Comp 411
Principles of Programming Languages
Lecture 15
Boxes as Values and Call-by-Reference

Corky Cartwright

October 8, 2012

Call-by-Value and Call-by-Reference

Consider the following program, which is illegal in Scala because **var** parameters are forbidden as method parameters. The corresponding code in Scheme and Java is legal.

```
def main(): Int = {
  var x: Int = 5
  var y: Int = 10
  swap(x, y)
  println(y)
}
def swap(var x: Int, y: Int) {
  val temp: Int = y
  y = x
  x = temp
}
```

What result is printed by evaluating such a program?

Since **x** and **y** are **var** identifiers, **main** program creates local boxes for **x** and **y**, and initializes the contents of these boxes to the **Int** values **5** and **10**, respectively. Note that these boxes are not accessible as data values in the program. The main program can only pass the *values* in these boxes to the procedure **swap**, which creates local boxes **x** and **y** to hold the passed values. The **swap** procedure exchanges the contents of the two local boxes and returns control to **main**. Then the **main** procedure calls **println** to print the contents of box **y**. What value does it print? **10**, because the boxes for **x** and **y** are only accessible via assignments in **main**.

Extending The Assignable Variables Model

In the Assignable Variables model of imperative programming, the implicit boxes in the environment holding the values of variables are not available as data values. As a result, it is impossible to write a `swap` procedure (in the absence of multiple assignment). Consequently, many languages with assignable variables include an extra form of parameter-passing is called *call-by-reference*.

Pascal and Fortran (66/77) are languages that supplement assignable variables with call-by-reference. In fact, Fortran passes everything (including constants!) by reference. Mutating a constant can cause havoc depending on the particular Fortran 66/77 implementation because mutation of ref parameters bound to (boxes containing) constants may change the value of contants used elsewhere in the program!

Another way to extend the Assignable Variables model is too add an explicit "address-of" operator, but it is very difficult to formulate a sound static, type system for such a language. The C type system, for example, is unsound.

Left-hand Evaluation

Algol-like languages (including C) 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 are considered values. Hence **`unbox`** is not applied to the box returned by the environment.

Right hand evaluation performs left-hand evaluation and then coerces boxes to values.

Variable and Data Aliasing

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

The absence of variable aliasing in Scheme does not mean that Scheme escapes the aliasing problem. Scheme only guarantees that distinct variable names do not refer to the same location (box). Scheme allows data aliasing, where more than selection path refers to the same location. For example, two elements of a vector 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 reference parameter is typically synonymous with the corresponding argument expression.

In the underlying implementation, each argument expression passed by reference is translated to a *suspension* (*thunk*) that yields a *box* (*reference*, *location*) when it is evaluated. In essence, *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) {
  // actual x must occur free in actual y
  int sum = 0;
  for (x = 0; x < n, x++) sum = sum + y;
  return sum;
}

int j, sum = 0;
int[10] a;
for (int i = 0; i < 10; i++) a[i] = i; // initialize a
sum = Sum(j, a[j], 10));       // compute the sum
```

Why Jensen's Device Has Become Obscure

The ugly convention of passing `j` and `a[j]` by name and using modifications to the formal parameter for `j` to determine different values for the formal parameter corresponding to `a[j]` is called *Jensen's device*. When cal-by-name arguments have side effects, parameter passing becomes so complex that simple reasoning about variables is no longer possible.

Imperative call-by-name is deservedly dead, except in Scala. But in Scala, combining visible side-effects and call-by-name is considered obscene.

In the imperative world, the call-by-need optimization of call-by-name does not work in general because reevaluations 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.*

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.

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 where shared memory may be unavailable or very expensive). 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. Scala has built-in tuple types to support this feature.

In languages with more conventional syntax and lighter weight constructs, a good 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 index as well as the matching value

lookup(e: Env, s: Symbol, result index: Int): JamVal

In C/C++, call-by-result can easily be simulated by passing the address of the "result" parameter by value and then assigning to the dereferenced formal parameter just before returning from the procedure.

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). Moreover, it is typically impossible to store a box inside a box (C pointers are an exception).

If boxes are first class, then boxes can be passed by value! Call-by-reference is superfluous complication. This approach is used in ML and imperative Jam. In such languages, boxes (refs) must be explicitly dereferenced because boxes are legitimate values.

In C/C++, boxes are automatically dereferenced in right-hand contexts but the & operator suppresses this operation (by defining a local left-hand context). I suspect that this convention is the source of troublesome bugs.