package: ofp 0.6.0

Object-Functional Programming (Draft)

Charlotte Maia

April 12, 2011

This vignette introduces the second part of the ofp package, a framework for object-functional programming.

Introduction

We consider object-functional programming, a programming paradigm that combines the strengths of object oriented programming with the strengths of functional programming. Object-functional programming builds on the ideas in the earlier vignette.

The current view of the author, is that object-functional programming revolves around certain concepts:

1. Functions are objects.
   They can have and do anything that any typical "object" can have and do, constructors, inheritance and attributes.
2. Functions can return other functions.
3. Functions can be evaluated directly.

Here, we do not require that functions are strictly functions in the mathematical sense, however this is still a desirable property. i.e. We allow functions to modify state, or to return different values given the same arguments.

To support these ideas, we consider enhanced functions. These are extended R functions, that are given their own environment (which we shall regard as a container object). We can assign values to the container object, and regard those values as function attributes.

A similar idea is seen in R's splinefun and ecdf functions.

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
```

Noting that we can print the function and access the attributes directly

```
> f
FUNCTION (x)
.$x + x
attributes:
x
> f$x
[1] 10
```

Extending Functions

Extending a function could mean different things. It could mean extending it's class attribute and giving it further attributes. It could mean changing or extending the body of the function. It could even mean changing or extending it's attribute list.

Here, we regard extending a function, as a combination of extending it's class attribute, potentially giving it more attributes, and potentially changing the body of the function. If we do not wish to change the body of the function, then we can use the extend function in the usual way.

```
> f = function (x) x
> linef1 = extend (FUNCTION (f), "line")
> linef1
FUNCTION (x)
x
```

However, if we do indeed wish to change the body, then we need the extendf function, which is the same as the extend function, except that the third argument is a function with the new body.

```
> f = function (x) a + b * x
> linef2 = extendf (linef1, "fancyline", f, a=0, b=1)
> linef2
FUNCTION (x)
a + b * x
attributes:
a b
```

S3 Methods

We can create S3 methods for functions, in the usual way.

```
> print.fancyline = function (f, ...)
          cat ("fancyline:", f$a, "+", f$b, "x\n")


> #same as print (linef2)
> linef2
fancyline: 0 + 1 x
```

Nested Functions

It's possible for a function to contain other functions (as attributes). If the child function needs to access the parent function's attributes, then the environment of the child function needs to be set the environment of the parent function.

```
> f = function (x) g (x)
> g = function (x) 2 * x + k
> f = FUNCTION (f, g, k=2)
> environment (f$g) = environment (f)


> f (4)
[1] 10
```