Notes on Objects

Programming Languages

October 18, 2016

Records

Records (or dictionaries) are simple structures mapping names to values.
In Python, records are constructed via:

```
x = {"foo":10, "bar":20+20}
```

and you access the components of a record (the fields) using the notation:

```
x["foo"]
```

In JavaScript, records are similar, for both construction

```
x = {foo:10, bar:20+20};
```

and access to components:

```
x["foo"]    x.foo
```

Moreover, records in those languages are *mutable*: you can change the value bound to a field.

It is straightforward to add records to a language such as our FUNC. For the sake of our examples, we'll in fact use REF, our language of S-expressions with reference cells.
We can easily add surface syntax to create records:

```
(record (x e1) (y e2) ...)
```

For instance, the examples above would be written

```
(record (foo 10) (bar (+ 20 20)))
```

This should evaluate down to a *record value* associating a value to every name, the result of evaluating the corresponding expression.

Corresponding to both record expressions and record values, we have nodes in our abstract representation:

```
class ERecord (Exp):
    def __init__ (self,bindings):
        self._bindings = bindings

    def eval (self,env):
        bindings = [ (id,e.eval(env)) for (id,e) in self._bindings]
        return VRecord(bindings)


class VRecord (Value):
    def __init__ (self,bindings):
        self.type = "record"
        self.bindings = bindings
```

This is all completely straightforward. Note that a record value makes its bindings (a list of name/value pairs) available as field `bindings`.

To access the fields of a record, we could easily add surface syntax such as

```
(field recExp name)
```

that extracts the value bound to `name` in the record obtained by evaluating `recExp`. I'll leave this one as an exercise to the reader.

A more interesting variant is to use the fact that a record value is really just a little environment (both associate values with names) to simplify the implementation.

Consider the following surface syntax

```
(with recExp bodyExp)
```

which evaluates by first evaluating `recExp` to a record value, adding the bindings in the record value to the top of the environment (so that they are first accessed when searching for an identifier's value) and then evaluating `bodyExp` in the resulting environment. You can think of it as *opening up* the record into the environment.

Thus,

```
(define r (record (a 10) (b (+ 20 20))))

(with r b)
```

would evaluate to 40, just like if we were using fields. But

```
(define r (record (a 10) (b (+ 20 20))))

(with r (+ a b))
```

would evaluate to 50, and would be slightly more inconvenient to write using field access. Of course, `with` expressions can be embedded into larger expressions.

```
(define r (record (a 10) (b (+ 20 20))))

(let ((c 99)
      (a 1000000))
   (with r (+ a c)))
```

Check your understanding: this last expression evaluates to 109.

Surface syntax `with` parses into a straightforward `EWith` node of the abstract representation:

```
class EWith (Exp):
    def __init__ (self,recExp,bodyExp):
        self._record = recExp
        self._exp = bodyExp

    def eval (self,env):
        record = self._record.eval(env)
        if record.type != "record":
            raise Exception("Runtime error: expected a record")
        return self._exp.eval(record.bindings+env)
```

Of course, fields of a record can hold any sort of value, including functions. So we can easily write

```
(define r (record (double (function (x) (* 2 x)))))

(with r (double 10))
```

the last expression evaluating to 10.

We can also do much more clever things, with functions in a record being able to access a shared reference cell:

```
(define r (let ((k (ref 2)))
            (record (mult (function (x) (* (deref k) x)))
                    (set! (function (y) (update! k y))))))
```

```
(with r (mult 10))    ;; --> 20

(with r (set! 5))

(with r (mult 10))    ;; --> 50
```

Objects

Object-oriented programming is a programming model where a set of functions has access to a shared encapsulated local state (i.e., an *object*).

To a first approximation, an object is a record where some of the fields are functions (i.e., *methods*).

Object-oriented systems generally split into two categories:

- **class-based**: a class is a template for objects (e.g., Java, C++, Python, Smalltalk)
- **prototype-based**: objects are derived from existing objects and modifying them (e.g., JavaScript, Self)

We're going to focus on class-based systems here.

Given a class, you can *instantiate* it to get an object (called an *instance* of the class). An operator such as `new` is often used to instantiate a class.

Other issues of interest is how the object-oriented system handles subclassing and code reuse. To a first approximation, *subclassing* is a client-side feature (that is, something that affects the user of a class). Class B is said to be a subclass of A if an instance of B can be used in any context where an instance of A is expected.

