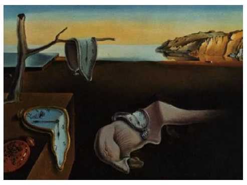


Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *http://www.jstatsoft.org/*

UNDER REVIEW

rscala: Integration of R and Scala

David B. Dahl
Brigham Young University

Abstract

This paper introduces the **rscala** package for the statistical software R. This package provides a two-way bridge between R and Scala enabling a user to exploit each language's respective strengths in a single project. The **rscala** package transparently brings Scala and Java methods into R as if they were native R functions. Specifically, Scala classes can be instantiated and methods of Scala classes can be called directly. Furthermore, arbitrary Scala code can be executed on the fly from within R, inline Scala functions can be defined, and callbacks to the original R interpreter are supported. Finally, **rscala** also enables arbitrary R code to be embedded within a Scala application. The **rscala** package is available on CRAN and requires no special installations or configurations of R or Scala.

Keywords: embedded interpreter, Java, Java Virtual Machine, JVM, R, Scala.

1. Introduction

Scala (Odersky *et al.* 2004) is a general-purpose, statically-typed programming language that supports both object-oriented and functional programming paradigms. The integration of these two paradigms allows a developer to choose the best paradigm for a given task. Code is generally concise, thanks in part to type inference. Martin Odersky, a professor at EPFL in Switzerland, is the primary architect of Scala. Previously, Odersky worked for Sun Microsystems where he developed the current generation of the Java (Oracle 2015) compiler. It is said that Scala's design is influenced by criticisms of Java though a high level of interoperability between Scala and Java is maintained. Indeed, Scala source code compiles to Java bytecode and runs on the Java Virtual Machine (JVM). Typesafe, a company co-founded by Odersky, provides an open source platform centered around Scala and makes its money providing training, consulting, and commercial support for the platform.

R (R Core Team 2015) is a scripting language and environment developed by statisticians for statistical computing and graphics. R has many contributors and a large base of statistically-oriented users. Scala and R each have distinct strengths and weaknesses. We believe that Scala deserves consideration when looking for a general-purpose programming language to complement R, but advocating Scala for statistical computing is beyond the scope of this paper. Instead, this paper introduces our **rscala** (Dahl 2015) package to those that are already somewhat familiar with R and Scala. The package allows users to seamlessly incorporate R and Scala (or Java) code in one program and utilize each language's respective strengths (including libraries, methods, speed, graphics, etc.). Specifically, **rscala** provides a two-way bridge between R and Scala, enabling R to execute Scala code and vice versa. Scala and Java classes can be instantiated, methods of Scala and Java classes can be called directly, inline Scala functions can be defined, and arbitrary Scala code can be executed on the fly from within R. Callbacks from Scala into the original R interpreter are also supported. Conversely, arbitrary R code can be evaluated from within a Scala application through an embedded R interpreter.

Thus the **rscala** package allows R developers to reuse Scala code and apply their Scala skills to make R extensions. On the other hand, **rscala** allows Scala developers to make use of R's broad array of data analysis and graphing capabilities from within a Scala application. The **rscala** package is intended to provide this bridge to those who have an interest in both Scala and R. This is not unlike what has already been done with R and other languages. Although largely transparent to the user, much of R's core is implemented with calls to C, C++, and Fortran code. These low-level languages provide more flexibility in memory management and compile to code that executes relatively quickly. The **rJava** (Urbanek 2013a) package extends this support to precompiled Java code. While results vary depending on the algorithm and testing methodology, it is generally recognized that well-designed C, C++, and Fortran code will execute faster and use less memory than Java. Nevertheless, Java arguably has several advantages and has attracted a substantial developer base. The **rJava** package allows developers to use existing Java code and to apply their Java skills to make R extensions. The **rscala** package aims to do for Scala what **rJava** has done for Java.

The paper is organized as follows. Section 2 provides a general usage guide to get users started quickly. Technical details of the package implementation are discussed in Section 3. Comparisons to other software, including numerical benchmarks, are provided in Section 4. Section 5 looks at two case studies using the **rscala** package. We conclude with future work in Section 6.

2. Usage of the package

In this section, we provide an introduction to the **rscala** package with the intent of getting users quickly started integrating R and Scala. Those interested in the more technical aspects of the **rscala** package — discussed in subsequent sections — will also benefit from understanding the syntax and ideas presented in this section.

2.1. Installation

The **rscala** package is available on the Comprehensive R Archive Network (CRAN) and can be installed through R's graphical interface or by executing the following R expression:

```
install.packages('rscala')
```

The package runs on a standard installation of R; no special compilation flags or installation procedures are needed. The **rscala** package can use an existing Scala installation (versions 2.10.x and 2.11.x are supported) or can install Scala using

```
rscala::scalaInstall()
```

This places Scala in the user's home directory: '~/.rscala'. System administrators will likely want to install Scala globally as described on the Scala webpage. In short, simply download the archive, unpack it, and add the "scala" script to the path. Note that Scala requires Java. The **rscala** package tries various methods to find a suitable Scala installation, including searching the path. Details on the search are available using

```
rscala::scalaInfo(verbose=TRUE)
```

2.2. Accessing Scala in R

Load the **rscala** package in an R session using

```
library('rscala')
```

A Scala interpreter instance is created using the `scalaInterpreter` function:

```
s <- scalaInterpreter()
```

Options are available to customize where Scala and Java are located and how they are invoked (e.g., setting the classpath and maximum heap size). Details on this and all other functions are provided in the R documentation for the package (e.g., `help(scalaInterpreter)`). Multiple instances of the Scala interpreter can be created and each instance runs independently with its own memory. Each interpreter can use multiple threads/cores but the bridge between R and Scala is not thread-safe. As such, multiple R threads/cores should not access the same interpreter simultaneously.

