CS61A Lecture 26

2011-08-03

Colleen Lewis

In the *REGULAR* version Where do arguments get evaluated?

A. In `mc-eval`

B. In `mc-apply`

C. In both

D. In neither

E. ???

Changes to `mc-eval` for the lazy evaluator

```
(define (mc-eval exp env)
  (cond ...
     ((application? exp)
        (mc-apply (mc-eval (operator exp) env)
           (list-of-values (operands exp) env)))
```

```
(define (mc-eval exp env)
  (cond ...
    ((application? exp)
      (mc-apply (actual-value (operator exp) env)
         (list-of-values (operands exp) env)))
```

The lazy mc-apply

`mc-apply`

DO evaluate the arguments → Primitive procedure

Don't evaluate the arguments → Compound (User defined) Procedure

---

THE RULES

The lazy mc-eval might return a Thunk ADT, we should Force these:

- Before you print something returned by `mc-eval`
- Before you pass arguments to a primitive procedure
  - `if` is LIKE a primitive procedure the predicate shouldn't be a Thunk ADT.

We should CREATE Thunk ADTs (delay stuff)

- Before you pass arguments to a compound procedure

The RANGE of `mc-eval` includes Thunk ADTs

```
STk> (load "lazy.scm")
okay
STk> (define g-env (setup-environment))
g-env
STk> (mc-eval '((lambda (x) x) (+ 2 3)) g-env)
(thunk (+ 2 3) env )
```

`mc-eval` may return a Thunk ADT

User-defined procedure: **Don't** evaluate the arguments

Tracing a call that returns a Thunk ADT from `mc-eval`

```
(define (mc-eval exp env)
  (cond ...
    ((application? exp)
     (mc-apply
       (actual-value   (operator exp) env)
       (list-of-values (operands exp) env)))
```

(`list-of-values` is struck through.)

```
STk> (mc-eval '((lambda (x) x) (+ 2 3)) g-env)
```

Tracing a call that returns a Thunk ADT from `mc-eval`

```
STk> (mc-eval '((lambda (x) x) (+ 2 3)) g-env)

      '(procedure (x) (x) exp )

(define (mc-apply procedure arguments env)
  (cond …
    ((compound-procedure? procedure)
       (eval-sequence
        (procedure-body procedure)
        (extend-environment
         (procedure-parameters procedure)
         (list-of-delayed-args arguments env)
         (procedure-environment procedure))))
```

Does this call force-it? **A. Y B. N C.?**

variables

values

old env

Replace a call to `mc-eval` to avoid printing a Thunk ADT

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

This was: **`mc-eval`**

`mc-eval` might return a delayed argument from a compound procedure

`if`'s need actual values!

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

`mc-eval` sometimes returns Thunk ADTs

```
(define (mc-eval exp env)
  (cond
    ((self-evaluating? exp)...
    ((variable? exp)...
    ((quoted? exp) ...
    ((assignment? exp) ...
    ((definition? exp) ...
    ((if? exp) ...
    ((lambda? exp) ...
    ((begin? exp) ...
    ((cond? exp) ...
    ((application? exp) ...
    (else (error "what?"))))
```

Should we add a Thunk? check to `mc-eval`?

A. No – not necessary

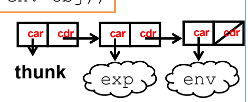
B. No handled by another case

B. Yes

D. ??

`actual-value`

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

(define (force-it obj)
  (if (thunk? obj)
      (actual-value (thunk-exp obj)
                    (thunk-env obj))
      obj))

(define (force-it-FAKE obj)
  (if (thunk? obj)
      (mc-eval      (thunk-exp obj)
                    (thunk-env obj))
      obj))
```

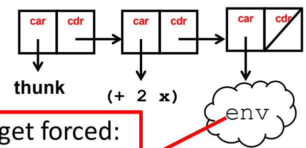


Example of why we call `actual-value`

```
STk> (load "lazy.scm")
okay
STk> (define g-env (setup-environment))
g-env
STk> (mc-eval
      '((lambda (x) x)
          ((lambda (y) y)
                 (+ 2 3)))
      g-env)
a.(thunk ((lambda (y) y) (+ 2 3)) env )
b.(thunk ((lambda (x) x) (+ 2 3)) env )
c.(thunk ((λ (x) x)((λ (y) y)(+ 2 3)))env )
d. 5      e. ??
```

If we're going to `delay-it` we need to keep track of the environment!

