Different Approaches to Argument Passing (Rough Draft)

Cameron Swords

February 6, 2014

cswords@indiana.edu

1 Introduction

This report provides an in-depth look at implementing different argument-passing semantics in the Racket C311 interpreter (written with `pmatch`), including call-by-value, call-by-reference, call-by-name, and call-by-need. It is, in many ways, an homage to the successes of some languages and the failures of others.

2 Boxes

Before the lecture notes, we need to discuss boxes in Scheme. These boxes work an awful lot like pointers in C, but, thankfully, they are far clearer in usage and much less likely to blow up in your face. (This is not to say that they will not blow up in your face; the capacity for explosion is certainly still there - as we may soon demonstrate.) While incomplete (and certainly not as efficient) as Scheme's own boxes, we can certainly implement a set of functions that *exhibit near-identical behavior* to boxes.

```
(define box
  (lambda (v)
    `(box . ,v)))

(define unbox
  (lambda (b)
    (cdr b)))

(define set-box!
  (lambda (box val)
    (set-cdr! box val)))
```

It may be helpful to keep this picture in your head while reading over the next several sections. Note that here, a box is a tagged list with the variable as the `cdr` of said list. Then unboxing something simply retrieves this `cdr`, and `set-box!` uses Racket's built-in `set-cdr!` to statefully change the list.

(Note that Racket has its own built-in boxes; you need not use the definitions above in your own code. They are simply here to help solidify the idea of how to use boxes.)

3 Side Effects

Unfortunately, implementing side effects is not nearly as difficult or craft as the side effects themselves may be. Dan Friedman has opposed them since 1968 - they have been around, and a bad idea, for a long time. To implement such a feature, we will explore the concept of 'state' - we need a stateful piece of code so that we can run side effects on it. We will do this by adding boxes to our interpreter. First, the vanilla interpreter:

```
(define val-of
  (lambda (exp env)
    (pmatch exp
      [`,x (guard (symbol? x)) (env x)]
      [`(lambda (,x) ,body)
        (lambda (a) (val-of body (lambda (y) (if (eq? x y) a (env y)))))]
      [`(,rator ,x) (guard (symbol? x)) ((val-of rator env) (env x))]
      [`(,rator ,rand) (guard (not (symbol? rand)))
        ((val-of rator env) (val-of rand env))])))
```

Notice the optimization we have introduce to the `rator/rand` line: if the rand is a symbol, it will simply get looked up in the environment on the recursive val-of call so we do it at the time we find it is a symbol, removing a single recursive step. Now, to add state, we will use Racket's `boxes` to rewrite our interpreter using these boxes:

```
(define val-of-boxes
  (lambda (exp env)
    (pmatch exp
      [`,x (guard (symbol? x)) (unbox (env x))]
      [`(lambda (,x) ,body)
        (lambda (a) (val-of-boxes body (lambda (y) (if (eq? x y) a (env y)))))]
      [`(,rator ,x) (guard (symbol? x)) ((val-of-boxes rator env) (box (unbox (env
        x))))]
      [`(,rator ,rand) (guard (not (symbol? rand)))
        ((val-of-boxes rator env) (box (val-of-boxes rand env)))])))
```

This interpreter works exactly like our previous interpreter except that the enviornment is now associating variables with *boxed* values instead of standard values. Note how we unbox the value the environment returns and box any value before application occurs (so that the lambda gets a boxed value). Similarly, to remove the scary prospect of passing around the *same box* - the same reference to memory - as the one in the environment, on symbol look-up in the `rator/x` line, we unbox and rebox the value, thus aquiring a new box.

Now adding side-effects becomes straight-forward; we need add only one line to the interpreter.

```
[(set! ,x ,rhs)
  (let ([vrhs (val-of rhs env)])
       (set-box! (env x) vrhs))]
```

Now we have introduced `set!` into our language. Note, however, that we don't have `set-box!` - that could be *bad*. These boxes, however, give us even more power than side effects: they let us change the calling convention used for evaluation in our langauge.

