package: ofp 0.7.0

Object-Functional Programming (Draft)

Charlotte Maia

May 23, 2011

This vignette introduces the ofp package, for object-functional programming. Object-functional programming is a programming paradigm that mixes features of object oriented programming with features of functional programming. There's a strong emphasis on enhanced functions, that have the properties of both objects and functions.

Notes

- **This package has been reduced, with previous features shifted to the s3x package.**
- **Tools for differentiation, integration, smoothing and interpolation, planned for ofp 0.8.**
- **This package is mildly unstable and contains several experimental features.**

Introduction

This package attempts to create a hybrid programming paradigm (which the author likes to refer to as, object-functional programming), that mixes features of object oriented programming with features of functional programming. Note that the approach use here, is slightly closer to object oriented programming.

In object oriented programming, objects are constructed, they can be extended, and can have methods and attributes. In functional programming, data is manipulated by calling functions, that map input data to output data. Here, we define enhanced functions as objects (with standard features of objects, such as attributes), that are also functions (though not necessarily pure functions), and are called using the same syntax as typical functions (e.g. f (x)).

A major application of object-functional programming is modelling functions defined via interpolation. The author is considering several other applications, and is hoping to discuss them more in the next revision.

As with the s3x package, here object attribute, means any (nested) object, accessible via the $ operator, it doesn't refer to R attributes, as such.

Currently, object attributes are implemented as elements of the function's environment. The same approach is used in R's splinefun and ecdf functions. Note that this approach is being re-considered and may be changed in the near future.

Enhanced Functions

Enhanced functions are created with the FUNCTION function. We provide a seed function, along with any attributes that we require. Here's 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
```