End-of-semester evaluations

1. Please fill out the EOS evaluation.

1

1

Lecture 23
Mutable Pairs – MP

T. METIN SEZGIN

2

Nugget

Now that we have a memory structure, we can add more sophisticated structures to our language

3

Adding lists/pairs to the language

```
let x = 4
in cons(x,
        cons(cons(-(x,1),
                  emptylist),
             emptylist))
```

4

In addition we want mutation

- New grammar

$$\begin{array}{ll} \textbf{newpair} & : \textit{Expval} \times \textit{Expval} \rightarrow \textit{MutPair} \\ \textbf{left} & : \textit{MutPair} \rightarrow \textit{Expval} \\ \textbf{right} & : \textit{MutPair} \rightarrow \textit{Expval} \\ \textbf{setleft} & : \textit{MutPair} \times \textit{Expval} \rightarrow \textit{Unspecified} \\ \textbf{setright} & : \textit{MutPair} \times \textit{Expval} \rightarrow \textit{Unspecified} \end{array}$$

- New set of
  - Denotables
  - Expressibles

$$\begin{array}{lcl} \textit{ExpVal} & = & \textit{Int} + \textit{Bool} + \textit{Proc} + \textit{MutPair} \\ \textit{DenVal} & = & \textit{Ref(ExpVal)} \\ \textit{MutPair} & = & \textit{Ref(ExpVal)} \times \textit{Ref(ExpVal)} \end{array}$$

```
(define-datatype expval expval?
  (num-val
    (value number?))
  (bool-val
    (boolean boolean?))
  (proc-val
    (proc proc?))
  (mutpair-val
    (p mutpair?))
  )
```

```
(define-datatype mutpair mutpair?
  (a-pair
    (left-loc reference?)
    (right-loc reference?)))
```

5

New scheme functions for pair management

make-pair : $\textit{ExpVal} \times \textit{ExpVal} \rightarrow \textit{MutPair}$

```
(define make-pair
  (lambda (val1 val2)
    (a-pair
      (newref val1)
      (newref val2))))
```

left : $\textit{MutPair} \rightarrow \textit{ExpVal}$

```
(define left
  (lambda (p)
    (cases mutpair p
      (a-pair (left-loc right-loc)
        (deref left-loc)))))
```

right : $\textit{MutPair} \rightarrow \textit{ExpVal}$

```
(define right
  (lambda (p)
    (cases mutpair p
      (a-pair (left-loc right-loc)
        (deref right-loc)))))
```

setleft : $\textit{MutPair} \times \textit{ExpVal} \rightarrow \textit{Unspecified}$

```
(define setleft
  (lambda (p val)
    (cases mutpair p
      (a-pair (left-loc right-loc)
        (setref! left-loc val)))))
```

setright : $\textit{MutPair} \times \textit{ExpVal} \rightarrow \textit{Unspecified}$

```
(define setright
  (lambda (p val)
    (cases mutpair p
      (a-pair (left-loc right-loc)
        (setref! right-loc val)))))
```

6

The Interpreter

```
(newpair-exp (exp1 exp2)
  (let ((val1 (value-of exp1 env))
        (val2 (value-of exp2 env)))
    (mutpair-val (make-pair val1 val2))))

(left-exp (exp1)
  (let ((val1 (value-of exp1 env)))
    (let ((p1 (expval->mutpair val1)))
      (left p1))))

(right-exp (exp1)
  (let ((val1 (value-of exp1 env)))
    (let ((p1 (expval->mutpair val1)))
      (right p1))))
```

```
(setleft-exp (exp1 exp2)
  (let ((val1 (value-of exp1 env))
        (val2 (value-of exp2 env)))
    (let ((p (expval->mutpair val1)))
      (begin
        (setleft p val2)
        (num-val 82)))))

(setright-exp (exp1 exp2)
  (let ((val1 (value-of exp1 env))
        (val2 (value-of exp2 env)))
    (let ((p (expval->mutpair val1)))
      (begin
        (setright p val2)
        (num-val 83)))))
```

