CSE341, Fall 2011, Lecture 21 Summary

Standard Disclaimer: This lecture summary is not necessarily a complete substitute for attending class, reading the associated code, etc. It is designed to be a useful resource for students who attended class and are later reviewing the material.

The purpose of this lecture is to consider the semantics of object-oriented language constructs, particularly calls to methods, as carefully as we have considered the semantics of functional language constructs, particularly calls to closures. As we will see, the key distinguishing feature is what `self` (Java's `this`) is bound to in the environment when a method is called. This concept is known as *late binding*, *dynamic dispatch*, or *virtual method calls*.

We will also see how to encode this concept as a *programming idiom* in Racket, using nothing more than pairs and functions. This exercise is nicely complementary to the exercise in lecture 11 where we encoded the concept of closures in Java and C. As a general point, the semantics in one approach to programming can usually be encoded as an idiom in another approach.

Resolving identifiers (variables, fields, method names, etc.)

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 programming language's semantics. For example, the lexical scope of ML and Racket 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. 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 are not declared, the actual answer depends on whether the method has 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. In other words, blocks are like closures in how variables are looked up, and the implementation of Ruby needs to associate environments with blocks just like the implementation of a functional language needs to associate environments with closures.

What it means to be first-class

A separate question from "lookup rules" such as what environment we use to look up a variable is the question of what can be bound to something like a variable, put in a field, passed to a method, etc. We call something *first class* if it can be computed with, stored in variables or fields, returned from functions/methods, etc. In Ruby, objects are first class and we have said before that "everything is an object," but to be more precise "every *value* is an object." Here are some things that are not objects in Ruby and therefore are not first-class in Ruby:

- Method names: If `m` and `n` are method names, you cannot write `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.

The notion of what is first-class is an important of any programming-language definitiona, and naturally the answer depends on the language.

Resolving (i.e., looking up) method calls: how to treat `self`

Since method names are not variables, we do not look them up in the environment like variables. We do evaluate the *receiver of a message (method call)* as a first-class expression, and then we use that object's class to determine what method to call.

In class-based object-oriented languages like Ruby and Java, the rule for evaluating a method call like `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 define `m` (with the right number of arguments).
- (Ruby's mixins complicate the lookup rules a bit more, as we will see in a future language. Furthermore, Ruby actually resolves undefined methods by calling the receiver's `method_missing` method, which is defined in `Object` to raise an error.)

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 method calls." It is central to the semantics of Java and Ruby. 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 the version of `distFromOrigin` that used getter methods. The method 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/Racket function calls. It may not seem that way to you because you learned it first. But it is truly more complicated: we have to treat the notion of `self` differently from everything else in the language. Complicated does not necessarily mean it is inferior or superior; it just means the language definition has more cases and takes more details to describe.

Contrasting dynamic dispatch and lexical scope

To understand how dynamic dispatch differs from the lexical scope we used for 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 `even` in the environment where `odd` was defined, it will get the function on the first line above. That is "good" for understanding how `odd` works *just from looking where 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 OOP, 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.

Encoding dynamic dispatch as a programming idiom

Let's now consider *coding up* objects and dynamic dispatch in Racket using nothing more than pairs and functions.[1] This serves two purposes:

[1]Though we did not study it, Racket has classes and objects, so you would not actually want to do this in Racket. The point is to understand dynamic dispatch by manually coding up the same idea.

- 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. This can help you be a better programmer in different languages that may not have the features you are used to.
- It gives a lower-level way to understand how dynamic dispatch "works" by seeing how we would do it manually in another language. An interpreter for an object-oriented language would have to do something similar for automatically evaluating programs in the language.

Also notice that we did an analogous exercise to better understand closures earlier in the course: We showed how to get the effect of closures in Java using objects and interfaces or in C using function pointers and explicit environments.

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. We could have done it that way instead.
- Real implementations are more efficient. They use better data structures (based on arrays or hashtables) for the fields and methods 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 Racket 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 Racket-like way in ML. (Conversely, typed OO languages are often no friendlier to ML-style programming unless they add separate constructs for generic types and closures.)

