Brandeis University

Computer Science (1)21b (Spring Term, 2016)

Structure and Interpretation of Computer Programs

Problem Set 4: Streams and Lazy Evaluation

Due Monday, March 21

Reading Assignment: Chapter 3, Section 3.5.

The purpose of this assignment is to give you some experience with the use of streams and delayed evaluation as a mechanism for organizing computation. Delayed evaluation is a powerful technique, so be careful. Some of the phenomena you will see here may be mind-boggling—hold onto your chairs. Many of the procedures you will need have been provided in the code attached, which should be loaded when you begin work on this problem set. You should be familiar with their contents before starting work in the laboratory. Most of the work in this laboratory assignment is conceptual. There are no large programs to write, but several difficult ideas to understand.

Some infinite streams

As explained in section 3.5.2, we can define an infinite stream of ones and use this to define the stream of positive integers.

```
(define ones (cons-stream 1 ones))

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

Problem 1. Type in these definitions and verify that they work by using the `print-stream` procedure, which has been loaded with this problem set.[1] [2] Show how to define the stream of all integers that are not divisible by either 2, 3, or 5. (Use `stream-filter`.)

The following procedure (included in the code for the problem set) takes two infinite streams and interleaves them, selecting elements alternately from each stream:

```
(define (interleave s t)
  (cons-stream (stream-car s)
               (interleave t (stream-cdr s))))
```

Problem 2. Test the procedure by interleaving the stream of integers that are not divisible by 7 with the stream of integers not divisible by 3. What are the first few elements of the resulting stream? What expression did you evaluate to find this result?

Problem 3. Define the following stream `alt`

```
(define alt (cons-stream 0 (interleave integers alt)))
```

What can you say about the structure of this stream? Do you notice any regularities, and can you explain how they arise? (For example, which elements are equal to 0? Equal to 1? Equal to 2?)

Interleaving is a somewhat *ad hoc* way to combine two streams. *Merging* is an alternative combination method, which we can use when there is some way to compare the sizes of stream elements.

The `merge` operation takes two ordered streams (where the elements are arranged in increasing order) and produces an ordered stream as a result, omitting duplicates of elements that appear in both streams:

[1] Check the WWW home page for the course for the code you need in order to complete this assignment.

[2] The `print-stream` procedure loaded here differs from the one in the text (a) by a clever hack (b) in that it doesn't start a new line for every stream element. If you would rather see each element printed on a separate line, just do `(stream-for-each print` *stream*`)` as shown in the text.

```
(define (merge s1 s2)
  (cond ((empty-stream? s1) s2)
        ((empty-stream? s2) s1)
        (else
         (let ((h1 (stream-car s1))
               (h2 (stream-car s2)))
           (cond ((< h1 h2)
                  (cons-stream h1 (merge (stream-cdr s1) s2)))
                 ((> h1 h2)
                  (cons-stream h2 (merge s1 (stream-cdr s2))))
                 (else
                  (cons-stream h1 (merge (stream-cdr s1)
                                         (stream-cdr s2)))))))))
```

Generating an infinite stream of pairs from a pair of infinite streams

The discussion beginning on page 338 of the book describes a method for generating pairs of integers (i, j). This part of the problem set deals with another method for doing so.

Consider the problem of generating the stream of all pairs of integers `(i,j)` with `i` less than or equal to `j`. More generally, suppose we have two streams $S = \langle s_1, s_2, \ldots \rangle$ and $T = \langle t_1, t_2, \ldots \rangle$. and imagine the infinite rectangular array:

$$\begin{array}{cccc} (s_1,t_1) & (s_1,t_2) & (s_1,t_3) & \ldots \\ (s_2,t_1) & (s_2,t_2) & (s_2,t_3) & \ldots \\ (s_3,t_1) & (s_3,t_2) & (s_3,t_3) & \ldots \\ . & . & . & \\ . & . & . & \\ . & . & . & \end{array}$$

Suppose we wish to generate a stream that contains all the pairs in the diagram that lie on or above the diagonal, i.e., the pairs:

$$\begin{array}{cccc} (s_1,t_1) & (s_1,t_2) & (s_1,t_3) & \ldots \\ & (s_2,t_2) & (s_2,t_3) & \ldots \\ & & (s_3,t_3) & \ldots \\ & & & \ldots \end{array}$$

(If we take S and T both to be the stream of integers, we get the stream of pairs of integers (i, j) with i less than or equal to j.)

Call the stream of pairs `(pairs S T)`, and consider it to be composed of three sets of elements: the element (s_1, t_1), the rest of the elements in the first row, and the remaining elements:

(s_1, t_1)	(s_1, t_2)	(s_1, t_3)	...
	(s_2, t_2)	(s_2, t_3)	...
		(s_3, t_3)	...
			...

Observe that the third part of this decomposition (elements not in the first row) is (recursively) `(pairs (stream-cdr s) (stream-cdr t))`. Also note that the rest of the first row is just

```
(stream-map (lambda (x) (cons (stream-car s) x))
            (stream-cdr t))
```

assuming that we use `cons` to construct the pairs (s_i, t_j). Thus we can form our stream of pairs as follows:

