6.001 SICP
Further Variations on a Scheme

Beyond Scheme – more language variants
Lazy evaluation
  • Complete conversion – normal order evaluator
  • Upward compatible extension – lazy, lazy-memo

Punchline: Small edits to the interpreter give us a new programming language

Evaluation model

Rules of evaluation:
• If expression is self-evaluating (e.g. a number), just return value
• If expression is a name, 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

Alternative models for computation

• Applicative Order:
  • evaluate all arguments, then apply operator

• Normal Order:
  • 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)

Applicative Order Example

{define (foo x)
  (write-line "inside foo")
  (+ x x)}

{foo (begin (write-line "eval arg") 222)}
  => (begin (write-line "eval arg") 222)
  => 222

=> (begin (write-line "inside foo")
  (+ 222 222))

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

\[
\text{(define (foo x)} \\
\text{ (write-line "inside foo")} \\
\text{ (+ x x))} \\
\text{(foo (begin (write-line "eval arg") 222))} \\
\Rightarrow \text{(begin (write-line "inside foo")} \\
\text{ (+ (begin (w-l "eval arg") 222) (begin (w-l "eval arg") 222))} \\
\text{As if we substituted the} \\
\text{unevaluated expression in the} \\
\text{body of the procedure} \\
\Rightarrow \text{444}
\]

Applicative versus Normal Order

\[
\text{(define (foo x)} \\
\text{ (write-line "inside foo")} \\
\text{ (+ x x))} \\
\text{(foo (begin (write-line "eval arg") 222))} \\
\text{Applicative} \\
\text{ Normal} \\
\text{eval arg} \\
\text{inside foo} \\
\text{eval arg} \\
\text{eval arg} \\
\text{• Note order of evaluation} \\
\text{• Note number of times arg is evaled}
\]

Normal order (lazy evaluation) versus applicative order

- How can we change our evaluator to use normal order?
  - Create “delayed objects” — expressions whose evaluation has been deferred
  - Change the evaluator to force evaluation only when needed
- Why is normal order useful?
  - What kinds of computations does it make easier?
    - Will primarily see next lecture with streams

The Core Evaluator

\[
\text{Eval} \\
\text{proc & args} \\
\text{Apply} \\
\text{• Core evaluator} \\
\text{• eval: dispatch on expression type} \\
\text{• apply: eval args then apply operator}
\]

Meval

\[
\text{(define (meval exp env)} \\
\text{ (cond ((self-evaluating? exp) exp)} \\
\text{ ((variable? exp) (lookup-variable-value exp env))} \\
\text{ ((quoted? exp) (text-of-quotation exp))} \\
\text{ ((assignment? exp) (eval-assignment exp env))} \\
\text{ ((definition? exp) (eval-definition exp env))} \\
\text{ ((if? exp) (eval-if exp env))} \\
\text{ ((lambda? exp) (make-procedure (lambda-parameters exp) (lambda-body exp) env))} \\
\text{ ((begin? exp) (eval-sequence (begin-actions exp) env))} \\
\text{ ((cond? exp) (eval (cond->if exp) env))} \\
\text{ ((application? exp) (mapply (meval (operator exp) env) (list-of-values (operands exp) env)))} \\
\text{ (else (error "Unknown expression type -- EVAL" exp)))}}
\]

Mapply

\[
\text{(define (mapply procedure arguments)} \\
\text{ (cond ((primitive-procedure? procedure) (apply-primitive-procedure procedure arguments))} \\
\text{ ((compound-procedure? procedure) (eval-sequence (procedure-body procedure) (extend-environment (procedure-parameters procedure) arguments (procedure-environment procedure)))} \\
\text{ (else (error "Unknown procedure type -- APPLY" procedure)))}}
\]
How can we implement lazy evaluation?

(define (l-apply procedure arguments env) ; changed
  (cond ((primitive-procedure? procedure)
             (apply-primitive-procedure
              procedure
              (list-of-arg-values arguments env)))
        (compound-procedure? procedure)
        (l-eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           [list-of-delayed-args arguments env]
           (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))
    (else (error "Unknown expression" exp))))

Meval versus L-Eval

(define (meval exp env)
  (cond
    ((self-evaluating? exp) exp)...
    ((cond? exp) (meval (cond->if exp) env))
    ((application? exp)
      (mapply (meval (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)
      ((application? exp
        (l-apply (actual-value (operator exp) env)
                 (operands exp)
                 env))
       (else (error "Unknown expression" exp))))

Actual vs. Delayed Values

(define (actual-value exp env)
  (force-it (l-eval exp env)))

(define (list-of-arg-values exps env)
  (if (no-operands? exps)
      ()
      (cons (delay-it (first-operand exps) env)
            (list-of-delayed-args (rest-operands exps) 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))))

Representing Thunks

• Abstractly – a thunk is a "promise" to return a value when later needed ("forced")

• Concretely – our representation:

Thunks – delay-it and force-it

(define (delay-it exp env) (list 'thunk exp env))
(define (thunk? obj) (tagged-list? obj 'thunk))
(define (thunk-thunk thunk) (cadr thunk))
(define (thunk-env thunk) (caddr thunk))

(define (force-it obj)
  (cond
    ((thunk? obj)
     (actual-value (thunk-thunk obj)
                  (thunk-env obj)))
    (else obj)))

(define (actual-value exp env)
  (force-it (l-eval exp env)))
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 reevaluation?

