package: s3x 0.1.0

Enhanced S3-Based Programming (Draft)

Charlotte Maia

May 23, 2011

This vignette provides an overview of the s3x package, for enhanced S3-based object oriented programming. A major goal of the package, is to support R programming that mixes object oriented programming with mathematical programming. The package builds on S3's system for class definition and method despatch, however provides utilities to simplify construction; it provides enhanced primitives (enhanced lists, environments and vectors), that further simplify construction and use an alternative form of attributes; plus it provides utilities for object referencing.

Important Notes

- **This package is a fork from the ofp package.**
- **This package is mildly unstable and contains several experimental features.**

Introduction

Roughly speaking, S3 represents the third version of the R language, with S3 adding object oriented capabilities to R. Unlike many object oriented programming languages, one can create an object, without defining a class. However, consistent with strongly object oriented languages, everything, even a simple number such as *1*, is an object. Hence, we can create a *1* object and assign attributes to it. S3 allows a programmer to define a class attribute, which in turn, supports method despatch. As a consequence of this, one can create an object of class X, then later change to same object to class Y.

One of the major strengths of R, is its support for mathematical programming, including strong support for symbolic programming, functional programming and vectorised programming. The power to combine object oriented programming (especially object oriented semantics) with mathematical programming (especially mathematical semantics), is a major goal of this package. Whilst R (and it's S3 system) support this goal, the author has three major concerns:

1. The source code required for a typical constructor is verbose, especially when one considers inheritance. This is common in class-based languages, however the view of the author is that verbose constructors tend to obscure mathematical meaning. Hence, mathematical models and object oriented implementations of those models, don't resemble each other.

2. Often, if one creates a class-based model with attributes, one would implement an object (of that class) as a list with it's attributes implemented as list elements (rather than R attributes), hence a class (or object) attribute has a different meaning in the context of a class-based model than it does in the context of the R language. However, in contrast, if one used a vector (rather than a list), a class attribute maps directly to an R attribute, further confusing the notion of an attribute. In the case of a list, an attribute (an element, from R's point of view) can be accessed, simply,

using the $ operator. However, in the case of a vector, an attribute is accessed in a somewhat awkward way, using the attributes or attr functions.

3. In S3, there's no direct support for object referencing, however it can be accomplished (indirectly) by creating an environment object. For basic referencing this works relatively well, however for more complex referencing, this becomes awkward, due to a lack of support features. e.g. Testing for equality of two environments, produces an error.

Further concerns include:

1. A major application of R, is processing data in tabular form, however the author has the following specific concerns:
   (a) The standard object for representing tabular data is a data.frame, however many data.frame features don't support object oriented programming well. e.g. By default, appending object to a data.frame (as a column), strips the attributes of that object. Also, data.frame(s) suffer from the same problems as vectors, when it comes to accessing attributes, namely the need for attributes or attr functions.
   (b) The standard function for mapping a data file (in a package) to an object (namely a data.frame), is the data function. However, the data function does not assign the object to a user-defined identifier (which is what one would expect in object oriented programming), rather it assigns it to a default identifier, and creates the object in the current environment.
   (c) There's a lack of (non-random) sampling functions, in the standard version of R. Nonrandom sampling is very useful when working with moderate to large datasets, especially when one wishes to reproduce data (or a sample of some data) in a report. i.e. If one has a table with a thousand rows, typically one would not want to reproduce the entire table in a report.
2. Method despatch, is based on the idea of generic functions. One problem that arises (if one uses a generic defined by someone else, including the generics from R's base package), is that a method's argument names (and the order of those arguments) are constrained by the generic. Whilst argument names may be meaningful in the context of the generic, they are not always meaningful (and sometimes very confusing) in the context of it's methods.

To address these concerns the s3x package implements:

1. Utility functions, extend and implant, that allow object construction (including element/attribute assignment) potentially in one line.
2. Enhanced lists, enhanced environments and enhanced vectors, that also allow objection construction (including element/attribute assignment) potentially in one line, plus provide a unified and simple attribute system (referred to as object attributes, in contrast to R attributes), whereby all enhanced primitives, have a $ operator defined, to access those attributes. Note, that the term "object attribute" is used in this package to describe any nested object accessible via the $ operator.
3. Enhanced environments, provide features to make them more suitable as object references.
4. In addition to enhanced environments, the package provides the functions, objref and deref, to further simplify object referencing (mainly in prototypes and top-level algorithms).
5. A temporary function (maybe changed in near future) datafile, for loading datasets to an object. Not discussed here, refer to man page, for datafile, for information.
6. A generic function samp, and several methods, for non-random sampling.

7. Temporary mask functions (may also be changed in near future) for handling some of the problems with generics arguments.

Enhanced table objects (to supersede data.frame(s)) are being considered for the near future.

Constructor Utilities

A typical design pattern for an S3 constructor, is a function that:

1. Creates an instance of the class, possibly by calling a superclass constructor.
2. Sets or concatenates the class attribute.
3. Assigns any elements/attributes, along with any associated computation.
4. Returns the object.

So a superclass/subclass example might be:

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


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

That's ok. One possible simplification, is to use the structure function. However, there are still two problems, firstly, naming each argument in the list is verbose, secondly, setting the class attribute is also verbose. Using the extend function, instead, we can write:

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


> circle (10, 10, 2.5)
$x
[1] 10

$y
[1] 10

$r
[1] 2.5

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

The extend function takes two or more arguments. The first argument is a seed object (to be extended), the second is the name of the subclass. The remaining arguments are the object attributes (re-iterating that the term object attribute has a special meaning the s3x package). By default, the extend function produces an error, if "..." is included in the call. The arguments can be named or unnamed. If unnamed, they default to the corresponding identifier (i.e. in the example above, an object attribute named "x" is created and assigned the value of x).

