Calling R from Java

Duncan Temple Lang

December 13, 2005

In the R-Java interface, we can pass an arbitrary R object to the Java interface as a *foreign reference*. This can then masquerade as a regular Java object (via dynamic byte-code generation), and the resulting Java methods are implemented as simple calls to R functions. This is one way to call R from Java.

Several people have expressed an interest in evaluating R expressions from within Java code. Typically, they wish to give a string containing legal R code and have it evaluated and get the result back. In this short document, we describe how to do that using the R-Java interface.

We note here that it is better to call functions rather than to use the S syntax so as to make ones code more robust to changes in design and implementation. In the near future, we will add a facility to call functions by name or be reference, with ordered and named arguments. It is even better to use the foreign reference above to hide the R object as a Java object and have oter code only rely on that Java object. This means that R is just one of many potential implementations, and one can slide in an alternative implementation in Java without altering the other code that calls this.

Suppose we want to get the names of the R variables in the first element of R's search path. We would do this with the R expression

```
objects()
```

What we want do is construct this as a string in our Java application and then pass it to an R evaluator to have it be parsed and evaluated. We do this by first creating a Java version of the R evaluator. To do this, we create an instance of the class `REvaluator`.

```
[]
 org.omegahat.R.Java.REvaluator e  = new REvaluator()
```

(Note that one does not need to create a separate instance for each evaluation and that the same object can be used any number of times to evaluate an expression.)

Now we want to have this object `e` parse and evaluate the expression `"objects()"` For this, we call one of the `eval()` methods provided by the `REvaluator` class. The simplest version requires only the expression in the form of a string.

```
[]
 Object val = e.eval("objects()")
```

This parses the expression and then evaluates it as a top-level R task. The result is then processed by the standard conversion mechanism used in the R-Java interface (including any user-registered converters) and the resulting Java object returned. In this particular example, the R value returned from the expression is a character vector. This is converted to an array of `String` objects. So we can cast the `val` to this type and work with it as a regular Java object.

```
[]
  if(val != null) {
   String[] objects = (String[])val;
   for(i = 0 ; i < objects.length; i++_
```

```
    System.err.println("("+i+") " + objects[i]);
  }
```

Objects that have no translation need to be returned as an `RForeignReference`. They can then be manipulated in subsequent Java code in an application-specific fashion.

The other versions of the `eval()` method in `REvaluator` allow one to control how the result of the R expression is converted back to a Java object. These allow one to specify whether an attempt should be made to find a converter or whether the value should be returned immediately as a foreign reference so that it can be used later in other R expressions. Alternatively, one can specify the required return type as a Java interface. The conversion code will then create a new class which extends `RForeignReference` and implement the methods of the Java interface by calling the corresponding R functions on that R object.

1 Calling R functions

One will quickly find that specifying calls to R functions via strings is quite limited. It is difficult to pass values computed in earlier computation as arguments to these calls. Instead, we want to be able to call the R functions in a more Java-like mechanism. We want to identify the function by name and then pass objects to it in a call of the form

```
REvaluator.call(functionName, argArray, namedArgTable)
```

In $\widehat{\Omega}$, we can use a simple syntax such as

```
functionName(arg1, arg2, .., name1=value, name2=value)
```

like we do in *S*.

The standard conversion mechanism used in the inter-system interfaces is used to convert the Java arguments to R objects. This means that one can register a C routine, Java method, or *S* function to perform the conversion.

Here are some simple examples of calling *R* functions that involve only primitive arguments. These are taken from the *org.omegahat.R.Java.Examples.JavaRCall* and one should consult that for more details.

