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

Corky Cartwright

February 16, 2022

Boxes as Values in Languages like ML

This approach is so simple and elegant that not much needs to be said. We only need to add the type `ref` including a setter (which is *not* available for ordinary functional data types) and a getter (`!` in Jam) for its only field.

The only disadvantage to this approach (which can be significant) is that every occurrence of a mutable variable denoting its contents must be explicitly dereferenced. In a typed language, this is typically easy to do since the absence of dereferencing almost always generates a type error. In addition, this annoyance discourages the use of mutable variables which is good IMO. In a dynamically typed language like Jam, these errors (failure to dereference boxes) are not detected until testing.

The remainder of this lecture focused on the complications involved in the assignable variables approach.

Call-by-Value and Call-by-Reference

Consider the following Scheme/Racket program, which contains a mutation:

```
(let [(f (lambda (x) (set! x 5)))]
  (let [(y 10)]
    (let [(_ (f y))]  ; mutate y [the value bound to x in f]
      y)))
```

What result is produced by evaluating this program? **`10`**

When **`f`** is applied (called), the value of **`y`**, which is **`10`**, is placed in a new box (for local **`x`**). Like all local variables, this new box (variable) and its contents (value) are *thrown away* after the procedure body for **`f`** (including the **`set!`**) has been evaluated. The application of "procedure" **`f`** returns the special value **`(void)`** which is the value returned by **`(set! x 5)`**. This special value is bound to the variable **`_`** immediately after **`f`** returns. (By convention, the name **`_`** is used in Scheme/Racket for bindings that are ignored in subsequent program text.) The innermost **`let`** returns the value of **`y`**, which is still **`10`**. Since this innermost **`let`** is the body of the enclosing (second-level **`let`**, which the body of the outermost, **`let`** the program returns the value **`10`**.

This behavior is based on the *call-by-value* semantics of **`lambda`** and **`let`** in Scheme/Racket. In the call on **`f`**, Scheme/Racket passes the *value* of **`y`** (not the containing box which is the variable **`y`**) rather than the variable **`y`** introduced by the second-level **`let`**. Hence, the body of **`f`** has no access to the variable **`y`** introduced by the second-level **`let`**; it only mutates its parameter (the local variable **`x`** created when **`f`** is called), leaving **`y`** bound to the value **`10`** after it returns **`(void)`**, which is bound to the variable **`_`**. To pass a variable like **`y`** to the "procedure" **`f`**, we need a different parameter passing mechanism than *call-by-value*. In a language like Scheme/Racket with assignable variables, *call-by-need* and *call-by-need* may suffice in some situations, but their semantics is so ugly (since mutation is possible) that most attempted uses (like writing **`swap`** using call-by-name) fail in some cases. We need a semantically tractable alternative to call-by-name/call-by-need that binds **`x`** to the variable **`y`** (a box [cell]) rather than the deferred left-hand/right-hand evaluation of **`y`**. The most common mechanism for this purpose in mainstream languages with assignable variables like Java is *call-by-reference*. Note that Java does not support call-by-reference while C++ and C# both support it. We will discuss this issue later.

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* 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

We need to introduce a new mode of parameter-passing is called *call-by-reference*. Our LC formulation is ***ugly*** for two reasons.

- It is a syntactic hack. Any expression other than an explicit variable `x` that has the same meaning when evaluated normally (such as `id(x)` where `id` denotes the identity function) has different semantics than `x` in this particular context, 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. Passing `id(x)` "works" because the argument expression (AST) is no longer a variable. Pascal and old Fortran (66/77) support more reasonable formulations of this parameter-passing technique. Old Fortran is still deeply flawed; it passes everything (including constants!) by reference. Mutating a constant in a Fortran subroutine (procedure) caused havoc – via mutation of shared constants. In many implementations, the Fortran compiler implementors did not bother to create a new copy for each constant parameter. So mutating constants passed as parameters mutated the shared cell!

Contemporary languages with assignable variables (such as Pascal, C, C#, Swift) escape the non-compositional critique by introducing two different forms of evaluation: the arguments in function calls corresponding to reference parameters must be evaluated differently (using *left-hand* evaluation) than other argument expressions which are right-hand evaluated. Mainstream languages that support call-by-reference force procedures (including explicit lambda-abstractions) to explicit declare reference parameters because they evaluate the corresponding argument expressions differently!

Left-hand vs. Right-hand Evaluation

Mainstream languages that support assignable variables and 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 (when it is supported).

The basic form of evaluation is left-hand evaluation; it essentially looks like our LC interpreter (without our ugly call-by-reference extension) except that boxes (variable cells) are considered values. Hence, in left-hand evalation, **`unbox`** is not applied to the box returned by the **`lookup`** operation of LC. Right hand evaluation simply performs left-hand evaluation followed by coercing boxes to values. In essence we have two mutually recursive meaning functions: *left-hand-eval* and *right-hand-eval*. Although left-hand evaluation is conceptually the primary form of evaluation (since right-hand evaluation is trivially implemented on top of left-hand evaluation), programming languages with assignable variables (including Java) stipulate that most contexts require right-hand evaluation which conditionally converts values that are boxes to their contents. In such languages, values that are not boxes are unchanged by right-hand evaluation, which merely strips the boxes wrapping values. Such languages prevent the direct nesting of boxes inside boxes so multiple stripping is not required. To understand the concept of left- and right-hand evaluation, consider a trivial assignment statement in C:

```
x = x+1;
```

The occurrence of **`x`** on the left-hand side of the assignment operator **`=`** is left-hand evaluated because left-hand contexts require a box. In this context, **`x`** simply evaluates to the box corresponding to **`x`**, which in compiled code is simply an address. In contrast, the occurrence of **`x`** on the right-hand side of **`=`** is right-hand evaluated which returns the contents of the box corresponding to **`x`**.