Instantiating Scala objects and calling methods

The **rscala** package allows R to hold a reference to any Scala object and to call the object's methods directly. There is also support for calling methods of a companion object and for instantiating objects. This functionality is illustrated in Table 1. These examples display classes and companion objects for `scala.util.Random`, `scala.math.BigInt`, and `scala.Array`, but any classes and companion objects in the classpath can be called in this manner. The only exception is inner classes which are not supported. If a method takes type parameters or is

Description Scala	R using the rscala package
Instantiate an object	
`val a = new scala.util.Random()`	`a <- s$do('scala.util.Random')$new()`
Method with arguments	
`a.setSeed(1234)`	`a$setSeed(1234L)`
Method without arguments	
`a.nextGaussian`	`a$nextGaussian()`
Companion object's method	
`val b = BigInt.probablePrime(8, a)`	`b <- s$do('BigInt')$probablePrime(8L, a)`
Companion object's special `apply` *method*	
`val c = Array(b)`	`c <- s$do('Array')$apply(b)`
Object's special `update` *method*	
`c(0) = b.pow(4)`	`c$update(0L, b$pow(4L))`
Object's special `apply` *method*	
`c(0).intValue`	`c$apply(0L)$intValue()`

Table 1: Examples showing how to replicate Scala expressions in R via the **rscala** package.

otherwise not a valid identifier in R (e.g., an operator method), the call to the method should be enclosed in quotes. For example, note that the methods `apply[Int]` and `:+` are quoted here:

```
h <- s$do('List')$'apply[Int]'(1L, 2L, 3L)
g <- h$":+"(100L)
g$toString()

## [1] "List(1, 2, 3, 100)"
```

Scala runs on the Java Virtual Machine and it supports instantiating Java classes and calling object and static methods. As such, **rscala** automatically provides this support as well. Scala provides the `scalap` executable to list the methods of class and companion objects. Its output is available directly in R using the `scalap` function. For example:

```
scalap(s, 'scala.util.Random')

## package scala.util
## class Random extends scala.AnyRef with scala.Serializable {
##   val self: java.util.Random = { /* compiled code */ }
##   def this(self: java.util.Random) = { /* compiled code */ }
##   def this(seed: scala.Long) = { /* compiled code */ }
##   def this(seed: scala.Int) = { /* compiled code */ }
##   def this() = { /* compiled code */ }
##   def nextBoolean(): scala.Boolean = { /* compiled code */ }
##   def nextBytes(bytes: scala.Array[scala.Byte]): scala.Unit = { /* compiled co...
##   def nextDouble(): scala.Double = { /* compiled code */ }
##   def nextFloat(): scala.Float = { /* compiled code */ }
##   def nextGaussian(): scala.Double = { /* compiled code */ }
##   def nextInt(): scala.Int = { /* compiled code */ }
##   def nextInt(n: scala.Int): scala.Int = { /* compiled code */ }
....
```

Return values when calling Scala

When calling Scala from within R, the return value is one of the following:

- A reference, i.e., an R object of class `ScalaInterpreterReference`. The methods of a reference can be called in subsequent statements, as exemplified by `a` and `c` in the right column of Table 1.
- A vector of integers, doubles, logicals, or characters.
- A matrix of integers, doubles, logicals, or characters.

The return value is guaranteed to be a reference if the optional argument `as.reference` is set to `TRUE`. For example, use `c$apply(0L)$intValue(as.reference=TRUE)`) to get a reference for the last expression of Table 1. When `as.reference=FALSE`, an error is generated when a vector or matrix cannot be returned. Finally, the default is `as.reference=NA` which results in a vector or matrix being returned if possible and a reference otherwise.

We now describe the special support for integers, doubles, logicals, and characters — jointly referred to as *primitives* — as well as vectors and matrices of these types. Vectors in R are implemented in Scala as arrays. Matrices in R are implemented in Scala as rectangular arrays of arrays, i.e., arrays of arrays of the same length. These types and data structures can be copied from R into Scala and *vice versa*. Table 2 summarizes this special support and how **rscala** converts data between the languages. In the table, variables with the same identifier (e.g., `b`) are equivalent in R and Scala. For example, a matrix of integers in R (e.g., `j <- matrix(c(1L, 2L), nrow=2)`) maps to a row-major rectangular array of arrays of Scala's `Int` (e.g., `val j = Array(Array(1), Array(2))`).

Arguments when calling Scala

When instantiating Scala classes, calling methods of objects, or calling methods of companion objects, an argument can be either: i. a Scala reference or ii. a vector or matrix of integers, doubles, logicals, or characters. Vectors and matrices supplied as arguments to Scala methods are automatically converted to Scala arrays and rectangular arrays as illustrated in Table 2. Of course, Scala reference objects require no conversion. For example, the Scala reference object `b` in the right-hand column of Table 1 is used as an argument for the `apply` method on the next line in that table.

Defining inline Scala *functions*

In addition to calling precompiled Scala methods, the **rscala** package allows the user to define Scala methods within an R session. These *inline* methods are compiled and cached for execution later in the R session. This feature is available using the `intpDef` method, or its shorthand substitute `s$def(args, body)`, where `args` gives the name and Scala type of each argument and `body` is arbitrary Scala code with access to the arguments defined in `args`. We demonstrate this functionality by defining a function to compute the number of partitions of n items, i.e., the Bell number (Bell 1938).

Primitive	Vector (array)	Matrix (rectangular array of arrays)

`a <- TRUE`
`val a = true`
	`e <- c(TRUE, FALSE)`
	`val e = Array(true, false)`
		`i <- matrix(c(TRUE, FALSE), nrow=2)`
		`val i=Array(Array(true), Array(false))`

`b <- 1L`
`val b = 1`
	`f <- c(1L, 2L, 3L)`
	`val f = Array(1, 2, 3)`
		`j <- matrix(c(1L, 2L), nrow=2)`
		`val j = Array(Array(1), Array(2))`

`c <- 1.0`
`val c = 1.0`
	`g <- c(1.0, 2.0, 3.0)`
	`val g = Array(1.0, 2.0, 3.0)`
		`k <- matrix(c(1.0, 2.0), nrow=2)`
		`val k = Array(Array(1.0), Array(2.0))`

`d <- 'a'`
`val d = 'a'`
	`h <- c('a', 'b', 'c')`
	`val h = Array('a', 'b', 'c')`
		`l <- matrix(c('a', 'b'), nrow=2)`
		`val l = Array(Array('a'), Array('b'))`

