Environment model

Rules of evaluation:

- If expression is self-evaluating (e.g. a number), just return value
- If expression is a variable, look up value associated with that name in environment
- If expression is a lambda, create procedure and return
- If expression is a special form (e.g. if) follow specific rules for evaluating subexpressions
- If expression is a compound expression
  - Evaluate subexpressions in any order
  - If first subexpression is primitive (or built-in) procedure, just apply it to values of other subexpressions
  - If first subexpression is compound procedure (created by lambda), evaluate the body of the procedure in a new environment, which extends the environment of the procedure with a new frame in which the procedure's parameters are bound to the supplied arguments

Applicative Order (aka Eager evaluation):
- evaluate all arguments, then apply operator

Normal Order (aka Lazy 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)
  - to test (if predicate)
  - to apply (operator)

Making Order of Evaluation Visible

- (define (notice x)
  (display "noticed")
  x)
- (+ (notice 52) (notice (+ 4 4))
  noticed
  noticed
  => 60

Applicative Order Example

```
(define (foo x)
  (display "inside foo")
  (+ x x))

(foo (notice 222))
=> (notice 222)
=> 222
=> (begin (display "inside foo")
       (+ 222 222))

noticed
inside foo
=> 444
```

We first evaluated argument, then substituted value into the body of the procedure
Normal Order Example

(define (foo x)
  (display "inside foo")
  (+ x x))

(foo (notice 222))

=> (begin (display "inside foo")
     (+ (notice 222) (notice 222)))

As if we substituted the unevaluated expression in the body of the procedure

inside foo
noticed
noticed

=> 444

Applicative Order vs. Normal Order

(define (foo x)
  (display "inside foo")
  (+ x x))

(foo (notice 222))

Applicative order

inside foo
noticed
noticed

Normal order

Think of as substituting values for variables in expressions

Think of as expanding expressions until only involve primitive operations and data structures

How can we change our evaluator to use normal order?

- Create "promises" – expressions whose evaluation has been delayed
- Change the evaluator to force evaluation only when needed

Why is normal order useful?

- What kinds of computations does it make easier?

m-apply – the original version

(define (m-apply procedure arguments)
  (cond ((primitive-procedure? procedure)
    (apply-primitive-procedure procedure arguments))
    ((compound-procedure? procedure)
      (eval-sequence (procedure-body procedure)
        (extend-environment (procedure-parameters procedure) arguments
        (procedure-environment procedure))))
    (else (error "Unknown procedure" procedure))))

Delayed expressions

Need to create delayed version of arguments that will lead to values

Need to convert to actual values

How can we implement lazy evaluation?

(define (l-apply procedure arguments env)
  ; changed
  (cond ((primitive-procedure? procedure)
    (apply-primitive-procedure procedure arguments))
    ((compound-procedure? procedure)
      (eval-sequence (procedure-body procedure)
        (extend-environment (procedure-parameters procedure) arguments
        (procedure-environment procedure))))
    (else (error "Unknown proc" procedure))))

Lazy Evaluation – l-eval

• Most of the work is in l-apply; need to call it with:
  – actual value for the operator
  – just expressions for the operands
  – the environment...

(define (l-eval exp env)
  (cond ((self-evaluating? exp) exp)
    ... (application? exp
      (l-apply (actual-value (operator exp) env)
        (operands exp env)
        env))
    (else (error "Unknown expression" exp))))
m-eval versus l-Eval

(define (m-eval exp env)
  (cond ((self-evaluating? exp) exp)
        (cond? exp (m-eval (cond->if exp) env))
        ((application? exp) (m-eval (m-eval (operator exp) env) (list-of-values (operands exp) env))
         (else (error "Unknown expression type -- EVAL" exp))))
  (define (l-eval exp env)
    (cond ((self-evaluating? exp) exp)
          (cond? exp ((l-eval (cond->if exp) env))
               ((application? exp) (l-eval (apply (actual-value (operator exp) env) (operands exp) env))
                (else (error "Unknown expression type -- EVAL" exp))))
    (define (actual-value exp env)
      (force-it (l-eval exp env)))
    (define (list-of-delayed-args exps env)
      (if (no-operands? exps) '()
          (cons (delay-it exp env)
                (list-of-delayed-args (rest-operands exps) env)))
    (define (list-of-arg-values exps env)
      (if (no-operands? exps) '()
          (cons (actual-value (first-operand exps) env)
                (list-of-arg-values (rest-operands exps) env)))
    (define (delay-it exp env)
      (list '-' promise exp env))
    (define (promise? obj)
      (tagged-list? obj 'promise))
    (define (promise-exp promise)
      (cadr promise))
    (define (promise-env promise)
      (caddr promise))
    (define (force-it obj)
      (cond ((promise? obj)
                  (actual-value (promise-exp obj) (promise-env obj)))
            (else obj)))
    (define (actual-value exp env)
      (force-it (l-eval exp env)))
    (define (force-it obj)
      (cond ((promise? obj)
                  (actual-value (promise-exp obj) (promise-env obj)))
            (else obj)))
    (define (force-it (l-eval exp env)))
    (define (actual-value exp env)
      (force-it (l-eval exp env)))
    (define (list-of-delayed-args exps env)
      (if (no-operands? exps) '()
          (cons (delay-it (first-operand exps) env)
                (list-of-delayed-args (rest-operands exps) env))))
    (define (list-of-arg-values exps env)
      (if (no-operands? exps) '()
          (cons (actual-value (first-operand exps) env)
                (list-of-arg-values (rest-operands exps) env))))
  (define (l-eval exp env)
    (cond ((self-evaluating? exp) exp)
          (cond? exp ((l-eval (cond->if exp) env))
               ((application? exp) (l-eval (apply (actual-value (operator exp) env) (operands exp) env))
                (else (error "Unknown expression type -- EVAL" exp))))