7

Nugget

We can get creative and devise a more efficient implementation

8

A different representation for mutable pairs

make-pair : *ExpVal* × *ExpVal* → *MutPair*
```
(define make-pair
  (lambda (val1 val2)
    (a-pair
      (newref val1)
      (newref val2))))
```

left : *MutPair* → *ExpVal*
```
(define left
  (lambda (p)
    (cases mutpair p
      (a-pair (left-loc right-loc)
        (deref left-loc)))))
```

- Note something about the addresses of the two values

9

A different representation for mutable pairs

mutpair? : *SchemeVal* → *Bool*
```
(define mutpair?
  (lambda (v)
    (reference? v)))
```

make-pair : *ExpVal* × *ExpVal* → *MutPair*
```
(define make-pair
  (lambda (val1 val2)
    (let ((ref1 (newref val1)))
      (let ((ref2 (newref val2)))
        ref1))))
```

left : *MutPair* → *ExpVal*
```
(define left
  (lambda (p)
    (deref p)))
```

right : *MutPair* → *ExpVal*
```
(define right
  (lambda (p)
    (deref (+ 1 p))))
```

setleft : *MutPair* × *ExpVal* → *Unspecified*
```
(define setleft
  (lambda (p val)
    (setref! p val)))
```

setright : *MutPair* × *ExpVal* → *Unspecified*
```
(define setright
  (lambda (p val)
    (setref! (+ 1 p) val)))
```

10

Learning outcomes of this lecture

- A student attending this lecture should be able to:
  1. Understand how pairs can be implemented, and do so
  2. Explain why the second implementation is more efficient
  3. Implement more sophisticated data structures (e.g., stack, arrays).

11

Lecture 24
Parameter Passing

T. METIN SEZGIN

12

Learning outcomes of this lecture

- A student attending this lecture should be able to:
  1. Understand that there are variations to parameter passing
  2. Understand CBV/CBR and how they work
  3. Understand the uses of CBR
  4. Trace and CBV/CBR evaluation using the env & store
  5. Implement CBR/CBR

13

Nugget

There are flavors to parameter passing.

14

What is the value of the following expression?

- What happens during evaluation?

```
let p = proc (x) set x = 4
in let a = 3
   in begin (p a); a end
```

15

Parameter Passing Variations

- Natural (PROC)
- Call-by-value
- Call-by-reference
- Call-by-name (lazy evaluation)
- Call-by-need (lazy evaluation)

16

PROC

```
let p = proc (x) set x = 4
in let a = 3
   in begin (p a); a end
```

Evaluates to 3

17

Call-by-value (IREF)

```
let p = proc (x) set x = 4
in let a = 3
   in begin (p a); a end
```

Evaluates to 3

18

IREF -- Call-by-reference

```
let p = proc (x) set x = 4
in let a = 3
   in begin (p a); a end
```

Evaluates to 4

19

Nugget

In Call by Value, a copy of the argument is passed

20

Another example

```
let f = proc (x) set x = 44
in let g = proc (y) (f y)
   in let z = 55
      in begin (g z); z end
```

CBV → 55

CBR → 44

21

```
let f = proc (x) set x = 44
in let g = proc (y) (f y)
in let z = 55
in begin
     (g z);
     z
  end
```

Evaluation trace

```
>  (run "
let f = proc (x) set x = 44
in let g = proc (y) (f y)
in let z = 55
in begin
     (g z);
     z
  end")
newref: allocating location 0
newref: allocating location 1
newref: allocating location 2
entering let f
newref: allocating location 3
entering body of let f with env =
((f 3) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2)))))
```

```
entering let g
newref: allocating location 4
entering body of let g with env =
((g 4) (f 3) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2))))
 (4 (procedure y ... ((f 3) (i 0) (v 1) (x 2)))))

entering let z
newref: allocating location 5
entering body of let z with env =
((z 5) (g 4) (f 3) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2))))
 (4 (procedure y ... ((f 3) (i 0) (v 1) (x 2))))
 (5 #(struct:num-val 55)))
```