```
(define (delay-it exp env)
  (list 'thunk exp env))
```



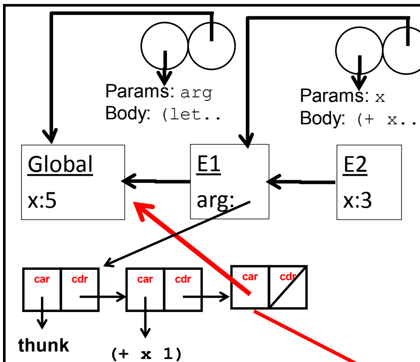
When I get forced: evaluate the `exp` in this environment

Why you need to evaluate Thunk ADTs in their original environment

```
STk> (define (crazy arg)
         (let ((x 3))
             (+ x arg arg)))
STk> (define x 4)
STk> (crazy (+ x 1))
```

`(crazy (+ x 1))`	`(crazy (+ x 1))`
`(crazy 5)`	`(* x arg arg)`
`(* x arg arg)`	`(* 3 (+ x 1)(+ x 1))`
`(* 3 5 5)`	`(* 3 (+ 3 1)(+ 3 1))`

WRONG way: ☹ without old environment



Does the third element point to:
A. Global
B. E1
C. E2
D. None
E. ??

Current frame: ~~Global~~ ~~E1~~ E2

Summary & Additional Notes

- Thunk ADTs could also be memoized
- We delayed arguments to compound procedures
  - Compound procedures are defined by the user
- We didn't delay arguments to primitive procedures
- We made sure we had the actual value to print it
- Ifs needed REAL values for predicates

Run `(query)` and tell it some facts

```
STk> (load "query.scm")
okay
STk> (query)
;;; Query input:
(assert! (colleen likes cookies))

Assertion added to data base.

;;; Query input:
```

Like `(mce)`: it starts an infinite loop

Tell the system facts

Some facts I told the query system

```
(assert! (colleen likes cookies))
(assert! (hamilton likes cookies))
(assert! (stephanie likes oreos))
(assert! (kevin likes pizza))
(assert! (eric likes pizza))
(assert! (phill likes everything))
```

We can ask the query system questions

```
;;; Query input:
(?who likes pizza)
```

What facts match this pattern?

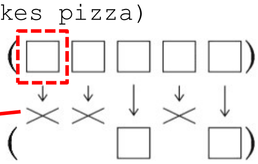
?who is a variable, it can be anything

```
;;; Query results:
(kevin likes pizza)
(eric likes pizza)
```

We can get multiple answers!!!!!!

The query system “filters” out facts that don’t match

```
;;; Query input:
(?who likes pizza)
```

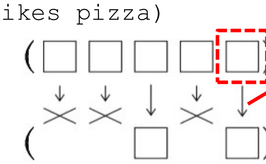


Don’t keep this one!

(colleen	likes	cookies)
(?who	likes	pizza)

The query system “filters” out facts that don’t match

```
;;; Query input:
(?who likes pizza)
```



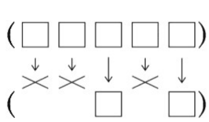
Keep this one!

(kevin	likes	pizza)
(?who	likes	pizza)

We can ask the query system questions

```
;;; Query input:
(?who likes pizza)
```

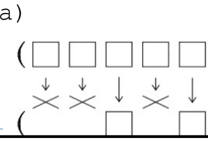
The variable name doesn’t matter



```
;;; Query results:
(kevin likes pizza)
(eric likes pizza)
```

Filtering allows us to get multiple things back!

~~(colleen likes cookies)~~
~~(hamilton likes cookies)~~
~~(stephanie likes oreos)~~
(eric likes pizza)
~~(phill likes everything)~~
(kevin likes pizza)



Write a query that matches ALL assertions that we've added!

