      (=
        (square (caddr triplet))
        (+  (square (car triplet))
            (square (cadr triplet)))))
    ppp))

(display-n-stream-elements pythagorean 20)
=>
(3 4 5)
(6 8 10)
...
```

Exercise 3.70

```
(defun merge-weighted (weight s1 s2)
  (cond
    ((stream-null? s1) s2)
    ((stream-null? s2) s1)
    (t
      (let* ( (s1car (stream-car s1))
              (s1w (funcall weight s1car))
              (s2car (stream-car s2))
              (s2w (funcall weight s2car)))
        (cond ((<= s1w s2w)
                (cons-stream s1car (merge-weighted weight (stream-cdr s1) s2)))
              (t
                (cons-stream s2car (merge-weighted weight s1 (stream-cdr s2)))))))))

(defun weighted-pairs (weight s1 s2)
  (cons-stream
    (list (stream-car s1) (stream-car s2))
    (merge-weighted
      weight
      (stream-map
        (lambda (x) (list (stream-car s1) x))
        (stream-cdr s2))
      (weighted-pairs weight (stream-cdr s1) (stream-cdr s2)))))
```

Note that `merge-weighted` is a bit different from the original `merge`. The original `merge` is supposed to merge elements which may be equal. In such case, it will leave only one of those elements in the result stream (removing duplications). On the other hand, we are merging according to weight, and we can't allow one of the pairs `(2 3)` and `(1 4)` to disappear just because they have the same weight, so we must merge in both.

a.

```
(deflex sump
  (weighted-pairs
    (lambda (pair) (apply #'+ pair))
    integers
    integers))

(display-n-stream-elements sump 10)
=>
(1 1)
(1 2)
(1 3)
(2 2)
(1 4)
(2 3)
(1 5)
(3 3)
(2 4)
(1 6)
```

b.

```
(deflex no-235-factors
  (stream-filter
    (lambda (n)
      (not (or  (divides? 2 n)
                (divides? 3 n)
                (divides? 5 n))))
    integers))

(deflex s235
  (weighted-pairs
    (lambda (pair)
      (let ((i (car pair)) (j (cadr pair)))
        (+ (* 2 i) (* 3 j) (* 5 i j))))
    no-235-factors
    no-235-factors))

(display-n-stream-elements s235 20)
=>
(1 1)
(1 7)
(1 11)
(1 13)
(1 17)
(1 19)
(1 23)
(1 29)
(1 31)
(7 7)
(1 37)
(1 41)
(1 43)
(1 47)
(1 49)
(1 53)
(7 11)
(1 59)
(1 61)
(1 67)
```

Exercise 3.71

```
(defun ij-cube (pair)
  (+ (cube (car pair)) (cube (cadr pair))))

(deflex cubew
  (weighted-pairs
    #'ij-cube
    integers
    integers))

(defun ramanujan (stream max-count)
  (do* ((s stream (stream-cdr s))
        (count 1))
      (nil)
    (if (= (ij-cube (stream-car s)) (ij-cube (stream-cadr s)))
      (if (> count max-count)
        (return)
        (progn
          (incf count)
          (format t "~a~%"
            (list (ij-cube (stream-car s))
                  (stream-car s)
                  (stream-cadr s))))))))

(ramanujan cubew 6)
=>
(1729 (1 12) (9 10))
(4104 (2 16) (9 15))
(13832 (2 24) (18 20))
(20683 (10 27) (19 24))
(32832 (4 32) (18 30))
(39312 (2 34) (15 33))
```

Exercise 3.72

This is very similar to the previous exercise:

```
(defun ij-square (pair)
  (+ (square (car pair)) (square (cadr pair))))

(deflex squarew
  (weighted-pairs
    #'ij-square
    integers
    integers))

(defun squares-3ways (stream max-count)
  (do* ((s stream (stream-cdr s))
        (count 1))
      (nil)
    (let ((a (stream-car s))
          (b (stream-cadr s))
          (c (stream-cadr (stream-cdr s))))
      (if (= (ij-square a) (ij-square b) (ij-square c))
        (if (> count max-count)
          (return)
          (progn
            (incf count)
            (format t "~a~%"
              (list (ij-square a) a b c))))))))

