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

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 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

Normal Order (Lazy) Evaluation
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))
```
We first evaluated argument, then substituted value into the body of the procedure

Normal Order Example
```
(define (foo x)
  (write-line "inside foo")
  (+ x x))

(foo (begin (write-line "eval arg") 222))
```
As if we substituted the unevaluated expression in the body of the procedure
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?

How can we implement lazy evaluation?

```scheme
(define (l-apply procedure arguments env)
  (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...

```scheme
(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

```scheme
(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))))
```

Actual vs. Delayed Values

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

```scheme
(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:

```
  thunk exp 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 (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))))
```
**Thunks – delay-it and force-it**

\[
\begin{align*}
\text{define} & \quad \text{(delay-it \, \text{exp \, env})} \quad \text{(list \ 'thunk \ exp \ env)} \\
\text{define} & \quad \text{(thunk? \ obj)} \quad \text{(tagged-list? \ obj \ 'thunk)} \\
\text{define} & \quad \text{(thunk-exp \ thunk)} \quad \text{(cadr \ thunk)} \\
\text{define} & \quad \text{(thunk-env \ thunk)} \quad \text{(caddr \ thunk)} \\
\text{define} & \quad \text{(force-it \ obj)} \\
\text{cond} & \quad \text{((thunk? \ obj)} \\
& \quad \text{\quad (\text{actual-value \ (thunk-exp \ obj)} \ (thunk-env \ obj)))} \\
& \quad \text{\quad (else \ obj))}) \\
\text{define} & \quad \text{(actual-value \ exp \ env)} \\
\text{define} & \quad \text{(force-it \ (l-eval \ exp \ env))} \\
\end{align*}
\]

**Memo-izing evaluation**

- In lazy evaluation, if we reuse an argument, have to reevaluate each time
- In normal evaluation, argument is evaluated once, and just referenced
- Can we keep track of values once we’ve obtained them, and avoid cost of reevaluation?

**Memo-izing Thunks**

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

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

**Thunks – Memoizing Implementation**

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

**Lazy Evaluation – other changes needed**

- **Example** – need actual predicate value in conditional if...
  \(\text{(define \ (l-eval-if \ exp \ env)}\)
  \(\quad \text{(if \ (true? \ (\text{actual-value \ (if-predicate \ exp) \ env)}))} \)
  \(\quad \text{(l-eval \ (if-consequent \ exp) \ env)}\)
  \(\quad \text{(l-eval \ (if-alternative \ exp) \ env))}\)
- **Example** – don’t need actual value in assignment...
  \(\text{(define \ (l-eval-assignment \ exp \ env)}\)
  \(\quad \text{(set-variable-value!} \)
  \(\quad \text{(assignment-variable \ exp)}\)
  \(\quad \text{(l-eval \ (assignment-value \ exp) \ env)}\)
  \(\quad \text{'ok})\)

**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: give programmer control!
**Variable Declarations: lazy and lazy-memo**

- Handle lazy and lazy-memo extensions in an upward-compatible fashion;

```lisp
(lambda (a (b lazy) c (d lazy-memo)) ...)
```

- "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.

**Syntax Extensions – Parameter Declarations**

```lisp
(define (first-variable var-decls) (car var-decls))
(define (rest-variables var-decls) (cdr var-decls))
(define declaration? pair?)
(define (parameter-name var-decl)
  (if (pair? var-decl) (car var-decl) var-decl))
(define (lazy? var-decl)
  (and (pair? var-decl) (eq? 'lazy (cadr var-decl))))
(define (memo? var-decl)
  (and (pair? var-decl) (eq? 'lazy-memo (cadr var-decl))))
```

**Controllably Memo-izing Thunks**

- **thunk** – never gets memoized
- **thunk-memo** – first eval is remembered
- **evaluated-thunk** – memoized-thunk that has already been evaluated

When forced

```plaintext
thunk-memo

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>thunk</td>
<td>exp</td>
<td>env</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>evaluated-thunk</td>
<td>result</td>
<td></td>
</tr>
</tbody>
</table>
```

**A new version of delay-it**

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

```lisp
(define (delay-it decl exp env)
  (cond ((not (declaration? decl))
          (l-eval exp env))
        ((lazy? decl)
          (list 'thunk exp env))
        ((memo? decl)
          (list 'thunk-memo exp env))
        (else (error "unknown declaration:" decl))))
```

**Change to force-it**

```lisp
(define (force-it obj)
  (cond ((thunk? obj) ; eval, but don’t remember it
          (actual-value (thunk-exp obj))
          (thunk-env obj)))
    ((memoized-thunk? obj) ; eval and remember
      (let ((result
              (actual-value (thunk-exp obj))
              (thunk-env obj))))
        (set-car! obj 'evaluated-thunk)
        (set-car! (cdr obj) result)
        (set-cdr! (cdr obj) '())
        result))
    ((evaluated-thunk? obj) (thunk-value obj))
    (else obj)))
```

