

Comp 411
Principles of Programming Languages
Lecture 16
Boxes as Values and Call-by-Reference

Corky Cartwright
March 1-3, 2021

Call-by-Value and Call-by-Reference

Consider this program, which contains a mutation:

```
(let [(f (lambda (x) (set! x 5)))]  
  (let [(y 10)]  
    (let [( _ (f y)) ] ; mutate y  
      y)))
```

What result is produced by evaluating this program?

The value of **y**, which is **10**, is placed in a new box (for local **x**) when **f** is applied; this new box (variable) and its contents (value) are *thrown away* after the procedure body for **f** (including the **set!**) has been evaluated, so the returned value is the value of **y**, which is still **10**. This behavior is *call-by-value*: we passed the *value* of **y**, not the containing box (variable **y**) itself. which provides the capability to change its value. (You can think of a box as object with a single field and getter and setter methods. Therefore, we cannot write a procedure that takes two variables as arguments and swaps their values. Why? A procedure cannot change variables passed as arguments because it only receives the values. It cannot obtain the names of the corresponding variables.

Supporting a swap operation

To express a swap operation as a program-defined procedure, a language must support passing the *boxes* (cells) corresponding to variables as *values* to the swap procedure.

We can support this capability with a small change to our LC interpreter based on the following observation: when the argument expression in an application is already a variable, it is associated with a box in the environment. Hence, we can pass this box to the procedure and don't need to create a new one locally:

```
((app? M)
 (apply (... fp ...)
         (if (var? (app-rand M))
             (lookup (var-name (app-rand M)) env) ; a box
             (...))))
```

This is ugly! So ugly that I almost cut this slide from the lecture. Why is it ugly?

Improving Our Ugly Design

This new mode of parameter-passing is called *call-by-reference*. Our LC formulation is *ugly* for two reasons.

- It is a syntactic hack. An expression other x that has the same meaning when evaluated as x (such as $\text{id}(x)$ where id denotes the identity function) implying the syntax of LC is not compositional!
- It does not provide a clean way to pass the *value* of a variable instead of the variable cell (box) itself. Pascal and old Fortran (66/77) support more reasonable formulations of this parameter-passing technique. But old Fortran is only slightly deeply flawed; it passes everything (including constants!) by reference. Mutating a constant in Fortran caused havoc – via mutation of shared constants. In many implementations Fortran implementors did not bother to create a new copy for each constant parameter. Call-by-reference barely escapes the non-compositional critique: the arguments in a function call with reference parameters must be evaluated differently (using *left-hand* evaluation).

Left-hand vs. Right-hand Evaluation

Algol-like languages (broadly speaking – C is included!) make a distinction between left-hand and right-hand contexts. Left-hand contexts typically include:

- the left-hand sides of assignments; and
- argument expressions passed by reference.

The basic form of evaluation is left-hand evaluation; it looks like LC (without our ugly call-by-reference extension) except that boxes (variable cells) are considered values. Hence **unbox** is not applied to the box returned by **lookup**. Right hand evaluation simply performs left-hand evaluation and then coerces boxes to values. In essence we have two mutually recursive meaning functions: *left-hand* and *right-hand*.

Variable and Data Aliasing

While passing references enables programmers to write procedures like swap, it also introduces a new phenomenon into the language: variable aliasing. Variable aliasing occurs when two syntactically distinct variables refer to the same mutable location in the environment. In Scheme and Java, such a coincidence is impossible; in Pascal and Fortran it is common.

The absence of variable aliasing in Scheme and Java does not mean that Scheme and Java escape the aliasing problem. Scheme and Java only guarantee that distinct variable names do not refer to the same location (box). Scheme and Java allow data aliasing, where more than one selection path refers to the same mutable cell. For example, in Scheme (Java), two elements of a vector (array) can be exactly the same box. All interesting programming languages permit data aliasing.

Imperative Call-by-Name

Algol 60 supports call-by-value and call-by-name, *but not call-by-reference*. In imperative languages (languages with mutable state), call-by-name has the same semantics as it does in functional languages, assuming that we equate *left-hand-evaluation* in imperative languages with evaluation in functional languages and coerce boxes to values in right-hand contexts (everywhere but the left-hand-sides of assignment and arguments passed by reference).

As a result, call-by-name is a baroque alternative to call-by-reference. A formal call-by-name parameter is typically synonymous with the corresponding argument expression.