(squares-3ways squarew 6)
=>
(325 (1 18) (6 17) (10 15))
(425 (5 20) (8 19) (13 16))
(650 (5 25) (11 23) (17 19))
(725 (7 26) (10 25) (14 23))
(845 (2 29) (13 26) (19 22))
(850 (3 29) (11 27) (15 25))
```

I suppose this could be generalized in some way: “find all numbers that can be written as *f(i,j)* of `i` and `j` in `n` different ways”. This will be left as an exercise to the diligent readers :-)

Exercise 3.73

First of all, I want to show how `integral` is implemented in Common Lisp. It is done a bit differently from Scheme, because it touches one of the points of difference between these two languages. In Scheme, variables and functions reside in the same namespace, while Common Lisp has separate namespaces for them[4].

Therefore, Common Lisp can’t treat variables the same way it treats functions, while Scheme can. Particularly, recursive definitions of variables are not as universal in Common Lisp. Scheme can use `define` to define recursive variables, and it can use `letrec` to do it locally. Common Lisp can’t[5]. What Common Lisp can do, however, is look at such variables explicitly as functions:

```
(defun integral (integrand initial-value dt)
  (labels (
      (int ()
        (cons-stream
          initial-value
          (add-streams (scale-stream integrand dt) (int)))))
    (int)))
```

This works just fine because, if you come to think of it, there isn't much of a difference between evaluating a variable for its value and calling a parameter-less function for its return value.

Now, to the `RC` circuit:

```
(defun RC (R C dt)
  (labels (
      (rc-model (v0 i-stream)
        (add-streams
          (scale-stream i-stream R)
          (integral
            (scale-stream i-stream (/ 1 C))
            v0
            dt))))
    #'rc-model))
```

Exercise 3.74

I'm pretty sure that what is expected as an answer here is:

```
(deflex zero-crossings
  (stream-map
    #'sign-change-detector
    sense-data
    (stream-cdr sense-data)))
```

However, there is a problem with this function. It is *non-causal* – a term in the study of signals and systems which describes functions that foresee future values. For instance, for the stream: `(1 2 -5 -6)` it produces `(0 -1 0 1)`. Note the -1 in the result, it comes at an earlier point of time than -5 in the input. This isn't a physical system! It would probably be more correct to implement it as:

```
(deflex zero-crossings
  (stream-map
    #'sign-change-detector
    (cons-stream 0 sense-data)
    sense-data))
```

Exercise 3.75

The fallacy in Louis's code is using the output values for the averaging. To make it right, the averaging must take only elements of input into account.

```
(defun make-zero-crossings-smoothing (input-stream last-value last-avpt)
  (let ((avpt (/ (+ (stream-car input-stream) last-value) 2)))
    (cons-stream
      (sign-change-detector avpt last-avpt)
      (make-zero-crossings-smoothing
        (stream-cdr input-stream)
        (stream-car input-stream)
        avpt))))
```

Note the clear separation of inputs and outputs here. `last-value` is only for the inputs, and it is used in the computation of the next `avpt`. `last-avpt` keeps track of the outputs for sign change detection.

Exercise 3.76

Here's the smoothing function:

```
(defun smooth (s)
  (stream-map
    (lambda (x1 x2) (/ (+ x1 x2) 2))
    (cons-stream 0 s)
    s))
```

It would be most modular to plug it in as follows:

```
(defun make-zero-crossings (in transform)
  (let ((smoothed-in (funcall transform in)))
    (stream-map
      #'sign-change-detector
      (cons-stream 0 smoothed-in)
      smoothed-in)))

(deflex sm-zero-crossings
  (make-zero-crossings sense-data #'smooth))
```

[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.

[4] This is, of course, the reason for the need to use `funcall` and `#'` in Common Lisp, which don't exist in Scheme.

[5] Well, not exactly. It works for some extent with `defvar`, but `defvar` is actually a macro which bends some of the rules, and works only for "toplevel" variables.

↑ Back to top