package: ofp 0.5.0

Environment-Based Programming

Charlotte Maia

June 12, 2010

This vignette is the second vignette in the ofp package, an R package for object oriented R programming, with a dual focus on enhanced S3 programming and object-functional programming. Here, we first provide a basic introduction to standard environments, then consider enhanced environments.

Introduction

Environments are very important and very powerful objects in R. However, at the time of writing this vignette, the author is not aware of any introductions per se, on how to write programs using environments. The standard R documentation is perhaps the most useful:

```
> ?environment
> ?sys.call
```

Environments are really easy to use, and the R interpreter handles environment identifiers like object references, making certain programs workable, that would otherwise not be workable.

The vignette starts off looking at the fundamentals of environments, then considers the differences between lists and environments. It considers extending environments, mainly with ENVIRONMENT objects from the ofp package. It also looks at a simple example of implementing a binary tree, using environments.

Furthermore, environments can be hashed. However, this topic is not covered here.

Fundamentals

Perhaps the most important environment is the global environment. Using it is somewhat trivial. Here we create the name "x" and assign the value 1 to it.

```
> x = 1
> x
[1] 1
```

We can achieve the same thing using the following (noting that whilst we are using an environment, the syntax is similar to lists):

```
> .GlobalEnv$x = 1
> x
[1] 1
```

Yet another way of doing it, is by using the assign and get functions (noting there is an exists function too).

```
> assign ("x", 1, .GlobalEnv)
> get ("x", .GlobalEnv)
```

```
[1] 1
```

We can create our own environment very easily, using the function new.env:

```
> e = new.env ()
```

Plus manipulate it, in the same way as the global environment above:

```
> e$x = 1
> e$x
[1] 1
```

However, printing the environment is not always that informative:

```
> e
<environment: 0x9e14980>
```

After we assign values to the environment, we can get simple information using the length, ls and ls.str functions. A further option is as.list. Noting that the order of the output is not necessarily the order we expect:

```
> e$y = 1:10
> e$z = "Something"
> length (e)
[1] 3
> ls (e)
[1] "x" "y" "z"
> ls.str (e)
x :  num 1
y :  int [1:10] 1 2 3 4 5 6 7 8 9 10
z :  chr "Something"
> as.list (e)
$z
[1] "Something"

$y
 [1]  1  2  3  4  5  6  7  8  9 10

$x
[1] 1
```

Environments are also used to evaluate calls (to functions). There is a stack of environments (also called frames). For each function call, an environment is created and pushed onto the stack. In general this should be left alone, however just to illustrate:

```
> recursive.function = function (n)
 {        frame.number = sys.nframe ()
         current.frame = format (sys.frame (frame.number) )
         calling.frame = format (sys.frame (-1) )
         print (data.frame (n, frame.number, current.frame, calling.frame),
                 row.names=FALSE)
         if (n > 1)
```

```
        {        cat ("\n")
                recursive.function (n - 1)
        }
}
> recursive.function (3)
 n frame.number            current.frame              calling.frame
 3           17 <environment: 0xa32c9c4> <environment: R_GlobalEnv>

 n frame.number            current.frame            calling.frame
 2           18 <environment: 0xa269b1c> <environment: 0xa32c9c4>

 n frame.number            current.frame            calling.frame
 1           19 <environment: 0x9fe6fb8> <environment: 0xa269b1c>
```

The function above simply calls itself a certain number of times. In the output above, the current.frame is obviously the frame that is used within that function call, the calling.frame is the frame where the call was made.

Lists and environments have much in common, however there are important differences. Some of which are discussed in the next section. However the most important difference we will mention now. The R interpreter handles environment identifiers as object references. We could use the word "name" rather than word "identifier", as name matches the R language more closely, however the word name is somewhat ambiguous in this context, so for clarity we shall use identifier.

There are many important implications to having object references, the full scope of which is beyond this vignette. Hopefully the reader is already familiar with the notion of an object reference, however if not, here is a taste. In the following example we create four environments. At face value, we assign a value x to environment e1.

```
> e1 = e2 = e3 = e4 = new.env ()
> e1$x = "environment e1's x value"
> e1$x
[1] "environment e1's x value"
```

However inspection of the other environments yields the following:

```
> e2$x
[1] "environment e1's x value"
> e3$x
[1] "environment e1's x value"
> e4$x
[1] "environment e1's x value"
```

Lists vs Environments

We have already mentioned that R treats environment identifiers as object references, however that is not the only difference between lists and environments. In general lists are more user-friendly, at least compared to standard environments. Say we wish to create an object, with three components, x, y and z. This is slightly easier with lists than environments.

```
> obj1 = list (x=1, y=2, z=3)
> obj2 = new.env ()
> obj2$x = 1
> obj2$y = 2
> obj2$z = 3
```

Another problem is what happens when we want make copies of the objects themselves (not the reference). For lists this is trivial:

```
> obj1.copy = obj1
```

However for environments more work is required.

```
> obj2.copy = new.env ()
> obj2.copy$x = obj2$x
> obj2.copy$y = obj2$y
> obj2.copy$z = obj2$z
```

For an environment with many values, we could try and be clever:

```
> obj2.copy = new.env ()
> names = ls (obj2)
> for (name in names) assign (name, get (name, obj2), obj2.copy)
> ls.str (obj2.copy)
x :  num 1
y :  num 2
z :  num 3
```

There are two traps here. Firstly, when calling ls, we may need to set all.names=TRUE. Secondly, it is possible for environments to contain references to other environments, and in order to copy the whole thing may require a reasonable amount of work. However a much simpler solution is provided in the next section.

