# Chapter 4. State
# (Essentials of Programming Languages)

Kwanghoon Choi

Software Languages and Systems Laboratory
Chonnam National University

## Outline

◇ Computational Effects

◇ EXPLICIT-REFS: A Language with Explicit References

◇ IMPLICIT-REFS: A Language with Implicit References

◇ MUTABLE-PAIRS: A Language with Mutable Pairs

◇ Parameter-Passing Variations

## 4.1 Computational Effects

A computation produces a value possibly with effects (e.g., read, print, alter the state of memory or a file system)

Here, we focus on a single effect: assignment to a location in memory
- binding vs. assignment

Store (i.e., memory): a finite map from *locations* to so called the *storable values*
- Env.: a finite map from variables to locations

Reference: a data structure that represents a location
- L-values (references), R-values (expressed values)

Two designs for a language with a store
- explicit references and implicit references

## 4.2 EXPLICIT-REFS: A Lang. with Expl. Refs.

◇ Examples in EXPLICIT-REFS

```
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*

## 4.2 EXPLICIT-REFS: A Lang. with Expl. Refs.

◇ Examples in EXPLICIT-REFS (cont.)

```
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)
```

Question. Do Exercise 4.1

## 4.2 EXPLICIT-REFS: A Lang. with Expl. Refs.

Three new operations *newref*, *deref*, and *setref* to create and use references
- In odd/even, let x = newref(0) in ..., deref(x), setref(x, 13)
- In counter, let counter = newref(0) in ..., setref(counter, -(deref(counter), -1))

Question. How do you write the three operations in your favorite languages such as C, C++, and Java?

References as a new kind of expressed values
- ExpVal = Int + Bool + Proc + Ref(Expval)
- DenVal = ExpVal

Question. Explain the behavior of "newref (newref (0))".

## 4.2.1 Store-Passing Specifications

In EXPLICIT-REFS, any expression may have an effect. To specify these effects, we need to describe what store should be used for each evaluation and how each evaluation can modify the store.

Store $\sigma$: a finite map from locations to (storable) values
- $[l = v]\sigma$ is a store $\sigma$ except that location $l$ is mapped to $v$
- A particular of value of $\sigma$ is called the $state$ of the store

A store-passing specification: Expression $exp$, evaluated in environment $\rho$ and with the store in state $\sigma_{init}$, returns the value $val$ and leaves the store in a possibly different state $\sigma$.
- (value-of $exp$ $\rho$ $\sigma_{init}$) $= (val, \sigma)$

## 4.2.1 Store-Passing Specifications (Cont.)

A store-passing specification: (value-of $exp\ \rho\ \sigma_{init}$) = ($val$, $\sigma$)

For const-exp:

(value-of (const-exp $n$) $\rho\ \sigma$) = ($n$, $\sigma$)

Note that const-exp is an effect-free operation.

For diff-exp:

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

The sequential behavior of diff-exp in the store-passing spec.

## 4.2.1 Store-Passing Specifications (Cont.)

For if-exp:

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

Note that if-exp shows a conditionally sequential behavior.

## 4.2.2 Specifying Operations on Explicit Refs

Syntax for the three new operations in EXPLICIT-REFS

$$\text{Expression} ::= \text{newref (Expression)}\ \boxed{\text{newref-exp (exp1)}}$$
$$\text{Expression} ::= \text{deref (Expression)}\ \boxed{\text{deref-exp (exp1)}}$$
$$\text{Expression} ::= \text{setref (Expression , Expression)}\ \boxed{\text{setref-exp (exp1 exp2)}}$$

The behavior of these operations:

$$\frac{(\text{value-of } exp\ \rho\ \sigma_0) = (val, \sigma_1) \quad l \notin \sigma_1}{(\text{value-of (newref-exp } exp)\ \rho\ \sigma_0) = (\text{ref-val } l, [l = val]\sigma_1)}$$

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

## 4.2.2 Specifying Ops on Explicit Refs (Cont.)

The behavior of these operations (Cont.):

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

## 4.2.3 Implementation

The implementation of the store-passing specification with the state in a single global variable

- No explicit passing and returning it

The representation of the store as a list of expressed values, and a reference is a number that denotes a position in the list

- A new ref. is allocated by appending a new value to the list
- Updating the store is modeled by replacing only the position $ref$ in the list with a new value $val$
- See Figure 4.1 and 4.2

Value-of clauses for explicit-reference operators in Figure 4.3

A trace of the effectful programs in Figure 4.4 and 4.5

## 4.3 IMPLICIT-REFS: A Lang. with Impl. Refs.

The explicit reference design gives a clear account of allocation, dereferencing, and mutation because all these operations are explicit in the programmer's code.

An alternative design with implicit references can free programmers from worrying about when to perform these operations.
  - In this design, every variable denotes a reference.

Question. In EXPLICIT-REFS, is every variable a reference?

## 4.3 IMPLICIT-REFS: A Lang. with Impl. Refs. (Cont.)

◇ Examples in IMPLICIT-REFS

```
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*

## 4.3 IMPLICIT-REFS: A Lang. with Impl. Refs. (Cont.)

◇ Examples in IMPLICIT-REFS (Cont.)

```
let g = let counter = 0 in
          proc (dummy)
            begin
             set counter = -(counter, -1);
             counter
            end
in let a = (g 11)
   in let b = (g 11)
      in -(a, b)
```

## 4.3 IMPLICIT-REFS: A Lang. with Impl. Refs. (Cont.)

In IMPLICIT-REFS, denoted values are references to locations that contain expressed values. References are no longer expressed values.

