State – Effects – Review

T. METIN SEZGIN

Languages considered so far

- LET
- PROC
- LETREC
- EXPLICIT-REFS (EREF)

New concepts

- Storable values
  - What sorts of things can we store?
- Memory stores
  - Where do we store things?
- Memory references (pointers)
  - How do we access the stores?

The new design

- Denotable and Expressed values

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

- Three new operations
  - `newref`
  - `deref`
  - `setref`

Example: references help us share variables

```
let x = newref(0)
in letrec even(dummy)
            = if zero?(deref(x))
                then 1
                else begin
                        setref(x, -(deref(x),1));
                        (odd 888)
                     end
         odd(dummy)
          = if zero?(deref(x))
                then 0
                else begin
                        setref(x, -(deref(x),1));
                        (even 888)
                     end
   in begin setref(x,13); (odd 888) end
```

Example: references help us create hidden state

```
let g = let counter = newref(0)
        in proc (dummy)
            begin
              setref(counter, -(deref(counter), -1));
              deref(counter)
            end
in let a = (g 11)
   in let b = (g 11)
      in -(a,b)
```
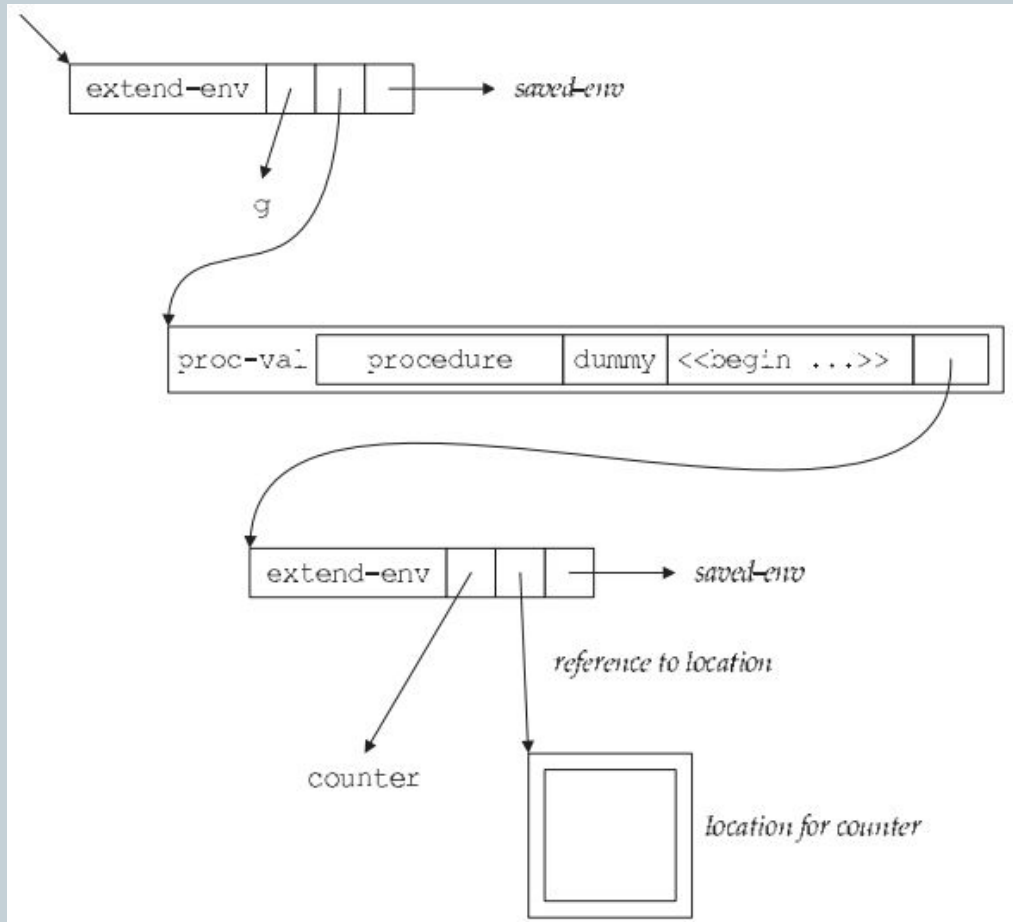
The entire expression evaluates to -1

Behind the scenes…

```
let g = let counter = newref(0)
        in proc (dummy)
             begin
              setref(counter, -(deref(counter), -1));
              deref(counter)
             end
in let a = (g 11)
   in let b = (g 11)
      in -(a,b)
```


extend-env | | | → saved-env

g

proc-val | procedure | dummy | <<begin ...>> |

extend-env | | | → saved-env

reference to location

counter

location for counter

Example: reference to a reference

```
let x = newref(newref(0))
in begin
    setref(deref(x), 11);
    deref(deref(x))
end
```

What does this evaluate to?

Lecture 21
State – Effects – Implementation

T. METIN SEZGIN

EREF implementation

- What happens to the store?
- How do we represent/implement stores?

