CSE 341, Spring 2011, Lecture 21 Summary

Standard Disclaimer: These comments may prove useful, but certainly are not a complete summary of all the important stuff we did in class. They may make little sense if you missed class, but hopefully will help you organize and process what you have learned.

The purpose of this lecture is to consider the semantics of object-oriented language constructs, particularly method calls, as carefully as we have considered the semantics of functional language constructs. As we will see, the key distinguishing feature is how `self` (Java's `this`) is bound to the environment when a method is called. We will also see how to encode this concept as a programming *idiom* in Scheme, even though Scheme does not have object-oriented *semantics*.

Before discussing `self` in particular, let's consider more generally the semantics of *resolving* (i.e., "looking up") various things like fields, variables, method names, etc. Such "name resolution" is a key part of a language's semantics. For example, the lexical scope of ML and Scheme is essential to understanding functional programming.

In Ruby, unlike Java, deciding whether some name like `f` refers to a field or a local variable is trivial because fields always start with a `@` and variables never do. (Relatedly, class fields always start with `@@`.) There is a general principle here: rules about shadowing and name collisions are trivially avoided when there is a syntactic distinction, as there is here between fields and variables. In Java, we need a separate rule that local variables shadow fields, though that is certainly not a big deal. Interestingly, Ruby's choice to let you omit `()` when calling a zero-argument method creates a different potential collision: Is `m+2` adding 2 to the local variable `m` or is it adding 2 to the result of calling `self.m()`? It turns out local variables shadow methods, but since variables aren't declared, the actual answer depends on whether the method has already assigned to a variable `m` or not. This is a Ruby detail that comes up rarely, but it does point out the subtle "gotchas" that can arise when your syntax is ambiguous.

To continue defining Ruby's lookup rules, inside a block we look up the variable from the environment where the block was defined. Nested blocks can shadow variables from outer blocks, and if a variable is not defined in the block (or an outer block), it can also be defined in the enclosing method. In other words, blocks are like closures in how variables are looked up, and the implementation of Ruby needs to build closures like you did when implementing your interpreter for a functional language.

However, Ruby blocks are not quite as directly useful as ML/Scheme closures because they are not *first-class*. A language construct is *first-class* if, by definition, it can be the result of an arbitrary computation, passed as an argument to a function, stored in another data structure (e.g., an object field), etc. Things that are values are first-class. In Ruby, the values are objects. Here are some things that are not first-class:

- Method names: You cannot write something like `x.(if b then m else n end)` which is trying to return the name of a method from an if-expression. Method names are not first-class; this is not a Ruby program. You have to write something like `if b then x.m else x.n end`.
- Argument lists: Suppose you have `f(1,2,3) + g(1,2,3)`. You cannot rewrite that as `x = (1,2,3); f x + g x`.
- Blocks: You cannot, for example, store a block in a field of an object. That is exactly why the class `Proc` exists. Objects are first-class, and of course instances of `Proc` are objects.

The notion of what is first-class is important in any programming language.

The remaining thing to define for Ruby is how method calls are resolved. That is, given a call, how do we determine which code is executed for the call. Note that in OO, we also sometimes refer to method calls as "message sends", which fits well with the notion of objects. Given `obj.m 2`, we can talk about "calling `obj`'s `m` method with argument 2" or "sending `obj` the message `m` with argument 2." These are synonymous phrases. In class-based object-oriented languages like Ruby and Java, the rule for evaluating `e0.m(e1,...,en)` is:

- Evaluate `e0`, `e1`, ..., `en` to values, i.e., objects `obj0`, `obj1`, ..., `objn`.

- Get the class of `obj0`. Every object "knows its class" at run-time. Think of the class as a special field of `obj0`.
- Suppose `obj0` has class `A`. If `m` is defined in `A`, call that method. Otherwise recur with the superclass of `A` to see if it defines `m`. Raise a "message not understood" error if neither `A` nor any of its superclasses defines `m` (with the right number of arguments).
- (Ruby's mixins complicate the lookup rules a bit more. See the next lecture.)

