Streams – the lazy way

Beyond Scheme – designing language variants:

- Streams – an alternative programming style

Streams – motivation

- State of the simulation captured in instantaneous values of state variables

| Clock: 1 | Ball: (x1 y1) | Wall: e1 |
| Clock: 2 | Ball: (x2 y2) | Wall: e2 |
| Clock: 3 | Ball: (x3 y3) | Wall: e2 |
| Clock: 4 | Ball: (x4 y4) | Wall: e2 |
| Clock: 5 | Ball: (x5 y5) | Wall: e3 |

Streams – Basic Idea

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

Streams – motivation

- Another view of the same information

| Clock: 1 | Ball: | Wall: |
| Clock: 2 | (x1 y1) | e1 |
| Clock: 3 | (x2 y2) | e2 |
| Clock: 4 | (x3 y3) | e2 |
| Clock: 5 | (x4 y4) | e2 |
| ... | (x5 y5) | e3 |

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.

\[
\text{(lambda (a (b lazy) c (d lazy-memo)) ...)}
\]

- "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

\[
\text{(define (cons-stream x (y lazy-memo)) (cons x y))}
\]

\[
\text{(define stream-car car)}
\]

\[
\text{(define stream-cdr cdr)}
\]

- Or, users could implement a stream abstraction:

\[
\text{(define (cons-stream x (y lazy-memo))}
\]

\[
\text{(lambda (msg) (cond ((eq? msg 'stream-car) x)
\text{((eq? msg 'stream-cdr) y)
\text{(else error "unknown stream msg" msg))}))}
\]

\[
\text{(define (stream-car s) (s 'stream-car))}
\]

\[
\text{(define (stream-cdr s) (s 'stream-cdr))}
\]

Stream Object

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

\[
\text{a\ value\ thunx}
\]

- Example

\[
\text{(define x (cons-stream 99 (/ 1 0)))}
\]

\[
\text{(stream-car x) => 99}
\]

\[
\text{(stream-cdr x) => error \text{ divide by zero}}
\]

Because stream-cdr is same as cdr, this is a primitive procedure application, hence forces evaluation

Decoupling computation from description

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

\[
\text{(list-ref}
\text{(filter (lambda (x) (prime? x))}
\text{(enumerate-interval 1 100000000))}
\text{100)}
\]

\[
\text{(define (stream-ref (stream-filter (lambda (x) (prime? x))}
\text{(stream-interval 1 100000000))}
\text{100)}
\]

\[
\text{Creates 100K elements}
\]

\[
\text{Creates 1M elements}
\]

\[
\text{(define (stream-filter pred str)
\text{(if (pred (stream-car str))}
\text{(cons-stream (stream-car str)
\text{(stream-filter pred
\text{(stream-cdr str))}))
\text{(stream-filter pred
\text{(stream-cdr str))})})}
\]

\[
\text{(stream-ref (stream-filter (lambda (x) (prime? x))}
\text{(stream-interval 1 1000000000))}
\text{100)}
\]

\[
\text{Creates 1 element, plus a promise}
\]

\[
\text{Creates 1 element, plus a promise}
\]

Decoupling Order of Evaluation

\[
\text{(stream-ref}
\text{(stream-filter (lambda (x) (prime? x))}
\text{(stream-interval 2 1000000000))}
\text{100)}
\]
Decoupling Order of Evaluation

```
(stream-filter prime? (str-in 1 10000000))
(stream-filter prime? (str-in 2 100000000))
(stream-filter prime? (str-in 2 10000000))
```

From recursive call

One Possibility: Infinite Data Structures!

- Some very interesting behavior
  ```lisp```
  (define ones (cons 1 ones))
  (stream-car (stream-cdr ones)) ⇒ 1
  ```

Finite list procs turn into infinite stream procs

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

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

Finding all the primes

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

Streams Programming

- Signal processing:
  ```
  \( x[n] \xrightarrow{\text{Delay}} y[n] \)
  ```

- Streams model:
Integration as an example

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

(integral ones 0 2)
=> 0 2 4 6 8
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))))

⇒ 1 2 6 24 ...

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

=> 1 2 6 24 ...

Think of (powers x) as giving all the powers of x starting at 1, then multiplying these two streams together gives a stream whose nth element is (n+1)!

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

An example: power series

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

For example:

\( \cos(x) = 1 - x^2/2 + x^4/24 - \ldots \)
\( \sin(x) = x - x^3/6 + x^5/120 - \ldots \)
Using streams to decouple computation

- So let’s pull apart the idea of generating estimates of a square root 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))

1.0  1.5  1.4166666666666665  1.4142156862745097
1.4142135623745899  1.414213562373095
1.414213562373095

Note how fast it converges!

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
          (iter (+ j 1) (+ sum (+ (f (+ a (* j dx)))
                                     (f (+ b (* j dx)))))))))

(* dx (iter 1 (+ (/ (f a) 2)
                 (/ (f b) 2))))

Do the same trick with integration

(define (witch x) (/ 4 (+ 1 (* x x))))

(trapezoid witch 0 1 0.1)

;Value: 3.1399259889071587

(trapezoid witch 0 1 0.01)

;Value: 3.14157598692313

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

(define (keep-halving R h)
  (cons-stream
   R h
   (keep-halving R (/ h 2)))))

(print-stream
  (keep-halving
   (lambda (h)
        (trapezoid 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!!

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