**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

```lisp
(define (l-apply procedure arguments env)
  (cond ((primitive-procedure? procedure)
          ...)
        ((compound-procedure? procedure)
          (l-eval-sequence
           (procedure-body procedure)
           (let ((params (procedure-parameters procedure))
                  (extend-environment
                   (map parameter-name params)
                   (list-of-delayed-args params arguments env)
                   (procedure-environment procedure))))
            (else (error "Unknown proc" procedure)))))
```
Deciding when to evaluate an argument...

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

\[
\text{(define (list-of-delayed-args var-decls exps env)}
\]

\[
\text{ (if (no-operands? exps)}
\]

\[
\text{ ('())}
\]

\[
\text{(cons (delay-it (first-variable var-decls) (first-operand exps) env)}
\]

\[
\text{(list-of-delayed-args (rest-variables var-decls) (rest-operands exps) env)))}
\]

Summary

- Lazy evaluation – control over evaluation models
- Convert entire language to normal order
- Upward compatible extension
  - lazy & lazy-memo parameter declarations

Streams – a different way of structuring computation

- Imagine simulating the motion of an object
- Use state variables, clock, equations of motion to update
- State of the simulation captured in instantaneous values of state variables

- OR – have each object output a continuous stream of information
- State of the simulation captured in the history (or stream) of values

Think about computation as if had entire sequence of values available at any point

How do we use this new lazy evaluation?

- Our users could implement a stream abstraction:

\[
\text{(define (cons-stream x y (lazy-memo))}
\]

\[
\text{(lambda (msg)}
\]

\[
\text{ (cond ((eq? msg 'stream-car) x)}
\]

\[
\text{ ((eq? msg 'stream-cdr) y)}
\]

\[
\text{ (else (error "unknown stream msg" msg))))})}
\]

\[
\text{(define (stream-car s) (s 'stream-car))}
\]

\[
\text{(define (stream-cdr s) (s 'stream-cdr))}
\]

OR

\[
\text{(define (cons-stream x (y (lazy-memo))}
\]

\[
\text{(cons x y))}
\]

\[
\text{(define stream-car car)}
\]

\[
\text{(define stream-cdr cdr)}
\]

Stream Object

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

\[
\text{a value}
\]

\[
\text{a thunk-memo}
\]

- Example

\[
\text{(define x (cons-stream 99 (/ 1 0)))}
\]

\[
\text{(stream-car x)} \Rightarrow 99
\]

\[
\text{(stream-cdr x)} \Rightarrow \text{error – divide by zero}
\]
Decoupling computation from description

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

```scheme
(list-ref (filter (lambda (x) (prime? x)) (enumerate-interval 1 100000000)) 100)
```

```scheme
(define (stream-interval a b)
  (if (> a b) the-empty-stream (cons-stream a (stream-interval (+ a 1) b))))
```

```scheme
(stream-ref (stream-filter (lambda (x) (prime? x)) (stream-interval 1 100000000)) 100)
```

Some details on stream procedures

```scheme
(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

```scheme
(stream-filter prime? (str-in 1 100000000))
```

```scheme
(stream-filter prime? (stream-in 2 10000000)) 1
(stream-filter prime? (stream-filter prime? (stream-in 2 10000000)))
```

Result: Infinite Data Structures!

- Some very interesting behavior

```scheme
(define ones (cons-stream 1 ones))
```

- Compare:

```scheme
(define ones (cons 1 ones)) => error, ones undefined
```

Finite list procs turn into infinite stream procs

```scheme
(define (add-streams s1 s2)
  (cond ((null? s1) '())
        ((null? s2) '())
        (else (add-streams (+ (stream-car s1) (stream-car s2))
                           (add-streams (stream-cdr s1) (stream-cdr s2))))))
```

```scheme
(define ints (cons-stream 1 (add-streams ones ints)))
```

Finding all the primes

```scheme
; Infinite Streams

; 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 97 101 103 107 109 113 127 131 137 139 149 151 157
```

```scheme
; Stream of primes

; 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 97 101 103 107 109 113 127 131 137 139 149 151 157
```
Building 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)))

(define ones (stream-filter (lambda (x) (= x 2)) (stream-cdr ints)))

(stream-car (stream-filter (lambda (x) (= x 2)) (stream-cdr ints)))

Integration as an example

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

(integral ones 0 2)
=> 0 2 4 6 8

Ones: 1 1 1 1 1
Scale 2 2 2 2 2

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