Records and Objects

March 26, 2020

Riccardo Pucella

In the days of yore

We extended our object language FUNC with *mutable state*

- explicit reference cells
- variant with implicit mutable variables using a syntactic transformation to reference cells

Today:

- we add records to FUNC
- we add objects to FUNC

Records

A record is a structure that maps **field names** to values

{ a: 10, b: 20, c: "hello", d: true }

Field names are like identifiers within a record

Record construction: { *id* : *expr*, … }

Field access: *record_expr.id*

Records vs dictionaries

The distinction is subtle

- a record maps **names** to values
- field lookup name *rec.f* is *syntax*
  - you evaluate *rec* to a record
  - you don't evaluate *f*
- found in statically typed languages
  - can associate a type to each field of the record
  - and the system can track it

E.g., records in OCaml

Records vs dictionaries

The distinction is subtle

- a dictionary maps values to values
  - can map strings (in lieu of names) to values
- field lookup *rec[f]*
  - you evaluate *rec*
  - you evaluate *f*
  - you index based on the value of *f* -- *rec["hello"]*
- found in dynamically typed languages

E.g., Python, JavaScript

VRecord

```
class VRecord (val entries : List[(String, Value)])
                                                extends Value {
  ...
  override def getBindings () : List[(String, Value)] =
    entries

  override def lookup  (id: String) : Value = {
    for (entry <- entries) {
      if (entry._1 == id) {
        return entry._2
      }
    }
    runtimeError("Runtime error : unbound field " + id)
  }
}
```

Surface syntax for records

```
expr ::= ...
         ( record ( id expr ) ... )
         ( field expr id )
```

`(record bindings)` creates *ERecord(bindings)*

`(field expr id)` creates *EField(expr, id)*

ERecord

```
class ERecord (val entries : List[(String, Exp)])
                                              extends Exp {
  ...

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

EField

```
class EField (val record : Exp, val field : String)
                                                extends Exp {
  ...

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

Opening a record

One things that records allow you to do is to **open up** a record so that you can use its fields as though they were identifiers in the environment

```
(open (record (a 10) (b 20) (c "hello"))
   (+ a b))
```

→ 30

EOpen

```
class EOpen (val record: Exp, val body: Exp) extends Exp {
  ...

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

Obvious surface syntax **(open** *expr expr***)**

Functions in records

Nothing prevents us from having functions stored in the fields of a record

```
(record (add (fun (a b) (+ a b)))
        (mult (fun (a b) (* a b))))
```

Sort of a baby module system

We can even be more clever, and hide **local state** in the record

Point example

```
(let ((x (ref 0)) (y (ref 0)))
  (record
     (getX (fun () (get x))
     (getY (fun () (get y))
     (move (fun (dx dy)
              (do (put x (+ (get x) dx))
                  (put y (+ (get y) dy)))))))

((field pt getX)) → 0
((field pt move) 2 3) → update x & y
((field pt getX)) → 2
```

Point example

```
(let ((x (ref 0)) (y (ref 0)))
  (record
     (getX (fun
     (getY (fun
     (move (fun
              (do
                                                  )))
```

This is starting to feel very much like an object, with the fields of pt holding methods

```
((field pt getX)) → 0
((field pt move) 2 3) → update x & y
((field pt getX)) → 2
```

Object-oriented programming

Objects:

- set of functions (methods) having access to some shared encapsulated state

Models:

- class-based: a **class** is used as a template for objects (Smalltalk, Java, C++, Python)
- prototype-based: objects are **cloned** from existing objects (Self, JavaScript)

Classes

Template:

- per-object local fields (instance variables)
- functions seeing local fields (methods)
- functions to create objects (constructors)

Classes are often **declarations** - since they usually define a type

We're not going to worry about classes as declarations

Simple classes

A class is just a constructor for objects

- arguments to the constructor
- local fields with initial values
- methods list

Surface syntax:

```
(class (id ...) ((fieldname expr) ...)
   (methodname (id ...) expr)
   ...)
```

Creating an object

Expression **new** taking a class and arguments, and calls the constructor with those arguments.

Surface syntax:

```
(new expr expr ...)
```

Calling a method

Need to get a method out of the object, and then can apply it like any other function.

Surface syntax:

```
(method expr id)
```

E.g. calling method *foo* with args 1, 2, 3:

```
((method obj foo) 1 2 3)
```

Accessing object from a method

Subtlety — for a method to invoke another method on the same object, we need a way to refer to the **current object** from within a method

Two approaches:

- explicitly pass the current object as an argument to the method (Python)
- implicitly make the current object available via a keyword (**this** in Java)

Implementation

We could create values

VClass, **VObject**

and expressions

EClass, **EObject**, **EMethod**

and we may want to do that anyway

(needed if we implement a type system)

But we can also do it via parser transformations

Encoding objects by example (1)

```
(class (a) ((f (ref a)))
   (value () (get f))
   (add (x) (put f (+ (get f) x))))
```

⇒

```
(fun (a)
  (let ((f (ref a)))
     (record
       (value (fun (this)
                (fun () (get f))))
       (add (fun (this)
              (fun (x)
                (put f (+ (get f) x))))))))
```

Encoding objects by example (2)

```
(new cls 3)
```

⇒

```
(cls 3)


(method obj add)
```

⇒

```
((field obj add) obj
```

Inheritance

Object systems also generally implement a form of inheritance

- code reuse mechanism that lets an object "inherit" implementation from an existing object
- often makes the new object a "special case" of the inherited object (subclassing)
- generalizes to multiple inheritance

Inheritance

Easiest implementation:

- if A inherits from B:
  - create an instance of B inside an instance of A
  - when looking up method m in A, look in B if the method doesn't exist in A
- need to manage creating the instance of B inside A

Details left to you — this may be the start of an interesting project on implementing a full object system