Table 2: Integers, doubles, logicals, and characters (labeled *primitives*), as well as vectors and matrices of these types can be seamlessly copied between R and Scala. Each cell in the table contains two lines: an R expression (top) and the equivalent Scala expression (bottom).

```
bell.using.rscala <- s$def('n: Int', '
  def snsk(n: Int, k: Int): BigInt = {
    if ( n == 0 && k == 0 ) BigInt(1)
    else if ( n == 0 || k == 0 ) BigInt(0)
    else k * snsk(n - 1, k) + snsk(n - 1, k - 1)
  }
  var sum = BigInt(0)
  for ( k <- 0 to n ) sum = sum + snsk(n, k)
  sum.doubleValue
')
```

Note that the object `bell.using.rscala` is an R function which calls the Scala code. Since `n` is defined as an `Int` in the example above, the argument `n` when calling the function is either a `ScalaInterpreterReference` to a Scala's `Int` object or is automatically cast to an integer with R's `as.integer` function. For example, although the literal 10 is stored in R as a double, it is cast to an integer here:

```
bell.using.rscala(10)

## [1] 115975
```

There are also three optional arguments, as seen here:

```
args(bell.using.rscala)

## function (n, as.reference = NA, quiet = "", gc = FALSE)
## NULL
```

The effect of the `as.reference` argument has already been discussed. The `quiet` and `gc` arguments are discussed later.

An R implementation of the same algorithm is

```
bell.using.R <- function(n) {
  snsk <- function(n, k) {
    if ( n == 0 && k == 0 ) 1
    else if ( n == 0 || k == 0 ) 0
    else k * snsk(n - 1, k) + snsk(n - 1, k - 1)
  }
  sum <- 0
  for ( k in 0:n ) sum <- sum + snsk(n, k)
  sum
}
```

Notice that the Scala implementation, depending on the value of n, can be significantly faster than the R implementation:

```
system.time(bell.using.rscala(16))

##    user  system elapsed
##   0.000   0.003   0.042

system.time(bell.using.R(16))

##    user  system elapsed
##   1.582   0.005   1.588
```

Evaluating arbitrary Scala code

The **rscala** package also allows arbitrary snippets of Scala code to be compiled and executed on the fly. These code snippets can be any mixture of statements, functions, class definitions, imports, etc. Consider an earlier example. Rather than evaluating the right column of Table 1 in R, the Scala code in the left column can be evaluated more directly in R as follows:

```
s %~% '
  val a = new scala.util.Random()
  a.setSeed(1234)
  println(a.nextGaussian)
  val b = BigInt.probablePrime(16, a)
  val c = Array(b)
  c(0) = b.pow(4)
  c(0).intValue
'

## 0.14115907833078006
## [1] 1871639857
```

Note that the output from the `println(a.nextGaussian)` statement is displayed. To suppress output from Scala, set the `quiet` global option of the `intpSettings` function:

```
intpSettings(s, quiet=TRUE)
s %~% '
  val a = new scala.util.Random()
  a.setSeed(1234)
  println(a.nextGaussian)
  val b = BigInt.probablePrime(16, a)
  val c = Array(b)
  c(0) = b.pow(4)
  c(0).intValue
'

## [1] 1871639857
```

The return value for the evaluation is the value of the last expression. It is an integer vector of length one because the Scala result's type `Int` could be converted to an integer vector in R. If a conversion is not possible, the return value is a reference. A reference is guaranteed when using the `%.~%` operator:

```
s %.~% 'c(0).intValue'

## ScalaInterpreterReference... res7: Int
```

By default, the `%~%` and `%.~%` operators perform string interpolation of the right-hand-side character vector before compiling and executing the Scala snippet. Any expression between '`@{`' and '`}`' is evaluated in the R session and its string representation (returned by the `toString` function) is used to replace '`@{...}`'. For example:

```
class.name <- 'STAT 624'
s %~% '"@{tolower(class.name)}".reverse'

## [1] "426 tats"
```

The code snippet can contain any number of '`@{...}`' embedded expressions. String interpolation is enabled or disabled globally using the `interpolate` option in the `intpSettings` function. The `%~%` and `%.~%` operators are shorthand substitutes for the `intpEval` function which provides call-specific settings for `interpolate` and `quiet`.

Getting and setting values

Values may be copied between Scala and R using `intpSet` and `intpGet` functions:

```
normal.Mean0.Sd1 <- rnorm(10)
normal.Mean0.Sd1

##  [1]  0.07073287  0.90572460 -0.78091914 -0.07300452 -0.32347637
##  [6] -1.90534795  0.64884201 -0.77225445  0.19224729 -0.28009645

intpSet(s, 'x', normal.Mean0.Sd1)
```

```
intpEval(s, 'val y = x.map(2 * _ + 1)')

normal.Mean1.Sd2 <- intpGet(s, 'y')
normal.Mean1.Sd2

##  [1]  1.1414657  2.8114492 -0.5618383  0.8539910  0.3530473 -2.8106959
##  [7]  2.2976840 -0.5445089  1.3844946  0.4398071
```

The shorthand substitutes for the previous calls to `intpSet` and `intpGet` are

```
s$x <- normal.Mean0.Sd1
normal.Mean1.Sd2 <- s$y
```

Notice that `s$def(...` and `s$do(...` do not conflict with the shorthand substitutes to `intpSet` and `intpGet` because `def` and `do` are reserved words in Scala and are, therefore, not valid identifiers.

Memory management

We now make a few remarks on memory management. To set the maximum Java heap size, use the optional `java.heap.maximum` argument in the `scalaInterpreter` function. Due to an unresolved bug (SI-4331) in the Scala REPL (read-eval-print-loop), memory associated with values returned by Scala's REPL cannot be recovered within that instance of the Scala interpreter even if the identifier is later set to another value (e.g., `null`). Specifically, bound values from calls to the `intpEval` function and its shorthand substitutes `%~%` and `%.~%` are subject to this memory leak (e.g., `a`, `b`, `c`, `res2`, and `y` in the previous code). Likewise, values bound using `intpSet` and its shorthand substitute are also subject to the memory leak (e.g., `x` in the previous code).

