Streams

CS21b:
Structure and Interpretation
of Computer Programs
Spring Term, 2004
We’ve already seen how evaluation order can change behavior when we program with state.

Now we want to investigate how evaluation order can change efficiency — and more important, how changing evaluation order can open the door to a whole new perspective on data structures. In particular, infinite data structures...
Streams and stream processing

We have already seen the "signal processing" or "circuit" style of programming:

\[
\text{(sum (map square}
\quad \text{(filter prime?}
\quad \text{(enumerate-interval 2 1000000)))})
\]

Now, what about the "waste motion" of the following?

\[
\text{(car (cdr (map square}
\quad \text{(filter prime?}
\quad \text{(enumerate-interval 2 1000000)))})
\]

Instead: produce elements of a new kind of list -- a stream -- on a "need to know" basis, while preserving the “standard invariants”

\[
\text{(stream-car (cons-stream x y)) = x}
\]
\[
\text{(stream-cdr (cons-stream x y)) = y}
\]

Also introduce the-empty-stream, stream-null? etc.
Building blocks:

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

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

(define (stream-for-each proc s) ...)

(define (display-stream s)
  (stream-for-each (lambda (x) (display x) (newline))
                   s))

What changes by changing just a name? Nothing. **We need a new implementation...**
We need a new implementation...

(cons-stream a b) =def= (cons a (delay b))

...but how do we delay evaluation?...

(delay b) =def= (lambda () b)

Thus

(cons-stream a b) =def= (cons a (lambda () b))

[Think of this as syntactic sugar.] Cons-stream gets a pair, evaluates the first argument, but builds a procedure which can evaluate the second argument (if called). What does this mean in terms of environment diagrams?

(define (stream-car stream) (car stream))

or if you prefer,

(define stream-car car)

(define (stream-cdr stream) (force (cdr stream)))

where

(force s) =def= (s)

thus

(define (stream-cdr stream) ((cdr stream)))
Stream implementation in action:

(stream-car  
  (stream-cdr  
    (stream-enumerate-interval 2 1000000)))

(stream-car  
  (stream-cdr  
    (stream-filter prime?  
      (stream-enumerate-interval 2 1000000))))

Note

(define (stream-enumerate-interval from to)
  (if (> from to)
      the-empty-stream
      (cons-stream from (stream-enumerate-interval (+ from 1) to))))

Efficiency issues

(define s (stream-filter prime? (stream-enumerate-interval 2 1000000)))

(+ (select 1000 s) (select 1001 s))

Notice work gets repeated. Also, we are using:

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

Answer: memoization
A memoized version of streams

(define head car)
;Value: head

(define (tail s)
  (let ((t (cdr s)))
    (if (or (pair? t)
            (null? t))
      t
      (begin
       (set-cdr! s (t))
       (cdr s)))))
;Value: tail

(define (ints n)
  (cons n
    (lambda ()
      (ints (1+ n)))))
;Value: ints

(define N (ints 1))
;Value: N

N
;Value: (1 . #[compound-procedure 2])

(tail N)
;Value: (2 . #[compound-procedure 4])

N
;Value: (1 2 . #[compound-procedure 4])

(tail (tail N))
;Value: (3 . #[compound-procedure 6])

N
;Value: (1 2 3 . #[compound-procedure 6])
Memoization:

(define (memo-proc proc)
  (let ((already-run? #f) (result #f))
    (lambda ()
      (if (not already-run?)
          (begin (set! result (proc))
            (set! already-run? #t)
            result)
          result)))

Question: what does this look like in the environment model?
Now instead, use the syntactic sugar:

(delay b) =def= (memo-proc (lambda () b))
Infinite streams

Lists are *inductively defined*, and every one of them has *finite length*. (Prove it!) In contrast, streams may be *infinite* in length!

\[
(\text{define ones (cons-stream 1 ones)})
\]

Compare this with

\[
(\text{define ones (cons 1 ones)})
\]

[what happens here?]

\[
(\text{define (integers-from n)}
\quad (\text{cons-stream n (integers-from (+ n 1)))})
\]

\[
(\text{select 100 (integers-from 101)})
\]
Infinite streams, cont.

