CS342: Organization of Prog. Languages

Topic 13: Lazy Evaluation

- Lazy Evaluation — The main idea
- Evaluation order
- Avoiding extra work...
- Delay and Force in Scheme
- An Implementation of Delay and Force
- An extended example using lazy evaluation.

Lazy Evaluation — The main idea

- Why do computations we might not need?

```
// c may be zero, sometimes...
r = c * big_hairy_computation(x,y,z);
```

```
;; We only count the entries...
(length (cons (hairy) (cons (big) (cons (comp) '()))))
```

- How can we tell whether the computation will be needed?

A General Framework

- Consider a general function call

  funExpr(argExpr1, ..., argExprN)

  We can write this in abstract syntax as

  call(funExpr, argExpr1, ... argExprN)

- The sub-expressions funExpr, argExpr1, ..., can themselves be function calls, e.g.

  call(**call**(f1,a1,a2),**call**(f2,a3,a4),**call**(f3,a5,a6))

Evaluation Orders

- There are several different orders the functions can be called:
- If the calls are made from *innermost* to the outermost, then we have **eager evaluation**.

  Many languages (e.g. C, Java, Scheme, ...) specify innermost to outermost evaluation of calls, without specifying the order of the calls within each level.
- If the calls are made starting from the *outermost*, then we have implicit **lazy evaluation**. (As opposed to the explicit kind, with delay and force).

  If we further specify that, at each level, the *leftmost* call is made first (i.e. outermost-leftmost), then we have what is called **normal-order evaluation**, an order with useful theoretical properties.

Avoiding extra work...

- Static analysis – local:

  ```
  if (Monday) {
      int c = 0;
      return c * big_hairy_deal(x,y,z);
  }
  ```

  Uses techniques such as *peep-hole optimization* and *constant propagation* within basic blocks.

- Static analysis – global:

  ```
  int i = 0;
  for (i = 0; i < 10; i++) printf("!");
  return (i - 10) * big_hairy_deal(x,y,z);
  ```

  Uses techniques such as *data flow analysis*.

Avoiding extra work... part 2

- Run-time methods:

  Leave decision until the value is required for some operation.

  We can do this by hand with `delay` and `force` in some languages.

- `delay` takes an expression and returns a so-called "promise". The expression is not evaluated in doing this.

- `force` takes a promise and evaluates the expression captured inside and returns the value.

  Subsequent uses of `force` on the same promise do not re-evaluate, they simply return the value.

Delay and Force in Scheme

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

An Implementation of Delay and Force

- `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 albe 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.

- Then `delay` and `force` can be implemented as

```
(define-syntax delay (syntax-rules ()
  ((_ expr) (cons #f (lambda () expr))) ))

(define (force p)
   (if (car p)
     (cdr p)
     (let ((x ((cdr p)))) ; Call the fn in (cdr p)
        (set-cdr! p x) (set-car! p #t) x ) ) )
```

- Example:

```
(define dd (delay (begin (write "Hello") (+ 1 1))))
; dd is (cons #f (lambda () (begin (write "Hello") (+ 1 1))))

(force dd)
; finds car dd is #f
; calls cdr dd, i.e. (lambda () ...)
;    writes "Hello"
;    computes (+ 1 1), giving 2
; saves 2 in f's variable x
; sets cdr of dd to be x, i.e. 2
; sets car of dd to be #t
; returns 2
```

Force and Delay – Recap

- Recall the two basic operators of explicit lazy evaluation:
- `(delay <expr>)`
  creates a "promise" object without evaluating `<expr>`.
- `(force <promise>)`
  takes a promise-valued expression and forces it to evaluate, giving its value (and causing whatever side effects it may have).
- Example:

```
> (define complaint
    (delay (begin
      (write "Gripe")(newline) (write "Gripe")(newline) (write "Gripe")(newline)
      99 )) )

> complaint
#<promise>

> (force complaint)
"Gripe"
"Gripe"
"Gripe"
99
```

Lazy Lists

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

```
> (require-library "synrule.ss")

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

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

- Example

```
> (define (say 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

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

> b
"car"             <-- Value
```

Lazy Series

- This uses some math.

  We will eventually use the following facts

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ...$$

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} - ...$$

- Represent an infinite series as a lazy list of coefficients.

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

```
0  1  0 -1/3!  0  1/5!  0 -1/7! ...
```

  which is

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

Making Infinite Lazy Series

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

```
(define (series-from-coef-fun f)
   (define (make-tail 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

- This fn converts the first n terms of the series to a string.
  Must give n, otherwise fn would never reach end of the infinite list!

```
(define (series->string 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

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

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

```
(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 x^n in the product $s1 \times s2$ is given by

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

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

```
(define (series-coef 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 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-* s1 s2)
   (define (make-tail 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 i) (if (= i 0) 1 (* i (fact (- i 1)))))
> (define (sin-coef i)
     (if (even? i) 0 (/ (expt -1 (/ (- i 1) 2)) (fact i))))
> (define (cos-coef 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 + ..."
```

- Compute $\sin^2 + \cos^2$.

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

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

Summary – Lazy Evaluation

- Lazy evaluation can be used when dealing with conceptually infinite data structures.
- Lazy evaluation can be used to avoid un-necessary computation even with finite data structures.
- There is an overhead to delay a computation.
- Either eager evaluation or lazy evaluation may be more useful/efficient depending on the circumstances.