In contrast, memory *not* subject to this leak are those associated with calls to `s$do(...`, functions returned by `s$def(...`, and calls to methods of Scala objects. No special handling is needed if the return value of these functions is a primitive, vector, or matrix. For example, the following code requires no memory management.

```
e <- s$do('scala.util.Random')$new()
sapply(1:1000, function(i) e$nextGaussian())
f <- s$def('n: Int', 'Array.fill(n) { scala.util.Random.nextGaussian }')
g1 <- f(1000)
```

If the return value of one of these functions is a reference, the associated memory in Scala will not be recovered until the R's `ScalaInterpreterReference` object no longer exists *and* the garbage collection function `intpGC` is called, or the `gc` argument is set to `TRUE`. Because this garbage collection is somewhat time consuming, the default value of the `gc` argument is `FALSE`. Continuing with the previous example, consider this code:

```
g2 <- f(1000, as.reference=TRUE)
```

The variable g2 is a reference to memory owned by the embedded Scala interpreter. The embedded Scala interpreter will not recover the associated memory until g2 is removed and intpGC is called:

```
rm(g2)
intpGC(s)
```

Callbacks from Scala into the original R interpreter

When an embedded Scala instance is created with the scalaInterpreter function, an instance of the Scala class org.ddahl.rscala.callback.RClient is bound to the identifier R. This object R provides access to the original R interpreter. Having the ability to callback into the original R interpreter facilitates writing generic functions in Scala that can be customized by an R user with no knowledge of Scala. The full Scaladoc for the RClient class, including examples, is available in the online supplement. Here we merely demonstrate this feature with a simple example that shows R calling an inline Scala method which itself calls an R function. Consider this first implementation:

```
myMean <- function(x) {
  cat('Here I am.\n')
  mean(x)
}

callRFunc <- s$def('functionName: String, x: Array[Double]', '
  R.xx = x
  R.evalD0(s"$functionName(xx)")
')

callRFunc('myMean', 1:100, quiet=FALSE)

## Here I am.
## [1] 50.5
```

In this example, the character vector containing the name of the myMean function is passed to the callRFunc function defined as an inline Scala function. This inline Scala method calls back into the original R interpreter using the evalD0 method of the R object.

A more advanced implementation utilizes the class org.ddahl.rscala.callback.RObject. Objects of this class are created in R with the intpWrap function and can represent arbitrary R objects which can later be used in embedded R code. Using the RObject class can reduce the need to copy data between R and Scala. The class is also documented in the Scaladoc and is demonstrated here:

```
callRFunc <- s$def('f: RObject, x: RObject', '
  R.evalD0(s"$f($x)")
')

callRFunc(intpWrap(s, myMean), intpWrap(s, 1:100), quiet=FALSE)

## Here I am.
## [1] 50.5
```

2.3. Accessing the R interpreter within a Scala or Java application

So far this paper has demonstrated calling Scala code from R. Conversely, the **rscala** package also allows an R interpreter to be embedded in a Scala, Java, or any other JVM-based application that has the **rscala** JAR in its classpath. Two versions of the JAR file are provided: one for each of Scala 2.10.x and Scala 2.11.x. Find the path of these JAR files using

```
rscala::rscalaJar()

## [1] "/home/dahl/local/R/devel/lib64/R/library/rscala/java/rscala_2.10-1.0.6.jar"
## [2] "/home/dahl/local/R/devel/lib64/R/library/rscala/java/rscala_2.11-1.0.6.jar"
```

The paths for specific versions of the JAR files are located using

```
rscala::rscalaJar('2.10')

## [1] "/home/dahl/local/R/devel/lib64/R/library/rscala/java/rscala_2.10-1.0.6.jar"

rscala::rscalaJar('2.11')

## [1] "/home/dahl/local/R/devel/lib64/R/library/rscala/java/rscala_2.11-1.0.6.jar"
```

To instantiate an R interpreter in a Scala application, use the special `apply` method of the companion object to the `org.ddahl.rscala.callback.RClient` class:

```
// Scala code
val R = org.ddahl.rscala.callback.RClient()
val side = R.evalS0("sample(c('heads', 'tails'), 1)")
println(s"Your coin landed $side.")

// Your coin landed heads.
```

The previous statements assume that, on Windows, the registry keys option was not disabled during installation. On other operating systems, `R` is assumed to be in the path. If these assumptions are not met or a particular installation of R is desired, the path to the R executable may be specified as the first argument to the `apply` function. The **rscala** package must be available to the selected R executable.

To use the `RClient` class in a Java or other JVM-based application, the Scala JARs and the **rscala** JAR must be in the classpath. These file system paths can be located from R using `rscala::scalaInfo()$jars` and `rscala::rscalaJar()`, respectively. To instantiate an R interpreter, call one of the `apply` static methods. The full Scaladoc for the `org.ddahl.rscala.callback.RClient` class and its companion object, including examples, is available in the online supplement.

2.4. Debugging and error propagation

We now describe how the **rscala** package supports Scala exceptions and R errors. If a run-time exception occurs while executing Scala code within R, a reference to the exception is returned:

```
x <- s %~% '
  val n = 10
  if ( n != 20 ) throw new RuntimeException("n is not 20.")
'
x$"isInstanceOf[RuntimeException]"()

## [1] TRUE

x$getMessage()

## [1] "n is not 20."
```

If `quiet=FALSE` is set using the `intpSettings` function, the stack trace is printed in R's console. Since `x` is a reference to the exception, the stack trace can also be displayed using

```
x$printStackTrace(quiet=FALSE)

## java.lang.RuntimeException: n is not 20.
##  at $line48.$read$$iw$$iw$.<init>(<console>:12)
##  at $line48.$read$$iw$$iw$.<clinit>(<console>)
##  at $line48.$eval$.$result$lzycompute(<console>:5)
##  at $line48.$eval$.$result(<console>:5)
##  at $line48.$eval.$result(<console>)
##  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
##  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
##  at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorIm...
##  at java.lang.reflect.Method.invoke(Method.java:606)
....
```

The above example is a run-time exception, but compile-time exceptions behave in the same way. For example, the code below tries to instantiate a class that is not defined in the current classpath:

```
x <- s %~% 'val a = new ClassThatDoesNotExist()'
x$getMessage()

## [1] "compile-time error"
```

Errors in R code embedded in a Scala or Java application result in a runtime exception. Likewise, errors in callbacks to the R interpreter using the `R` object lead to a reference to the exception. For example, the code below is valid Scala but the function does not exist in the R interpreter and leads to an error:

```
x <- s %~% 'R.eval("nonexistantFunctionInR()")'
x$getMessage()

## [1] "Error in R evaluation."
```

2.5. Developing packages depending on rscala

We now describe how to develop an R package with Scala code using the **rscala** package. Just as several packages on CRAN are implemented using precompiled Java code accessed through the **rJava** package, we provide tools such that Scala developers can develop packages that run embedded Scala source code or execute precompiled Scala or Java code.