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

Locations are created with each binding operations; at each procedure call, let or letrec.

Question. Explain where new locations are created in the examples.

Syntax for the set expression in IMPLICIT-REFS based on LETREC

Expression :: = set Identifier = Expression $\boxed{\text{assign-exp (var exp)}}$

Question. Can we make chains of references as “newref (newref exp)” in EXPLICIT-REFS?

## 4.3.1 Specification

The behavior of variable and assignment:

(value-of $var\ \rho\ \sigma$) $=$ ($\sigma(\rho(var))$, $\sigma$)

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

The behavior of procedure:

(apply-procedure (procedure $var\ body\ \rho$) $val\ \sigma$

$=$ (value-of $body\ [var = l]\rho\ [l = val]\sigma$) where $l \notin dom(\sigma)$

- $l$ is a new location.

The behavior of let is similar.

## 4.3.2 Implementation

Now we are ready to modify the previous interpreter or to write a new code for var-exp, assigin-exp, let-exp, apply-procedure, and (multi-declaration) letrec.

- (together with extend-env-rec)

A trace of the effectful programs in Figure 4.8

c.f. Dynamic assignment (also called fluid binding) in Exercise 4.21

## 4.4 MUTABLE-PAIRS: A Lang. with Mutable Pairs

MUTABLE-PAIRS: Add mutable pairs to IMPLICIT-REFS.

Example:

```
let glo = pair (11,22) in
let f = proc(loc)
     let d1 = setright(loc, left(loc)) in
     let d2 = setleft(glo, 99) in
       -(left(loc), right(loc))
in (f glo)
```

Question. Describe the pair referred by glo at the end?

## 4.4 MUTABLE-PAIRS: A Lang. with Mut. Pairs (Cont.)

A mutable pair consists of two locations, each of which is independently assignable.

- ExpVal = Int + Bool + Proc + MutPair
- DenVal = Ref(ExpVal)
- MutPair = Ref(ExpVal) $\times$ Ref(ExpVal)

Interfaces for mutable pairs

- make-pair : ExpVal $\times$ ExpVal $\rightarrow$ MutPair
- left : MutPair $\rightarrow$ ExpVal
- right : MutPair $\rightarrow$ ExpVal
- setleft : MutPair $\times$ ExpVal $\rightarrow$ ExpVal
- setright : MutPair $\times$ ExpVal $\rightarrow$ ExpVal

## 4.4.1 Implementation

We can implement the interfaces for mutable pairs using the reference data type from our preceding examples.

- See Figure 4.9

The implementation of MUTABLE-PAIRS:

- Figure 4.10

A trace of the example in Figure 4.11 and 4.12

## 4.4.12 Another Representation of Mutable Pairs

The two locations in a pair are independently assignable, but they are not independently allocated. They will be allocated together: if the left part of a pair is one location, then the right part is in the next location.

Another representation of a pair by a reference to its left.
- Figure 4.13

Question. Write ExpVal, DenVal, and MutVal for this representation of mutable pairs.

## 4.5 Parameter-Passing Variations

Two ways of call-by-value parameter-passing:

- In PROC, the denoted value is the same as the expressed value of the actual parameter (See the def. of apply-procedure)
- In EXPLICIT-REFS and IMPLICIT-REFS, the denoted value is a reference to a location containing the expressed value of the actual parameter (See the def. of apply-procedure)

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

Question. What is a at the end?

Question. Modify the example to use mutable pairs in order to produce 4.

## 4.5.1 Call-By-Reference

Though the isolation between the caller and callee is generally desirable, there are times when it is valuable to allow a procedure to be passed locations with the expectation that they will be assigned by the procedure.

Call-by-reference
- ExpVal = Int + Bool + Proc
- DenVal = Ref(ExpVal)

The only thing that changes is the allocation of new locations.
- Under call-by-reference, a new location is created every evaluation of an operand other than a variable.
- Under call-by-value, a new location is created for every evaluation of an operand.

## 4.5.1 Call-By-Reference (Cont.)

Example 1:

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

Example 2:

```
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
```

## 4.5.2 Lazy Evaluation; Call-By-Name and Call-By-Need

Under call-by-name, an operand in a procedure call is not evaluated until it is needed by the procedure body. If the body never refers to the parameter, then there is no need to evaluate it.

```
letrec infinitloop (x) = inifiniteloop( -(x,1) ) in
let f = proc (z) 11 in
  (f (infiniteloop 0))
```

Unevaluated operand is represented by so called *thunk*, consisting of an expression and an environment.

Call-by-reference

- ExpVal = Int + Bool + Proc
- DenVal = Ref(ExpVal + Thunk)
- Thunk = Expression $\times$ Environment

## 4.5.2 Lazy Evaluation; Call-By-Name and Call-By-Need (Con

Lazy evaluation (or called call-by-need) = Call-by-name + sharing:

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