We start by creating the R interpreter instance in Java (`ROmegahatInterpreter`) which initializes the R engine, etc. Next, we create then the R evaluator `REvaluator`) instance which we will use to make the calls to the R engine.

```
[Examples:RfromJava]
  ROmegahatInterpreter interp = new ROmegahatInterpreter(ROmegahatInterpreter.fixArgs(args), false);
  REvaluator e = new REvaluator();
```

Now we are ready to create the different calls to the *R* functions. The first is a simple call to *objects()* with no arguments. This returns the names of the objects in the default, global environment. We know this returns an array of Strings (which may be `null`) and so we can perform the cast.

```
[]
 String[] objects = (String[])  e.call("objects");
```

The second call specifies the element of *R*'s search path whose contents are to be returned. We can specify this by name or by index. In this case, we provide the name as a Java `String`. To do this, we create an array containing each of the arguments (1 in this case).

```
[]
 String[] objects = (String[])  e.call("objects", new Object[]{"package:base"});
```

Now, we turn to calling the function *seq()* and giving it two and more arguments. The first call is equivalent to the S expression

```
seq(as.integer(1), as.integer(10))
```

Again, we create an array to store the arguments. In this case, we specify the arguments as `Integer` objects. And finally, we invoke the `REvaluator`'s `call()` method with these arguments.

```
[]
  int[] seq;
  funArgs = new Object[2];
   funArgs[0] = new Integer(1);
   funArgs[1] = new Integer(10);

  seq = (int[])e.call("seq", funArgs);
```

Now we add a third and named argument, specifically for the *by* parameter. We do this by creating We reuse the first two arguments in the array `funArgs`.

```
[Example:RfromJava]

  java.util.Hashtable namedArgs = new java.util.Hashtable(1);
  namedArgs.put("by", new Integer(2));
  seq = e.call("seq", funArgs, namedArgs);
```

In some cases, it would be simpler to create an array with three arguments and to specify the names of the arguments separately. The following is equivalent to the previous example as it specifies two parallel arrays, one giving the argument values and the other giving the argument names

```
[]
 funArgs = new Object[3];
 funArgs[0] = new Integer(1);
 funArgs[1] = new Integer(10);
 funArgs[2] = new Integer(2);

 String[] names = new String[3];
 names[2] = "by";

  value =  e.call("seq", funArgs, names);
```

These examples involve simple primitive data types. Suppose we want to pass a matrix to R and have it produce a pair-wise plot. (Note that what we do with the matrix within the R call is the relevant point of this example.)

There are two basic approaches (as is true in general for all the inter-system conversion approaches we use.)

by-value We can convert the Java matrix into an S matrix by copying its contents to S.

by-reference We can create in R a proxy for the Java object and have the code that operates on it call methods on it using the $ operator or the *.Java()* function explictly.

The by-value appears simpler but requires code specific to the Java class. It also means that one cannot share the object between Java and S so that modifciations in one system are visible to the other.

In our example, the code that generates the pairwise scatterplots (*pairs()*) is not written in a way to handle a reference to a Java object. Thus, we should use the *by-value* approach. We should note however, that we must start writing *S* code in a more general and flexible manner so that we can pass objects from different systems as arguments to functions and have the same behaviour. Too many *S* functions are written using explicit knowledge of the representation of the data type and are not abstracted to use methods on the object. This is not only reduces the re-use of the code with respect to inter-system interfaces, but makes for code that is not robuts to small changes in design. The S4-style classes and more importantly, object oriented style classes are being added to SPlus and R and are important tools for serious software development in the *S* language.

Back to our example and converting the Java matrix to an *S* matrix. As with all inter-system interfaces and distributed computing environments, we have to decide where to do the conversion. In other words, in which language do we write the computations that creates the *S* matrix object from the Java instance. We can do this in R/S-Plus, Java or C.

Let's take what should be the simplest approach and create an R function that converts the Java matrix. We will assume that we have a `DenseDoubleMatrix2D` from the `http://www.colt.org`Colt package.