A number of common tasks (perhaps things we take for granted with lists) will not work with environments.

```
> e = f = new.env ()
> e == f
Error in e == f :
comparison (1) is possible only for atomic and list types
> for (obj in e) print (obj)
Error in for (obj in e) print(obj) :
invalid type/length (environment/1) in vector allocation
> is.na (e)
logical(0)
Warning message:
In is.na(e) : is.na() applied to non-(list or vector) of type 'environment'
> e [[1]]
Error in e[[1]] : wrong arguments for subsetting an environment
```

Now whilst lists are more user-friendly, environments offer substantial performance gains. Roughly speaking, when R calls a function, it copies each argument. This isn't completely true, however we are

still using a copy-by-value system. Environments allow a copy-by-reference system, which is potentially much faster.

In the following example, we create two large objects, that are essentially the same, except that one is based on a list, and the other is based on an environment. We also create two functions, to modify each of the objects, which again are essentially the same.

```
> obj1 = list ()
> obj2 = new.env ()
> for (i in 1:100)
 {        obj1 [[i]] = 1:1000
         assign (as.character (i), 1:1000, obj2)
 }
> f1 = function (obj) {obj [[1]] = rnorm (1); obj}
> f2 = function (obj) {obj$`1` = rnorm (1); obj}
```

For each object we apply the corresponding function 1000 times and record the total time. This isn't the best experimental design, as there are other factors which effect the performance, however the result is reasonably clear, with the environment based example performing much better.

```
> #list example
> start = Sys.time ()
> for (i in 1:1000) obj1 = f1 (obj1)
> finish = Sys.time ()
> finish - start
Time difference of 1.366078 secs


> #environment example
> start = Sys.time ()
> for (i in 1:1000) obj2 = f2 (obj2)
> finish = Sys.time ()
> finish - start
Time difference of 0.04305983 secs
```

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. The first problem that was noted using environments was 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 is 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
```

We can also clone the ENVIRONMENT object. This will account for other environments, even if there are circular references.

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


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

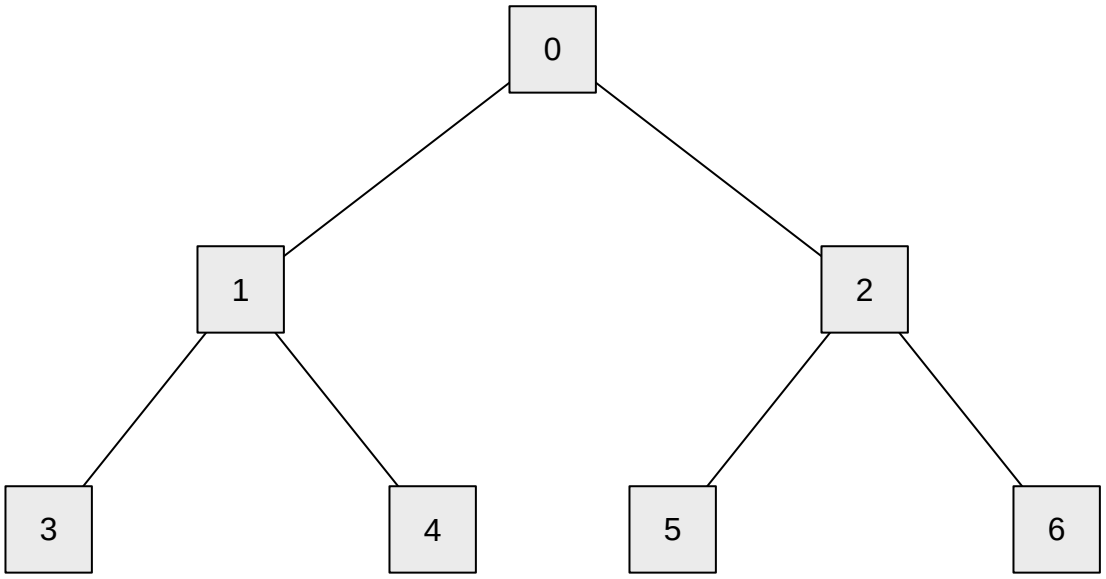
Extending ENVIRONMENT objects is trivial:

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

Binary Trees

In this section we create and traverse a binary tree, using environments. Whilst the benefits of using environments are relatively small here, this example is relatively simple, plus for more complex graphs, such as large cyclic graphs, we really need object references, otherwise the implementation becomes difficult.

Lets say we have the following binary tree:



We can create the tree, by defining (informally) a node class. The example allows specification of a parent node (this isn't completely necessary).

```
> node = function (value, parent=NULL, child0=NULL, child1=NULL)
 {        node = extend (ENVIRONMENT (value, child0, child1), "node")
         if (!is.null (parent) )
         {        if (is.null (parent$child0) ) parent$child0 = node
                 else if (is.null (parent$child1) ) parent$child1 = node
                 else stop ("parent already has two children")
         }
         node
 }
> print.node = function (node, ...) cat (node$value, "\n")
```

Now creating the tree:

```
> node0 = node (0)
> node0.0 = node (1, node0)
> node0.1 = node (2, node0)
> node0.0.0 = node (3, node0.0)
> node0.0.1 = node (4, node0.0)
> node0.1.0 = node (5, node0.1)
> node0.1.1 = node (6, node0.1)
```

We can write a very simple traversal algorithm:

```
> preorder = function (node)
 {        print (node)
         if (!is.null (node$child0) ) preorder (node$child0)
         if (!is.null (node$child1) ) preorder (node$child1)
 }
> preorder (node0)
```

```
0
1
3
4
2
5
6
```