Inheritance is a *code reuse* mechanism, as is delegation. Both are ways to create a new class by reusing the code from another class. In the case of inheritance, this usually has the side effect of creating a subclass. If a class B inherits from class A, the idea is that if a method on an instance of B is called and B itself doesn't define the method, then the method is looked for in A. Things get a bit murkier when thinking about inheritance for fields.

Let's add classes to REF, our languages of S-expressions with reference cells. (Reference cells are not needed, but they make for nicer examples.) We are going to mimick the way we introduced records above.

We are going to first introduce classes, then introduce objects. Classes will be first-class, meaning that classes will be values themselves. (In most languages, classes are declarations. That's because classes are often tied to the type system, which we'll return to later in the course.)

The surface syntax for creating a class is:

```
(class (x ...) ((a e1) ...) ((m f1) ...))
```

where `x ...` are the parameters of the class (that can be used when constructing an instance of the class) while `((a e1) ...)` are the fields of the class (with `e1`,... evaluating to the initial value of those fields when the class is instantiated) and `((m f1) ...)` are the methods of the class, with each `f1`,... a `(function ...)` expression.

As an example, here is a simple class with a field `k` and a method `mult`. It doesn't do anything particularly interesting.

```
(class (init)
  ((k (ref init)))
  ((mult (function (x) (* 2 x)))))
```

Classes evaluate to *class values*, that are basically constructor functions (functions that create an instance of the class). You can see that from the fact that class values store a closure that when called creates an instance of the class using `EObject` which we'll see below.

The class surface syntax has an immediate abstract representation:

```
class EClass (Exp):

    def __init__ (self,params,fields,methods):
        self._params = params
        self._fields = fields
        self._methods = methods

    def eval (self,env):
        constructor = VClosure(self._params,
                               EObject(self._fields,
                                       [(id,EFunction(["this"],exp)) for (id
   ,exp) in self._methods]),
                               env)
        return VClass(self._params,constructor)

class VClass (Value):

    def __init__ (self,params,constructor):
        self._params = params
        self._constructor = constructor
        self.type = "class"

    def instantiate (self,args):
        return self._constructor.apply(args)
```

Let's notice that the methods passed to the `EObject` in the constructor are modified somewhat. More on that below.

To instantiate a class, we have surface syntax:

```
(new clexp e1 ...)
```

where `clexp` is an expression that should evaluate to a class value to be instantiated, and `e1`,... are the arguments passed as the parameters of the class to instantiate it.

The abstract representation corresponding to the `new` surface syntax is basically a call to the closure (representing the constructor function) stored in a class value:

```
class ENew (Exp):
    def __init__ (self,cls,args):
        self._class = cls
        self._args = args

    def eval (self,env):
        cls = self._class.eval(env)
        if cls.type != "class":
            raise Exception("Runtime error: new() on a non-class")
        args = [ e.eval(env) for e in self._args ]
        return cls.instantiate(args)
```

Objects themselves are constructed using the `EObject` abstract representation node, which roughly corresponds to a record. The one difference from records is that fields and methods are stored separately, because methods will require a bit of work.

```
class EObject (Exp):

    def __init__ (self,fields,methods):
        self._fields = fields
        self._methods = methods

    def eval (self,env):
        fields = [ (id,e.eval(env)) for (id,e) in self._fields]
        methods = [ (id,e.eval(env)) for (id,e) in self._methods]
        return VObject(fields,methods)


class VObject (Value):

    def __init__ (self,fields,methods):
        self.type = "object"
        self._fields = fields
        self._methods = methods
        self.env = fields + [ (id,v.apply([self])) for (id,v) in methods]
```

To call a method on an object or access a field of an object, we use a similar trick as we did for records. We have surface syntax

```
(with objexp e)
```

that takes an object expression, evaluates it to an object, makes its fields and methods available by adding them to the current environment, and evaluates `e` in the resulting environment. The implementation is the same as for records:

```
class EWithObj (Exp):
    def __init__ (self,exp1,exp2):
        self._object = exp1
        self._exp = exp2

    def eval (self,env):
        object = self._object.eval(env)
        if object.type != "object":
            raise Exception("Runtime error: expected an object")
        return self._exp.eval(object.env+env)
```

All of this is pretty straightforward, and looks a lot like our records implementation. The one difference is in the environment stored in a `VObject`, the one that gets added to the current environment in the `EWith` class.

