Object-Functional Programming (Rough Draft)

Charlotte Maia

November 23, 2009

Abstract

This vignette looks at combining the paradigms of object oriented programming and functional programming, with an emphasis on using object-level methods as functions. We create an object with at least one method, and treat the method as an object in it's own right. We then call the method as a function, without any reference to the container object, however the method can still access the container object's members.

Introduction

At the time of writing this vignette, the author is not aware of any standard definition of, or approach to, "object-functional programming". It might seem intuitive to think of object-functional as having objects which are functions, however this is somewhat trivial.

Here we present one approach (using the oosp package) where we regard object-functional programming as a programming paradigm that combines object oriented programming with functional programming.

One of the major principles of object oriented programming is encapsulation, where an object contains methods and attributes. Therefor, for a paradigm to be both object oriented and functional it should should incorporate this major principle.

The approach used here is that if we have an almost arbitrary object (it must contain at least one method), which we shall regard as a container object, then we can regard the methods of that object as objects in their own right. If we set things up correctly, then we can associate the method object with the container object, and invoke the method directly, in the same way that we would call any other function, without any reference to the container object, which we shall regard as a free method. (Note that free methods are not m class methods, however we can conveniently wrap an m object along with an m class method).

This approach allows us to avoid much of the computational cost associated with normal method despatch and can simplify both function calls and function definitions. A simple example is a lookup function, where in principle, all we need to provide (for each function call) is a key value.

For the simple case, we can implement a free method, by treating a function's environment as a container object, and storing attributes in that environment. If we wish to have a free method that represents an m object method, which we shall regard as a hypermethod, we still use the function's environment, however there are two container objects, the function's environment (which is becomes merely an implementation artifact), and the m object itself. More on this later...

Whilst not object-functional in the sense presented here, we can also this approach to implement C-like pointers (C-like in the sense we can casually reference and deference objects, and we have references to vectors that are vaguely similar to pointers to arrays). Here a pointer is a freemethod that when evaluated returns the object, however for lists and vectors, we provide extraction methods. More on this later too...

Free Methods

Here free methods are methods that are objects in their own right, have a container object and have access to all the container objects members, and can be called as ordinary functions, without reference to the container object. Standard free methods (using the freemethod function) which we are about to present, have their container object created automatically, the next section presents a more explicit approach. Free methods are implemented via environments, and before showing the direct freemethod approach, we will show the more fundamental approach, using standard R commands.

In R, functions have an environment (which we can assign). We can treat that environment as a container object, and hence the function as a method of that object. In the simplest sense we can give a function attributes (distinct from it's arguments). Let us consider an example that wraps R's match function. Typically when calling match, we would need to provide a vector (to look things up in) for each call:

```
> key = LETTERS [1:6]
> value = c ("A's value", "B's value", "C's value",
         "D's value", "E's value", "F's value")
> table = data.frame (key, value, stringsAsFactors=FALSE)
> table [match ("D", table [,1]), 2]
[1] "D's value"
```

Let us consider the case where (for each call) we provide the key values only:

```
> lookup1 = function (key)
 {       table = environment (sys.function () )$table
         table [match (key, table [,1]), 2]
 }
> e = new.env ()
> environment (lookup1) = e
> e$table = table


> lookup1 ("D")
[1] "D's value"
```

We can simplify this process (slightly). The oosp package provides the function freemethod. Here a dot is a self-reference. Attributes are accessed like elements of a list.

```
> f = function (key) table [match (key, .$table [,1]), 2]
> lookup2 = freemethod (f, table)
> lookup2 ("D")
[1] "D's value"
```

The first argument of the freemethod function, is the function we wish to turn into a freemethod. The remaining arguments are the attributes for the container object. Generally the naming is automatic, however where calling freemethod inside another function using ..., the attributes should be named, i.e. table=table, rather than just table.

Hypermethods

The m class system allows the creation of m objects. Whilst in principle m object contain methods, the methods are an illusion, and methods are stored at the class-level. We can use free methods to wrap (pretentiously) a m object method, by wrapping the m class method and taking care of the method despatch details (noting this is more efficient that normal method despatch, used by the m class system).

Let's say we have the following class:

```
> MyObject = function ()
 {
        MyObject = function () .$x = 1

        methoda = function () .$x = 2 * .$x
        methodb = function (y) .$x + y

        mclass ()
 }
```

We create the object in the usual way, however create hypermethods using the hypermethod function:

```
> obj = MyObject ()
> m1 = hypermethod (obj, "methoda")
> m2 = hypermethod (obj, "methodb")
```

We can then call the methods as functions:

```
> obj$x
[1] 1
> m1 ()
> obj$x
[1] 2
> m2 (10)
[1] 12
```

Pointers

There are many situations where we wish to create a reference to a single object. Many of those, the object is a vector. Pointer objects here (not strictly pointers) provide a simple mechanism for this.

Let's say we have a matrix, we can create a pointer to it, using the pointer function.

```
> mptr = pointer (matrix (1:16, nrow=4) )
> mptr
pointer:matrix
```

We deference the pointer, by calling it as a function:

```
> mptr ()
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
```

Extraction methods have been implemented, to make the pointers feel like pointers to arrays:

```
> mptr [,1] = 0
> mptr ()
     [,1] [,2] [,3] [,4]
[1,]    0    5    9   13
[2,]    0    6   10   14
[3,]    0    7   11   15
[4,]    0    8   12   16
> mptr.copy = mptr
> mptr.copy [,1] = 1
> mptr ()
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    1    6   10   14
[3,]    1    7   11   15
[4,]    1    8   12   16
```