package: ofp 0.6.0

Enhanced S3 Programming (Draft)

Charlotte Maia

November 27, 2010

This vignette introduces the first part of the ofp package, a framework for enhanced S3 programming.

Introduction

There are two major object oriented systems in R, S3 and S4. Arguably, S3 deviates from conventional object oriented programming, however is relatively simple, has an inherent functional flavour and is very flexible. In contrast, S4 is relatively complex, very structured and very verbose.

The first part of this package is designed to enhance S3 capabilities, firstly, giving it a more standard object oriented flavour, and secondly, emphasizing simplicity and readability.

The largest section of this vignette discusses enhanced primitives, which include:

- Enhanced lists.
- Enhanced environments.
- Enhanced functions.
- Enhanced vectors.

Simplified Constructors

A typical design pattern for a constructor, is a function (roughly speaking), that:

1. Creates an instance of the class (including creating an instance of any superclass, where often the superclass constructor is the first call within function).
2. Extends the class attribute (for S3).
3. Sets any further attributes.
4. Returns the object.

So a superclass/subclass example might be:

```
> point = function (x=0, y=0)
          structure (list (x=x, y=y), class=c ("point", "list") )
```

```
> circle = function (x=0, y=0, r=1)
 {        obj = point (x, y)
         class (obj) = c ("circle", class (obj) )
         obj
 }


> colouredcircle = function (x=0, y=0, r=1,
         line.colour="black", fill.colour="white")
 {        obj = circle (x, y, r)
         class (obj) = c ("colouredcircle", class (obj) )
         obj$line.colour = line.colour
         obj$fill.colour = fill.colour
         obj
 }
```

Mostly that's fine. However, in the superclass constructor, explicitly naming each argument of the list is cumbersome. Secondly, the subclass constructors are way too long. Using the ofp package, we can instead write:

```
> point = function (x=0, y=0) extend (LIST (x, y), "point")


> circle = function (x=0, y=0, r=1) extend (point (x, y), "circle", r)


> colouredcircle = function (x=0, y=0, r=1,
         line.colour="black", fill.colour="white")
         extend (circle (x, y, r), "colouredcircle", line.colour, fill.colour)


> colouredcircle (fill="blue")
$x
[1] 0

$y
[1] 0

$r
[1] 1

$line.colour
[1] "black"

$fill.colour
[1] "blue"

attr(,"class")
[1] "colouredcircle" "circle"         "point"          "LIST"
[5] "list"
```

Alternatively (for the one of subclass constructors):

```
> circle = function (x=0, y=0, r=1)
 {        obj = extend (point (x, y), "circle")
         implant (obj, r)
 }
```

We shall discuss LIST objects later. The import parts are the extend and implant functions, which serve to make our code succinct.

The extend function, is intended to take an object (as it's first argument), the name of the subclass (as it's second argument), and potentially further arguments representing attributes for the object. Then it returns the extended object.

The implant function is almost identical to the extend function, except that it doesn't take the name of a subclass as an argument, and doesn't adjust the class attribute.

One restriction, is that we can not call either extend or implant using dots. The following is not allowed:

```
> extend (circle (x, y, r), "colouredcircle", ...)
```

One further word of warning. The view of the author, is that it's advisable to always extend the class attribute, by appending a value to it, rather than simply setting it to some scalar value.

Occasionally, R programs use calls such as `inherits (obj, "something")`, which may fail to produce the expected result, if we simply do something like `class (obj) = "circle"`.

Object References

There are many situations where we wish to create an object reference. The objref function allows us to do this (noting that environments, discussed later, provide a more efficient and flexible system). To create an object reference, we call objref with an object as it's single argument. The reference that is returned, it actually a function itself, which when evaluated (with no arguments) returns the object.

Let's say we want an object reference to a matrix.

```
> m = objref (matrix (1:16, nrow=4) )
> m
objref:matrix
> m ()
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
```

To create multiple references to the same object:

```
> q = m
```

Extraction methods have been implemented, to simplify working with references to lists and vectors. These also provide the main process for modifying the object.

```
> m [1, 1]
[1] 1
> m [1, 1] = 0
> m [1, 1]
[1] 0
```

Because it's a reference, dereferencing q, will yield our modified matrix:

```
> q [1, 1]
[1] 0
```

Noting that the following are not equivalent.

```
> m () [1, 1] = 1
> m [1, 1] = 1
```

Note that the objref system may be changed in future versions.