In addition to the extend function, is the implant function, which is the same, except that there's no subclass argument. A point object (without setting the class) could be created as follows:

```
> point = function (x=0, y=0) implant (list (), x, y)
```

Note that both the extend and implant function, return the object.

Enhanced Primitives

Enhanced primitives are primitives with an extended class attribute (and support methods), enhanced constructors reflecting the approach in the previous section, and an alternative form of attributes. A common feature of all enhanced primitives is that the term "attribute" refers to an object attribute (i.e. an attribute defined in the context of a class or object model) rather than an actual R attribute (i.e. an attribute set using the attr function). Such attributes, are accessed via the $ operator, even for vectors. Currently, enhanced primitives include:

- Enhanced lists (LIST objects).
- Enhanced environments (ENVIRONMENT objects).
- Enhanced functions (FUNCTION objects), are implemented via the ofp package.
- Enhanced vectors (VECTOR objects), which include INTEGER, REAL, COMPLEX, TEXT and LOGICAL objects.

Each of these (except for functions), is discussed separately in the following sections. Note that the following are being considered for future versions of the s3x package:

- Enhanced matrix objects.
- Enhanced table objects.
- Enhanced call or expression objects.
- Enhanced exception objects.
- For enhanced vectors, rational and enumeration subtypes.

Enhanced Lists

The main purpose of LIST objects is to remove the need to explicitly name each argument in the list constructor. Given the following:

```
> x = 1
> y = 2
```

We can simply:

```
> obj = list (x=x, y=y, z=3)
```

To:

```
> obj = LIST (x, y, z=3)
```

Note that LISTs extend lists, and object attributes, correspond to it's elements.

Enhanced Environments

ENVIRONMNENT constructors follow the same convention as LIST constructors, plus provide a print method (non-recursive for environments). So we can write (using x and y, from the previous section):

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

Note that ENVIRONMENTSs extend environments, and environment attributes, correspond to it's elements.

We can also compare two ENVIRONMENT objects for equality:

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

Enhanced Vectors

The main purpose of enhanced vectors is to support the use of vectors with attributes. In contrast to enhanced lists and enhanced environments, enhanced vectors implement object attributes as R attributes (more precisely as elements of an R attribute named "."). In contrast to typical R vectors, enhanced vectors have a $ operator defined, to access their object attributes, removing the need to call the attributes and attr functions.

The VECTOR class is abstract and has five subclasses INTEGER, REAL, COMPLEX, TEXT and LOGICAL. Each class extends VECTOR. Each class also extends integer, numeric, complex, character and logical, respectively.

Each class, has two constructors, a standard constructor (with the same name as the class) that requires the length of the vector as it's first argument and a seed (or coercion) constructor (with the letter v appended) that takes an existing vector as it's first argument.

We can create enhanced integers as follows:

```
> x = INTEGER (10, some.attribute=TRUE, some.other.attribute=2011)
> y = INTEGERv (1:10, some.attribute=FALSE, some.other.attribute="abcdef")
> x
 [1] 0 0 0 0 0 0 0 0 0 0
INTEGER
object_attributes: some.attribute some.other.attribute
> y
 [1]  1  2  3  4  5  6  7  8  9 10
INTEGER
object_attributes: some.attribute some.other.attribute
```

We can access their attributes:

```
> x
 [1] 0 0 0 0 0 0 0 0 0 0
INTEGER
object_attributes: some.attribute some.other.attribute
> x$someattribute = FALSE
> x$someattribute
[1] FALSE
```

We can access and modify their elements in the normal way:

```
> x [1:5] = 2
> x
 [1] 2 2 2 2 2 0 0 0 0 0
INTEGER
object_attributes: some.attribute some.other.attribute someattribute
> x [4:7]
[1] 2 2 0 0
INTEGER
object_attributes: some.attribute some.other.attribute someattribute
```

Object References

Indirect References via Environments

Environments (both standard and enhanced) can be used for object referencing:

```
> e = ENVIRONMENT (x=0)
> pass.by.ref = function (ref) ref$x = 1
> pass.by.ref (e)
> e$x
[1] 1
```

Indirect References via The "objref" Function

The package implements the objref function (and deref function) to simplify object referencing, in prototypes and top-level algorithms. The functions simplifies syntax, however increase computational cost, hence should be avoided in high(er) performance code.

An object reference is created via the objref function. This function returns an extended environment. However, operators such as $ and bracket operators, apply to the referenced object. We can write:

```
> obj1 = obj2 = objref (1:3)
> obj1
objref ->  integer
> obj2
objref ->  integer
```

Intuitively:

```
> deref (obj1)
[1] 1 2 3
> obj1 [4] = 4
> deref (obj1)
[1] 1 2 3 4
> deref (obj2)
[1] 1 2 3 4
```

Nonrandom Sampling

The samp function produces non-random samples for table-based objects and vectors. For table-based objects it returns the first n and last m rows, for vectors, the first n elements and last m elements. By default, n=3 and m=n. Using the cars dataset (comes with R).

```
> samp (cars)
   speed dist
1      4    2
2      4   10
3      7    4
48    24   93
49    24  120
50    25   85
> samp (cars, 3, 1)
   speed dist
1      4    2
2      4   10
3      7    4
50    25   85
> samp (cars, 1)
   speed dist
1      4    2
50    25   85
```

Mask Generics

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 s3x 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 s3x package masks print, summary, format, plot, lines and points. Now, if we load s3x, 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 (...)
```

Note that this approach may be changed in the near future version (due to some undesirable side effects).