Normal Order Example

\begin{align*}
&\text{(define (foo x)} \nonumber \\
&\text{\quad (write-line “inside foo”)} \nonumber \\
&\text{\quad (+ x x)} \nonumber \\
&\text{\quad (foo (begin (write-line “eval arg”) 222))} \nonumber \\
\end{align*}

\[
\Rightarrow (\begin{align*}
&\text{(begin (write-line “inside foo”) \nonumber} \\
&\text{\quad (+ (begin (w-l “eval arg”) 222) \nonumber} \\
&\text{\quad (begin (w-l “eval arg”) 222)))} \nonumber \\
\end{align*}
\]

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

\[
\Rightarrow 444
\]

Memo-izing Thunks

- Idea: once thunk \text{exp} has been evaluated, remember it
- If value is needed again, just return it rather than recompute

- Concretely — \text{mutate} a thunk into an evaluated-thunk

Thunks – Memoizing Implementation

\begin{align*}
&\text{(define (evaluated-thunk? obj)} \nonumber \\
&\text{\quad (tagged-list? obj ‘evaluated-thunk)} \nonumber \\
&\text{\quad (define (thunk-value evaluated-thunk)} \nonumber \\
&\text{\quad (cadr evaluated-thunk))} \nonumber \\
&\text{\quad (define (force-it obj)} \nonumber \\
&\text{\quad \quad (cond ((thunk? obj)} \nonumber \\
&\text{\quad \quad \quad (let ((result (actual-value (thunk-exp obj) \nonumber \\
&\text{\quad \quad \quad \quad (thunk-env obj))))} \nonumber \\
&\text{\quad \quad \quad \quad (set-car! obj ‘evaluated-thunk)} \nonumber \\
&\text{\quad \quad \quad \quad (set-car! (cdr obj) result))} \nonumber \\
&\text{\quad \quad \quad \quad (set-cdr! (cdr obj) ‘()} \nonumber \\
&\text{\quad \quad \quad \quad result))))} \nonumber \\
&\text{\quad \quad \quad \quad [(evaluated-thunk? obj) \nonumber \\
&\text{\quad \quad \quad \quad (thunk-value obj)] \nonumber \\
&\text{\quad \quad \quad \quad (else obj))]} \nonumber \\
\end{align*}

Lazy Evaluation – other changes needed

- Example – need actual predicate value in conditional if...
  \begin{align*}
  &\text{(define (l-eval-if exp env)} \nonumber \\
  &\text{\quad (if (true? (actual-value (if-predicate exp) env))} \nonumber \\
  &\text{\quad \quad (l-eval (if-consequent exp) env))} \nonumber \\
  &\text{\quad \quad \quad (l-eval (if-alternative exp) env)} \nonumber \\
  \end{align*}

- Example – don’t need actual value in assignment...
  \begin{align*}
  &\text{(define (l-eval-assignment exp env)} \nonumber \\
  &\text{\quad (set-variable-value! \nonumber \\
  &\text{\quad \quad (assignment-variable exp))} \nonumber \\
  &\text{\quad \quad \quad (l-eval (assignment-value exp) env)} \nonumber \\
  &\text{\quad \quad \quad \quad env)} \nonumber \\
  &\text{\quad \quad \quad \quad ‘ok)} \nonumber \\
  \end{align*}

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: \text{give programmer control!!}
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 usual 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.

Syntax Extensions – Parameter Declarations

- Look at the variable declaration to do the right thing...

  \( \text{define} \ (\text{delay-it} \ \text{decl} \ \text{exp} \ \text{env}) \)

  \( \text{define} \ (\text{force-it} \ \text{obj}) \)

  \( \text{define} \ (\text{l-apply} \ \text{procedure} \ \text{arguments} \ \text{env}) \)

Controllably Memo-izing Thunks

- \text{thunk} – never gets memoized
- \text{thunk-memo} – first eval is remembered
- \text{evaluated-thunk} – memoized-thunk that has already been evaluated

\( \text{thunk-memo} \ \text{exp} \ \text{env} \)

\( \text{evaluated-thunk} \ \text{result} \)

Change to force-it

\( \text{define} \ (\text{force-it} \ \text{obj}) \)

\( \text{define} \ (\text{l-apply} \ \text{procedure} \ \text{arguments} \ \text{env}) \)

A new version of delay-it

- Look at the variable declaration to do the right thing...

  \( \text{define} \ (\text{delay-it} \ \text{decl} \ \text{exp} \ \text{env}) \)

Changes to l-apply

- Key: in l-apply, only delay "lazy" or "lazy-memo" params
- make thunks for "lazy" parameters
- make memoized-thunks for "lazy-memo" parameters
Deciding when to evaluate an argument...

- Process each variable declaration together with application subexpressions – delay as necessary:

```scheme
(define (list-of-delayed-args var-decls exps env)
  (if (no-operands? exps)
      '()
      (cons (delay-it (first-variable var-decls)
                    (first-operand exps)
                    env)
            (list-of-delayed-args
             (rest-variables var-decls)
             (rest-operands exps)
             env))))
```

Summary

- Lazy evaluation – control over evaluation models
  - Convert entire language to normal order
  - Upward compatible extension
    - lazy & lazy-memo parameter declarations
  - We have created a new language (with new syntax), using only relatively small changes to the interpreter.

Summary

- We can control when arguments are evaluated
  - By making a lazy evaluator
  - By changing the evaluator to allow specification of arguments
  - Changing the evaluator requires a small amount of work but dramatically shifts the behavior of the system
  - Applicative order versus Normal order
  - Using a lazy evaluator lets us separate the apparent order of computation inherent in a problem from the actual order of evaluation inside the machine