- Behavior specification
- Implementation

Nugget

In order to add the memory feature to the language, we need a data structure

Store passing specifications

- The new **value-of**
$$(\text{value-of } exp_1 \ \rho \ \sigma_0) = (val_1, \sigma_1)$$

Nugget

We also need to rewrite the rules of evaluation to use the memory

Store passing specifications

- The new **value-of** $(\text{value-of } exp_1 \; \rho \; \sigma_0) = (val_1, \sigma_1)$

- Example $(\text{value-of } (\text{const-exp } n) \; \rho \; \sigma) = (n, \sigma)$

- More examples

$$\frac{(\text{value-of } exp_1 \; \rho \; \sigma_0) = (val_1, \sigma_1) \qquad (\text{value-of } exp_2 \; \rho \; \sigma_1) = (val_2, \sigma_2)}{(\text{value-of } (\text{diff-exp } exp_1 \; exp_2) \; \rho \; \sigma_0) = (\lceil \lfloor val_1 \rfloor - \lfloor val_2 \rfloor \rceil, \sigma_2)}$$

$$\frac{(\text{value-of } exp_1 \; \rho \; \sigma_0) = (val_1, \sigma_1)}{(\text{value-of } (\text{if-exp } exp_1 \; exp_2 \; exp_3) \; \rho \; \sigma_0)}$$
$$= \begin{cases} (\text{value-of } exp_2 \; \rho \; \sigma_1) & \text{if } (\text{expval->bool } val_1) = \#t \\ (\text{value-of } exp_3 \; \rho \; \sigma_1) & \text{if } (\text{expval->bool } val_1) = \#f \end{cases}$$

Nugget

We also need to write the rules of evaluation for the new expressions

Grammar specification

- The new grammar

$Expression ::= \texttt{newref} \ (Expression)$
$\boxed{\texttt{newref-exp (exp1)}}$

$Expression ::= \texttt{deref} \ (Expression)$
$\boxed{\texttt{deref-exp (exp1)}}$

$Expression ::= \texttt{setref} \ (Expression \ , \ Expression)$
$\boxed{\texttt{setref-exp (exp1 exp2)}}$

- Specification

$$\frac{(\texttt{value-of } exp \ \rho \ \sigma_0) = (val, \sigma_1) \qquad l \notin \mathrm{dom}(\sigma_1)}{(\texttt{value-of } (\texttt{newref-exp } exp) \ \rho \ \sigma_0) = ((\texttt{ref-val } l), [l{=}val]\sigma_1)}$$

$$\frac{(\texttt{value-of } exp \ \rho \ \sigma_0) = (l, \sigma_1)}{(\texttt{value-of } (\texttt{deref-exp } exp) \ \rho \ \sigma_0) = (\sigma_1(l), \sigma_1)}$$

$$\frac{\begin{array}{c}(\texttt{value-of } exp_1 \ \rho \ \sigma_0) = (l, \sigma_1)\\ (\texttt{value-of } exp_2 \ \rho \ \sigma_1) = (val, \sigma_2)\end{array}}{(\texttt{value-of } (\texttt{setref-exp } exp_1 \ exp_2) \ \rho \ \sigma_0) = (\lceil 23 \rceil, [l{=}val]\sigma_2)}$$

Nugget

The implementation will require adding and initializing a **store** structure

Implementation

```
value-of-program : Program → ExpVal
(define value-of-program
  (lambda (pgm)
    (initialize-store!)
    (cases program pgm
      (a-program (exp1)
        (value-of exp1 (init-env)))))))
```

Nugget

We need ways of accessing and manipulating the **store**

Implementation of Stores

```
empty-store : () → Sto
(define empty-store
  (lambda () '()))

get-store : () → Sto
(define get-store
  (lambda () the-store))

reference? : SchemeVal → Bool
(define reference?
  (lambda (v)
    (integer? v)))

deref : Ref → ExpVal
(define deref
  (lambda (ref)
    (list-ref the-store ref)))
```

```
usage: A Scheme variable containing the current state
  of the store. Initially set to a dummy value.
(define the-store 'uninitialized)


initialize-store! : () → Unspecified
usage: (initialize-store!) sets the-store to the empty store
(define initialize-store!
    ambda ()
    (set! the-store (empty-store))))

newref : ExpVal → Ref
(define newref
  (lambda (val)
    (let ((next-ref (length the-store)))
      (set! the-store (append the-store (list val)))
      next-ref)))
```

setref!

```
setref! : Ref × ExpVal → Unspecified
usage:  sets the-store to a state like the original, but with
  position ref containing val.
(define setref!
  (lambda (ref val)
    (set! the-store
      (letrec
        ((setref-inner
          usage:  returns a list like store1, except that
          position ref1 contains val.
          (lambda (store1 ref1)
            (cond
              ((null? store1)
               (report-invalid-reference ref the-store))
              ((zero? ref1)
               (cons val (cdr store1)))
              (else
                (cons
                  (car store1)
                  (setref-inner
                    (cdr store1) (- ref1 1))))))))
        (setref-inner the-store ref)))))
```

Implementation
newref-exp, deref-exp, setref-exp

```
(newref-exp (exp1)
  (let ((v1 (value-of exp1 env)))
    (ref-val (newref v1))))

(deref-exp (exp1)
  (let ((v1 (value-of exp1 env)))
    (let ((ref1 (expval->ref v1)))
      (deref ref1))))

(setref-exp (exp1 exp2)
  (let ((ref (expval->ref (value-of exp1 env))))
    (let ((val2 (value-of exp2 env)))
      (begin
        (setref! ref val2)
        (num-val 23)))))
```