```
(define (pairs s t)
  (cons-stream (cons (stream-car s) (stream-car t))
               (combine-in-some-way
                  (stream-map (lambda (x) (cons (stream-car s) x))
                              (stream-cdr t))
                  (pairs (stream-cdr s) (stream-cdr t)))))
```

In order to complete the procedure, we must decide on some way to `combine-in-some-way` the two inner streams.

Problem 4. Define a procedure `interleave-pairs` that uses the general method for forming pairs as above, using `interleave`—as in problem 2 above—to combine the two streams. Examine the resulting stream

```
(interleave-pairs integers integers)
```

Can you make any general comments about the order in which the pairs are placed into the stream? For example, about how many pairs precede the pair (1,100)? the pair (99,100)? the pair (100,100)?[3]

[3]If you can make precise mathematical statements here, all the better. But feel free to give more qualitative answers if you find yourself getting bogged down.

For the bogged, here's a hint: suppose we interleave the upper right and lower right *streams* in the above diagram, with the latter going *first*. Let $f(i, j)$ be the position of (s_i, t_j) in the output stream, with the first position numbered 1. Then you should be able to *verify* that $f(1, n) = 2n - 1$ and $f(m + 1, n + 1) = 2f(m, n)$.

Ordered streams of pairs

Now we want to be able to generate streams where the pairs appear in some useful order, rather than in the order that results from an *ad hoc* interleaving process. If we want to use a strategy as in the `merge` procedure of problem 2, we need to define an order on *pairs* of integers – that is, we need some way to say that a pair (i, j) is "less than" a pair (k, l). Assume that we can compute a positive integer *weight* for any pair (i, k), and to order pairs by their weight. We also assume that the weighting function is "compatible" with the ordering on integers, as follows:

- if $m < n$ then for any k, $weight((m, k)) < weight((n, k))$; and
- if $m < n$ then for any k, $weight((k, m)) < weight((k, n))$.

(In other words, $weight((i, j))$ increases if you increase either i or j.)

There are many different ways to choose a compatible weighting function. For instance, we could take the weight of (i, j) to be the sum $i + j$, or the sum of the cubes $i^3 + j^3$.

Problem 5. Write a procedure `merge-weighted`, which is like `merge`, except for two differences:

- `merge-weighted` takes an additional argument `weight`, which is a procedure that computes the weight of a pair. This is used to determine the order that elements should appear in the resulting stream.
- Unlike `merge`, `merge-weighted` should not discard elements with duplicate weights. Indeed, we may find two different pairs with the same weight, and this may be interesting, as we shall see below. (This stipulation makes the structure of `merge-weighted` simpler than that of `merge`, since there is one less case to worry about. Also, you can leave out the `empty-stream?` test from `merge`, since we will be concerned here only with infinite streams.)

Hand in your definition of `merge-weighted`.

Now we can implement the `pairs` procedure as described above, using the procedure `merge-weighted` to combine the two streams. The following version of the procedure takes an additional argument `pair-weight`, which is a function of two variables that will be used to compute the weights:

```
(define (weighted-pairs s t pair-weight)
  (cons-stream (cons (stream-car s) (stream-car t))
               (merge-weighted
                  (stream-map
                    (lambda (x) (cons (stream-car s) x))
                    (stream-cdr t))
                  (weighted-pairs (stream-cdr s) (stream-cdr t) pair-weight)
                  (lambda (p) (pair-weight (car p) (cdr p))))))
```

Note how `lambda` is used to convert `pair-weight`, a function of two arguments, into a procedure that expects a pair, as required by `merge-weighted`.

Problem 6. `Weighted-pairs`, as listed above, is included in the code for this problem set. Using this together with your procedure `merge-weighted` from problem 5 define the streams:

(a) all pairs of positive integers (i, j) ordered according to the sum $i + j$.

(b) all pairs of positive integers (i, j) ordered according to the sum $i^3 + j^3$.

(c) all pairs of positive integers (i, j), where neither i nor j is divisible by 2, 3, or 5, ordered according to the sum $2i + 3j + 5ij$.

What are the first few pairs in each stream? What expression(s) did you type in order to generate each stream?

Pairs with the same weight

Ordered streams of pairs provide an elegant solution to the problem of computing the *Ramanujan numbers*—numbers that can be written as the sum of two cubes in more than one way. This problem is given in exercise 3.71 on page 342 of the text. You should read that problem now.

Observe that to find a number that can be written as the sum of two cubes in two different ways, we need only generate the stream of pairs of integers (i, j) weighted according to the sum $i^3 + j^3$, then search the stream for two consecutive pairs with the same weight.

To implement and generalize the observation, we use a procedure `combine-same-weights`. This procedure takes a stream of pairs together with a weighting function `pair-weight`, which is a function of two arguments, as in the `weighted-pairs` procedure. We assume that the pairs appear in the stream according to increasing weight. `Combine-same-weights` returns a stream of lists. The `car` of each list is a weight, and the `cdr` is all the pairs that have that weight. The lists appear in the stream in order of increasing weight. For instance, if the input stream is the one that you should have obtained in problem 6(a):

