Introduction to The "ofp" Package (Draft)

Charlotte Maia

May 8, 2010

Abstract

This vignette introduces the ofp package, along with it's most fundamental features.

Introduction

Despite the name of the package, the package's main focus is on R programming that's both object oriented and highly computational.

The package includes two frameworks for object oriented programming, firstly, a framework for environment-based programming (supporting object references), and secondly, a framework for object-functional programming. The package also includes enhanced list, environment, function and vector objects, as well as simplified processes for object construction, a work around to R check restrictions on S3 method arguments, object references and some other highly experimental features.

In many respects, "object-functional programming" represents the culmination of all the features in this package, hence the author's decision to name the package after it (plus she just didn't like the sound of the alternatives...).

The following sections of this vignette, discuss object construction, method arguments, object references and enhanced list and vector objects. The other vignettes discuss environment-based programming and object-functional programming, along with enhanced environment and function objects, respectively.

As mentioned above, the package also contains some other highly experimental features (intended as future utility classes/functions). Users are recommended not to use features, other than those discussed in the vignettes.

It's worth noting that the view of the author is, that in producing a language (or language extension or library) to support programming that's both object oriented and highly computational, such a language should:

- Support code that is succinct, human-readable and flexible, in turn supporting a simple and relatively direct mapping from mathematical model to program code, where the program code is clearly reflective of the mathematical model.
- Expanding on the point above, issues of elegance, robustness and consistency, should not be enforced by the language, rather be optional extras, applied if and only if a programmer so desires.
- All mathematical primitives, should be "objects" and be capable of having attributes, that can be accessed in a simple way.

Note that we use the term "attribute" in it's more general object oriented way (not it's traditional R way). For lists and environments we will implement attributes as elements of those objects. For vectors and standard functions, they're R attributes. For enhanced functions, well, we'll get there...

Succinct 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".

Unrestricted S3 Method Arguments

In the previous section, we created a point class, not 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, points, fitted, residuals, as.list, as.data.frame and mean.

Now, if we load ofp, we can use p instead of x, and R Check is still content.

Object References

There are many situations where we wish to create an object reference. The objref function allows us to do this (noting that the vignette on environments discusses a more flexible and efficient 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:

```
> x = y = 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, dereference either x or y, will yield our modified matrix:

```
> x [1, 1]
[1] 0
```

Noting that the following are not equivalent.

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

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, 4)
> obj
$x
[1] 1

$y
[1] 2

$z
[1] 3

[[4]]
[1] 4
```

Can be simplified to (noting the LIST arguments):

```
> x = 1
> y = 2
> obj = LIST (x, y, z=3, 4)
> obj
$x
[1] 1

$y
[1] 2

$z
[1] 3

[[4]]
[1] 4

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

Note that the current implementation is limited when calling LIST with dots. The extend and implant functions used earlier, actually make use of LIST and prohibit the use of dots. LIST itself, will allow dots, however automatic naming does not work.

```
> #calling LIST with dots
> f = function (x, ...) LIST (x, ...)


> #calling LIST indirectly with named arguments
> #works...
> f (1, y=2)
$x
[1] 1

$y
[1] 2

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


> #calling LIST indirectly unnamed arguments
> #fails...
> y = 2
> f (1, y)
$x
[1] 1

$..1
[1] 2

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

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 a list-like syntax. There are currently four kinds of enhanced vectors, LOGICAL, TEXT, INTEGER and REAL, where TEXT extends character vectors, and REAL extends numeric vectors. Rational and enhanced complex vectors may be added in the next version. Call, function, and polynomial vectors are being considered.

To create an INTEGER 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

$someattribute
[1] TRUE

$someotherattribute
[1] FALSE
```

```
> x$someattribute = FALSE
> x$someattribute
[1] FALSE
```

Alternatively, we can create a vector (including a matrix) by omitting the first argument and providing a dimension value.

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

The process to create the other vectors is the same.