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

```
1  let a = 3
2  in let p = proc(x) {
3                      set x = 4 }
4  in begin
5           (p a);
6           a
7       end
```

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

Evaluating Procedure Calls

The current call-by-value implementation

```
1  eval_expr (en:env) (e:expr) :exp_val =
2   match e with
3   | App(e1,e2)      ->
4     let v1 = eval_expr en e1 in
5     let v2 = eval_expr en e2 in
6     apply_proc v1 v2
```

- Note how a new copy is placed in the store

```
1  let rec apply_proc f a =
2    match f with
3    | ProcVal (x,b,env) ->
4        let l = Store.new_ref g_store a
5        in eval_expr (extend_env env x (RefVal l)) b
6    | _ -> failwith "apply_proc: Not a procVal"
```

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`	$a \to 0$	$0 \to$ `NumVal 3`
`let p = proc(x)set x` ↪ `= 4`	$a \to 0$, $p \to 1$	$0 \to$ `NumVal 3`, $1 \to$ `ProcVal` ↪ ...
`(p a)`	$a \to 0$, $p \to 1$, $x \to 0$	$0 \to$ `NumVal 3`, $1 \to$ `ProcVal` ↪ ...
`set x = 4`	$a \to 0$, $x \to 0$	$0 \to$ `NumVal 4`, $1 \to$ `ProcVal` ↪ ...
`a`	$a \to 0$, $p \to 1$	$0 \to$ `NumVal 4`, $1 \to$ `ProcVal` ↪ ...
4		

Call-by-reference - Example 2

```
1  let p = proc(x) { 5-x }
2  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

```
1  let swap = proc (x) { proc (y) {
2                          let temp = x
3                          in begin
4                               set x = y;
5                               set y = temp
6                             end
7                          }
8              }
9          in let a = 33
10         in let b = 44
11         in begin
12              ((swap a) b);
13              a-b
14             end
```

- What is the output of this program under call-by-value and under call-by-reference?
- Execution trace on the board

Implementing Call-by-reference in IMPLICIT-REFS

- Expressed and denoted values do not change

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

- 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-proc:: exp_val -> exp_val -> expval`

```
let rec apply_proc f a =
  match f with
    | ProcVal (x,b,env) ->
        let l = Store.new_ref g_store a
        in eval_expr (extend_env env x (RefVal l)) b
    | _ -> failwith "apply_proc: Not a procVal"
```

Now: `apply-proc::exp_val -> int -> exp_val`

```
let rec apply_proc f refv =
  match f with
    | ProcVal (x,b,env) ->
          eval_expr (extend_env env x refv) b
    | _ -> failwith "apply_proc: Not a procVal"
```

Modifying the interpreter

Before: `eval_expr`: the case for function application

```
let rec eval_expr (en:env) (e:expr) :exp_val =
  match e with
  ...
  | App(e1,e2)      ->
    let v1 = eval_expr en e1 in
    let v2 = eval_expr en e2 in
    apply_proc v1 v2
```

Now: `eval_expr`: the case for function application

```
let rec  eval_expr (en:env) (e:expr) :exp_val =
  match e with
  ...
  | App(e1,e2)      ->
    let v1 = eval_expr en e1 in
    let v2 = value_of_operand en e2 in
    apply_proc v1 v2
```

Modifying the interpreter

`eval_expr`: the case for function application

```
let rec  eval_expr (en:env) (e:expr) :exp_val =
  match e with
  ...
  | App(e1,e2)      ->
    let v1 = eval_expr en e1 in
    let v2 = value_of_operand en e2 in
    apply_proc v1 v2
```

`value_of_operand:: env -> expr -> exp_val` (returns a RefVal, in fact)

```
let rec value_of_operand env e =
  match e with
    | Var id -> from_some @@ apply_env g_store env id
    | _ -> RefVal (Store.new_ref g_store (eval_expr env e))
```

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

```
type exp_val =
  | NumVal of int
  | BoolVal of bool
  | ProcVal of string*expr*env
  | RefVal of int
  | ThunkVal of expr*env
```

- 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:: env -> expr -> expval`

```
let rec value_of_operand env e =
  match e with
  | Var id -> from_some @@ apply_env g_store env id
  | _ -> RefVal (Store.new_ref g_store (eval_expr env e))
```

Now: `value_of_operand:: env -> expr -> exp_val`

```
let rec value_of_operand env e =
  match e with
  | Var id -> from_some @@ apply_env g_store env id
  | _ -> RefVal (Store.new_ref g_store (ThunkVal(e,env)))
```

Modifying the interpreter

```
let rec value_of_operand env e =
  match e with
  | Var id -> from_some @@ apply_env g_store env id
  | _ -> RefVal (Store.new_ref g_store (ThunkVal(e,env)))
```

The case for variables in `eval_expr` for call-by-name.

```
eval_expr (en:env) (e:expr) :exp_val =
  match e with
  ...
  | Var id              ->
    (match apply_env g_store en id with
    | None -> failwith @@ "Variable "^id^" undefined"
    | Some aRef -> match Store.deref g_store @@
      ↪ refVal_to_int aRef with
          | ThunkVal(e,en) -> eval_expr en e
          | ev -> ev )
```

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 (d) {
             begin
              set count = count + 1;
              count
             end }
in (proc (x) { x+x } (g 0))
```

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 id                ->
  (match apply_env g_store en id with
  | None -> failwith @@ "Variable "^id^" undefined"
  | Some aRef -> match Store.deref g_store @@ refVal_to_int
      ↪ aRef with
          | ThunkVal(e,en) -> eval_expr en e
          | ev -> ev )
```

The case for variables in `eval_expr` for call-by-need.

```
| Var id                ->
  (match apply_env g_store en id with
  | None -> failwith @@ "Variable "^id^" undefined"
  | Some aRef -> match Store.deref g_store @@ refVal_to_int
      ↪ aRef with
    | ThunkVal(e,en) -> let value_of_thunk = eval_expr en e
      in Store.set_ref g_store (refVal_to_int aRef)
          ↪ value_of_thunk;
        value_of_thunk
    | ev -> ev )
```