Parameter Passing

CS469

Parameter Passing Variations

What are the values that are passed as actual parameters when a procedure is called?

1. Call-by-value
2. Call-by-reference
3. Call-by-name
4. Call-by-need

Call-by-value

- The most common parameter passing mechanism.
- It is the one currently implemented in IMPLICIT-REFS

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

Under call-by-value semantics `a` and `x` point to different references

Call-by-value

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

Code	Env	Store
	empty-env	empty-store
`let a = 3`	$a \to 0$	$0 \to$ `num-val 3`
`let p = proc(x) set x = 4`	$a \to 0$, $p \to 1$	$0 \to$ `num-val 3`, $1 \to$ `proc-val ...`
`(p a)`	$a \to 0$, $p \to 1$, $x \to 2$	$0 \to$ `num-val 3`, $1 \to$ `proc-val ...`, $2 \to$ `num-val 3`
`set x = 4`	$a \to 0$, $p \to 1$, $x \to 2$	$0 \to$ `num-val 3`, $1 \to$ `proc-val ...`, $2 \to$ `num-val 4`
`a`		
3		

Evaluating Procedure Calls

The current call-by-value implementation

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

- Note how a new copy is placed in the store

```
;;  {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))))))
```

Assessment

- The caller cannot see the modifications of the callee.
- The procedure gets a copy of the value asociated with its parameters.
- The procedure has a private reference for each parameter.

Call-by-Value

Call-by-Reference

Lazy Evaluation

Call-by-reference - Example 1

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

Code	Env	Store
	empty-env	empty-store
`let a = 3`	$a \to 0$	$0 \to$ `num-val 3`
`let p = proc(x) set x = 4`	$a \to 0$, $p \to 1$	$0 \to$ `num-val 3`, $1 \to$ `proc-val ...`
`(p a)`	$a \to 0$, $p \to 1$, $x \to 0$	$0 \to$ `num-val 3`, $1 \to$ `proc-val ...`
`set x = 4`	$a \to 0$, $p \to 1$, $x \to 0$	$0 \to$ `num-val 4`, $1 \to$ `proc-val ...`
`a`		
4		

Call-by-reference - Example 2

```
let p = proc(x)  -(5,x)
in (p 3)
```

- What should be passed to `p`?
- A copy of 3, of course

Call-by-reference in IMPLICIT-REFS

- If an operand is a variable reference, then
  - A reference to the variable location is passed
  - The formal parameter is bound to the location of the operand
- If the operand is not a variable reference (for example: the number 3), then
  - A new reference is created, and it is treated as in call-by-value

Example 3

```
let a = 3
in let f = proc(x) proc(y) set x = -(x,y)
   in begin
        ((f a) 2);
        a
      end
```

- Since `a` is a variable, it is passed by reference
- Since 2 is a constant, it is passed by value
- This program evaluates to 1

Example 4 – Swapping the values of two variables

```
let swap = proc (x) proc (y)
                        let temp = x
                        in begin
                             set x = y;
                             set y = temp
                           end
          in let a = 33
          in let b = 44
          in begin
               ((swap a) b);
               -(a,b)
             end
```

- What is the output of this program under call-by-value and under call-by-reference? (call-by-value) (num-val -11)
- Execution trace on the board

(call-by-reference) (num-val 11)

Implementing Call-by-reference in IMPLICIT-REFS

- Expressed and denoted values do not change

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

- The only thing that changes is the allocation of new locations when parameters are passed
  - Call-by-value:
    - a new location is created for every evaluation of an operand
  - Call-by-reference:
    - a new location is created for every evaluation of an operand other than a variable.

Modifying the interpreter

Before: `apply-procedure::{proc, expval} -> expval`

```
(define apply-procedure
 (lambda (proc1 arg)
   (cases proc proc1
     (procedure (var body saved-env)
        (value-of body
          (extend-env var (newref arg) saved-env))))))
```

Now: `apply-procedure::{proc, ref} -> expval`

```
(define apply-procedure
  (lambda (proc1 ref)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of body
          (extend-env var ref saved-env))))))
```

Modifying the interpreter

Before: `apply-procedure::{proc, expval} -> expval`

```
(define apply-procedure
 (lambda (proc1 arg)
   (cases proc proc1
     (procedure (var body saved-env)
        (value-of body
          (extend-env var (newref arg) saved-env))))))
```