```
(colleen   likes cookies)
(hamilton  likes cookies)
(stephanie likes oreos)
(eric      likes pizza)
(phill     likes everything)
(kevin     likes pizza)
```

A. Not possible
B. Need 1 variable
C. Need 2 variables
D. Need 3 variables
E. Stuck

What can a query return

```
(colleen   likes cookies)
(hamilton  likes cookies)
(stephanie likes oreos)
(eric      likes pizza)
(phill     likes everything)
(kevin     likes pizza)

;;; Query input:
(?who likes elephants)
```

How many results?
A. 0 B. 1 C. 2 D. 3-6 E. ??

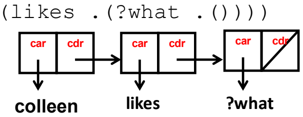
Do these match?

```
(assert! (colleen likes ice cream))
(assert! (colleen likes cookies))

;;; Query input:
(colleen likes ?what)

;;; Query results:
```

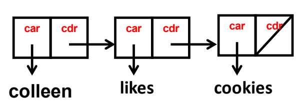
A. Only cookies
B. Only ice cream
C. Both
D. Neither
E. ??

We need to think about pairs

```
;;; Query input:
(colleen likes ?what)
(colleen .(likes .(?what .())))
```



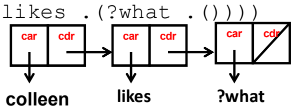

```
(colleen likes cookies)
(colleen .(likes .(cookies.())))
```



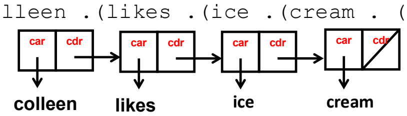
We need to think about pairs

```
;;; Query input:
(colleen likes ?what)
(colleen .(likes .(?what .())))
```

Do these match?
A. Yes B. No C. ??



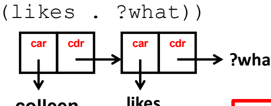

```
(colleen likes ice cream)
(colleen .(likes .(ice .(cream . ()))))
```



We need to think about pairs

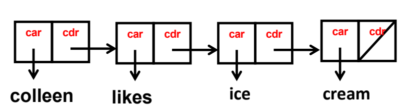
```
;;; Query input:
(colleen likes . ?what)
(colleen .(likes . ?what))
```

Equivalent




```
(colleen likes ice cream)
```

matches!



Facts with variables: `rules`

We can add things WITH variables to the "facts"

```
(assert! (rule (car ?a (?a . ?b))))

;;; Query input:
(car ?x (5 6 7))
  (car ?x (5 .(6 .(7 .())))))
;;; Query results:
(car 5 (5 6 7))
```

Facts with variables: `rules`

We can add things WITH variables to the "facts"

```
(assert! (rule (car ?a (?a . ?b))))

(car   ?a  (?a . ?b))
(car   ?x  (5  . (6.(7.()))))
?a = ?x
?a = 5
?b = (6.(7.()))
(car 5 (5 6 7))
```

We figured stuff out about the world

We can return things with variables

```
;;; Query input:
(car 1 ?y)

;;; Query results:
(car 1  (1 . ?b))
(car 1 (1 . ?b-27))
```

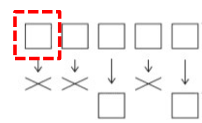
Matches with this but we still don't know all the values

This is what is really printed: Query system adds #'s to avoid naming conflicts

Rules can have Bodies

Body Treat as a new query

```
(assert! (rule (awesome ?x)
           (?x likes cookies)))
```

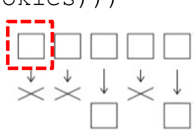


```
;;; Query input:
(awesome ?y)
?x = ?y ;figured out
(?x         likes cookies)
(stephanie  likes oreos)
?x = stephanie ; figured out
```

Rules can have Bodies

Body Treat as a new query

```
(assert! (rule (awesome ?x)
           (?x likes cookies)))
```



```
;;; Query input:
(awesome ?y)
?x = ?y ;figured out
(?x        likes cookies)
(hamilton  likes cookies)
?x = hamilton; figured out
```

Write 2nd

```
;;; Query input:
(2nd ?x (4 5 6))
;;; Query results:
(2nd 5 (4 5 6))
;;; Query input:
(2nd 3 ?x)
;;; Query results:
(2nd 3 (?a-29 3 . ?c-29))
```