Notes on Objects

Programming Languages

March 15, 2018

Records

Records are simple structures mapping names to values.

Records are not unlike Javascript objects, constructed via

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

Components `foo` and `bar` are called fields, and can be accessed via:

```
x.foo
```

Records in Javascript are *mutable*: you can change the value bound to a field.

Dictionaries in Python are not quite records. They don't bind names to values, but rather values to values. (Which is why you have to use quotes around the keys of a dictionary in Python, because they are string values, as opposed to names.)

It is straightforward to add records to a language like those we have developed. For the sake of our examples, we'll use REF, our language of S-expressions with reference cells, extended with the ability to use `define` at the shell to define new top-level bindings.

We can easily define 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.

The `record` surface syntax parses into a record expression, and record expressions evaluate to record values. We add record expressions and record values to our abstract representation:

```
class VRecord (bindings: List[(String,Value)]) extends Value {

  override def toString () : String = "record(" + bindings + ")"
  override def isRecord () : Boolean = true

  override def getBindings () : List[(String,Value)] = {
     return bindings
  }

  override def lookup (s:String) : Value = {
     for ((n,v) <- bindings) {
       if (n==s) {
          return v
       }
     }
     error("No field "+s+" in record")
  }
}

class ERecord (val bindings: List[(String,Exp)]) extends Exp {

   override def toString () : String =
       "ERecord(" + bindings + ")"

    def eval (env : Env) : Value = {
       val bvs = bindings.map((p) => (p._1,p._2.eval(env)))
       return new VRecord(bvs)
    }
}
```

This is all completely straightforward. Note that a record value makes its bindings (a list of name/value pairs) available via method `getBindings`, and also implements a `lookup` method to look up the value associated with a name.

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

```
(field recExp name)
```

that extracts the value bound to `name` in the record obtained by evaluating `recExp`. The `field` surface syntax parses into an `EField` node of the abstraction representation:

```
class EField (val r:Exp, val f:String) extends Exp {
```

```
  override def toString () : String =
     "EField(" + r + "," + f + ")"

  def eval (env : Env) : Value = {
    val vr = r.eval(env)
    return vr.lookup(f)
  }
}
```

An interesting variant to access fields of a record is to allow a programmer to *open up* a record, and make its names available in the environment like any other binding.

Consider the following surface syntax

```
(with recExp bodyExp)
```

which evaluates by first evaluating `recExp` to a record value and then 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) before evaluating `bodyExp` in the resulting 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 `EWith` node of the abstract representation:

```
class EWith (val r:Exp, val body:Exp) extends Exp {

   override def toString () : String =
      "EWith(" + r + "," + body + ")"

   def eval (env : Env) : Value = {
     val vr = r.eval(env)
     var newenv = env
     for ((n,v) <- vr.getBindings()) {
       newenv = newenv.push(n,v)
     }
     return body.eval(newenv)
   }
}
```

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

```
(define r (record (double (fun (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 hidden in the closure of the record:

```
(define r (let ((k (ref 2)))
            (record (mult (fun (x) (* (read k) x)))
                    (set! (fun (y) (write! k y))))))

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

(with r (set! 5))

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

This starts very much to look like an object in the object-oriented programming sense, with `k` as a field of the object, and `mult` and `set!` as methods.

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, Scala, C++, Python, Smalltalk)
- **prototype-based**: objects are derived from existing objects (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 sometimes used to instantiate a class, but not always.

Class A is said to be a subclass of B if an instance of A can be used in any context where an instance of B is expected. This is a behavioral definition, and moreover concerns the *users* of a class.

The two main code reuse mechanisms in object-oriented programming are delegation and inheritance. They concern the *implementers* of a class. Delegation is just what it says on the tin: a class A delegates to another class B the implementation of some of its methods. This is often achieved by explicitly putting in an object of class A an object of class B which is the target of the delegation. Class A and class B need not related.

Inheritance is a form of implicit delegation. Class A inherits from class B if whenever A does not define a method, it is looked for automatically in B. In pretty much every language that supports inheritance, inheritance will imply subclassing: if A inherits from B, then A will be a subclass of B.

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 imitate the way we introduced records above.

We first introduce classes, then introduce objects. 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. Here, we will interpret a class as simply a function that creates objects of that class (basically, a class will be its constructor).

The surface syntax for creating a class is:

```
(class c (x ...) (fields (a e1) ...) (methods (m (p ...) body) ...)
```

where `c` is a name by which the class can be referred to within its own methods (in case a method wants to construct a new instance of the class), `x ...` are the parameters of

the class that can be used when constructing an instance of the class, `(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 (p ...) body) ...` are the methods of the class, with each method taking parameters `p ...` and body `body`.

