Parameter Passing
Lazy evaluation

T. METIN SEZGIN

End-of-semester evaluations

1. Please fill out the EOS evaluation.

What is the value of the following expression?

- What happens during evaluation?

```
let p = proc (x) set x = 4
in let a = 3
   in begin (p a); a end
```

Review

Parameter Passing

Call by Value

* Creates the copy of actual value
* Passes the copied variable's memory address

```
let p = proc (x) set x = 4
in let a = 3
   in begin (p a); a end
```

→ What happens during evaluation

CBV → 3
CBR → 4

Call by value creates a copy and pushes it into the memory then passes the memory address of copied value as parameter. Therefore if you change the value of a into the scope of the function, you change the value of the copied reference not actual a

Call by Reference

* Sends the original memory address of variable

Call by reference modifies the value of original variable since modifications are made on the actual memory address of the variable

```
let f = proc (x) set x = 44
in let g = proc (y) (f y)
   in let z = 55
      in begin (g z); z end
```

CBR → 44
CBV → 55

Review

* When it is called that create the CRR and CBV versions of a function, the first thing you should care variables. If functions do not have variables, CBV and CBR will be same since CBR and CBV have effect on variables.

Implementing CBR

- Expressed and denoted values remain the same

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

- Location allocation policy changes
  - If the formal parameter is a variable, pass on the reference
  - Otherwise, put the value of the formal parameter into the memory, pass a reference to it

```
(call-exp (rator rand)
  (let ((proc (expval->proc (value-of rator env)))
        (arg (value-of-operand rand env)))
    (apply-procedure proc arg))))
```

creates the copy of the actual variable's value here

```
value-of-operand : Exp × Env → Ref
(define value-of-operand
  (lambda (exp env)
    (cases expression exp
      (var-exp (var) (apply-env env var))
      (else
        (newref (value-of exp env))))))
```

- Bora Karagul

Learning outcomes of this lecture

- A student attending this lecture should be able to:
  1. Understand that there are variations to parameter passing
  2. Understand CBV/CBR and how they work
  3. Understand the uses of CBR
  4. Trace and CBV/CBR evaluation using the env & store
  5. Implement CBR/CBR

Parameter Passing Variations

- Natural (PROC)
- Call-by-value
- Call-by-reference
- Call-by-name (lazy evaluation)
- Call-by-need (lazy evaluation)

Lazy evaluation

- Call-by-name
- Call-by-need

```
letrec infinite-loop (x) = infinite-loop(-(x,-1))
in let f = proc (z) 11
   in (f (infinite-loop 0))
```

Thunks

- Save any future work for the future

```
(define-datatype thunk thunk?
  (a-thunk
    (exp1 expression?)
    (env environment?)))
```

Implementation (call-by-name)

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

value-of-operand : $Exp \times Env \to Ref$

```
(define value-of-operand
  (lambda (exp env)
    (cases expression exp
      (var-exp (var) (apply-env env var))
      (else
        (newref (a-thunk exp env))))))
```

```
(var-exp (var)
  (let ((ref1 (apply-env env var)))
    (let ((w (deref ref1)))
      (if (expval? w)
        w
        (value-of-thunk w)))))
```

value-of-thunk : $Thunk \to ExpVal$

```
(define value-of-thunk
  (lambda (th)
    (cases thunk th
      (a-thunk (exp1 saved-env)
        (value-of exp1 saved-env)))))
```

Memoization (call-by-need)

```
(var-exp (var)
  (let ((ref1 (apply-env env var)))
    (let ((w (deref ref1)))
      (if (expval? w)
        w
        (let ((val1 (value-of-thunk w)))
          (begin
            (setref! ref1 val1)
            val1))))))
```