Interpretation and Compilation of Programming Languages
Part 9 - Object-oriented languages and recursive types

João Costa Seco

May 28, 2014

1 Introduction

Object-oriented programming combine functional and data abstraction with one essential imperative feature, object identity and self reference (the keyword `this`). From previous lectures we take both kind of fundamental and orthogonal abstraction mechanisms. However, the `this` keyword, denoting the object itself cannot be encoded using any existing feature. We do need recursive definitions. Consider the recursive construction `declrec` introduced before (LN07 will be updated to include it). We can encode an object construction expression (similar to Javascript) such as

```
{ a = 0, b = fun x -> this.a + 1 }.b(0)
```

into a recursive definition

```
decl o =
  declrec this = { a = 0, b = fun x -> this.a + 1 } in this in
o.b(0)
```

In this case, the identifier `this` denotes a record with a closure associated to label `b`. The closure refers to a environment that recursively contains the association of `this` to the record itself. This approach is an encoding, that adds new features to a language by encoding it into existing features. Although it is possible, in general, to preserve the semantics of a construct in the encoding, that is not generally true for the typing relation.

```
data Value =
    Integer Int
    ...
  | ObjectV [(String,Value)]
```

Figure 1: Abstract syntax for objects

Notice that in a *call-by-value* evaluation strategy the keyword can only be used when guarded by a function declaration. Otherwise the evaluation does not terminate. Using a lazy evaluation strategy one can define mutually recursive values without entering an infinite loop. One example is available in Figure 2 described below.

In order to define the operational semantics of a language with objects, we first define the abstract semantics, that is exactly the same as for records, see Figure 1. The operational semantics we present here does not use the general recursion as above, but custom modifies the environment to introduce the cycle in the environment definition. In Figure 2 the definition of the result `record` is based on an association between `labels` and `values`, whose definition uses record. The recursivity here is obtained by the lazy evaluation of the base language (Haskell). Functions `unzip` and `zip` are used to separate labels from field expressions, and vice-versa.

When using a *call-by-value* strategy, like in OCaml or Java, one may use a long known technique called the Landin's knot, that consists in using memory references to make a loop using memory references, and make the functional program follow them. Notice that a Landin's knot is not typable using a strict type system without recursive types. Figure 4 shows that using an abstract representation for objects using a state variable, declared in Figure 3, first declares an empty object that gets stored in all closures of the object, and then gets an updated the list of fields. Unguarded evaluations of the identifier `this` will find an empty list of fields, but all others will follow the loop created by memory references that lead to the intended object value.

```
eval (Object fields) env mem =
          if Undefined `elem` values
             then (Undefined,mem)
             else (object, mem)
          where
          object = ObjectV $ zip labels values
          values = map (\e -> Delay e envr) fieldExps
          envr = assoc "this" object $ beginScope env
          (labels,fieldExps) = unzip fields
```

Figure 2: Operational semantics for objects

```
type result =
  | Integer of int
  ...
  | ObjectV of ((string * result) list) ref
```

Figure 3: Abstract syntax for objects OCaml

2 Inheritance

Implementation inheritance (which is a different thing from subtyping) can be implemented using objects alone. This style is based on the so-called prototypes, and can be found in languages like smalltalk and Javascript.

The main effect is that, in the absence of a field in the definition of an object, its definition is imported from the prototype (or in class based languages, from the super class). Consider the following encoding for an object that works by copying all the fields

```
decl prototype = {
    counter = var(0),
    inc = fun this -> fun y -> counter := !counter } in
decl object = {
    counter = prototype.counter,
    inc = prototype.inc,
    get = fun this -> fun y -> !counter,
    dup = fun this -> fun y -> this.inc(0); this.inc(0) } in
object.dup(object, 0)
```

Notice that, instead of being based in a recursive definition as before, methods are parameterized, and method invocation needs to point out the

```
let eval e env =
  match e with
  ...
  | Object le ->
      let r = ref [] in
      let env' = assoc "this" (ObjectV r) (beginScope env) in
    let lv = List.map (fun (l,e) -> (l,eval e env')) le
    in r := lv; ObjectV r

  | Select (e,l) ->
      (match c_eval e env with
        ObjectV lv -> (try
                  lookup l !lv
                with
                 Not_found -> Undefined)
    | _ -> Undefined)
```

Figure 4: Operational semantics for objects (OCaml)

target object. This is true for instance in Javascript, where it is possible to call a function in a primitive way and explicitly saying the object that will be bound to the name `this`. A function in Javascript provides an apply operation taking two parameters, the target object and an array of the function arguments. Notice that in a statically typed language is able to statically produce the instantiation code that copies the prototype's fields. In a dynamically typed language must implement a runtime lookup inside the method invocation operation.

```
dup.apply(object,0)
```

Final note More reading about typing of objects, using recursive types, can be done using the set of slides made available through CLIP.