Let's explain things a little bit.

The problem we need the solve, the reason why we cannot easily implement objects using records directly, is that we would like to be able to refer to other fields or methods of an object from within methods of the object. That basically means that we need to have access to the object itself from within the object. The common solution to this problem is to have every method of an object take an extra argument, which will get the object itself when a method is called in an object. That makes the object available from within the body of the method.

Languages have two ways of handling this extra argument:

- many languages treat that extra argument *implicitly:* that extra argument does not appear in the argument list of methods, and instead is available using a special keyword such as `this`. (Examples: C++, Java, JavaScript)
- some languages require that argument to be listed *explicitly.* Python is one such example.

We're treating this extra argument explicitly in our implementation of objects. If you look at `EClass`, you see that it calls `EObject` passing it the fields and methods that the object should have. The methods are wrapped inside another function that expects a single argument, `this`, which will represent the current object when the method is eventually called. Thus, a method defined by

```
(double (function (x) (* 2 x)))
```

in a class will be treated as a function of the form

```
(double (function (this) (function (x) (* 2 x))))
```

in `EObject` and stored that way. If you think *curry-style*, this is a function that expects an argument `this` and an argument `x` and returns 2`x`. The argument `this` is the implicit argument containing the current object.

When we call a method, we better make sure to pass an object to the `this` argument. That's actually achieved in `VObject`. A `VObject` is an object value, and it has an environment for fields and an environment for methods, put together into a single environment that's available as field `env`, and that gets used by `EWith` to know what to add to the current environment. When that single environment is constructed in an object *obj*, the methods are called and are passed *obj*, so that the results are the methods as they were originally defined in the class by the user, and in the body of which identifier `this` is bound to the current object *obj*.

We can now reimplement the "adjustable" multiplier from earlier, in our object-oriented system:

```
(define multiplier (class (init)
                     ((k (ref init)))
                     ((mult (function (x) (with this (* (deref k) x)))))))

(define m (new multiplier 10))

(with m (mult 3))    ;;  --> 30

(update! (with m k) 20)

(with m (mult 3))    ;;  --> 60
```

Here is another classical example, points:

```
obj> (define point (class (_x _y) ((x (ref _x)) (y (ref _y)) (code 1))
                  ((move (function (nx ny) (with this (do (update! x nx)
                                                          (update! y ny))))))))
point defined
obj> (define p (new point 10 20))
p defined
obj> (with p (move 11 21))
none
```

```
obj> (with p (deref x))
11
obj> (with p (deref y))
21
```

Inheritance

When a class B inherits from class A, intuitively, we want to make the methods (and fields) of A available to instances of B.

One way to implement inheritance in our setting is to create instance of A (called a super-object) whenever we create an instance of B, so that when we look for a method in B, if we don't find it in the instance of B, we look for it in the super-object. The trickiest think is to get *dynamic dispatch* to work correctly: the *this* identifier in a method of the super-object of an instance of B should refer to the instance of B, and not to the super-object.

We can extend the surface syntax of classes to allow for the declaration of a class from which to inherit (the super-class). When we create an instance *inst* of a class, we also instantiate any super-class to form a super-object that will be associated with the instance *inst* created.

```
(class (x ...) (clexp g1 ...) ((a e1) ...) ((m f1) ...))
```

Here, `clexp` is an expression that should evaluate to a class value representing the super-class, and `g1`,... are expressions that will be evaluated and passed to the constructor of the super-class to create the super-object when the class itself is instantiated. (Both `clexp` and `g1`,... are evaluated when an instance is created, so that they may depend on the arguments passed to the constructor.)