22

Evaluation trace

```
let f = proc (x) set x = 44
in let g = proc (y) (f y)
in let z = 55
in begin
    (g z);
    z
  end
```

```
entering let g
newref: allocating location 4
entering body of let g with env =
((g 4) (f 3) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2))))
 (4 (procedure y ... ((f 3) (i 0) (v 1) (x 2)))))

entering let z
newref: allocating location 5
entering body of let z with env =
((z 5) (g 4) (f 3) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2))))
 (4 (procedure y ... ((f 3) (i 0) (v 1) (x 2))))
 (5 #(struct:num-val 55)))
```

```
entering body of proc y with env =
((y 5) (f 3) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2))))
 (4 (procedure y ... ((f 3) (i 0) (v 1) (x 2))))
 (5 #(struct:num-val 55)))

entering body of proc x with env =
((x 5) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2))))
 (4 (procedure y ... ((f 3) (i 0) (v 1) (x 2))))
 (5 #(struct:num-val 55)))

#(struct:num-val 44)
>
```

23

Uses of call-by-reference

- Multiple return values

```
let swap = proc (x) proc (y)
            let temp = x
            in begin
                set x = y;
                set y = temp
               end
in let a = 33
   in let b = 44
      in begin
          ((swap a) b);
          -(a,b)
         end
```

24

Learning outcomes of this lecture

- A student attending this lecture should be able to:
  1. Understand that there are variations to parameter passing
  2. Understand CBV/CBR and how they work
  3. Understand the uses of CBR
  4. Trace and CBV/CBR evaluation using the env & store
  5. Implement CBR

25

Questions?

26

Implementing CBR

- Expressed and denoted values remain the same
- Location allocation policy changes
  - If the formal parameter is a variable, pass on the reference
  - Otherwise, put the value of the formal parameter into the memory, pass a reference to it

$$ExpVal = Int + Bool + Proc$$
$$DenVal = Ref(ExpVal)$$

```
(call-exp (rator rand)
  (let ((proc (expval->proc (value-of rator env)))
        (arg (value-of-operand rand env)))
    (apply-procedure proc arg)))
```

apply-procedure : $Proc \times Ref \to ExpVal$

```
(define apply-procedure
  (lambda (proc1 val)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of body
          (extend-env var val saved-env))))))
```

value-of-operand : $Exp \times Env \to Ref$

```
(define value-of-operand
  (lambda (exp env)
    (cases expression exp
      (var-exp (var) (apply-env env var))
      (else
        (newref (value-of exp env))))))
```

27

Another example

```
let b = 3
in let p = proc (x) proc(y)
            begin
             set x = 4;
             y
            end
   in ((p b) b)
```

- Here there is variable aliasing
- This evaluates to 4

28

Lazy evaluation

- Call-by-name
- Call-by-need

```
letrec infinite-loop (x) = infinite-loop(-(x,-1))
in let f = proc (z) 11
   in (f (infinite-loop 0))
```

29

Thunks

- Save any future work for the future

```
(define-datatype thunk thunk?
  (a-thunk
    (exp1 expression?)
    (env environment?)))
```

30

Implementation (call-by-name)

$$
\begin{aligned}
DenVal &= Ref(ExpVal + Thunk) \\
ExpVal &= Int + Bool + Proc
\end{aligned}
$$

value-of-operand : $Exp \times Env \to Ref$

```
(define value-of-operand
  (lambda (exp env)
    (cases expression exp
      (var-exp (var) (apply-env env var))
      (else
        (newref (a-thunk exp env))))))
```

```
(var-exp (var)
  (let ((ref1 (apply-env env var)))
    (let ((w (deref ref1)))
      (if (expval? w)
        w
        (value-of-thunk w)))))
```

value-of-thunk : $Thunk \to ExpVal$

```
(define value-of-thunk
  (lambda (th)
    (cases thunk th
      (a-thunk (exp1 saved-env)
        (value-of exp1 saved-env)))))
```

31

Memoization (call-by-need)

```
(var-exp (var)
  (let ((ref1 (apply-env env var)))
    (let ((w (deref ref1)))
      (if (expval? w)
        w
        (let ((val1 (value-of-thunk w)))
          (begin
            (setref! ref1 val1)
            val1))))))
```

32

Quiz #8

- In the previous lecture we introduced two implementations for mutable pairs – the second one being a more space-efficient version. The implementation for **`setleft`** is given below. Write down the implementation for **`setright`**.

setleft : *MutPair* × *ExpVal* → *Unspecified*

```
(define setleft
  (lambda (p val)
    (setref! p val)))
```

33