IREF & Mutable Pairs

T. METIN SEZGIN

Announcements

- The environment question graded generously

EREF - Review

T. METIN SEZGIN

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

The new grammar

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

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

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

Specification

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

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.

$$\frac{\begin{array}{c}(\texttt{value-of}\ exp_1\ \rho\ \sigma_0) = (val_1, \sigma_1) \\ (\texttt{value-of}\ exp_2\ \rho\ \sigma_1) = (val_2, \sigma_2)\end{array}}{(\texttt{value-of}\ (\texttt{diff-exp}\ exp_1\ exp_2)\ \rho\ \sigma_0) = (\lceil \lfloor val_1 \rfloor - \lfloor val_2 \rfloor \rceil, \sigma_2)}$$

In case exp1 changes the memory, must use sigma 1, not 0

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

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

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

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

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

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

New grammar

- A set operation for assignment

$$\mathit{Expression} ::= \texttt{set}\ \mathit{Identifier}\ \texttt{=}\ \mathit{Expression}$$

$$\boxed{\texttt{assign-exp (var exp1)}}$$

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**

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

- **assign-exp**

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

- **apply-procedure**

$$\texttt{(apply-procedure (procedure } var\ body\ \rho\texttt{)}\ val\ \sigma\texttt{)}$$
$$= \texttt{(value-of } body\ [var = l]\rho\ [l = val]\sigma\texttt{)}$$

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 : $Proc \times ExpVal \to ExpVal$

```
(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

Adding lists/pairs to the language

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

Nugget

Having a memory feature allows us to have

mutable pairs

In addition we want mutation

- New grammar

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

- New set of
  - Denotables
  - Expressibles

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

```
(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?)))
```

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)))))
```

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)))))
```

Nugget

We can get creative and devise a more efficient implementation

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

A different representation for mutable pairs

mutpair? : $\mathit{SchemeVal} \rightarrow \mathit{Bool}$

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

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

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

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

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

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

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

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

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

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

```
(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).