Enhanced Primitives

Enhanced Lists

We touched on enhanced lists earlier, namely the LIST object. The main purpose of LIST objects is to remove the need to explicitly name each argument in the list. So the following:

```
> x = 1
> y = 2
> obj = list (x=x, y=y, z=3)
> obj
$x
[1] 1

$y
[1] 2

$z
[1] 3
```

Can be simplified to (noting the LIST arguments):

```
> x = 1
> y = 2
> obj = LIST (x, y, z=3)
> obj
$x
[1] 1

$y
[1] 2

$z
[1] 3

attr(,"class")
[1] "LIST" "list"
```

Note that the current version prevents creating enhanced lists with dots as an argument (unless we specify the call object, see the Rd file, which is slightly complex). This feature is may be changed in future versions. So the following is not allowed:

```
> f = function (...) LIST (...)
> f (x, y)
```

Enhanced Environments

Extending environments is relatively simple, however it doesn't seem to be common practice. One reason we might want to extend an environment is to write our own print method.

```
> e = extend (new.env (), "myenv")
> print.myenv = function (e) print ("a myenv object")
> e
[1] "a myenv object"
```

The ofp package, also provides ENVIRONMENT (enhanced environments) objects, that extend environments. These provide workarounds to some of the limitations of standard environments. Furthermore, ENVIRONMENT objects can be extended. A problem with using standard environments is that creating an environment required separate calls, for each assignment. Using ENVIRONMENT objects we can create environments in a similar way to lists. Presently, ENVIRONMENT objects, print themselves, by calling as.list, however it's not recursive (so nested environments print the usual way).

```
> e = ENVIRONMENT (x=1, y=2, z=3)
> e
$z
[1] 3
$y
[1] 2
$x
[1] 1
```

We can compare two ENVIRONMENT objects for equality:

```
> e = f = ENVIRONMENT ()
> g = ENVIRONMENT ()
> e == f
[1] TRUE
> e == g
[1] FALSE
```

Extending ENVIRONMENT objects is trivial:

```
> e = extend (ENVIRONMENT (), "myenv2")
```

Enhanced Functions

Enhanced functions, are created with the FUNCTION function. This is discussed in the next vignette.

Enhanced Vectors

The purpose of enhanced vectors is to support the use of vectors with attributes. R already allows vectors to have attributes, however the process is not as simple as accessing the elements of a list. So we provide enhanced vectors, using list-like syntax. There are currently five kinds of enhanced vectors, TEXT, REAL, COMPLEX, INTEGER and LOGICAL, where TEXT extends character vectors, and REAL extends numeric vectors. To create an enhanced vector, with some attributes.

```
> x = INTEGER (1:10, someattribute=TRUE, someotherattribute=FALSE)
> x
```

```
INTEGER
 [1]  1  2  3  4  5  6  7  8  9 10
attributes:
someattribute someotherattribute
> x$someattribute = FALSE
> x$someattribute
[1] FALSE
```

Alternatively, we can create a vector by omitting the first argument and providing a dimension value. Noting that there are currently some problems using this approach to create matrices.

```
> INTEGER (dimension=2)
INTEGER
[1] 0 0
```

The process to create the other vectors is the same.

Simplified Methods

In the earlier section on constructors, we created a point class, now let's create a method, an intuitive one...

```
> #a possible print method
> print.point = function (p, ...) cat ("x:", p$x, "\ny:", p$y, "\n")

> p = point (0, 0)
> p
x: 0
y: 0
```

At face value, if works fine. However, let's try and make a package...

```
> R CMD check My1stRPackage
* checking S3 generic/method consistency ... WARNING
print:
  function(x, ...)
print.point:
  function(p)
```

After a few changes...

```
> #another possible print method
> print.point = function (x, ...) cat ("x:", x$x, "\ny:", x$y, "\n")
```

Now, R Check is content, however I don't want to call my object x, I want to call p. So the ofp package implements mask functions, that "mask" a subset of the standard generics. In principle, we should still include the dots argument, however otherwise we can use what ever arguments we want. Currently (these may change) the ofp package masks print, summary, format, plot, lines and points. Now, if we load ofp, we can use p instead of x, and R Check is still content.

One can mask other generics, say mean, using a declaration such as:

```
> mean = function (...) base::mean (...)
```

This will create some overhead, however using method despatch at all, creates overhead. R check will think that mean is a regular function, rather than a generic. Noting that calling mean.myobject will still call the mean generic.