Lazy Evaluation – other changes needed

- Example: Need actual predicate value in conditional if...
  (define (l-eval-if exp env)
    (if (true? (actual-value (if-predicate exp) env))
        (l-eval (if-consequent exp) env)
        (l-eval (if-alternative exp) env)))

- Example: Don’t need actual value in assignment...
  (define (l-eval-assignment exp env)
    (set-variable-value!
     (assignment-variable exp)
     (l-eval (assignment-value exp) env)
     env) 'ok)

Examples

- (define identity (lambda (x) x)) identity: <proc>
- (define a (notice 3)) a: promise 3 Noticed!
- (define b (identity (notice 3)) b: promise (notice 3)
- (define c b) c: Noticed! Noticed!
- (define d (+ b c)) d: 6 Noticed! Noticed!
- (define plus (identity +)) plus: promise +
- (define d (+ b a)) => 6 Noticed!
- c => 3 Noticed!
Memo-izing evaluation

- In lazy evaluation, if we reuse an argument, have to reevaluate each time
- In usual (applicative) evaluation, argument is evaluated once, and just referenced
- Can we keep track of values once we've obtained them, and avoid cost of re-evaluation?

Memo-izing Promises

- Idea: once promise exp has been evaluated, remember it
- If value is needed again, just return it rather than recompute
- Concretely—mutate promise into an evaluated promise

Why mutate?—because other names or data structures may point to this promise!

Promises – Memoizing Implementation

```scheme
(define (evaluated-promise? obj) (tagged-list? obj 'evaluated-promise))
(define (promise-value evaluated-promise) (cadr evaluated-promise))
(define (force-it obj) (cond ((promise? obj) (let ((result (actual-value (promise-exp obj) (promise-env obj)))) (set-car! obj 'evaluated-promise) (set-car! (cdr obj) result) (set-cdr! (cdr obj) '()) (cons (evaluated-promise? obj) (promise-value obj)) (else obj)))
```

Examples - Memoized

- (define identity (lambda (x) x)) identity: <proc>
- (define a (notice 3)) a: promise 3 Noticed!
- (define b (identity (notice 3))) b: promise (notice 3)
- (define c b) c: promise +
- (define d (+ b c)) d: 6 Noticed! *CHANGE*
- (define plus (identity +)) plus: promise +
- (plus a b) => 6 *CHANGE*
- c => 3 *CHANGE*

Summary of lazy evaluation

- This completes changes to evaluator
  - Apply takes a set of expressions for arguments and an environment
    - Forces evaluation of arguments for primitive procedure application
    - Else defers evaluation and unwinds computation further
    - Need to pass in environment since don’t know when it will be needed
  - Need to force evaluation on branching operations (e.g. if)
  - Otherwise small number of changes make big change in behavior of language

Laziness and Language Design

- We have a dilemma with lazy evaluation
  - Advantage: only do work when value actually needed
  - Disadvantages
    - not sure when expression will be evaluated; can be very big issue in a language with side effects
    - may evaluate same expression more than once
  - Memoization doesn't fully resolve our dilemma
  - Advantage: Evaluate expression at most once
  - Disadvantage: What if we want evaluation on each use?
  - Alternative approach: Selective Laziness
Streams – the lazy way

Beyond Scheme – designing language variants:

• Streams – an alternative programming style
to infinity, and beyond…

Decoupling computation from description

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

(list-ref
  (filter (lambda (x) (prime? x))
    (enumerate-interval 1 1000000))
100)

Generate only what you actually need...

Stream Object

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

  stream-car
  
  stream-cdr

  a value
  a promise

• Example

  (define x (cons-stream 99 (/ 1 0)))
  (stream-car x) => 99
  (stream-cdr x) => error – divide by zero

  Stream-cdr forces the promise wrapped around (/ 1 0), resulting in an error

Implementing Streams

• Stream is a data structure with the following contract:
  – (cons-stream a b) – cons together a with promise to compute b
  – (stream-car s) – Returns car of s
  – (stream-cdr s) – Forces and returns value of cdr of s

• Implement in regular evaluator with a little syntactic sugar
  – (define (cons-stream a b) (cons a b))

  • Example

    (define (ints-starting-with i)
      (cons-stream i (ints-starting-with (+ i 1))))

  Recursive procedure with no base case!
  – Why does it work?

Ints-starting-with

Stream-ref

(define (stream-ref s i)
  (if (= i 0)
    (stream-car s)
    (stream-ref (stream-cdr s) (- i 1))))

• Like list-ref, but cdr’s down stream, forcing
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

(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

(stream-ref (stream-filter (lambda (x) (prime? x))
                          (ints-starting-with 2)) 4)
Finding all the primes

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)))

2 sieve (filter ints 2)
3 sieve (filter ints 3)

Interleave

Produce a stream that has all the elements of two input streams:

(define (interleave s1 s2)
  (cons-stream
    (stream-car s1)
    (interleave s2 (stream-cdr s1)))))

Rationals

(define (div-by-stream s n)
  (cons-stream (/ n (stream-car s))
               (div-by-stream (stream-cdr s) n))))

(define (make-rats n)
  (cons-stream n
               (interleave (div-by-stream (stream-cdr ints) n)
                           (make-rats (+ n 1)))))

(define rats (make-rats 1))

Power Series

- Approximate function by summation of infinite polynomial
- Great application for streams!
  <We’ll do this in recitation!>