Integers not divisible by 7:

(define integers (integers-from 0))

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

(define no-sevens
  (stream-filter (lambda (x) (not (divisible? x 7))) integers))

Fibonacci numbers, yet again:

(define (fibgen a b) (cons-stream a (fibgen b (+ a b))))

(define fibs (fibgen 0 1))

Primes:

(define (sieve stream)
  (cons-stream
    (stream-car stream)
    (sieve (stream-filter (lambda (x)
                             (not (divisible? x (stream-car stream))))
             (stream-cdr stream))))))

(define primes (sieve (integers-from 2)))
Defining streams implicitly

Assume the streams are infinite...

(define (add-streams s1 s2)
    (cons-stream (+ (stream-car s1) (stream-car s2))
                (add-streams (stream-cdr s1)
                             (stream-cdr s2))))

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

(define fibs
    (cons-stream 0
                (cons-stream 1
                             (add-streams (stream-cdr fibs) fibs))))
Defining streams implicitly

(define primes
  (cons-stream 2
    (stream-filter prime?
      (integers-from 3))))

(define (prime? n)
  (define (iter ps)
    (if (> (square (stream-car ps)) n)
      #t
      (if (divisible? n (stream-car ps))
        #f
        (iter (stream-cdr ps))))
  (iter primes))

Note the list of primes is use to generate candidate prime divisors -- we just need to "prime the pump" (no pun intended...)

Defining infinite data structures

An infinite data structure is just a finite piece of “ordinary” data structure, together with a rule (i.e., procedure) for generating more of the structure. A simple example: infinite binary trees.

Syntactic sugar:

\[
\text{(make-tree root left right)} = \text{def=} \text{(list root (lambda () left) (lambda () right))}
\]

\[
\text{(define root car)}
\]
\[
\text{(define (left-tree tree) ((cadr tree)))}
\]
\[
\text{(define (right-tree tree) ((caddr tree)))}
\]

Suppose S1 and S2 are streams, and you want to make a tree of all the possible interleavings of S1 and S2 ...

\[
\text{(define (interleave S1 S2 rootvalue)}
\]
\[
\text{(make-tree rootvalue (interleave (stream-cdr S1) S2 (stream-car S1))) (interleave S1 (stream-cdr S2) (stream-car S2))-})
\]
Joint bank accounts: “Don’t cross the streams!”

One-client bank account, with a stream of deposits (- means withdrawal)

(define (make-deposit-account balance deposit-stream)
  (cons-stream
   balance
   (make-deposit-account (+ balance (head deposit-stream))
                        (tail deposit-stream))))

Two-client account (with two streams of deposits):

(define (joint-account balance harry-stream anne-stream)
  (make-tree balance
           (joint-account (+ balance (head harry-stream))
                         (tail harry-stream)
                         anne-stream)
           (joint-account (+ balance (head anne-stream))
                         harry-stream
                         (tail anne-stream))))

Every path through the tree indicates a specific interleaving...
Mapping a function over an infinite tree:

(define (map-tree proc tree)
  (list (proc (root tree))
        (lambda () (map-tree proc (left-tree tree)))
        (lambda () (map-tree proc (right-tree tree))))

Picking a (stream) path through an infinite tree:

(define (random-path-stream tree)
  (cons-stream (root tree)
               (random-path-tree
                 ((pick-random (subtrees tree)))))
What’s the difference between programs and data?

Here’s a program:

(define (square n) (* n n))
What’s the difference between programs and data?

(Philosophical analogue: do we represent knowledge extensionally [list], declaratively [specification of what is], or procedurally [how to]?)

Here’s a program:

```
(define (square n) (* n n))
```

No, this is data! You see, there’s an infinite stream of squares, and `(square n)` selects the nth element of this stream... Maybe a better differentiation is... static vs. dynamic?