

EREF - Review



T. METIN SEZGIN

Newref create an address

Deref takes an address and return value

Setref takes an address and changes its content to the value that is given as second variable of the setref.

The new grammar

```
Expression ::= newref (Expression)
               newref-exp (exp1)

Expression ::= deref (Expression)
               deref-exp (exp1)

Expression ::= setref (Expression , Expression)
               setref-exp (exp1 exp2)
```

Specification

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

Here it returns 23, since **every expression has to turn something**, setref also returns a value. This is just a value that we don't care about.

Must note any potential change to the memory.

$$(\text{value-of } \text{exp}_1 \ \rho \ \sigma_0) = (\text{val}_1, \sigma_1)$$

In case exp_1 changes the memory, must use sigma 1, not 0

$$(\text{value-of } \text{exp}_2 \ \rho \ \sigma_1) = (\text{val}_2, \sigma_2)$$

$$(\text{value-of } (\text{diff-exp } \text{exp}_1 \ \text{exp}_2) \ \rho \ \sigma_0) = ([\text{val}_1] - [\text{val}_2], \sigma_2)$$

$$(\text{value-of } \text{exp}_1 \ \rho \ \sigma_0) = (\text{val}_1, \sigma_1)$$

Again, must use new sigma just in case the memory is changed

$$\begin{aligned} & (\text{value-of } (\text{if-exp } \text{exp}_1 \ \text{exp}_2 \ \text{exp}_3) \ \rho \ \sigma_0) \\ &= \begin{cases} (\text{value-of } \text{exp}_2 \ \rho \ \sigma_1) & \text{if } (\text{expval} \rightarrow \text{bool } \text{val}_1) = \#t \\ (\text{value-of } \text{exp}_3 \ \rho \ \sigma_1) & \text{if } (\text{expval} \rightarrow \text{bool } \text{val}_1) = \#f \end{cases} \end{aligned}$$

IREF & Mutable Pairs



T. METIN SEZGIN

Learning outcomes of this lecture



- A student attending this lecture should be able to:
 1. Understand and implement implicit references
 2. Understand how pairs can be implemented, and do so
 3. Explain alternative implementations of pairs
 4. Implement more sophisticated data structures (e.g., stack, arrays).

Nuggets



- One can have implicitly managed references as well
- Now that we have a memory structure, we can add more sophisticated structures to our language

Nugget



One can have implicitly managed references as well

Implicit references



● EREF

- References instantiated explicitly
- References explicitly stored in the store
- Expressed and denoted values include references

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

Implicit references



● IREF

- References are instantiated by the interpreter
- All denoted values are references to expressed values
- Each binding operation introduces a location
 - Let
 - letrec
 - proc
- Pointers to stores are saved in the environment

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

New grammar



- A set operation for assignment

Expression ::= set *Identifier* = *Expression*
assign-exp (var exp1)

Examples



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

```
let g = let count = 0
        in proc (dummy)
            begin
                set count = -(count,-1);
                count
            end
  in let a = (g 11)
    in let b = (g 11)
      in -(a,b)
```

Behavior specification



- **var-exp**

$$(\text{value-of } (\text{var-exp } var) \ \rho \ \sigma) = (\sigma(\rho(var)), \sigma)$$

- **assign-exp**

$$\frac{(\text{value-of } exp_1 \ \rho \ \sigma_0) = (val_1, \sigma_1)}{(\text{value-of } (\text{assign-exp } var \ exp_1) \ \rho \ \sigma_0) = ([27], [\rho(var) = val_1]\sigma_1)}$$

- **apply-procedure**

$$\begin{aligned} &(\text{apply-procedure } (\text{procedure } var \ body \ \rho) \ val \ \sigma) \\ &= (\text{value-of } body \ [var = l]\rho \ [l = val]\sigma) \end{aligned}$$

Implementation



● **var-exp**

```
(var-exp (var) (deref (apply-env env var)))
```

● **assign-exp**

```
(assign-exp (var exp1)
  (begin
    (setref!
      (apply-env env var)
      (value-of exp1 env))
    (num-val 27)))
```

● **apply-procedure**

```
apply-procedure :  $Proc \times ExpVal \rightarrow ExpVal$ 
(define apply-procedure
  (lambda (proc1 val)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of body
          (extend-env var (newref val) saved-env)))))))
```

Implementation



Reference instantiations

- **apply-procedure**

```
apply-procedure : Proc × ExpVal → ExpVal
(define apply-procedure
  (lambda (proc1 val)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of body
          (extend-env var (newref val) saved-env)))))))
```

- **let**

```
(let-exp (var exp1 body)
  (let ((val1 (value-of exp1 env)))
    (value-of body
      (extend-env var (newref val1) env))))
```

- **letrec**

```
(extend-env-rec (p-names b-vars p-bodies saved-env)
  (let ((n (location search-var p-names)))
    (if n
      (newref
        (proc-val
          (procedure
            (list-ref b-vars n)
            (list-ref p-bodies n)
            env))))
      (apply-env saved-env search-var))))
```

Nugget



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

Nugget



Having a memory feature allows us to have
mutable pairs

In addition we want mutation



- New grammar

newpair : $Expval \times Expval \rightarrow MutPair$
left : $MutPair \rightarrow Expval$
right : $MutPair \rightarrow Expval$
setleft : $MutPair \times Expval \rightarrow Unspecified$
setright : $MutPair \times Expval \rightarrow Unspecified$

- New set of

- Denotables
- Expressibles

ExpVal = $Int + Bool + Proc + MutPair$
DenVal = $Ref(ExpVal)$
MutPair = $Ref(ExpVal) \times Ref(ExpVal)$

```
(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?)))
```

New scheme functions for pair management



make-pair : $ExpVal \times ExpVal \rightarrow MutPair$

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

left : $MutPair \rightarrow ExpVal$

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

right : $MutPair \rightarrow ExpVal$

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

setleft : $MutPair \times ExpVal \rightarrow Unspecified$

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

setright : $MutPair \times ExpVal \rightarrow Unspecified$

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

The Interpreter



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

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

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

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

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

Nugget



We can get creative and devise a more efficient implementation

A different representation for mutable pairs



- Note something about the addresses of the two values

make-pair : $ExpVal \times ExpVal \rightarrow MutPair$

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

left : $MutPair \rightarrow ExpVal$

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

A different representation for mutable pairs



mutpair? : *SchemeVal* \rightarrow *Bool*

```
(define mutpair?  
  (lambda (v)  
    (reference? v)))
```

make-pair : *ExpVal* \times *ExpVal* \rightarrow *MutPair*

```
(define make-pair  
  (lambda (val1 val2)  
    (let ((ref1 (newref val1)))  
      (let ((ref2 (newref val2)))  
        ref1))))
```

left : *MutPair* \rightarrow *ExpVal*

```
(define left  
  (lambda (p)  
    (deref p)))
```

right : *MutPair* \rightarrow *ExpVal*

```
(define right  
  (lambda (p)  
    (deref (+ 1 p)))))
```

setleft : *MutPair* \times *ExpVal* \rightarrow *Unspecified*

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

setright : *MutPair* \times *ExpVal* \rightarrow *Unspecified*

```
(define setright  
  (lambda (p val)  
    (setref! (+ 1 p) val)))
```


Learning outcomes of this lecture



- A student attending this lecture should be able to:
 1. Understand and implement implicit references
 2. Understand how pairs can be implemented, and do so
 3. Explain alternative implementations of pairs
 4. Implement more sophisticated data structures (e.g., stack, arrays).