Now that we have defined what method to call, we still have to define what environment to use when evaluating its body. If the method has *formal arguments* (i.e., argument names or parameters) `x1`, `x2`, ..., `xn`, then the environment for evaluating the body will map `x1` to `obj1`, `x2` to `obj2`, etc. But there is one more thing that is the essence of object-oriented programming and has no real analogue in functional programming: We always have `self` in the environment. **While evaluating the method body, `self` is bound to `obj0`, the object that is the "receiver" of the message.**

This last phrase is what is meant by the synonyms "late-binding," "dynamic dispatch", and "virtual function calls". It is central to Java and Ruby's semantics. It means that when the body of `m` calls a method on `self` (e.g., `self.someMethod(34)` or just `someMethod(34)`), we use the class of `obj0` to resolve `someMethod`, *not* (necessarily) the class where `someMethod` is defined.

This semantics is:

- Exactly why our example from last lecture "worked" when we defined a `PolarPoint` class and did not have to override `distFromOrigin`. The `distFromOrigin` in the superclass made calls to `self.x` and `self.y`, so when `self` was bound to a `PolarPoint`, we used the definitions of `x` and `y` in `PolarPoint`.
- "Old news" to you because you learned it in your introductory Java courses. Now we can just give it a more precise definition because we understand that it is really about what `self` is bound to in the environment.
- Quite a bit more complicated than ML/Scheme function calls. It may not seem that way to you because you learned it first. But it took many weeks in your introductory course before it could be explained.

To understand how dynamic dispatch differs from function calls, consider this simple ML code that defines two mutually recursive functions:

```
fun even x = if x=0 then true else odd (x-1)
and odd  x = if x=0 then false else even (x-1)
```

This creates two closures that both have the other closure in their environment. If we later shadow the `even` closure with something else, e.g.,

```
fun even x = false
```

that will not change how `odd` behaves. When `odd` looks up `env` in the environment where `odd` was defined, it will get the first line above. That is "good" for understanding how odd works *just from looking at where odd is defined*. On the other hand, suppose we wrote a better version of `even` like:

```
fun even x = (x mod 2) = 0
```

Now our `odd` is not "benefitting from" this optimized implementation.

In OO programming, we can use (abuse?) subclassing, overriding, and dynamic dispatch to change the behavior of `odd` by overrding `even`:

```
class A
  def even x
     if x=0 then true else odd(x-1) end
  end
```

```
  def odd x
     if x=0 then false else even(x-1) end
  end
end
class B < A
  def even x # changes B's odd too!
     x % 2 = 0
  end
end
```

Now `(B.new.odd 17)` will execute faster because `odd`'s call to `even` will resove to the method in `B` – all because of what `self` is bound to in the environment. While this is certainly convenient in the short example above, it has real drawbacks. We cannot look at one class (`A`) and know how calls to the code there will behave. In a subclass, what if someone overrode `even` and did not know that it would change the behavior of `odd`? Basically, any calls to methods that might be overridden need to be thought about very carefully. It is likely often better to have private methods that cannot be overridden to avoid problems. Yet overriding and dynamic dispatch is the biggest thing that distinguishes object-oriented programming from functional programming.

Let's now consider *coding up* objects and dynamic dispatch in Scheme. This serves two purposes:

- It demonstrates that one language's *semantics* (how the primitives like message send work in the language) can typically be coded up as an *idiom* (simulating the same behavior via some helper functions) in another language.
- It gives a lower-level way to understand how dynamic dispatch "works" by seeing how we would do it manually in another language.

Our approach will be different from what Ruby (or Java) actually does in these ways:

- Our objects will just contain a list of fields and a list of methods. This is not "class-based", in which an object would have a list of fields and a class-name and then the class would have the list of methods.
- Real implementations are more efficient. They use better data structures (based on arrays) for the fields and objects rather than simple association lists.

Nonetheless, the key ideas behind how you implement dynamic dispatch still come through. By the way, we are wise to do this in Scheme rather than ML, where the types would get in our way. In ML, we would likely end up using "one big datatype" to give all objects and all their fields the same type, which is basically awkwardly programming in a Scheme-like way in ML. (On the other hand, typed OO languages are often no friendlier to ML-style programming.)