In the underlying implementation, each argument expression passed by name is translated to a *suspension* (*thunk* in Algol 60 terminology) that yields a *box* (*location*) when it is evaluated. *Call-by-name* repeatedly evaluates the actual parameter to produce a box every time the corresponding formal parameter is referenced. If the suspension produces the same location each time, then call-by-name is equivalent to call-by-reference. But the suspension can contain references to variables that change (from assignment) during the execution of the procedure body. In the special case where an argument expression does not have box type (*e.g.*, a constant like 10), the calling program generates a dummy box and copies the value into the box.

Abusing Call-by-Name: Jensen's Device

Consider the following Algol-like code (written in C syntax) that uses assignment to change the box denoted by a call-by-name parameter.

```
procedure Sum(int x, int y, int n) { // call-by-name
// Jensen's Device: in the call on Sum, the arguments must
// be a var x and an expression M where x occurs free in M
// encoding a function f(x) = M

    int sum = 0;
    for (x = 0; x < n, x++) sum = sum + y;
    return sum;
}

int j, sum = 0;
sum = Sum(j, j*j, 10); // compute the sum 0*0 + 1*1 + ... 9*9
```


Why Jensen's Device Has Become Obscure

The ugly convention of passing x and $x*x$ by name and using modifications to the formal parameter for x to determine different values for the formal parameter corresponding to $x*x$ is called Jensen's device. Parameter passing has become so complex that simple reasoning about variables is no longer possible.

Imperative call-by-name is deservedly dead but perhaps for the wrong reason. It has come back in Scala for the right reason: when programming in a functional style, deferring argument evaluation may be important, albeit subtle. Consider the Y operator.

In the imperative world, the call-by-need optimization of call-by-name does not work because re-evaluations of the suspension for a call-by-name parameter does not necessarily produce the same result!

Call by Value-Result

Call-by-reference has a clean semantic definition but some programming methodologists have shunned it because of variable aliasing. In its place, they have proposed *call-by-value-result*. In the context of distributed computing, this mechanism makes sense for efficiency reasons (assuming the costs of copying arrays can be amortized), but its semantics is ugly from the perspective of program reasoning.

When an actual parameter is passed by *value-result*, the calling procedure left-hand-evaluates the actual parameter exactly as it would for call-by-reference. It passes the address of the box to the called procedure which saves it, creates a new local variable (a box) for the corresponding formal parameter and copies the contents of the passed box into the local box.

During the execution of the procedure body, the local copy is used whenever the formal parameter is accessed.

On exit from the called procedure, the called procedure copies the contents of the local box into the corresponding actual parameter box. In essence, call-by-value-result creates a temporary copy of the actual parameter box and copies the contents of this copy into the actual parameter box on exit.

Value-result is sometimes called *copy-in/copy-out* or *in/out*, particularly in the context of languages for distributed computing.

Call by Result

- Given the availability of *call-by-value-result* (*copy-in, copy-out*) which can be viewed as an enhancement of *call-by-value* (*copy-in*), it makes sense to consider *call-by-result* (*copy-out*) in isolation. This mechanism is actually more useful in conventional languages than *call-by-value-result* (which IMO is inferior to *call-by-reference* except in context of distributed computing). In many situations, it is natural to define a function/method that returns multiple values. Scheme has an explicit syntax (not covered in Comp 210/211) for doing this. But Scheme has an unusual syntax that makes inclusion of such a convention relatively easy. In languages with more conventional syntax, a common way to return multiple results is to return the primary result normally and the other (auxiliary) results using *call-by-result*.
- **Example:** a lookup function on environments that returns the matching binding as **Binding lookup(value Env e, value Symbol s, result JamVal val)** and the value in a result parameter if it is available without additional evaluation. In Java, the **Env** argument **e** would probably be the receiver rather than an explicit argument. In principle, Java could support *call-by-result* (and *call-by-reference*.)

Call-by-Reference vs. Boxes as Values

- In call-by-reference, boxes are not “first-class” values because they can only be used in limited (left-hand) contexts.
 - Everywhere else they are coerced to their contents (right-hand evaluation).
 - It is typically impossible to store a box inside a box (C pointers are an exception because of weak typing rules); in fact, boxes generally cannot be elements of composite data structures in languages where boxes are not first-class values.
- If boxes are first class, then boxes can be passed by value! In this case, call-by-reference is a superfluous. The price of “boxes as first-class values” is that they must be dereferenced to obtain the contents (as in ML). In C, boxes are first-class (represented by pointers). C also performs implicit dereferencing depending on context but pointer variables are not implicitly dereferenced). C also provides a prefix operators for forcing the dereferencing of pointers (*) and suppressing the dereferencing of assignable variables (&).