Object-Functional Programming (Draft)

Charlotte Maia

February 6, 2010

Abstract

This vignette looks at combining the programming paradigms of object oriented programming and functional programming, arguably with a slightly stronger object oriented (rather than functional) flavour. There is a very strong emphasis on (1) enhanced functions (as objects), where functions can have self-references and attributes; and (2) supporting a diverse range of function-valued functions.

Introduction

At the time of writing this vignette, the author is not aware of any standard definition of, or approach to, "object-functional programming". Here we present a definition and an approach (via the oosp package), where we define object-functional programming as a programming paradigm that combines object oriented programming with functional programming.

Arguably, the approach used here is closer to object oriented programming, in that we build our approach on top of object oriented principles, however do not strictly adhere to functional principles. However, like functional programming, functions are not only objects, they are the main kind of object. Whilst we allow computation in a variety of ways, we encourage computation via functions and support function-valued functions.

Rather than simply having a programming language that provides some degree of support for both object oriented programming and functional programming (which R, indeed has), or merely treat functions as objects (which is kind of trivial really), here we create enhanced kinds functions, where functions (as serious objects) can have self-references and attributes (and potentially methods).

A similar idea is seen in R's splinefun and ecdf functions. Here wish wish to develop this idea further.

The Fundamental Approach

Let consider this idea, using standard R commands (noting that we are going to make use of environments). 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:

```
> #simple use of match
> 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 ("A", table [,1]), 2]
[1] "A's value"
```

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

```
> #slightly enhanced form
> lookup1 = function (key) d [match (key, d [,1]), 2]
> e = new.env ()
> environment (lookup1) = e
> e$d = table

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

The Enhanced Approach

We can simplify this process (slightly). The oosp package provides the function FUNCTION, for defining enhanced functions. This function takes care of the environment issues for us.

```
> #more enhanced version
> f = function (key) table [match (key, d [,1]), 2]
> lookup2 = FUNCTION (f, d=table)

> lookup2
FUNCTION (key)
attributes:
[1] "d"

> lookup2 ("D")
[1] "D's value"
```

The current preference of the author (and this issue is still being explored), is to use an explicit self-reference operator (here a single dot), which is used like a list. This is clearer (with respect to the function body), and allows a function to have attributes and arguments with the same name. Plus at present, there is no guarantee that the way that attributes are stored will stay the same, hence the following approach is recommended.

```
> #safer version
> f = function (key) table [match (key, .$d [,1]), 2]
> lookup3 = FUNCTION (f, d=table)

> lookup3 ("E")
[1] "E's value"
```

More on Attributes (and Methods)

Sometimes we want to access (including modify) the attributes of an enhanced function. This is done the same way as a list or environment.

```
> lookup3$d
```

```
  key     value
1   A A's value
2   B B's value
3   C C's value
4   D D's value
5   E E's value
6   F F's value
```

We can include a function as (roughly speaking) an attribute, arguably as a method. However, there are some special issues which are still being explored.