Languages with Assignable Variables without Call-by-reference

Some languages with assignable variables do not explicitly support call-by-reference because either:

- they have a very-low level semantics that eschews type-safety like C; or
- they treat the boxes (cells) corresponding to variables as local artifacts that cannot be passed as arguments to methods/procedures like Java, Scala, Kotlin (?).

Low-level languages like C directly provide an operator (unary prefix & in C) which simply returns the address of its argument (which must have an intelligible left-hand meaning), so the address corresponding to any variable (or other assignable expression such as an array element) can be passed as an argument. The corresponding procedure parameter must be a pointer type. This perspective is incompatible with defining a semantics that is more abstract than a generic machine implementation. It does not make sense for any language that I would call "high-level" which by definition do not have a semantics based on the concept of machine addresses. In particular, machine addresses do not appear in the semantic definitions of Java, C#, Scala, or Kotlin.

In higher level languages with assignable variables, passing local variables by reference must be restricted if the corresponding boxes are allocated on the call stack, an issue that we will discuss in detail when we discuss the Algol 60 runtime (an efficient framework for implementing environments in compiled code) and its descendants (which includes the runtimes of essentially all contemporary languages). Why must the usage of reference parameters be restricted? Because stack-allocated boxes for variables are deallocated when the execution of the procedure/method introducing the variable exits (returns). A passed reference to such a box can potentially become a "dangling pointer" (a pointer to an object that no longer exists, which can easily happen in C code).

In Java, there is a simple reason why you cannot pass the box corresponding to a local variable directly as a method parameter. If you look at architecture of the JVM, local variables are stored in slots in the activation record (stack frame) for the method where the variable is introduced. The JVM only supports direct access to slots in the current activation record, so there is no way to directly pass the box for a local variable as an argument because it is not accessible from the called methods (which have their own activation records). The same restriction by the way exists for object fields; you cannot pass them directly as parameters (using say their machine addresses). There are various implementation hacks to work around this limitation, but they have significant costs. Wait until we discuss language runtimes.

Variable and Data Aliasing

While passing references enables programmers to write procedures like `swap`, it also introduces a new phenomenon into the language called *variable aliasing*. Variable aliasing occurs when two syntactically distinct variables refer to the same mutable location in the environment. In Scheme and Java (where call-by-reference is not supported), 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 completely 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 (object). All interesting programming languages permit data aliasing. Both variable and data aliasing can devastating semantic consequences. Two apparently disjoint references to data may be synonymous. Variable aliasing is particularly insidious because we naturally assume distinct simple variables are disjoint rather than synonymous.

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, inefficient, and fragile alternative to call-by-reference. A 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 (by executing assignment operations) 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 mutations of the formal parameter for **x** to determine different values for the formal parameter corresponding to **x*x** is called Jensen's device. In this idiom, the semantics of parameter passing is so complex that simple reasoning about variables is no longer possible. Believe it or not, the popular buzz regarding this particular idiom was positive when it debuted. (How clever!) But software developers eventually learned that software written using this device was slow and fraught with potential bugs. Beware of the opinions of crowds.

Imperative call-by-name is deservedly dead but perhaps for the wrong reason (execution cost rather than convoluted semantics?). 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 do not necessarily produce the same result! Hence, imperative call-by-name combines convoluted semantics with horrendous inefficiency.

More Pathologies of Imperative Call-by-name

In the preceding example illustrating Jensen's device, both `x` and `y` have the same type `int`. But they actually have different types: `x` is bound to a thunk (suspension) that evaluates to a box that s coerced to its contents (of type `int`) in right-hand contexts while `y` is bound to a thunk that returns an `int` value rather than a box. Using `y` on the left-hand side of an assignment must generate an error! There is no box corresponding to the argument `j*j`! The thunk bound to `y` does not evaluate to a box. In the world of assignable variables, call-by-name is a huge conceptual fail.

As a result of these pathologies, call-by-name is now completely discredited and misunderstood. As we have already learned, call-by-name in a purely functional (no mutation) context is simpler in some respects than call-by-value. Termination analyses and debugging are more subtle (and hence more complex) for call-by-name than for call-by-value, but that is the price of greater expressiveness. Look at the difference between the call-by-name Y operator and call-by-value Y operator. Which is simpler? Call-by-name wins here because computing fixed-points is non-trivial and the subtle (but mathematically rigorous) behavior of call-by-name incremental evaluation can be exploited in this domain. Call-by-value can only clumsily mimic the incremental behavior of call-by-name at the cost of significantly more complex code that manually inserts eta-conversion where evaluation must be suspended. On the other hand, I don't know of a simple logic for reasoning about functional call-by-name programs. The current research buzz in this regard is to use co-induction from the realm of category theory which I find opaque. I prefer a simple first-order logical system incorporating the notion of *admissibility*. That is an interesting topic for future research.

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 data structures can be amortized), but its semantics is actually quite 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 (which was saved). 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 (rarely covered in introductory courses) 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) but implementations using the JVM might be clumsy.

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 (`&`). The ugly aspect of these features in C is the lack of type safety and automatic storage management (which can invalidate heap pointers).