package: s3x 0.2.0

Enhanced S3 Programming (Draft)

Charlotte Maia

September 5, 2011

This vignette provides an overview of the s3x package. The package builds on top of R's S3 system for object oriented programming. Major focal points include simplified constructors, enhanced functions, enhanced vectors and object references.

Introduction

Roughly speaking, S3 represents the third version of the R language, with S3 adding object oriented capabilities to R.

Whilst S3 supports object oriented programming, standard S3 can be difficult to use. However, S3 has some redeeming features, not only does S3 support standard (class-based) object oriented programming, S3 also supports nonstandard object oriented programming, including list-based, table-based, object-functional and object-vectorised programming. This creates a number of interesting possibilities.

There are three over-arching goals of the s3x package. Firstly, to simplify standard object oriented programming, with particular respect to readability. Secondly, to support a model-based development approach, where in general, models come first and programs second and it's desirable for program semantics to clearly reflect their models. Thirdly, to support both standard and nonstandard object oriented programming, such that object oriented methodology can be used to design, reformulate and implement a wide range of models encountered in computer science, computer graphics, mathematics and applied statistics.

To support these over-arching goals, the package provides the following:

1. Utility functions, extend and implant, to simplify object construction.
2. Enhanced lists, functions and vectors, namely
   LIST, FUNCTION and VECTOR objects, respectively.
3. Utility functions, for working with tabular data (namely data.frame objects).
4. Environment-based object references.
5. Copycat generics, to remove constraints on S3 method argument names.
6. An alternative attribute system.
7. A $ operator, to access enhanced function and vector attributes.
8. Self-referencing, for functions to access their own attributes.

Note that author of the s3x package is also the author of the ofp package. The original ofp package contained some of the features of this package. The current ofp package is in a transitional state and

is intended to contain tools object-functional programming, such as tools for interpolation and function visualisation.

Also note that the package distinguishes between three kinds of attributes, "R attributes" which refer to standard R attributes, "object attributes" which refer to any nested objects accessible via a $ operator and "plot attributes" (not discussed). In this vignette, an attribute always refers to an object attribute, unless stated otherwise.

Construction Utilities

A typical design pattern for an R/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
 }
```

In theory this is fine. However, as one of the over-arching goals of this package is simplicity, the author feels that object construction should be more succinct. Hence, the s3x package provides the utility functions extend and implant.

Extend takes two or more arguments. The first argument is a seed object (to be extended), the second argument 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). For object's without an explicit class attribute (such as vectors), extend will set the class to the subclass. For object's with an explicit class attribute (such as data.frame(s)), extend will concatenate the class. Extend 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 following above, an object attribute named "x" is created and assigned the value of x). Extend returns the object.

This allows a single call, to set an object's class and assign attributes. The above example can be rewritten as:

```
> 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"
```

Implant 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 extend and implant are mostly restricted to "s3x-like" objects. An s3x-like object can be regarded as any object, where a $ operator can be used to set named elements of that object.

Enhanced Primitives

Enhanced primitives are (quite pretentiously) extended versions of R's standard primitives and include enhanced lists, functions and vectors. They are same as standard primitives, except that:

1. They have a capitalised class attribute, namely LIST, FUNCTION or VECTOR.
2. Enhanced lists and vectors have the same constructor arguments as standard lists and vectors, however enhanced functions are created using a somewhat different syntax.
3. A copycat generic system is used, so
   print.LIST calls s3x_print, which by default, calls s3x_print.LIST.
   This adds computational overhead, however removes constraints on S3 method argument names.
4. Enhanced functions and vectors, have an alternative attribute system, with a $ operator defined.
5. Enhanced function support self-referencing, currently read-only, so a function can (within the function body) access it's own attributes.

Enhanced tables (possibly extended data.frame objects) and rational types are being considered for the future. Note the following changes from earlier versions:

1. Constructors no longer support attributes, however one can still use extend or implant.
2. Mask generics, were removed.
3. Enhanced functions, no longer use environments.
4. Enhanced vectors, no longer have subclasses.

A discussion of function and vector attribute implementation issues are given in Appendix B. The vector subclasses were removed because many R features automatically change vector types (say from integer to numeric) and it's very difficult for subclasses support such changes.

Enhanced Functions and Vectors

Enhanced functions and enhanced vectors, support mixing object oriented programming with functional and vectorised programming. The general idea is that, one creates an object (say a function), does object oriented things with it (constructs it, extends it, gives it methods and attributes) and also does mathematical things with it (for a function, evaluate it, directly).

Trivial function example:

```
> f.seed = function () 1
> f = implant (FUNCTION (f.seed), my.name="trivial example")
> f$my.name
[1] "trivial example"
> f ()
[1] 1
```

More sophisticated function example (interpolation of two values, given a value between zero and one):

```
> interpolator = function (a, b)
 {        f.seed = function (x) .$a + x * .$interval
         implant (FUNCTION (f.seed), a, interval = b - a)
 }

> f = interpolator (10, 12)
> f
FUNCTION (x)
{  .$a + x * .$interval
}
object attributes:
a, interval

> f (0.25)
[1] 10.5
> f (0.75)
[1] 11.5
```

If we want, we can get a copy of the object attributes:

```
> objattr (f)
$a
[1] 10

$interval
[1] 2
```

Simple vector example:

```
> #simple distance class, no validation
> distance = function (x, unit="m")
         extend (VECTOR (x), "distance", unit)
> print.distance = function (x)
 {        print (as.vector (x) )
         cat ("unit =", x$unit)
 }

> #prototype conversion function
> in.mm = function (x)
 {        if (x$unit == "m") {x = x * 1000; x$unit="mm"}
         else stop ("unsupported unit")
         x
 }

> d = distance (25)
> d
[1] 25
unit = m
> in.mm (d)
[1] 25000
unit = mm
```

Note that arithmetic operations on two enhanced vectors, generally return an object with the class and attributes of the first vector.

Tabular Data Utilities

The function, read.package.data, can be used to create a data.frame from a package dataset. Another function, read.data.file can be used read other data files. These functions call read.table, however have different defaults. Refer to the man file for more info.

The samp function produces non-random samples for table-based objects and vectors. This is particularly useful to printing a snippet of a dataset in a vignette or other article. 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
```

Object Referencing

Whilst environments can be used directly, to support object referencing, this approach isn't discussed here. Instead we shall consider objref objects, which are extended environments, intended to mimic traditional object references. These objects aren't intended for high performance programs (standard environments should be used instead), they're intended purely for simplicity.

An object reference is created via the objref function. Various methods are defined, such as print, $ and bracket operators, Many of these apply to the referenced object. The object can be dereferenced using deref.

```
> #a reference
> ref1 = objref (1:3)
> ref1
objref -> integer
> deref (ref1)
[1] 1 2 3


> #another reference
> ref2 = ref1
> ref2
objref -> integer
> deref (ref2)
[1] 1 2 3


> #as both refer to same object, a modification in ref1's object is reflected in ref2
> ref1 [1] = 0
> ref1 [1]
[1] 0
> ref2 [1]
[1] 0
```

Copycat Generics

In R, a generic system is used for method despatch. Whilst this is considerably different from many other object oriented languages, the system is both convenient and effective.

A problem developing R packages is that R's generic system, places restrictions on S3 method argument names. A major example is print, which requires that the first argument is named x. This is problematic because we may wish to name our argument something else.

A number of workarounds have been considered. Previously, a masking system was used. The current approach uses a combination of copycat generics (e.g. s3x_print instead of print) and bridge functions, where a method (for the standard generic) calls the s3x generic.

Enhanced lists, have a print.LIST method. Calling print (with an enhanced list) calls print.LIST. Unlike a standard print method, this method simply calls s3x_print, which then calls s3x_print.LIST (or potentially a subclass method).

A full list of copycat generics and bridge functions is given in Appendix A.

Note that not all bridge functions have default methods associated with them. Enhanced lists also have a plot.LIST method that calls s3x_plot. As no default method is implemented, trying to plot an enhanced list will produce an error (unless a subclass provides a suitable method).

Appendix A: Copycat Generics and Bridge Functions

Currently, the s3x package contains the following copycat generics (aka s3x generics):

- s3x_print
- s3x_format
- s3x_as.data.frame
- s3x_plot
- s3x_lines
- s3x_points

Currently, the s3x package contains the following bridge functions (grouped by class):

- print.LIST
- format.LIST
- plot.LIST
- lines.LIST
- points.LIST
- print.FUNCTION
- format.FUNCTION
- plot.FUNCTION
- lines.FUNCTION
- points.FUNCTION
- print.VECTOR
- as.data.frame.VECTOR
- print.objref
- as.data.frame.objref

Appendix B:
Function and Vector Attribute Implementation Issues

Whilst R already allows a programmer to give an object, attributes, via the attributes and attr functions, the author has two concerns with the standard approach. Firstly, many R features automatically assign (or change) these attributes, hence, if a programmer decides that object X, should have attributes A and B, the actual object may have further attributes, say dim. Secondly, setting and accessing attributes, via attributes or attr is verbose.

For lists (R's general purpose objects), we can ignore these problems and implement an object's attributes as it's elements.

For (enhanced) functions and vectors, an R attribute "." is defined when the object is created and assigned an empty list. Object attributes are implemented as elements of that list. A function, objattr, can be used to get a copy of the entire list. Plus a $ operator is defined to allow setting and accessing those attributes.

It's convenient for functions (that is, their bodies) to be able to access their own attributes.

The simplest way this can be achieved, is by setting a function's environment (generally to a new environment) and assigning values to that environment. An alternative approach, is for a function to access standard attributes or ".").

Originally, the author used the first approach, however she had some concerns:

1. A simple assignment, say f=g, results in a mixed copy-by-value and copy-by-reference result.
   In most respects, function f is a copy of function g, hence modifying f's body or formal arguments, doesn't effect g. However, f's environment is the same as g's, hence whilst changes to f's body don't effect g, changes to it's environment will.

2. As standard R programming, uses copy-by-value, the use of environments, can be confusing.

3. Where a function's attribute, has the same name as another function, errors (which are not always obvious) can be produced.

Later, she changed to the second approach. When a function is created, a line is inserted into the seed function's body, to create a self reference object, also called ".". The self reference is simply a copy of the function's attributes. This approach isn't perfect:

1. If function f wishes to call function g and g needs access to f's attributes, then f needs to call g, with "." as an argument.

2. If a function wished to modify it's own attributes, there may be a substantial computational cost.

3. There's some increased computation cost,
   however for highly vectorised implementations, this cost should be negligible.

Currently, function's can't set their attributes, only access them.