Dynamic Binding
Normal-Order Evaluation

CS21b: Structure and Interpretation of Computer Programs
Brandeis University
Spring Term, 2015
In Scheme, the environment in which a procedure is \textit{evaluated} is statically determined from the environment in which the procedure is \textit{defined}. Why not instead let the environment in which evaluation occurs be determined from the \textit{dynamic} environment in which the application occurs?

\begin{verbatim}
(define (accumulate op nullvalue fun lst)
    (acc lst))

(define (acc lst)
    (if (null? lst)
        nullvalue
        (op (fun (car lst))
            (acc (cdr lst)))))
\end{verbatim}

This instead of the usual

\begin{verbatim}
(define (accumulate op nullvalue fun lst)
    (if (null? lst)
        nullvalue
        (op (fun (car lst))
            (accumulate op nullvalue fun (cdr lst)))))
\end{verbatim}

How does the Scheme implementation need to change to accommodate this \textit{dynamic binding} discipline? How and where are frames attached to the environment?
Dynamic binding: eval and apply

(define (eval exp env)
  (cond ...
    ((lambda? exp)
      (make-procedure (lambda-parameters exp)
         (lambda-body exp)))
    ...
    ((application? exp)
      (apply (eval (operator exp) env)
         (list-of-values (operands exp) env)
         env))
    (else
      (error "Unknown expression type — EVAL" exp))))

(define (apply procedure arguments env)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
       ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment
             (procedure-parameters procedure)
             arguments
             env)); <=== used to be: (procedure-environment procedure))
       (else
        (error "Unknown procedure type — APPLY" procedure))))
Normal-order evaluation

In normal order evaluation, we delay the computation of everything we can. Delaying means the construction of a thunk: a data object having (unevaluated) code, and an environment in which it is to be (later) evaluated. (When have we done this before?!) When a user-defined (compound) procedure is applied, we bind each parameter to a thunk instead of binding it to a value.

Procrastination stops at the following points:

Read-eval-print loop: You shouldn't print a thunk---instead, you should evaluate it.

Applying procedures: You can't apply a thunk---you need to force its evaluation so you get a procedure to apply.

Applying primitive procedures: You can't add two thunks, or take the car of one. What about cons? (A design decision.)

Conditionals: When evaluating an if expression, a thunk isn't good enough---you need to evaluate the predicate to see if it is true or false.
Driver loop

(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output
      (actual-value
        input
        the-global-environment)))
      (announce-output output output-prompt)
      (user-print output)))
  (driver-loop))

If an expression evaluates to a thunk, procedure actual-value forces evaluation of the thunk until a value is produced.
Procedure application

(define (eval exp env)
  (cond ...
    ((application? exp)
     (apply (actual-value (operator exp) env)
            (operands exp) env))
    (else
     (error "Unknown expression type — EVAL" exp))))

Procedure apply takes unevaluated arguments and their environment as parameters.

(define (apply procedure arguments env)
  (cond ((primitive-procedure? procedure)
     (apply-primitive-procedure
      procedure
     (list-of-arg-values arguments env))) ; changed
      ((compound-procedure? procedure)
     (eval-sequence
      (procedure-body procedure)
     (extend-environment
      (procedure-parameters procedure)
     (list-of-delayed-args arguments env) ; changed
     (procedure-environment procedure))))
    (else
     (error
     "Unknown procedure type — APPLY" procedure)))))

In compound procedure application, the parameters are thunked. In primitive procedure application, all thunked parameters are forced.
List of values:

(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))))

List of thunks:

(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)))))
In evaluating a conditional, force evaluation of the predicate:

(define (eval-if exp env)
  (if (true? (actual-value
    (if-predicate exp)
    env))
    (eval (if-consequent exp) env)
    (eval (if-alternative exp) env)))
Thunks

\[
\begin{align*}
\text{(define (actual-value exp env)} & \quad \text{(define (delay-it exp env)} \\
\text{  (force-it (eval exp env)))} & \quad \text{  (list 'thunk exp env))}
\end{align*}
\]

\[
\begin{align*}
\text{(define (force-it obj)} & \quad \text{(define (thunk-exp thunk))} \\
\text{  (if (thunk? obj)} & \quad \text{(define (thunk-env thunk))} \\
\text{    (actual-value)} & \quad \text{(cadr thunk))} \\
\text{    (thunk-exp obj)} & \quad \text{(caddr thunk))} \\
\text{    (thunk-env obj))} & \quad \text{)}
\end{align*}
\]

\[
\begin{align*}
\text{(define (thunk? obj)} & \quad \text{(define (A x) (B x))} \\
\text{  (tagged-list? obj 'thunk))} & \quad \text{(define (B y) (C y))} \\
\text{)} & \quad \text{(define (C z) (D z))} \\
\text{)} & \quad \text{(define (D w) (1+ w))}
\end{align*}
\]

\[
\begin{align*}
\text{(A 10)} & \quad \text{(A 10)}
\end{align*}
\]

...to see why, draw the environment diagram...

Why thunks need recursive forcing

...to see why, draw the environment diagram...
Why thunks need recursive forcing

(define (A x) (B x))
(define (B y) (C y))
(define (C z) (D z))
(define (D w) (1+ w))

(A 10)

...to see why, draw the environment diagram...

Another, more pernicious example:

(define (A n) (if (= n 0) #t (A (- n 1)))))