The function is quite simple. It must obtain the values to put into the matrix and the dimensions of the matrix. The former can be retrieved by calling the `toArray()` method of the `DenseDoubleMatrix2D` object. Similarly, the number of rows and columns in the matrix can be obtained by calling the corresponding methods in the target object being converted. Note that the `toArray()` method returns an array of arrays containing `double` values. This is converted to an R object as ...

```
[]

 coltConverter <-
 function(x, klassName) {
   vals <- unlist(x$toArray())

   matrix(vals, x$rows(), x$columns())
 }
```

Along with the converter, we have to register a function that determines whether the converter can handle the object being converted. In this case, we only handle `DenseDoubleMatrix2D` objects and so the function need only compare names.

```
[]
 coltMatch <-
 function(x, klassName) {
   klassName == "cern.colt.matrix.impl.DenseDoubleMatrix2D"
 }
```

The final step is to register these functions with the basic conversion mechanism used by the R-Java interface.

```
[]
setJavaFunctionConverter(converter, match,
                         description="Colt DenseDoubleMatrix2D to R matrix",
                         fromJava=T)
```

Exactly where and when this R expression is evaluated depends on the Java application and the R session. One can add it to a *.First()*, evaluate it using the `eval()` method in the `REvaluator` class, and so on.

2 Returning R Objects to Java

The result of evaluating an R expression or invoking an R function is an R object. When this is passed back to the Java method that called it, the standard mechanism for converting R objects to Java objects is applied to the R value. Any user-level converters that have been registered for this direction of translation can be applied to the object. If no converter is found (either user-level or built-in), a proxy for the R object is returned to Java in the form of an *RForeignClass* object. Essentially, this defines the Java-to-R interface. However, we will present some examples here to illustrate the idea. But before we do that, it should be noted that this is a very different situation that arises when one embeds Java within R. In that case, implicit calls to R are made from Java via callbacks to R reference objects. For example, we might export a reference to an R matrix as an argument to a Java method. In that case, the class of the Java object is known because of the context in which the R object is being passed, i.e. the method and its parameter list. Hence, we have type information about the Java object being expected. When calling R from Java via the *REvaluator*, we have no such type information. All we know is we are getting back an R object and must convert it. When this is done from a Java application, we must know the class of the return value and can therefore specify a converter when we compile the application.

Let's suppose a simple R function that fits a linear model to simulated data. It takes a single argument specifying the number of observations to use. It then generates X and Y as

$$X = seq(1, length = n) Y = 10 + .5 * X + rnorm(n)$$

So a suitable function is

```
[lmConvert.R]
simLM <-
function(n, sigma=1)
{
 x <- seq(1, length=n)
 y <- 10 + .5 * x + sigma*rnorm(n)
 val <- lm(y ~ x, data=data.frame(x = x, y = y))

cat("Finished simLM\n")
 val
}
```

This returns an object of class lm. We want to return this to the Java code. We have a variety of different options.

2.1 Java Class

One thing we can do is to define a new Java class, say `SLinearModelFit`. We define it to have accessor methods for the different elements the S lm object has such as coefficients, residuals, rank, terms, etc. And then we create a converter from R to Java that passes these different elements by value to the new instance of this class.

```
[SLinearModelFit.java]
package org.omegahat.R.Java.Examples;

public class SLinearModelFit
```

```
{
 double[] coefficients;
 double[] residuals;
 int rank;

 public SLinearModelFit(double[] coeffs, double[] resids, int rank) {
  setCoefficients(coeffs);
  setResiduals(resids);
  setRank(rank);
 }

 public int getRank() {
   return(rank);
 }

 public void setRank(int v) {
   rank = v;
 }

 public double[] getResiduals() {
    return(residuals);
 }

 public void setResiduals(double[] vals) {
    residuals = vals;
 }

 public double[] getCoefficients() {
   return(coefficients);
 }

 public void setCoefficients(double[] vals) {
   coefficients = vals;
 }

}
```

We should note that we can partially automate the creation of this class definition. We can create an instance of the lm class and then examine its elements and their types.

