Lecture 19
Lexical Addressing
INTERPRETATION
Review

T. METIN SEZGIN

1

Nameless interpreter

- Use lexical addresses instead of variable names
- Implement a new model of environment
  - Use addresses
  - Much like a memory model

2

On the top level

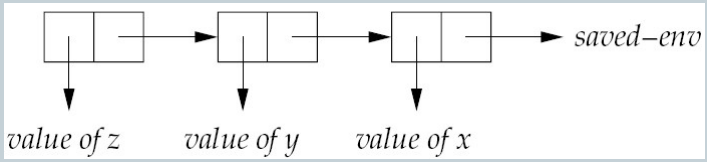
```
run : String → ExpVal
(define run
  (lambda (string)
    (value-of-program
      (translation-of-program
        (scan&parse string)))))
```

3

New environment interface

nameless-environment

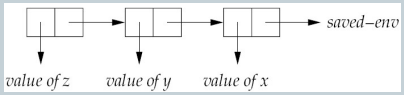
nameless-environment?	: *SchemeVal* → *Bool*
empty-nameless-env	: () → *Nameless-env*
extend-nameless-env	: *Expval* × *Nameless-env* → *Nameless-env*
apply-nameless-env	: *Nameless-env* × *Lexaddr* → *DenVal*



4

New environment interface

nameless-environment?	: *SchemeVal* → *Bool*
empty-nameless-env	: () → *Nameless-env*
extend-nameless-env	: *Expval* × *Nameless-env* → *Nameless-env*
apply-nameless-env	: *Nameless-env* × *Lexaddr* → *DenVal*




```
nameless-environment? : SchemeVal → Bool
(define nameless-environment?
  (lambda (x)
    ((list-of expval?) x)))

empty-nameless-env : () → Nameless-env
(define empty-nameless-env
  (lambda ()
    '()))

extend-nameless-env : ExpVal × Nameless-env → Nameless-env
(define extend-nameless-env
  (lambda (val nameless-env)
    (cons val nameless-env)))

apply-nameless-env : Nameless-env × Lexaddr → ExpVal
(define apply-nameless-env
  (lambda (nameless-env n)
    (list-ref nameless-env n)))
```

5

Procedure specification and implementation

```
(apply-procedure (procedure body ρ) val)
= (value-of body (extend-nameless-env val ρ))
```

```
procedure : Nameless-exp × Nameless-env → Proc
(define-datatype proc proc?
  (procedure
    (body expression?)
    (saved-nameless-env nameless-environment?)))
```

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

6

Interpreter for the new language

value-of : *Nameless-exp* × *Nameless-env* → *ExpVal*

```
(define value-of
  (lambda (exp nameless-env)
    (cases expression exp

      (const-exp (num)     ...as before...)
      (diff-exp (exp1 exp2)   ...as before...)
      (zero?-exp (exp1)       ...as before...)
      (if-exp (exp1 exp2 exp3)  ...as before...)
      (call-exp (rator rand)   ...as before...)

      (nameless-var-exp (n)
        (apply-nameless-env nameless-env n))

      (nameless-let-exp (exp1 body)
        (let ((val (value-of exp1 nameless-env)))
          (value-of body
            (extend-nameless-env val nameless-env))))

      (nameless-proc-exp (body)
        (proc-val
          (procedure body nameless-env)))

      (else
        (report-invalid-translated-expression exp)))))
```

7

Lecture 20
State – Effects

T. METIN SEZGIN

8

Languages considered so far

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

9

Computational Effects

- So far we have considered
  - Expressions generating values
  - Everything local
  - No notion of global state
  - No global storage
- We want to be able to
  - Read memory locations
  - Print values in the memory
  - Write to the memory
  - Have global variables
  - Share values across separate computations
- We need
  - A model for memory
    - Access memory locations
    - Modify memory contents

10

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?

11

The new design

- Denotable and Expressed values

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

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

12

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
```

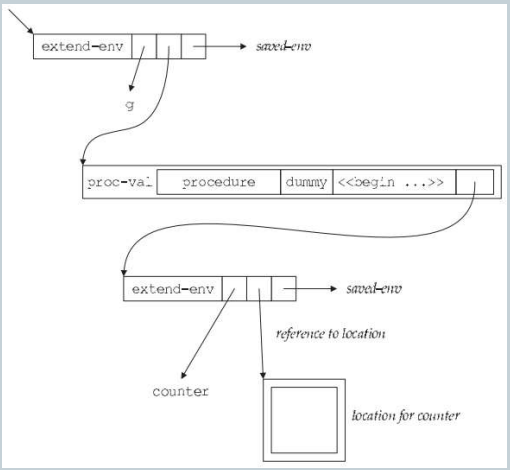

Example: references help us create hidden state

```
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)
```



15

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?

16

EREF implementation

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

- Behavior specification
- Implementation

17

Nugget

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

18

Store passing specifications

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

19

Nugget

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

20

Store passing specifications

- The new **value-of** $(\texttt{value-of}\ exp_1\ \rho\ \sigma_0) = (val_1, \sigma_1)$
- Example $(\texttt{value-of}\ (\texttt{const-exp}\ n)\ \rho\ \sigma) = (n, \sigma)$
- More examples

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

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

21

Nugget

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

22

Grammar specification

- The new grammar

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

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

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

- Specification

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

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

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