Object-Functional Programming (Draft)

Charlotte Maia

May 8, 2010

Abstract

This vignette presents a hybrid programming paradigm, that combines the strengths of object oriented programming with functional programming. The paradigm focuses on two main concepts: firstly, enhanced R functions, where functions are treated as objects, potentially with self-references and attributes; and secondly, 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 ofp 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, and 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 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.

Note that in order to build a useful object-functional system, not only do we need the features described in this vignette, we also need more general object oriented features (such as those described in the other vignettes), as well as more general computational features (such as vectorised computation, which is not discussed).

Enhanced Functions

Enhanced functions are created with the FUNCTION function. Creating FUNCTIONs is similar to creating VECTORs described in the first vignette. Rather than providing a seed vector, we provide a seed function, along with any attributes that we require. Here, is an example, for a lookup function.

```
> #first, a suitable data structure to look things up in
> 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
```

```
  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
```

```
> #second, the function itself
> f = function (key) table [match (key, d [,1]), 2]
> lookup = FUNCTION (f, d=table)
```

Calling the function.

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

Sometimes me may wish to have a function, where an attribute name is the same as an argument name. Probably not the best design pattern. However it can still be achieved using a self reference.

```
> f = function (x) .$x + x
> f = FUNCTION (f, x=10)
> f (2)
[1] 12
```

Function Valued Functions

Here, we consider function valued functions achieved via setting attributes (it can also be achieved via symbolic manipulation). Here's a very simple example for creating a second degree polynomial.

```
> poly2 = function (a, b, c)
 {        f = function (x) a + b * x + c * x^2
         FUNCTION (f, a, b, c)
 }
> p = poly2 (0, 0, 1)
> p (1:10)
 [1]   1   4   9  16  25  36  49  64  81 100
```

Noting that FUNCTION, is also a function valued function.

More on Attributes

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

```
> f$x
[1] 10
```

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.