Next, we can write a converter function in R that creates an instance of this new class.

```
[lmConvert.R]
lmCvt <-
function(obj,...)
{
 .JNew("org.omegahat.R.Java.Examples.SLinearModelFit", obj$coefficients, obj$residuals, obj$rank)
}
```

Before we register the converter, we must load the Java library to make the *S* functions it provides available to the session. Note that at this point, we can make calls to the different functions that access the Omegahat interpreter (e.g. *.Java()*, *.JNew()*, etc.).

```
[lmConvert.R]
library(Java)
```

Again, we must register the converter function along with its matching function which determines whether the converter can handle a given object.

```
[lmConvert.R]
setJavaFunctionConverter(lmCvt, function(x,...){inherits(x,"lm")},
                         description="lm object to Java",
                         fromJava=F)
```

We are now in a position to write some Java code that actually calls this R code and makes use of the converters. The steps are relatively simple.

- We create the Omegahat interpreter and the Java version of the *REvaluator*.
- We source the *R* code that defines the *simLM()* and the converter into the *R* session. We do this by evaluating an *R* expression. Note that we use a call to `voidEval()` since we are not interested in the return value.
- Now we are ready to invoke the *simLM()* and we do so by invoking the `call()` method of the *REvaluator*. We give it an array of the arguments, here containing just a single value which is an integer (`Integer`).
- The remainder of the code manipulates the result and prints out the coefficients, residuals and rank using the `show()` method of the $\widehat{\Omega}$ interpreter.

```
[Main]
static public void main(String[] args) {
  ROmegahatInterpreter interp = new ROmegahatInterpreter(ROmegahatInterpreter.fixArgs(args), false);
  REvaluator e = new REvaluator();

  String rfile = "system.file('data', 'lmConvert.R', pkg='Java')";

  System.err.println("executing: source(" + rfile +")");
  e.voidEval("source(" + rfile +")");

  Object val = e.call("simLM", new Object[]{new Integer(10)});
  System.err.println("Result of simLM: " + val + " (" + val.getClass() + ")");

  org.omegahat.R.Java.Examples.SLinearModelFit f = (org.omegahat.R.Java.Examples.SLinearModelFit) val;
  interp.show("Coefficients:");
   interp.show(f.getCoefficients());
  interp.show("Residuals:");
   interp.show(f.getResiduals());
  interp.show("Rank:");
   interp.show(new Integer(f.getRank()));

  val = e.call("simLM", new Object[]{new Integer(10)});

}
```

Having compiled the Java code and created the `lmConvert.R` file, we can invoke this Java application using the `RJava` script that is installed with the Java package. In my setup, I invoke it as follows.

```
[]
/tmp/R/tmp/Java/scripts/RJava --example --class org.omegahat.R.Java.Examples.lmTest --gui=none --silent
```

We specify the *lmTest* as the class whose `main()` method is to be run. The remaining arguments are passed to the R startup and turn off the loading of the graphics device code and the startup message or banner.

The following is the code that actually defines the *lmTest* and allows it to be created directly from this document.

```
[lmTest.java]
package org.omegahat.R.Java.Examples;

import org.omegahat.R.Java.REvaluator;
import org.omegahat.R.Java.ROmegahatInterpreter;

public class lmTest  {

@use Main

}
```

The makefile in the `R/Java/Examples/` directory is responsible for creating the different files from this

```
[]
 make
 make # Do it twice for the moment!
```

2.2 Proxy Class

An alternative approach is to pass back a reference to the lm object in R. We can define a Java interface which mirrors the operations supported by R on an lm object. Suppose this interface is named `LinearModelFitInt` and has methods accessing the elements of the lm object in R and perhaps some other methods such as *plot()*.

```
[]
public interface LinearModelFitInt {
 public double[] getResiduals();
 public double[] getCoefficients();
 public int      rank();
 public void     plot();
}
```