An R package based on **rscala** should include `rscala` in the `Depends` or `Imports` field of the package's `DESCRIPTION` file. If using the `Imports` field, the `NAMESPACE` file should also include `import(rscala)`. As with packages that depend on **rJava**, JAR files that the package developer wishes to be in the classpath are placed in the package's `java` directory. The package should define an `.onLoad` function which calls `rscalaPackage(pkgname)`, where `pkgname` is the package's name. Since the namespace is not yet sealed, this function is able to define an environment `E` in the package's namespace. This environment contains `pkgname` (the package's name) and `jars` (the path of all the JAR files in the package's `java` directory). Finally, the R package should call `rscalaLoad(classpath, ...)`, where `classpath` and `...` are passed to the `scalaInterpreter` function and the result is set to `E$s`. If `classpath` is `NULL`, then `E$jars` is used. An acceptable `.onLoad` function may be as simple as

```
.onLoad <- function(libname, pkgname) {
  rscalaPackage(pkgname)
  rscalaLoad()
}
```

The package then has access to an embedded Scala interpreter through `E$s` with which the package can run embedded Scala source code or execute precompiled Scala or Java code. If desired, the call to the `rscalaLoad` function can be outside the `.onLoad` function, waiting until the Scala interpreter `E$s` is first needed. Subsequent calls to `rscalaLoad` have no effect and return immediately.

Because **rscala**'s syntax for calling precompiled code is very similar to **rJava**'s high-level "$ convenience operator," developing a package based on the **rscala** package can be very familiar to those accustomed to the **rJava** package. Take, for example, CRAN's **mailR** package: "Interface to Apache Commons Email to send emails from R" (Premraj 2015). This package depends on the **rJava** package and mostly uses its high-level `$` convenience operator. As a proof of concept, we ported the **mailR** package to **rscala**, replacing the dependency on **rJava**. Among the 264 lines in the R scripts for version 0.4.1 of the **mailR** package, we rewrote 13 lines and added one more line to complete the port. Further, 11 lines in the original **rJava** implementation could be removed because of the way in which **rscala** handles exceptions. As an example of what simple changes needed to be made, consider one expression from the original **rJava**-based package:

```
base_dir <- .jnew("java.io.File", normalizePath(getwd()))
```

In our port to **rscala**, we replaced this line with

```
base_dir <- E$s$do("java.io.File")$new(normalizePath(getwd()))
```

Of course, porting a package that makes frequent use of **rJava**'s low-level interface (e.g., the `.jcall` function) will require more changes. The online supplement contains both the original **mailR** package and our port to **rscala**.

3. Implementation

The **rscala** package is designed around a client/server model using local TCP/IP sockets.

3.1. Implementation of Scala code in R

We first describe how Scala code is accessed from R. The `scalaInterpreter` function asynchronously spawns a new instance of the Java Virtual Machine using the `java` executable by calling the `main` static method of the `scala.tools.nsc.MainGenericRunner` class. This is the same class that the `scala` executable shell script uses when Scala is started from the command line. The arguments supplied to the `java` executable are also the same with the exception that we prepend the classpath to give precedence to a modified `scala.Console` class. Our modification facilitates redirecting console output to the R console via TCP/IP sockets.

The `scalaInterpreter` function uses the `java` process's standard input to start a custom TCP/IP server that is bound to a randomly selected port. The TCP/IP server accepts only a single local connection. The Scala server is implemented using the Java standard library. R repeatedly tries to connect to the Scala server as a client using functions from the **base** R package. The Scala server has an instance of `scala.tools.nsc.interpreter.IMain`, the class representing Scala's REPL. The R client accesses the functionality of the server's REPL class by means of a custom protocol.

Scala's REPL is capable of compiling and executing Scala code on the fly as well as setting and getting values of identifiers. When a particular Scala method is called for the first time in R, a one-time compilation occurs behind the scenes. The compiled method is then cached and re-used for subsequent calls to the same method. As an illustration, consider the code below and notice that the elapsed time is significantly less in the second call to the same method:

```
system.time( s$do("sys")$props("user.name") )

##    user  system elapsed
##   0.003   0.000   0.186

system.time( s$do("sys")$props("user.name") )

##    user  system elapsed
##   0.001   0.000   0.001
```

This difference is most noticeable when the actual execution time is short relative to the one-time compilation time. The `system.time` function in R does *not* measure the CPU time and system time spent in Scala, but the elapsed time is indeed an accurate measure of the wall time.

Callback functionality is implemented over the same TCP/IP sockets. Immediately after the R client requests the execution of Scala code, R becomes a temporary, embedded server for the Scala code to access the original R interpreter through the `R` object (an instance of the `RClient` class). The embedded R server interprets code and sets/gets values using the `parse`, `eval`, `assign`, and `get` functions. After the Scala server executes the Scala code, it tells the embedded R server to exit and the Scala server resumes its usual behavior.

3.2. Implementation of R code in Scala and other JVM applications

The functionality that allows Scala, Java, and other JVM-based applications to access an embedded R interpreter is achieved by reusing the code implementing the previously-discussed callback functionality. In this case, however, there is not an existing instance of the R interpreter. The class `RClient` spawns an R instance, uses the same code to connect the R client to the Scala server, and then immediately starts the embedded R server. There are actually two versions of the `RClient` class. The Java version (`org.ddahl.rscala.java.RClient`) is a thin wrapper over the Scala version (`org.ddahl.rscala.callback.RClient`). The wrapper provides familiar Javadoc to Java developers and also hides various Scala-specific features.