```
(1 . 1) (1 . 2) (2 . 2) (1 . 3) (2 . 3) (1 . 4) (3 . 3)
(2 . 4) (1 . 5) (3 . 4) (2 . 5) (1 . 6) (4 . 4) (3 . 5)
(2 . 6) (1 . 7) ...
```

and `pair-weight` is `+`, then the procedure should generate the stream

```
(2 (1 . 1))
(3 (1 . 2))
(4 (1 . 3) (2 . 2))
(5 (1 . 4) (2 . 3))
(6 (1 . 5) (2 . 4) (3 . 3))
(7 (1 . 6) (2 . 5) (3 . 4))
(8 (1 . 7) (2 . 6) (3 . 5) (4 . 4))
        .
        .
        .
```

Problem 7. Implement the procedure `combine-same-weights` as described above.

Problem 8. The following procedure is included in the code for this problem set:

```
(define (same-weight-pairs s t pair-weight)
  (combine-same-weights (weighted-pairs s t pair-weight)
                        pair-weight))
```

Use this code to generate the Ramanujan numbers by first generating

```
(same-weight-pairs integers
                   integers
                   (lambda (i j) (+ (cube i) (cube j))))
```

and filtering the result for lists that contain more than one pair. What are the first five Ramanujan numbers?

Problem 9. In a similar way to problem 8, generate streams of

(a) All numbers that can be written as the sum of two squares in three different ways (showing how they can be so written). What are the five smallest such numbers?

(b) All numbers that can be written, in two different ways, in the form $i^3 + j^2$ where i is an odd positive integer and j is an even positive integer. (Also give, for each number, the appropriate values for i and j.) What are the five smallest such numbers?

What expressions did you use to generate these streams?

```
;; This is the code for Streams and Lazy Evaluation

;; The Scheme we're using does not have ``stream-car'' and ``stream-cdr''
;; but rather ``head'' and ``tail''.  Let's make it compatible with
;; the text.  (The first edition of A+S used ``head'' and ``tail''.)

(define stream-car head)
(define stream-cdr tail)

;; Now, on with the show...

(define (divisible? x y) (= (remainder x y) 0))

;; Useful stream utility functions

(define (stream-filter pred stream)
  (cond ((empty-stream? 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)))))

;;  Mapping functions

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

;;  Iterating along a stream.

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

```
;;  Streams of numbers

(define (add-streams s1 s2)
  (cond ((empty-stream? s1) s2)
        ((empty-stream? s2) s1)
        (else (cons-stream (+ (stream-car s1) (stream-car s2))
                           (add-streams (stream-cdr s1) (stream-cdr s2))))))

(define (scale-stream c stream)
  (stream-map (lambda (x) (* x c)) stream))

;;  Differs from book by not checking for empty streams
(define (interleave s1 s2)
  (cons-stream (stream-car s1)
               (interleave s2
                           (stream-cdr s1))))

(define (merge s1 s2)
  (cond ((empty-stream? s1) s2)
        ((empty-stream? s2) s1)
        (else
         (let ((h1 (stream-car s1))
               (h2 (stream-car s2)))
           (cond ((< h1 h2) (cons-stream h1 (merge (stream-cdr s1) s2)))
                 ((> h1 h2) (cons-stream h2 (merge s1 (stream-cdr s2))))
                 (else (cons-stream h1
                                    (merge (stream-cdr s1)
                                           (stream-cdr s2)))))))))

;; This next procedure is to be used in forming streams of pairs,
;; once you have defined the procedure MERGE-WEIGHTED
(define (weighted-pairs s t pair-weight)
  (cons-stream (cons (stream-car s) (stream-car t))
               (merge-weighted
                  (stream-map (lambda (x) (cons (stream-car s) x))
                              (stream-cdr t))
                  (weighted-pairs (stream-cdr s) (stream-cdr t) pair-weight)
                  (lambda (p) (pair-weight (car p) (cdr p))))))

;; This procedure forms streams of weighted pairs, where pairs of the
;; same weight have been combined.  In order to use it, you must
;; define an appropriate procedure COMBINE-SAME-WEIGHTS
(define (same-weight-pairs s t pair-weight)
  (combine-same-weights (weighted-pairs s t pair-weight)
                        pair-weight))
```

```
;;  Stream I/O.

(define print-stream
  (let ()
    (define (iter s)
      (if (empty-stream? s)
          (display "]")
          (begin (write (stream-car s))
                 (write " ")
                 (iter (stream-cdr s)))))
    (lambda (s)
      (write "")
      (iter s))))

;; You may wonder why PRINT-STREAM has been written in such an obscure
;; way, when
;; (define (print-stream s)
;;   (write "[")
;;   (stream-for-each (lambda (x) (write x) (write " ")) s)
;;   (write "]"))
;; would have the same effect.  If you think about the "actor model"
;; and tail recursion, however, you may begin to see why.

;;  For exercise 3.43
(define (show x)
  (write-line x)
  x)

(define (nth-stream n s)
  (if (= n 0)
      (stream-car s)
      (nth-stream (- n 1) (stream-cdr s))))

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