Lecture
Mutable Pairs – MP

T. METIN SEZGIN

1

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).

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

Nugget

Having a memory feature allows us to have

mutable pairs

5

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

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

```
(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?)))
```

6

New scheme functions for pair management

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)))))
```

right : *MutPair* → *ExpVal*

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

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

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

setright : *MutPair* × *ExpVal* → *Unspecified*

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

7

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)))))
```

8

Nugget

We can get creative and devise a more efficient implementation

9

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

10

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)))
```

11

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).

12