4. Comparisons to other software

In this section we compare and contrast the **rscala** package to other related software.

4.1. Comparison to rJava

The **rJava** package, together with the included JRI (Java/R interface) software for callbacks, serves as inspiration for our **rscala** package. Indeed, we adopt **rJava**'s high-level `$` convenience operator in **rscala**. The implementations of the two packages are, however, quite different. The **rscala** package uses TCP/IP sockets for communication between processes. As a consequence, R, Scala, and/or Java can be updated without reinstalling the package. In contrast, **rJava** uses Java's JNI (Java native interface) to dynamically load the JVM shared library in R's process space. The versions of R and Java must be compatible (i.e., both 32-bit or both 64-bit binaries) and updates to either R or Java usually require reinstalling the package or running `R CMD javareconf`. Callbacks are provided in **rJava** using the JRI software included with the **rJava** package. This requires that R be compiled with the option `--enable-R-shlib`. This option is enabled by default on Windows and Mac OS X, but it is not enabled for most Linux distributions. While installing **rJava** requires a bit more care and maintenance, we will see in Section 4.4 that it invokes methods faster than **rscala** does. JRI also provides a richer interface to R objects than does **rscala**.

Since Scala compiles to Java bytecode and runs on the Java Virtual Machine, some of the functionality in the **rscala** package can be replicated using the **rJava** package. Both **rscala** and **rJava** can call precompiled Java bytecode. Features that **rJava** does not support include defining inline functions (Section 2.2.4), evaluating arbitrary code (Section 2.2.5), and explicit support for Scala.

We now consider the difficulty of calling Scala from **rJava**. After working on several projects using **rJava** to access Scala code in R, we found that Scala was cumbersome to use. Scala provides several features that do not map directly to Java equivalents. The Scala compiler uses name mangling, behind-the-scenes code generation, and other techniques when compiling to Java bytecode. Calling Scala code that makes use of these features from **rJava** requires some understanding of Scala's compiler at a minimum, but often the developer is required to write a Java-friendly wrapper method in Scala that hides advanced Scala features. We illustrate this scenario with a few examples. Consider the following Scala code:

```
// Scala code
val a = List[String]("apples", "bananas")

var b = List.empty[String]
b = "bananas" :: b
b = "apples"  :: b

println(a == b)

// true
```

The equivalent R code to create the objects `a` and `b` in Scala using the **rscala** package is a straight-forward syntactic translation:

```
a <- s$do('List')$apply('apples', 'bananas')

b <- s$do('List')$'empty[String]'()
b <- b$'::'('bananas')
b <- b$'::'('apples')

cat( a$'=='(b) )

## TRUE
```

There appears to be no translation of the Scala code to create the object `a` using the **rJava** package. This is because the Scala compiler uses internal classes to implement the *varargs* in the `apply` method. One solution would be to write a Java-friendly wrapper method that the developer could call using **rJava**. The object `b` can be created in R using the **rJava** package, but the developer has to know: i. that the `::` method gets renamed to `$colon$colon` because `::` does not conform to JVM specifications, ii. that the Scala companion object `List` is represented in Java bytecode by the class `List$`, and iii. that Scala's `==` method compiles to Java's `equals` method. Below is the equivalent R code using the **rJava** package, except one cannot translate the Scala code for the object `a` to an **rJava** equivalent:

```
library('rJava')
.jinit()
.jaddClassPath(rscala::scalaInfo()$jars)

b <- .jnew('scala.collection.immutable.List$')$empty()
b <- b$'$colon$colon'('bananas')
b <- b$'$colon$colon'('apples')

b$equals(b)

## [1] TRUE
```

4.2. Comparison to Rcpp

From a user's perspective, the **rscala** package is very similar to the **Rcpp** package (Eddelbuettel and François 2011), with the obvious difference that **rscala** supports Scala and **Rcpp** supports C++ and C. Both packages allow the user to define an inline function within R and return an R function which wraps the newly-compiled function. Both also support callbacks to the original R interpreter and both provide a foundation for building other packages. Whereas **Rcpp** loads dynamically-compiled code into R's process space, **rscala** uses TCP/IP sockets for communication between the R and JVM processes. The benchmarks in Section 4.4 will show that **Rcpp** is generally unmatched in execution speed, especially for short-lived function calls.

4.3. Comparison to Rserve

Rserve (Urbanek 2013b) "is a TCP/IP server which allows other programs to use facilities of R... without the need to initialize R or link against [the] R library" (`http://www.rforge.net/Rserve`). Whereas the **rscala** package only supports Scala, Java, and other JVM-based applications, a client for **Rserve** can be written in any language that supports TCP/IP sockets. Clients are currently available for Java, C++, R, Python, C#, .Net/CLI, and Ruby. Note that **rJava**'s JRI links against the R library and one of the benefits of the **Rserve**'s client/server approach is that it "prevent[s] multi-threading problems that arise when linking against R library directly" (`http://www.rforge.net/Rserve/doc.html`). The **rscala** package shares this benefit because it too is implemented using a client/server approach. Like **rJava**'s JRI, however, **Rserve** requires that R be compiled with the option `--enable-R-shlib` which is the default on Windows and Mac OS X, but is not the default on most Linux distributions. The **rscala** package uses a standard installation of R on all platforms. **Rserve** provides the same rich interface to R objects as **rJava**. Benchmarks in Section 4.4 will show that **Rserve** is faster than **rscala**. Finally, we note that **rscala** is neutral with respect to choice of operating system, but **Rserve** has some limitations on Windows and the user is advised to not "use Windows unless you really have to" (`http://www.rforge.net/Rserve/doc.html`).