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

```
(class c (init)
  (fields (k (ref init)))
  (methods (double (x) (* 2 x))))
```

Classes do not have an abstract representation — they translate via a syntactic transformation into a function that creates an object via `EObject`. Here is the syntactic transformation function:

```
def mkClass (name:String, params:List[String],
             fields:List[(String,Exp)], meths:List[(String,Exp)]) : Exp =
 {
  return new ERecFunction(name,params,new EObject(fields,meths))
}
```

An object is basically a record, meaning that we have a value class `VObject` to represent object values, and an abstract representation node `EObject` to represent the expression used to create a `VObject` value.

```
class VObject (val fields: List[(String,Value)], val methods:List[(String,
   Value)]) extends Value {

  override def toString () : String = "object(" + fields + "," + methods +
   ")"
  override def isObject () : Boolean = true

  override def lookupField (s:String) : Value = {
     for ((n,v) <- fields) {
       if (n==s) {
          return v
       }
     }
     error("No field "+s+" in object")
  }

  override def lookupMethod (s:String) : Value = {
     for ((n,v) <- methods) {
       if (n==s) {
```

```
          return v
        }
      }
      error("No method "+s+" in object")
  }
}

class EObject (val fields: List[(String,Exp)],
               val meths:List[(String,Exp)]) extends Exp {

   override def toString () : String =
     "EObject(" + fields + "," + meths + ")"

   def eval (env : Env) : Value = {
     // evaluate all the field expressions
     val fields_val = fields.map((p) => (p._1,p._2.eval(env)))
     // wrap every method with as function expecting "this" as an argument
     val meths_val =
          meths.map((p) => (p._1,new VRecClosure("",List("this"),p._2,env)))
     return new VObject(fields_val,meths_val)
   }
}
```

The one difference from records is that fields and methods are stored separately, because methods need to be handled specially.

To access a field of an object is like accessing a field of a record:

```
(field objExp name)
```

where `objExp` is an expression evaluating to an object, and `name` is the name of the field to access. This parses into the abstraction represetion node `EField`:

```
class EField (val r:Exp, val f:String) extends Exp {

  override def toString () : String =
     "EField(" + r + "," + f + ")"

  def eval (env : Env) : Value = {
    val vr = r.eval(env)
    return vr.lookupField(f)
  }
}
```

To access a method of an object, we do something similar, but with a twist. The surface syntax is straightforward:

```
(method objExp name)
```

where `objExp` is an expression evaluating to an object, and `name` is the name of the method to access. This evaluates to a function, which can then be applied to arguments to invoke the method, such as in:

```
((method test double) 25)
```

which calls method `double` in object `test` with argument 25.

The twist is that methods have a characteristic that fields don't have. We generally want to be able to refer to other fields or methods of an object from within methods of the object. That 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 receive the object itself when a method is called on an object. This extra argument (called `this`, or `self`) makes the object available in 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`. This is what C++, Java, JavaScript, and Scala do.
- some languages require that argument to be listed *explicitly.* This is what Python does — every method of an object in Python looks like `def foo (self,x,y): ...`

We're treating this extra argument explicitly in our surface syntax. We're using the keyword `this` in a method to represent the object on which the method is invoked. How is that achieved? Well, look at the code for `EObject`. You see that when we create the list of methods to store in the resulting `VObject`, we wrap every single method with a function that expects a single argument `this`. Thus, a method defined by

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

in a class will be treated as a function:

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

The argument `this` is the implicit argument containing the current object.

We access the method of an object using expression `EMethod`, which is similar to `EField`:

```
class EMethod (val r:Exp, val f:String) extends Exp {

  override def toString () : String =
     "EMethod(" + r + "," + f + ")"

  def eval (env : Env) : Value = {
    val vr = r.eval(env)
    val m = vr.lookupMethod(f)
    return m.apply(List(vr))
  }
}
```

The difference is that when we pull the method out of the object, we immediately call it, passing the current object (from which we extracted the method!) as an argument. This will return the function that had been wrapped, the function that expects the actual parameters of the method. But that function is defined in an environment where `this` has been bound to the object from which the method was accessed, meaning that the body of the method can freely refer to identifier `this`. Boom.

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

```
  (define multiplier (class c (init)
                       (fields (k (ref init)))
                       (methods (mult (x) (* (read (field this k)) x)))))

  (define m (multiplier 10))

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

  (write! (field m k) 20)

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

More interestingly, we can actually implement class `Coordinates` from Homework 1. There we implemented it in Scala. Now we implement it in our own language:

```
(define coordinates
   (class c (initx inity)
      (fields (x initx) (y inity))
      (methods
         (translate (dx dy)
            (c (+ (field this x) dx) (+ (field this y) dy)))
```

```
         (scale (s)
            (c (* (field this x) s) (* (field this y) s))))))
```

And here are some sample outputs:

```
OBJ> (define pr! (function (coord) (print! (field coord x) (field coord y))))
pr! defined
OBJ> (define e (coordinates 10 20))
e defined
OBJ> (pr! e)
10 20
None
OBJ> (pr! ((method e translate) 100 200))
110 220
None
OBJ> (pr! e)
10 20
None
OBJ> (pr! ((method e scale) 2))
20 40
None
OBJ> (pr! ((method e scale) 5))
50 100
None
```