Our objects will just have fields and methods:

```
(struct obj (fields methods))
```

We will have `fields` hold an immutable list of *mutable* pairs where each element pair is 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 Racket functions, with no special features or language additions. We do need to define our own function, called `assoc-m` below, because Racket's `assoc` expects an immutable list of immutable pairs.

```
(define (assoc-m v xs)
  (cond [(null? xs) #f]
        [(equal? v (mcar (car xs))) (car xs)]
        [#t (assoc-m v (cdr xs))]))

(define (get obj fld)
  (let ([pr (assoc-m fld (obj-fields obj))])
    (if pr
        (mcdr pr)
        (error "field not found"))))

(define (set obj fld v)
  (let ([pr (assoc-m 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 (no mutation needed since we will be less dynamic than Ruby). 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 Racket 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)
        (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`. (The code above uses Racket's support for variable-argument functions because it is convenient — we could have avoided it if necessary. Here, `send` can take any number of arguments greater than or equal to 2. The first argument is bound to `obj`, the second to `msg`, and all others are put in a list (in order) that is bound to `args`. Hence we expecte `(cdr pr)` to be a function that takes two arguments: we pass `obj` for the first argument and the list `args` for the second argument.)

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

```
(define (make-point _x _y)
  (obj
   (list (mcons 'x _x)
         (mcons 'y _y))
   (list (cons 'get-x (lambda (self args) (get self 'x)))
         (cons 'get-y (lambda (self args) (get self 'y)))
         (cons 'set-x (lambda (self args) (set self 'x (car args))))
         (cons 'set-y (lambda (self args) (set self 'y (car args))))
         (cons 'distToOrigin
               (lambda (self args)
                 (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`, which has no special meaning here in Racket. We then use `self` as an argument to `get`, `set`, and `send`. If we had some other object we wanted to send a message to or access a field of, we would just pass that object to our helper functions by putting it in the `args` list. In general, the second argument to each function is a list of the "real arguments" in our object-oriented thinking.

By using the `get`, `set`, and `send` functions we defined, making and using points "feels" just like OOP:

```
(define p1 (make-point 4 0))
(send p1 'get-x)        ; 4
(send p1 'get-y)        ; 0
(send p1 'distToOrigin) ; 4
(send p1 'set-y 3)
(send p1 'distToOrigin) ; 5
```

A Simple "Subclass"

Our encoding of objects does not use classes, but we can still create something that reuses the code used to define points. Here is code to create points with a color field and getter/setter methods for this field. The key idea is to have the constructor create a point object with `make-point` and then extend this object by creating a new object that has the extra field and methods:

```
(define (make-color-point _x _y _c)
  (let ([pt (make-point _x _y)])
    (obj
     (cons (mcons 'color _c)
           (obj-fields pt))
     (append (list
              (cons 'get-color (lambda (self args) (get self 'color)))
              (cons 'set-color (lambda (self args) (set self 'color (car args)))))
           (obj-methods pt)))))
```

We can use "objects" returned from `make-color-point` just like we use "objects" returned from `make-point`, plus we can use the field `color` and the methods `get-color` and `set-color`.

A "Subclass" Using Dynamic Dispatch

The essential distinguishing feature of OOP is dynamic dispatch. Our encoding of objects "gets dynamic dispatch right" but our examples do not yet demonstrate it. To do so, we need a "method" in a "superclass" to call a method that is defined/overridden by a "subclass." As in the previous lecture, let's define polar points by adding new fields and overriding the `get-x`, `get-y`, `set-x`, and `set-y` methods. A few details about the code below:

- As with color-points, our "constructor" uses the "superclass" constructor.
- As would happen in Java, our polar-point objects still have `x` and `y` fields, but we never use them.
- For simplicity, we just override methods by putting the replacements earlier in the method list than the overridden methods. This works because `assoc` returns the first matching pair in the list.

Most importantly, the `distToOrigin` "method" 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` just like dynamic dispatch requires. The correct behavior results from our `send` function passing the whole object as the first argument.

```
(define (make-polar-point _r _th)
  (let ([pt (make-point #f #f)])
    (obj
     (append (list (mcons 'r _r)
                   (mcons 'theta _th))
             (obj-fields pt))
     (append
      (list
       (cons 'set-r-theta
             (lambda (self args)
               (begin
                 (set self 'r (car args))
                 (set self 'theta (cadr args)))))
```

```
    (cons 'get-x (lambda (self args)
                   (let ([r (get self 'r)]
                         [theta (get self 'theta)])
                     (* r (cos theta)))))
    (cons 'get-y (lambda (self args)
                   (let ([r (get self 'r)]
                         [theta (get self 'theta)])
                     (* r (sin theta)))))
    (cons 'set-x (lambda (self args)
                   (let* ([a     (car args)]
                          [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 args)
                   (let* ([b     (car args)]
                          [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 a polar-point object and send it some messages like this:

```
(define p3 (make-polar-point 4 3.1415926535))
(send p3 'get-x) ; 4
(send p3 'get-y) ; 0 (or a slight rounding error)
(send p3 'distToOrigin) ; 4 (or a slight rounding error)
(send p3 'set-y 3)
(send p3 'distToOrigin) ; 5 (or a slight rounding error)
```