4.4. Benchmarks

In this section we explore the performance of the **rscala** package relative to other software used to accomplish the same tasks. All benchmarks were run in R 3.2.0 compiled from source with the option `--enable-R-shlib`. The following packages were installed from CRAN: **rscala** 1.0.6, **rJava** 0.9.6, **Rcpp** 0.11.5, **Rserve** 1.7.3, and **microbenchmark** 1.4.2 (Mersmann, Beleites, Hurling, and Friedman 2014). The benchmarking workstation runs Ubuntu 14.04.02, has 32GB of RAM, and has an Intel i7-3930K CPU with 6 cores and hyperthreading enabled yielding a total of 12 virtual cores. The workstation has Scala 2.11.6 and Java 1.7.0_17. Unless stated otherwise, our benchmarking functions are single-threaded.

Computing the mean of a sample

Consider the simple task of computing the mean of a random sample of size n from the uniform distribution on the unit interval. We explore several implementations of this task through two functions based on standard R, one function based on **Rcpp**, three functions based on **rJava**, and five functions based on **rscala**. For example, `standardR2`, `rJava1`, `rJava2`, `rscala1`, and `rscala2` all involve looping in R instead of the compiled code. Other reasons for the different

Function	1 Sample		10 Samples		100 Samples		1,000 Samples		10,000 Samples	
	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio
`standardR2`	0.0000	2	0.0000	4	0.0002	19	0.002	84	0.02	135
`rJava1`	0.0003	65	0.0008	109	0.0054	481	0.042	1,864	0.40	2,685
`rJava2`	0.0023	442	0.0176	2,275	0.1567	14,005	1.524	67,266	15.22	101,530
`rscala1`	0.0017	336	0.0091	1,174	0.0643	5,744	0.455	20,097	4.47	29,808
`rscala2`	0.0018	351	0.0083	1,077	0.0582	5,200	0.405	17,873	3.94	26,263

Table 3: Benchmarks of functions whose performance relative to the **Rcpp** function gets worse as the sample size n increases. The definitions of the functions are provided in the online supplement.

Function	1 Sample		100 Samples		10,000 Samples		1,000,000 Samples		100,000,000 Samples	
	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio
`standardR1`	0.0000	2.9	0.0000	2.9	0.0006	2.3	0.0379	2.31	3.2078	2.27
`rJava3`	0.0001	18.3	0.0001	14.5	0.0005	2.1	0.0232	1.41	2.1679	1.53
`rscala3`	0.0005	123.5	0.0009	144.0	0.0019	8.1	0.0436	2.65	3.4078	2.41
`rscala4`	0.0005	117.5	0.0005	84.6	0.0013	5.5	0.0259	1.58	2.1679	1.53
`rscala5`	0.0005	117.1	0.0005	84.9	0.0012	5.0	0.0258	1.57	2.1686	1.53

Table 4: Benchmarks of functions whose performance relative to the **Rcpp** function gets better as the sample size n increases. The definitions of the functions are provided in the online supplement.

versions include calling inline functions versus precompiled code and using package-specific features such as **rJava**'s low-level and high-level interfaces. The online supplement has the complete source code for functions and benchmarking. Note that the R and **Rcpp** functions use the same underlying C code to sample random uniforms. Likewise, **rJava** and **rscala** use the same code to sample random uniforms, but this code is different from that used by R and **Rcpp**. We find that the **Rcpp** function is the fastest (not listed in the table) and choose it as the reference from which performance ratios of the other methods are calculated.

Using the package **microbenchmark**, we run the benchmarking functions 50 times for each of the following sample sizes for n: 1, 10, 100, …, 10,000. We find that as n increases, the performance relative to **Rcpp** gets worse for some functions. Their results are displayed in Table 3. Each of these functions involves looping in R instead of the compiled code. Among these slow implementations, we note that `standardR2` is the fastest and that the performance of the **rscala** implementations lies between two **rJava** implementations. The only difference in these two **rJava** implementations is that the slower version uses the high-level `$` convenience operator.

We test the other functions for even larger n: 1, 100, …, 100,000,000. The performance of the other functions improves with n but seems to asymptote. These results are displayed in Table 4. The functions `rJava3` and `rscala5` call exactly the same Java bytecode and the initial performance difference is due to the overhead for the TCP/IP protocol used by **rscala**. As n increases, the performance difference virtually disappears. The function `rscala4` is written in Scala using a while-loop and compiles to essentially the same Java bytecode used in `rJava3` and `rscala5`. The function `rscala3` demonstrates the well-known fact that for-loops in Scala are not optimized for performance.

Function	1 Second Sleep		0.01 Seconds Sleep		0 Seconds Sleep	
	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio
`callback.Rcpp`	1.0011	1.000	0.0101	1.003	0.0000	6.246
`callback.rJava`	1.0013	1.000	0.0102	1.013	0.0001	22.559
`callback.rscala`	1.0031	1.002	0.0116	1.156	0.0012	237.130

Table 5: Benchmarks of functions calling a simple R function that sleeps for 1, 0.01, and 0 seconds. Performance ratios are relative to the **Rcpp** function. The definitions of the functions are provided in the online supplement.

Calling into the JVM and back into R

To further investigate the performance of **rscala** relative to other software, we benchmark the performance of **Rcpp**, **rJava**, and **rscala** in calling a very simple R function:

```
sleep <- function(secs) {
  Sys.sleep(secs)
  rnorm(1)
}
```

This function takes an argument `secs` and pauses the function for the specified time. Although this function is of little practical utility, we use it to simulate functions in R whose execution may be very fast (by sleeping 0 seconds), somewhat computationally expensive (by sleeping 1 second), and something in between (by sleeping 0.01 seconds). It is interesting to investigate the invocation overhead of the three methods in calling the functions with the different sleep times. The online supplement has complete code for the functions and benchmarking. The **rJava** function uses the low-level interface (as opposed to the high-level `$` convenience operator) for performance reasons. The **Rcpp** function is again the fastest and we use it as the reference when calculating performance ratios for the other methods.

We see from Table 5 that the **rscala** function has the highest invocation latency. The ratios indicate that the protocol overhead is important for short-lived R functions (i.e., when sleeping 0 seconds) but negligible for long-lived R functions (i.e., when sleeping 1 second).