To go hand in hand with this extended surface syntax, `EClass` and `EObject` both take optional arguments holding the super-class expression `clexp` and the super-class argument expressions `g1`,...

```
class EClass (Exp):

    def __init__ (self,params,fields,methods,sp=None,args=[]):
        self._params = params
        self._fields = fields
        self._methods = methods
        self._super = sp
        self._super_args = args

    def eval (self,env):
        constructor = VClosure(self._params,
                               EObject(self._fields,
```

```
                                        [(id,EFunction(["this"],exp)) for (id
   ,exp) in self._methods],
                                        self._super,
                                        self._super_args),
                                env)
        return VClass(self._params,constructor)
```

`EClass` simply passes those optional parameters to `EObject` in the constructor function. `EObject` creates the super-object and passes it to `VObject` (again, as an optional argument).

```
class EObject (Exp):

    def __init__ (self,fields,methods,sp=None,args=[]):
        self._fields = fields
        self._methods = methods
        self._super = sp
        self._super_args = args

    def eval (self,env):
        fields = [ (id,e.eval(env)) for (id,e) in self._fields]
        methods = [ (id,e.eval(env)) for (id,e) in self._methods]
        if self._super:
            # instantiate super-object if one is needed
            sp = self._super.eval(env)
            args = [ e.eval(env) for e in self._super_args ]
            if sp.type != "class":
                raise Exception("Runtime error: super class not a class")
            super_obj = sp.instantiate(args)
        else:
            super_obj = None
        return VObject(fields,methods,super_obj)
```

The two final changes are in `VObject` which deals with the (optional) super-object by adding its bindings to the environment provided by the object itself, layered in such a way that the bindings for the object are searched before the bindings for the super-object.

```
class VObject (Value):

    def __init__ (self,fields,methods,sp=None):
        self.type = "object"
        self._fields = fields
        self._methods = methods
        self._super = sp
```

```
    def env (self,obj):
        env = self._fields + [ (id,v.apply([obj])) for (id,v) in self.
    _methods ]
        if self._super:
            # if there's a super object, add its environment in _after_ the
    current one
            env += self._super.env(obj)
        return env
```

The main difference here is that what used to be an `env` field in `VObject` is now a `env()` method: it takes an object to use as *this* as an argument. (The object is used as the first argument to the wrapped methods, which resolves the *this* parameter introduced by the constructor of the class. When the environment of the super-object is added, that same object is passed as well, which makes the methods in the super-object refer to that same object through their *this* argument. This takes care of dynamic dispatch: the *this* argument always refers to the original object on which the method was invoked, even if the method ultimately called is one that belongs to a super-object.

That last bit is ensured by the evaluation method for `EWithObj` that gets the environment from the object and adds it to the current environment, making sure to pass the object itself to the `env()` method of the object to resolve all the *this* arguments to the object itself:

```
class EWithObj (Exp):
    def __init__ (self,exp1,exp2):
        self._object = exp1
        self._exp = exp2

    def eval (self,env):
        object = self._object.eval(env)
        if object.type != "object":
            raise Exception("Runtime error: expected an object")
        return self._exp.eval(object.env(object)+env)
```

By way of example, here is an interaction with the system where a point class and a colored point class are created. Method `move` moves a point to another position, and is inherited by the colored point class. To illustrate dynamic dispatch, we have a field `code` that returns 1 or 2 depending on whether the object is a color point or a point, and a method `print` defined only in the point class that prints the value of code—which field `code` is used depends on the actual instance on which `print` is invoked.

```
obj/inh> (define point (class (_x _y) ((x (ref _x)) (y (ref _y)) (code 1))
                  ((move (function (nx ny) (with this (do (update! x nx)
                                                          (update! y ny)))))
                   (print (function () (print! (with this code)))))))
```

```
point defined
obj/inh> (define p (new point 10 20))
p defined
obj/inh> (with p (move 11 21))
none
obj/inh> (with p (deref x))
11
obj/inh> (with p (deref y))
21
obj/inh> (with p (print))
1
none

obj/inh> (define cpoint (class (_x _y _c) (point _x _y) ((color _c) (code 2)) ()))
cpoint defined
obj/inh> (define cp (new cpoint 100 200 99))
cp defined
obj/inh> (with cp color)
99
obj/inh> (with cp (move 101 202))
none
obj/inh> (with cp (deref x))
101
obj/inh> (with cp (deref y))
202
obj/inh> (with cp color)
99
obj/inh> (with cp (print))
2
none
```

Exercise: the above does not allow you to call a method in the super-object if it's defined in the current object. Most language will allow you to invoke methods on the super-object via a special identifier such as *super*. (Special identifier in the same sense that *this* is a special identifier.) Extend the above implementation of classes and objects so that you can refer to any super-object of an object via identifier *super*.