Lazy Evaluation

CS 1025 Computer Science Fundamentals I

Stephen M. Watt
University of Western Ontario

Lazy Evaluation

- Sometimes you might or might not use a value that is expensive to compute.
- Why compute it if you don't need it.
- Sometimes you might need it, but not right away.
  (So you could wait until a processor is free...)
- *Why do today what you can put off to tomorrow?*

  (Isn't this the opposite of what you'd expect me to say?)

Evaluation Order

- Sometimes a programming language doesn't specify evaluation order.
- "Applicative order" evaluates the arguments before calling the function. This is "eager". Used in most programming languages.
- "Normal order" evaluates the arguments just before they are used inside a function.
  This is "lazy". Used in a lot of theory and some programming languages.

Lazy Evaluation in Scheme

- “delay” creates a *promise* ... An object that may be evaluated later.
- “force” causes the promise to be evaluated to give a value.

- Example:

```
(define do-it (lambda (a b)
     (write “Hello”) (newline) (+ a b)))

(define five (delay (do-it 2 3))) ; do-it not called yet
<#promise>
...
...
(define n (force five))           ; do-it called here.
“Hello”
5
```

Another Example

```
(define big   (lambda () (write "big")   (newline) (+ 1 1)))
(define hairy (lambda () (write "hairy") (newline) (+ 2 2)))
(define comp  (lambda () (write "comp")  (newline) (+ 3 3)))

(define l (cons (delay (big))
                (cons (delay (hairy))
                      (cons (delay (comp)) '()) ) ) )
l
(#<promise> #<promise> #<promise>)
(length l)
3
(force (cadr l))
"hairy"
4
(force (cadr l))
4
```

Delay and Force in Scheme

- `delay` must capture an expression so it can be evaluated later.

  It can be implemented in terms of a macro which puts the expression inside a lambda.

  The resulting "promise" object would then refer to this function.

- `force` must be able to tell whether a promise needs to be evaluated (and then do the evaluation) or whether it simply contains the result (and then return it).

- Let us represent a promise, then, as a pair whose car is either #t, indicating the cdr is the value desired or #f, indicating that the cdr is the lambda to compute the value.

Delay and Force in Scheme

- Then delay and force can be implemented as

```
 ;; A Scheme macro:  (delay foo) -> (cons #f (lambda () fo
(define-syntax delay (syntax-rules ()
   ((_ expr) (cons #f (lambda () expr))) ))

 (define force (lambda (p)
     (if (car p)
         (cdr p)
         (let ((x ((cdr p)))) ; Call the fn in p’s cdr
             (set-cdr! p x) (set-car! p #t) x ) ) ))
```

Lazy Lists

- We define the basic operators:
  lazy-cons   lazy-car   lazy-cdr lazy-null?

```
(define-syntax lazy-cons (syntax-rules ()
        ((_ <car-expr> <cdr-expr>)
         (delay (cons <car-expr> <cdr-expr>))) ))

(define lazy-null? (lambda (ll) (null? (force ll))))
(define lazy-car   (lambda (ll) (car   (force ll))))
(define lazy-cdr   (lambda (ll) (cdr   (force ll))))
```

Lazy List Example

```
(define say (lambda (a)
       (write "Say") (write a) (newline) a ))

(define ll (lazy-cons (say "My")
                 (lazy-cons (say "car")
                       (lazy-cons (say "drives!") ’()) )))
ll
#<promise>

(lazy-car ll)
"Say""My" <-- Printed as side effect
"My"      <-- Value

(lazy-car ll)
  “My”      <-- Value

(define b (lazy-car (lazy-cdr ll)))
"Say""car" <-- Side effect
"car"      <-- Value
```

Infinite Series!

- This use some math.

  We will eventually use the following facts:

  sin(*x*) = x – x^3/3! + x^5/5! – x^7/7! + ...
  cos(x) = 1 – x^2/2! + x^4/4! – x^6/6! + ...

- We can represent an infinite series as a lazy list of coefficients.

  E.g. sin(x) would be the lazy list of

  0   1   0   -1/6   0   1/120   0   -1/5040   ...

Making Infinite Series

- This function makes a series, given a function to compute the *i*-th coefficient.

```
(define series-from-coef-fun (lambda (f)
    (define make-tail (lambda (i)
             (lazy-cons (f i) (make-tail (+ i 1))) ))
    (make-tail 0) ))
```

- Note the inner recursive function has no if statement, and so *has no base case!!!*

- We use lazy-evaluation to delay the infinite recursion when making an infinite list.

