Streams – the lazy way

Streams – the lazy way to infinity, and beyond…

Beyond Scheme – designing language variants:

- Streams – an alternative programming style

Streams – motivation

- Imagine simulating the motion of a ball bouncing against a wall
  - Use state variables, clock, equations of motion to update

Streams – Basic Idea

- Have each object output a continuous stream of information
  - State of the simulation captured in the history (or stream) of values
Remember our Lazy Language?

- Normal (Lazy) Order Evaluation:
  - go ahead and apply operator with unevaluated argument subexpressions
  - evaluate a subexpression only when value is needed
    - to print
    - by primitive procedure (that is, primitive procedures are "strict" in their arguments)
    - on branching decisions
    - a few other cases
- Memoization -- keep track of value after expression is evaluated
- Compromise approach: give programmer control between normal and applicative order.

Variable Declarations: lazy and lazy-memo

- Handle lazy and lazy-memo extensions in an upward-compatible fashion:
  
  \[
  \text{lambda (a (b lazy) c (d lazy-memo))} \ldots
  \]
- "a", "c" are normal variables (evaluated before procedure application)
- "b" is lazy; it gets (re)-evaluated each time its value is actually needed
- "d" is lazy-memo; it gets evaluated the first time its value is needed, and then that value is returned again any other time it is needed again.

The lazy way to streams

- Use cons
  
  \[
  \text{define (cons-stream x (y lazy-memo))}
  \]
  
  \[
  \text{(cons x y)}
  \]
  
  \[
  \text{(define stream-car car)}
  \]
  
  \[
  \text{(define stream-cdr cdr)}
  \]
- Or, users could implement a stream abstraction:
  
  \[
  \text{define (cons-stream x (y lazy-memo))}
  \]
  
  \[
  \text{(lambda (msg)}
  \]
  
  \[
  \text{((eq? msg 'stream-car) x)}
  \]
  
  \[
  \text{((eq? msg 'stream-cdr) y)}
  \]
  
  \[
  \text{else error "unknown stream msg" msg)\text{)}}}
  \]
  
  \[
  \text{(define (stream-car s) (s 'stream-car))}
  \]
  
  \[
  \text{(define (stream-cdr s) (s 'stream-cdr))}
  \]

Stream Object

- A pair-like object, except the cdr part is lazy (not evaluated until needed):

  \[
  \text{cons-stream} \quad \text{stream-car} \quad \text{stream-cdr}
  \]

  \[
  a \quad \text{value} \quad \text{thunk}
  \]

- Example
  
  \[
  \text{define x (cons-stream 99 (/ 1 0))}
  \]
  
  \[
  \text{(stream-car x) => 99}
  \]
  
  \[
  \text{(stream-cdr x) => error - divide by zero}
  \]

Decoupling computation from description

- Can separate order of events in computer from apparent order of events in procedure description

  \[
  \text{list-ref}
  \]
  
  \[
  \text{filter (lambda (x) (prime? x))}
  \]
  
  \[
  \text{(enumerate-interval 1 100000000)}
  \]
  
  \[
  \text{100)}
  \]
  
  \[
  \text{(define (stream-ref (stream-interval a b))}
  \]
  
  \[
  \text{(if (> a b)}
  \]
  
  \[
  \text{the-empty-stream}
  \]
  
  \[
  \text{(cons-stream a (stream-interval (+ a 1) b))})}
  \]
  
  \[
  \text{(define (stream-ref (stream-filter (lambda (x) (prime? x))}
  \]
  
  \[
  \text{(stream-interval 1 100000000))}
  \]
  
  \[
  \text{100)}
  \]
Stream-filter

```
(define (stream-filter pred str)
  (if (pred (stream-car str))
      (cons-stream (stream-car str)
                   (stream-filter pred
                                   (stream-cdr str)))
      (stream-filter pred
                      (stream-cdr str))))
```

Decoupling Order of Evaluation

- Demo
- I need three volunteers.

```
(stream-ref
  (stream-filter (lambda (x) (prime? x))
                  (stream-interval 2 10000000))
  4)
```

Decoupling Order of Evaluation

```
(stream-filter prime? (str-in 1 10000000))
(stream-filter prime? (stream-cdr 2 10000000))
(stream-filter prime? (stream-cdr ones))
```

One Possibility: Infinite Data Structures!