Our objects will just have fields and methods:

```
(define-struct obj (fields methods)) ; a cons cell would also work
```

We will have `fields` hold an association list (a list of pairs), where elements are a symbol (the field name) and a value (the current field contents). With that, we can define helper functions `get` and `set` that given an object and a field-name, return or mutate the field appropriately. Notice these are just plain Scheme functions, with no special features or language additions, except that to get things to work in Racket, we need to use Racket's mutable lists for mutable fields:

```
(require racket/mpair)
```

```
(define (get obj fld)
  (let ([pr (massoc fld (obj-fields obj))])
    (if pr
        (mcdr pr)
        (error "field not found"))))
(define (set obj fld v)
  (let ([pr (massoc fld (obj-fields obj))])
    (if pr
        (set-mcdr! pr v)
        (error "field not found"))))
```

More interesting is calling a method. The `methods` field will also be an association list, mapping method names to functions. The key to getting dynamic dispatch to work is that these functions will all take an extra explicit argument that is implicit in languages with built-in support for dynamic dispatch. This argument will be "self" and our Scheme helper function for sending a message will simply pass in the correct object:

```
(define (send obj msg . args)
  (let ([pr (assoc msg (obj-methods obj))])
    (if pr
        ((cdr pr) obj args) ; do the call
        (error "method not found" msg))))
```

Notice how the function we use for the method gets passed the "whole" object `obj`, which will be used for any sends to the object bound to `self`.

Now we can define `make-point`, which is just a Scheme function that produces a point object. That's what constructors do; they take arguments and return new objects:

```
(define make-point
  (lambda (_x _y)
    (make-obj
     (mlist (mcons 'x _x)
            (mcons 'y _y))
     (list (cons 'get-x (lambda (self lst) (get self 'x)))
           (cons 'get-y (lambda (self lst) (get self 'y)))
           (cons 'set-x (lambda (self lst) (set self 'x (car lst))))
           (cons 'set-y (lambda (self lst) (set self 'y (car lst))))
           (cons 'distToOrigin
                 (lambda (self lst)
                   (let ([a (send self 'get-x)]
                         [b (send self 'get-y)])
                     (sqrt (+ (* a a) (* b b))))))))))
```

Notice how each of the methods takes a first argument, which we just happen to call `self`. We then use `self` as an argument to `get`, `set`, and `send`. Of course, if we had some other object we wanted to send a message or access a field of, we would just pass that object to our helper functions.

Finally, we can define a "subclass" of polar points (defined at the end since it gets a bit long). We can append new fields and new/overriding methods to the front of the lists made by creating a point. (This relies on the fact that `assoc` returns the first matching pair in the list.) So our "constructor" uses the `make-point` constructor (just like Java constructors always call `super()`).

Most importantly, the procedure paired with `'distToOrigin` still works for a polar point because the method calls in its body will use the procedures listed with `'get-x` and `'get-y` in the definition of `make-polar-point`.

```
(define make-polar-point
  (lambda (_r _th)
    (let ([pt (make-point #f #f)])
      (make-obj
       (mappend (mlist (mcons 'r _r)
                       (mcons 'theta _th))
                (obj-fields pt)) ; Java-style field extension
       (append
        (list
         (cons 'set-r-theta
               (lambda (self lst)
                 (begin
                   (set self 'r (car lst))
                   (set self 'theta (car (cdr lst))))))
         (cons 'get-x (lambda (self lst)
                        (let ([r (get self 'r)]
                              [theta (get self 'theta)])
                          (* r (cos theta)))))
         (cons 'get-y (lambda (self lst)
                        (let ([r (get self 'r)]
                              [theta (get self 'theta)])
                          (* r (sin theta)))))
         (cons 'set-x (lambda (self lst)
                        (let* ([a     (car lst)]
                               [b     (send self 'get-y)]
                               [theta (atan (/ b a))]
                               [r     (sqrt (+ (* a a) (* b b)))])
                          (send self 'set-r-theta r theta))))
         (cons 'set-y (lambda (self lst)
                        (let* ([b     (car lst)]
                               [a     (send self 'get-x)]
                               [theta (atan (/ b a))]
                               [r     (sqrt (+ (* a a) (* b b)))])
                          (send self 'set-r-theta r theta)))))
        (obj-methods pt))))))
```

We can create an "object" and send it some messages like this:

```
(define p1 (make-polar-point 4 3.1415926535))
(send p1 'set-y 4)
(send p1 'get-y)
(send p1 'distToOrigin)
```