4 Call By Reference

Until now, all of our interpreters have used call-by-value semantics: arguments are evaluated and the resulting value is passed into the function. We will now change our interpreter to use *call-by-reference* semantics—instead of passing a value around, we will pass in a *reference* to the value. Consider the following expression:

```
((lambda (x)
  ((lambda (y)
    (begin2
      (set! y 2)
      x))
  x))
5)
```

In the standard call-by-value semantics, this would return `5`. However, given the call-by-reference semantics, `y` will be bound to the same reference in memory as the `x` was and so when one is changed, the other will also change. In call-by-reference semantics, a function receives an implicit reference to a variable used as argument, rather than a copy of its value. We can implement this evaluation style by doing exactly this - we will pass a reference to a variable in any time we call a function with a variable for input. This means our `rator/x` line will see the only change:

```
(define val-of-cbr
  (lambda (exp env)
    (pmatch exp
      [`,x (guard (symbol? x)) (unbox (env x))]
      [`(lambda (,x) ,body)
        (lambda (a) (val-of-cbr body (lambda (y) (if (eq? x y) a (env y)))))]
      [`(set! ,x ,rhs) (guard (symbol? x)) (set-box! (env x) (val-of-cbr rhs env))]
      [`(,rator ,x) (guard (symbol? x)) ((val-of-cbr rator env) (env x))]
      [`(,rator ,rand) (guard (not (symbol? rand)))
        ((val-of-cbr rator env) (box (val-of-cbr rand env)))])))
```

Now our intepreter will yield a `2` on the input above (assuming we add begin2, a minor task). Before moving on, for the sake of simplicity we will rewrite this interpreter using representation-independent environments. We do this to clean up the code and to free ourselves from dealing with the representation of the environment itself.

```
(define val-of-cbr
  (lambda (exp env)
    (pmatch exp
      [`,x (guard (symbol? x)) (unbox (apply-env env x))]
      [`(lambda (,x) ,body) (lambda (a) (val-of-cbr body (extend-env x a env)))]
      [`(set! ,x ,rhs) (guard (symbol? x))
        (set-box! (apply-env env x) (val-of-cbr rhs env))]
      [`(,rator ,x) (guard (symbol? x)) ((val-of-cbr rator env) (apply-env env x))]
      [`(,rator ,rand) (guard (not (symbol? rand)))
        ((val-of-cbr rator env) (box (val-of-cbr rand env)))])))
```

Now, onward to the other calling conventions!

5 Lazy Evaluation

For our next step, we can dispense with the concept of side effects - our next two semantics shifts deal with *when* things are evaluated. Consider the following:

```
((lambda (f)
   (if (zero? 0)
     5
     f))
 ((lambda (x) (x x)) (lambda (x) (x x))))
```

If we call this function using Scheme's call-by-value semantics, we would expect an infinite loop - it will attempt to execute omega and never come back. However, in this example a valid answer is `5` - omega will never be called, and so safely returning `5` would be a nice behavior to have. We will accomplish this by filling our boxes with a more complex expression. In our previous two interpreters, the boxes were simply filled with values. This time, we will suspend the entire computation to be performed until its result is asked for. And we can accomplish this using *thunks*.

6 Thunks

A *thunk* is a function that takes no arguments. So far we have used thunks to represent the empty environment:

```
> (define empty-env
    (lambda ()
      (lambda (y) (errorf who "invalid variable ~s" y))))
> (empty-env)
#<procedure>
> (define empty-env
    (lambda ()
      '()))
> (empty-env)
()
```

So a thunk will evaluate and return whatever expression is in the body. For example:

```
> (define five
    (lambda ()
      (+ 2 3)))
> (five)
5
```

Note here that the computation is not performed at definition, but at invocation. A more extreme example might go something like this:

```
> (define omega
    (lambda ()
      ((lambda (x) (x x)) (lambda (x) (x x)))))
> (omega)
;; infinite loop
```

This demonstrates the power of a thunk. The entire computation is wrapped inside of a lambda of no arguments and left unexecuted until invocation - we have *shunted* the work. This is a large hint toward how we will perform a similar task inside of our interpreter - we will put off evaluation using thunks.