- Some very interesting behavior

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

Finite list procs turn into infinite stream procs

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

```
(define ints
  (add-streams (stream-car ones) (stream-car ints)))
```

Finding all the primes

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```
Using a sieve

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

(define primes
  (sieve (stream-cdr ints)))

Streams Programming

• Signal processing:

  \[ x[n] \rightarrow \text{Delay} \rightarrow y[n] \]

• Streams model:

Integration as an example

(define (integral integrand init dt)
  (define int
    (cons-stream
     init
     (add-streams (stream-scale dt integrand)
                  init)))
  int)

(integral ones 0 2)

⇒ 0 0 0 0 0

Ones: 1 1 1 1 1
Scale 2 2 2 2 2

An example: power series

Think about this in stages, as a stream of values

(define (powers x)
  (cons-stream 1
               (scale-stream x (powers x))))

(define facts
  (cons-stream 1
               (mult-streams (stream-cdr ints) facts)))

⇒ 1 2 6 24 ...

An example: power series

(define (series-approx coeffs)
  (lambda (x)
    (mult-streams
     (div-streams (powers x) (cons-stream 1 facts))
     coeffs)))

\[ g(x) = g(0) + x g'(0) + x^2/2 g''(0) + x^3/3! g'''(0) + \ldots \]

For example:

\[ \cos(x) = 1 - x^2/2 + x^4/24 - \ldots \]

\[ \sin(x) = x - x^3/6 + x^5/120 - \ldots \]

An example: power series

(define (stream-accum str)
  (cons-stream (stream-car str)
               (add-streams (stream-accum
                             str) (stream-cdr str))))
An example: power series

(define (power-series g)
  (lambda (x)
    (stream-accum ((series-approx g) x))))

(define sine-coeffs
  (cons-stream 0
    (cons-stream 1
      (cons-stream -1 sine-coeffs)))))

(define cos-coeffs (stream-cdr sine-coeffs))

Using streams to decouple computation

• Here is our old SQRT program

(define (sqrt x)
  (define (try guess)
    (if (good-enough? guess)
        guess
        (try (improve guess))))

(define (improve guess)
  (average guess (/ x guess))))

(define (good-enough? guess)
  (close? (square guess) x))

(try 1))

• Unfortunately, it intertwines stages of computation

Using streams to decouple computation

• So let's pull apart the idea of generating estimates of a sqrt from the idea of testing those estimates

(define (sqrt-improve guess x)
  (average guess (/ x guess))))

(define (sqrt-stream x)
  (cons-stream
    1.0
    (stream-map (lambda (g) (sqrt-improve g x))
      (sqrt-stream x))))

(print-stream (sqrt-stream 2))

Using streams to decouple computation

• That was the generate part, here is the test part...

(define (stream-limit s tol)
  (define (iter s)
    (let ((f1 (stream-car s))
          (f2 (stream-car (stream-cdr s))))
      (if (close-enough? F1 f2 tol)
          f2
          (iter (stream-cdr s))))
  (iter s))

(stream-limit (sqrt-stream 2) 1.0e-5)

;Value: 1.412135623746899

• This reformulates the computation into two distinct stages: generate estimates and test them.

Do the same trick with integration

(define (trapezoid f a b h)
  (let ((dx (* (- b a) h))
         (n (/ 1 h)))
    (define (iter j sum)
      (if (> j n)
          sum
          (let ((f1 (* - b a))
                (n (/ 1 h)))
            (define (iter j sum)
              (if (> j n)
                  sum
                  (iter (+ j 1) (+ sum (f (+ a (* j dx)))))
              (* dx (iter 1 (+ (/ (f a) 2)
                            (/ (f b) 2))))))))

• So this gives us a good approximation to pi, but quality of approximation depends on choice of trapezoid size. What happens if we let \( h \to 0? \)
Accelerating a decoupled computation

(\text{define} \ (\text{keep-halving} \ R \ h) \\
\ (\text{cons-stream} \\
\ (R \ h) \ \\
\ (\text{keep-halving} \ R \ (/ \ h \ 2))))

(\text{print-stream} \\
\ (\text{keep-halving} \\
\ (\text{lambda} \ (h) \ (\text{trapezoid} \ \text{witch} \ 0 \ 1 \ h)) \\
\ 0.1))

Convergence – getting about 1 new digit each time, but each line takes twice as much work as the previous one!!!

(\text{stream-limit} \ (\text{keep-halving} \\
\ (\text{lambda} \ (h) \ (\text{trapezoid} \ \text{witch} \ 0 \ 1 \ h)) \\
\ .5) \\
\ 1.0e-9)

Value: 3.14159265343456 – takes 65,549 evaluations of \text{witch}

Summary

• Lazy evaluation – control over evaluation models
  • Convert entire language to normal order
  • Upward compatible extension
    – lazy & lazy-memo parameter declarations

• Streams programming:
  a powerful way to structure and think about computation