Now, we can automatically generate a new class that implements this interface and inherits from *RForeignRefrence*. (See the function *jdynamicCompile()* in the Java package.) We create an instance of this class by first registering the lm object with the foreign reference manager.

```
[]
 ref <- foreignReference(lmValue)
```

and then passing the result R reference object as the argument to the constructor of this new class

```
[]
 .JNew("LinearModelFitForeignReference", ref)
```

3 Setup

There are two ways in which we might want to call R from Java. One is that we have a regular Java application and we want to embed Java within it as a *worker*. Alternatively, we are in an R session and use R as the main controller and from there call some Java code that needs to evaluate an R expression. In the first case, you will need to have built R as a shared library. You can do this by passing the argument `--enable-shared` to the `configure` script when building R from source. Then, make the regular R installation and the R shared library. That is, from within the top-level directory of the R distribution issue the following commands: `make`cd src/main make libR

When Java is embedded in R, one need not do anything special other than installing and loading the `Java` package and following the steps to instantiate the `REvaluator` object and evaluating the expression(s).

4 Example

4.1 Command Line and Text Examples

When the `Java` package is installed, it provides a script – `scripts/RJava` – to allow one to run a Java application that can dynamically load R (assuming the shared library is available). You can use this to run the example provided in `REvaluator` provided by that classes' `main()` method. Invoke it as

```
Java/scripts/RJava --example --class org.omegahat.R.Java.Examples.JavaRCall --gui=none
```

This prints the search path and the result of the R expression

```
objects('package:base')
```

It uses the Omegahat evaluator to print the results (and this is why they are truncated)

Another example allows you to type R commands at a Java-controlled prompt. Invoke this as

```
Java/scripts/RJava --example --gui=none
```

Then, one sees the prompt

```
[omegahat->R]
```

and can type R expressions such as

```
[omegahat->R]  sum(1:3)
6
[omegahat->R]  letters
a
b
c
d
e
f
g
h
i
j
k
```

```
l
m
n
o
p
q
r
s
t
u
v
w
x
y
z
```

4.2 Graphics Example

Another example involves using the R graphics engine. Again, we can use the `eval(String)` method of the `ROmegahatInterpreter` to evaluate an S expression that produces a plot. In the simplest case, this produces a window and displays a plot. Our example produces a plot of 100 random normals. It can be called with following shell command

```
RJava --example --class org.omegahat.R.Java.Examples.JavaRPlot
```

(You will probably need to provide a fully qualified path for the RJava script, as it is in the `scripts/` directory of the installed Java package.) This will display the plot in a graphics device window. If one wanted to produce postscript output, simply append the argument `-gui=none` to the call which will indicate to R that it should not use the interactive devices. So the command

```
RJava --example --class org.omegahat.R.Java.Examples.JavaRPlot --gui=none
```

will produce a file named `Rplots.ps` which will contain the plot.

There are issues with the event handling and hence resizing the graphics device, etc. Also, the interaction of threads, Java's linking of the X11 libraries, etc. make for further complications. To avoid these problems and also get ehanced event handling at the Java level, one should consider using the R graphics device that uses Java and the Graphics2D API to render the contents of the plot. This is the `RJavaDevice` package available from the Omegahat Project.

5 Plots & Graphics Devices

Note that in the examples above when running R from within a Java application via the R shared library, we turned off the graphical interface and hence the (X11) graphics devices. This is a result of the way in which the R library is loaded into R and then how the X11 graphics device code is loaded into R. If one does want to use the device, you will have to compile the `R_X11.so` by hand having made a marginal modification to the Makefile in the `src/unix/X11/` directory of the R source distribution. Append

```
-L${R_HOME}/bin -lR
```

to the definition of the R_X11_la_LIBADD variable so that it reads

```
R_X11_la_LIBADD = $(ALL_X_LIBS) $(LIBPATHS) $(BITMAP_LIBS) -L${R_HOME}/bin -lR
```

Then type make in that directory to create a version of `R_X11.so` that is linked against `libR.so`.

This is not an official recommendation in any way and a different mechanism will be introduced in the future.