Now: `apply-procedure::{proc, ref} -> expval`

```
(define apply-procedure
  (lambda (proc1 ref)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of body
          (extend-env var ref saved-env))))))
```

Modifying the interpreter

Before: `value-of`: the case for function application

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

Now: `value-of`: the case for function application

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

Modifying the interpreter

`value-of`: the case for function application

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

`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))))))
```

Call-by-Value

Call-by-Reference

Lazy Evaluation

Lazy evaluation

- Another parameter-passing mechanism.
- In a given call a procedure may never refer to one or more of its formal parameters.
- Time devoted to evaluating the operands is wasted.
  - Such evaluation may yield an error or never terminate.
  - It is better to postpone it
- Deciding if a parameter is going to be used or not is generally undecidable.

Lazy evaluation

- We can postpone the evaluation of a procedure by encapsulating the operand as the body of a thunk.
  - thunk: a procedure of no arguments.
  - freezing: creating a thunk.
  - thawing: evaluating a thunk.
- Lazy evaluation mechanisms vary in the way they handle multiple references to the same parameter.
  - Call-by-name
  - Call-by-need

Lazy evaluation variations

- Call-by-name: invokes the thunk every time the parameter is referred to.
- Call-by-need: records the value of the thunk the first time it is invoked and saves the value for future invocations of the thunk.

- Without side effects, call-by-name and call-by-need give the same answer.
- In the presence of side effects, they yield different results.
- Let's see an example

Example of benefit of lazy evaluation

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

- Here infinite-loop is a procedure that, when called, never terminates.
- f is a procedure that, when called, never refers to its argument and always returns 11.
- What happens when evaluating this expression using call-by-value or call-by-reference?

Example of benefit of lazy evaluation

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

- In lazy parameter passing, the expression `(infinite-loop 0)` is encapsulated into a thunk
- This has the effect of freezing the expression
- The procedure `proc (z) 11` is then called, where `z` is associated to the thunk
- Since the body of the procedure, namely `11`, does not require looking up `z`, the thunk is never evaluated and the result `11` is immediately returned

Freezing an expression for evaluating it later

- In order to implement lazy evaluation we must freeze the argument expression without evaluating it
- That way it can be evaluated only if it is really needed
- A thunk is a data structure that holds a frozen expression

```
1 (define-datatype thunk thunk?
2   (a-thunk
3     (exp1 expression?)
4     (env environment?)))
```

- Why do we need env? Same reason as in closures

Expressed and Denoted Values

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

- An environment now holds a reference either to an expressed value or to a thunk
- Thunks, however, are not expressed values: they cannot be returned as the result of a computation

Modifying the interpreter

Before (call-by-reference): `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))))))
```

Now: `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 (a-thunk exp env))))))
```

Modifying the interpreter

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

The case for variables in `value-of` for call-by-name.

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

Modifying the interpreter

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

The case for variables in `value-of` for call-by-name.

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

Modifying the interpreter

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

The case for variables in `value-of` for call-by-name.

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

Modifying the interpreter

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

The case for variables in `value-of` for call-by-name.

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

Modifying the interpreter

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

The case for variables in `value-of` for call-by-name.

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

Modifying the interpreter(cont.)

`value-of-thunk:: thunk ->expval`

```
1 (define value-of-thunk
2   (lambda (th)
3     (cases thunk th
4       (a-thunk (exp1 saved-env)
5         (value-of exp1 saved-env)))))
```

Lazy evaluation variations

- When a parameter is frozen inside a thunk, every time it is looked-up it has to be re-evaluated
- This is inefficient
- Also, this may cause effects to be executed multiple times
- We review this situation with an example and then introduce a variation of lazy evaluation called call-by-need

Lazy evaluation variations

```
let g = let count = 0
        in proc ()
             begin
              set count = +(count,1);
              count
             end
in (proc (x) +(x,x)
    (g))
```

- The procedure g returns the number of times it is called.
- Under call-by-name each reference to variable x invokes g and the result is 3.
- Under call-by-need the result is 2, because g is invoked only once.

Call-by-need

- Once we find the value of the thunk, we can install that expressed value in the same location, so that the thunk will not be evaluated again.

```
(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))))))
```