6.001 SICP
Streams – the lazy way

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

![Diagram of a ball and a wall with variables: position, velocity, elasticity, time]

Streams – motivation

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

<table>
<thead>
<tr>
<th>Clock: 1</th>
<th>Ball: (x1 y1) Wall: e1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock: 2</td>
<td>Ball: (x2 y2) Wall: e2</td>
</tr>
<tr>
<td>Clock: 3</td>
<td>Ball: (x3 y3) Wall: e2</td>
</tr>
<tr>
<td>Clock: 4</td>
<td>Ball: (x4 y4) Wall: e2</td>
</tr>
<tr>
<td>Clock: 5</td>
<td>Ball: (x5 y5) Wall: e3</td>
</tr>
</tbody>
</table>

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)
  • 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:

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

- "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

\[
\begin{align*}
&\text{define} \ (\text{cons-stream } x \ {\text{y lazy-memo}}) \\
&\text{(cons } x \ y) \\
&\text{define stream-car car} \\
&\text{define stream-cdr cdr}
\end{align*}
\]

- Or, users could implement a stream abstraction:

\[
\begin{align*}
&\text{define} \ (\text{cons-stream } x \ {\text{(y lazy-memo)}}) \\
&\text{(lambda } (\text{msg}) \\
&\quad \text{(cond } (\text{eq? msg 'stream-car}) x \\
&\quad \quad (\text{eq? msg 'stream-cdr}) y \\
&\quad \quad \text{(else error "unknown stream msg" msg}))))) \\
&\text{define} \ (\text{stream-car } s) \ (s \ '\text{stream-car}) \\
&\text{define} \ (\text{stream-cdr } s) \ (s \ '\text{stream-cdr})
\end{align*}
\]

Stream Object

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

\[
\begin{align*}
\text{cons-stream} & \quad \text{stream-car} \\
\text{stream-cdr} & \quad \text{a value} \\
\text{a thunk} & \quad \text{cons-stream}
\end{align*}
\]

- Example

\[
\begin{align*}
&\text{define} \ x \ (\text{cons-stream} \ 99 \ (/ \ 1 \ 0))) \\
&(\text{stream-car } x) \Rightarrow 99 \\
&(\text{stream-cdr } x) \Rightarrow \text{error - divide by zero}
\end{align*}
\]

Stream-filter

\[
\begin{align*}
&\text{define} \ (\text{stream-filter} \ \text{pred} \ str) \\
&\quad (\text{if} \ (\text{pred} \ (\text{stream-car} \ str)) \\
&\quad \quad (\text{cons-stream} \ (\text{stream-car} \ str) \\
&\quad \quad \ (\text{stream-filter} \ \text{pred} \\
&\quad \quad \quad (\text{stream-car} \ str))) \\
&\quad \quad (\text{stream-filter} \ \text{pred} \\
&\quad \quad \quad (\text{stream-cdr} \ str)))
\end{align*}
\]

Decoupling computation from description

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

\[
\begin{align*}
&\text{list-ref} \\
&\quad (\text{filter} \ (\text{lambda} \ (x) \ (\text{prime?} \ x))) \\
&\quad \ (\text{enumerate-interval} \ 1 \ 100000000)) \\
&\quad 100 \\
&\quad (\text{define} \ (\text{stream-interval} \ a \ b) \\
&\quad \ (\text{if} \ (> \ a \ b) \\
&\quad \quad \text{the-empty-stream} \\
&\quad \quad \ (\text{cons-stream} \ a \ (\text{stream-interval} \ (+ \ a \ 1) \ b)))) \\
&\quad (\text{stream-ref} \\
&\quad \quad (\text{stream-filter} \ (\text{lambda} \ (x) \ (\text{prime?} \ x))) \\
&\quad \quad (\text{stream-interval} \ 1 \ 1000000000)) \\
&\quad 100
\end{align*}
\]

Decoupling Order of Evaluation

\[
\begin{align*}
&\text{stream-filter prime?} \ (\text{str-in} \ 1 \ 100000000)) \\
&\quad (\text{stream-filter prime?} \ 1 \ (\text{str-in} \ 2 \ 100000000)) \\
&\quad (\text{stream-filter prime?} \ (\text{str-in} \ 3 \ 100000000)) \\
&\quad (\text{stream-filter prime?} \ (\text{str-in} \ 3 \ 100000000))
\end{align*}
\]

Decoupling Order of Evaluation
One Possibility: Infinite Data Structures!

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

  Ones: 1 1 1 1 1 ....

- Compare:
  
  ```lisp
  (define ones (cons 1 ones)) => error, ones undefined
  ```

---

Finite list procs turn into infinite stream procs

```lisp
(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
  (cons-stream 1 (add-streams ones 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  97 101 103 107 109 113
```

Remember our sieve?

```lisp
(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] y[n] Delay
```

- Streams model:

```
  x  +  +  +
  |  |  |
  +  +  +
  add-streams stream-cdr stream-scale G  y
```

Integration as an example

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

(integral ones 0 2)
```

(integral ones 0 2)

Ones: 1 1 1 1 1

Scale: 1 2 3 4 5

1

=> 0 2 4 6 8
An example: power series

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

For example:

\[ \cos(x) = 1 - \frac{x^2}{2} + \frac{x^4}{24} - \ldots \]
\[ \sin(x) = x - \frac{x^3}{6} + \frac{x^5}{120} - \ldots \]

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))))
```

Note how fast it converges!
Using streams to decouple computation

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

\[
\text{(define (stream-limit s tol)}
\]
\[
\text{(define (iter s))}
\]
\[
\text{(/ (stream-car s)) (if (close-enough? (stream-car (stream-cdr s))) tol s))}
\]

\[
\text{define (stream-limit (sqrt-stream 2) 1.0e-5)}
\]

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

Do the same trick with integration

\[
\text{define (trapezoid f a b h)}
\]
\[
\text{(let ((dx (* (- b a) h)) (n (/ 1 h)))}
\]
\[
\text{(define (iter j sum)) (if (> j n) sum (sum (+ sum (f (+ a (* j dx)))))) (* dx (iter 1 (+ (/ (f a) 2) (/ (f b) 2)))))}
\]

- 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

\[
\text{(define (keep-halving R h)}
\]
\[
\text{(cons-stream (R h)} (keep-halving (lambda (h) (trapezoid witch 0 1 h)) (/ h 2))))}
\]

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