7 Call-By-Name

In call-by-name semantics, the evaluation itself is stored and repeated every time the value is asked for. We will accomplish this by placing the evaluation to be performed *inside of a thunk* inside of a box in our environment. Here is an association list environment that associates `x` with `2`:

$((x.\boxed{(\lambda()(\text{value-of } 2\ '()))}))$

Notice that the call to `value-of` (with an empty environment) is sitting inside of the thunk—when the thunk is applied, the call is issued and returns 2. But this computation is not performed until the value of x is looked up in the environment. We add this behavior to our interpreter by modifying the `symbol` and `rator/rand` lines:

```
(define val-of-cbname
  (lambda (exp env)
    (pmatch exp
      [`,x (guard (symbol? x)) ((unbox (apply-env env x)))]
      [`(lambda (,x) ,body) (lambda (a) (val-of-cbname body (extend-env x a env)))]
      [`(,rator ,x) (guard (symbol? x))
         ((val-of-cbname rator env) (apply-env env x))]
      [`(,rator ,rand) (guard (not (symbol? rand)))
         ((val-of-cbname rator env) (box (lambda () (val-of-cbname rand env))))])))
```

The two major changes are clear: the unboxed thunk is being applied in the `symbol` line, causing evaluation of its body (and yielding a value) and we have added a lambda to our `rator/rand` line to package up the `rator`'s argument appropriately. And thus we have implemented call-by-name semantics.

8 Call-By-Need

The final calling convention we will discuss is known as *call-by-need* - it is similar to call-by-name but differs significantly. In call-by-name, the computation is repeated every time the variable is looked up. In call-by-need, however, the computation is performed the *first time* the variable is used and the result is then stored back in its place. We will implement this by defining a helper that will unbox our expression and deal with evaluation and storage all at once:

```
(define unbox/need
  (lambda (b)
    (let ([val ((unbox b))]) ;; (unbox b) returns a thunk
      (set-box! b (lambda () val)) ;; update the box to hold val
      val)))
```

Here we perform the unboxing and evaluation of the boxed thunk. We then take the result of this evaluation, wrap it up in its own thunk, and place our new thunk inside the box. (Note that this approach applies the thunk, re-wraps it, and places a new thunk in the box each time. That said, after the first computation each subsequent call should be a thunk containing a single value, which is *almost* free in Racket.) Now we need only call this in the symbol line of our interpreter and we are done:

```
(define val-of-cbneed
  (lambda (exp env)
    (pmatch exp
      [`,x (guard (symbol? x)) (unbox/need (env x))]
      [`(lambda (,x) ,body)
        (lambda (a) (val-of-cbneed body (extend-env x a env))]
      [`(,rator ,x) (guard (symbol? x)) ((val-of-cbneed rator env) (apply-env env
        x))]
      [`(,rator ,rand) (guard (not (symbol? rand)))
        ((val-of-cbneed rator env) (box (lambda () (val-of-cbneed rand env))))])))
```

Note that the application parentheses in the `symbol` line also disappear as we are performing this invocation in the helper function instead. As a parting thought, consider how a function that returns random numbers would behave over multiple calls using call-by-need semantics.

9 Conclusion

In conclusion, we have covered four calling conventions and how to implement each:

- **Call-By-Value**: Scheme's standard calling convention; arguments are evaluated and then passed into a function.
- **Call-By-Reference**: Arguments are evaluated and a reference to the result is passed into a function. This allows multiple variables to point to, reference, and alter the same piece of memory.
- **Call-By-Name**: Evaluation of any argument is put off until the argument is looked up in the environment. This convention evaluates the argument every time it is used.
- **Call-By-Need**: Evaluation of arguments is put off until the argument is looked up. After the first evaluation, the result is stored back and used in all future requests.

For further reading and details on the topic, look into Dan Friedman's "CONS should not Evaluate its Arguments" (a.k.a. "CONS the Magnificent") and Haskell, a functional programming language that uses lazy (call-by-need) evaluation.