Printing Lazy Series

```
(define series->string (lambda (s n)
   (let ((r '())) ; Collected parts in reverse order

      (do ((ll s (lazy-cdr ll)) ; Current tail
           (i 0 (+ i 1))) ; Current exponent
          ((> i n)) ; End when i > n.

          (let ((ci (lazy-car ll))) ; Current coefficient
             (cond ((> ci 0) (set! r (cons " + " r)))
                   ((< ci 0) (set! r (cons " - " r)) (set! ci (- ci)) ) )

             (if (not (= ci 0)) (begin
               (set! r (cons (number->string ci) r))
               (if (> i 0) (set! r (cons " x" r)))
               (if (> i 1) (set! r (cons (number->string i)(cons "^" r)))

      ;; Now the parts are collected, finish up.
      (if (null? r) (set! r '("0")))
      (set! r (cons " + ..." r))
      (apply string-append (reverse r)) ) ))
```

Lazy Series Example

```
(define s (series-from-coef-fun (lambda (i) (* i i)) ))

(series->string s 4)

" + 1 x + 4 x^2 + 9 x^3 + 16 x^4 + ..."
```

Question: How to Implement + ?

- How would you go about writing an addition function which makes a new series by adding two existing ones coefficient by coefficient?

Answer

- This program adds series:

```
(define series-+ (lambda (sa sb)
      (lazy-cons (+ (lazy-car sa) (lazy-car sb)))
                 (series-+ (lazy-cdr sa) (lazy-cdr sb)) ))
```

- Again, note that with lazy eval we can have a recursive function with no base case.

- Examples:

```
(define s1 (series-from-coef-fun (lambda (i) (* 2 i)) ))
(series->string s1 4)
" + 2 x + 4 x^2 + 6 x^3 + 8 x^4 + ...“

(define s2 (series-from-coef-fun (lambda (i) i) ))
(series->string s2 4)
" + 1 x + 2 x^2 + 3 x^3 + 4 x^4 + ...“

(define s3 (series-+ s1 s2))
(series->string s3 4)
" + 3 x + 6 x^2 + 9 x^3 + 12 x^4 + ..."
```

Another Example: Multiplication

- The coefficient of xn in the product s1 × s2 is given by

```
coef(s1,n)*coef(s2,0) + coef(s1,n-1)*coef(s2,1) + ... + coef(s1,0)*coef
```

- We will need a program to find the i-th coefficient of a given series:

```
(define series-coef (lambda (s i)
     (if (= i 0) (lazy-car s)
                 (series-coef (lazy-cdr s) (- i 1)) ) ))
```

- The program for the n-th term of the product is:

```
(define series-*-term (lambda (n s1 s2)
     (do ((i 0 (+ 1 i)) (sum 0))
         ((> i n) sum)
       (set! sum (+ sum (* (series-coef s1 (- n i)) (series-coef s2 i))
```

- The program for the product is then

```
(define series-* (lambda (s1 s2)
       (define make-tail (lambda (i)
                (lazy-cons (series-*-term i s1 s2) (make-tail (+ i 1))
       (make-tail 0) ))
```

Tying It All Together

- Let's test our package by seeing whether sin^2 (x) + cos^2 (x) = 1
- The functions below calculate the coefficients of sin and cos.

```
(define fact (lambda (i) (if (= i 0) 1 (* i (fact (- i 1))))))
(define sin-coef (lambda (i)
      (if (even? i) 0 (/ (expt -1 (/ (- i 1) 2)) (fact i)))))
(define cos-coef (lambda (i)
      (if (odd? i) 0 (/ (expt -1 (/ i 2)) (fact i)))))
```

- See that the series for sin and cos are right:

```
(define s (series-from-coef-fun sin-coef))
(series->string s 8)
" + 1 x - 1/6 x^3 + 1/120 x^5 - 1/5040 x^7 + ..."

(define c (series-from-coef-fun cos-coef))
(series->string c 8)
" + 1 - 1/2 x^2 + 1/24 x^4 - 1/720 x^6 + 1/40320 x^8 + ..."
```

Tying It All Together

- Compute sin^2+cos^2.

```
(define sscc (series-+ (series-* s s) (series-* c c)))
(series->string sscc 10)
" + 1 + ...“

(series->string sscc 100)
" + 1 + ..."
```