Calling into R from a JVM application

We now consider the performance and simplicity of calling R code from within a Java application. Suppose there is an array of doubles in the Java application and we wish to produce a normal q-q plot of the data using R. We compare an **rscala** solution against two alternatives: i. spawning a separate R process using the Rscript executable, and ii. using the Java client for **Rserve**. We measure the time needed to produce 200 plots of randomly-generated data. The online supplement contains the related Rscript and Java source code.

Spawning a separate R process using Rscript requires somewhat tedious coding to save the data to a file for R to read and plot. Further, since a new R process is started for each plot, the performance is rather poor as well, taking a total of 26.7 seconds for the 200 plots. Conversely, the **Rserve** and **rscala** solutions are both more convenient and quite similar. The **Rserve** solution takes only 0.6 seconds and the **rscala** solution takes 1.4 seconds demonstrating that the protocol overhead of **Rserve** is less than that of **rscala**.

5. Case studies

5.1. Bayesian logistic regression

In this section we use a custom Markov chain Monte Carlo (MCMC) algorithm to fit a Bayesian logistic regression model in R. This model can also be estimated in R through a variety of other algorithms using R packages such as RStan (Stan Development Team 2014), rbugs (Yan and Prates 2013), R2jags (Su and Yajima 2015), rjags (Plummer, Stukalov, and Denwood 2015), R2WinBUGS (Sturtz, Ligges, and Gelman 2005a), R2OpenBUGS (Sturtz, Ligges, and Gelman 2005b), and BRugs (Thomas, O'Hara, Ligges, and Sturtz 2006). Our interest here, however, is *not* to determine the best algorithm for Bayesian model fitting, but rather to compare the convenience and computational speed of Scala, C, and standard R in implementing the same commonly-used, computationally-expensive algorithm.

Studies show that babies who breastfeed have a significantly reduced risk of health problems such as HIV, obesity, and neurological defects due to the fact that milk contains natural immune components that are not present in baby formula (Stuebe Fall 2009). Medical professionals have observed that premature babies take longer to learn how to breastfeed and many are not breastfeeding by the time they are ready to leave the hospital (Briere, Lucas, McGrath, Lussier, and Brownell 2015). To quantify this relationship, a statistician wishes to fit a logistic regression model in which the probability of breastfeeding before leaving the hospital is modeled as a function of gestational age. Data is available on $n = 64$ infants where $y_i = 1$ if the baby is breastfeeding at departure and x_i is the gestational age. The model is

$$\Pr(y_i = 1) \; = \; \frac{1}{1 + \exp\{-(\beta_0 + \beta_1 x_i)\}}$$

where β_0 and β_1 are unknown parameters to estimate. We assume independent Normal(-15, 5) and Normal(1, 1) priors on β_0 and β_1, respectively.

Since the model is not conjugate, we obtain samples from the joint posterior distribution using a Metropolis sampling algorithm with a uniform random walk. We examine the convenience and computational speed of running a single MCMC chain as well as simultaneously running 6 and 12 independent chains in parallel. Recall that the test machine has 6 cores and hyperthreading is enabled making it appear that there are 12 cores. We run the chains for 1,001,000 iterations using the prior means as the starting state. We discard the first 1,000 iterations for burn-in. This same MCMC algorithm is implemented in Scala using **rscala**, in C using the `.C` interface, and in standard R. Note that a C implementation using the `.Call` interface requires a more intimate knowledge of R's C API. The implementations, data, and benchmarking code are all available in the online supplement.

The results are in Table 6 along with wall times and performance ratios (relative to Scala). Surprisingly, we find that the Scala implementation is slightly faster than the C implementation. This may be due to the fact that the Scala code uses a different library for evaluating the log of the binomial probabilities or because the `.C` interface is not as performant as the `.Call` interface. The standard R implementation is about 13-15 times slower.

The breastfeeding infants example involves $n = 64$ observations at one of six unique values of the predictor. To see how the results might change for larger datasets, we simulated a dataset of $n = 20,000$ observations at one of 200 unique values of the predictor. The results are found in Table 7. The most striking finding is that the standard R implementation

	1 core		6 cores		12 cores	
Language	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio
Scala using **rscala**	1.12	1.00	0.29	1.00	0.18	1.00
C using `.C`	1.48	1.32	0.31	1.09	0.24	1.36
Standard R	15.79	14.11	3.78	13.15	2.70	15.08

Table 6: Breastfeeding infants example: Wall time and relative performance of three implementations of the same algorithm using 1, 6, and 12 cores.

	1 core		6 cores		12 cores	
Language	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio
Scala using **rscala**	47.35	1.00	10.82	1.00	7.87	1.00
C using `.C`	53.17	1.12	11.49	1.06	7.67	0.97
Standard R	75.51	1.59	16.15	1.49	11.30	1.44

Table 7: Simulated data example: Wall time and relative performance of three implementations of the same algorithm using 1, 6, and 12 cores.

improves dramatically, now being only 1.4-1.6 times slower than the Scala implementation. The explanation is that the standard R implementation, for this simulated data example, spends 72.5% of the time in the function `dbinom`. This function is implemented in C using the `.Call` interface. It is not surprising that a C implementation and another implementation based mostly on C (i.e., the standard R implementation) perform similarly.

5.2. Scala web application using R

Suppose a Scala developer maintains a website based on the **Play Framework** (Typesafe 2015) for web development. He is tasked with adding new functionality to the website which, for any airport and any year, displays: i. a graph of daily high and low temperatures and ii. the yearly high and low temperatures with their dates. He can look for a Scala or Java library to obtain the weather data and another library for plotting. If he is already familiar with R, however, the data, computation, and plotting tasks are easily accomplished using the **weatherData** (Narasimhan 2014) and **ggplot2** (Wickham 2009) packages:

```
library("weatherData")
library("ggplot2")
theme_set(theme_gray(base_size=18))

.cache <- new.env()
cache <- function(location, year) {
  key <- paste0(location, "-", year)
  if ( exists(key, envir=.cache) ) get(key, envir=.cache)
  else {
    d <- na.omit(getWeatherForYear(location, year))

    maxT <- d[which.max(d$Max_TemperatureF), c("Max_TemperatureF", "Date")]
    minT <- d[which.min(d$Min_TemperatureF), c("Min_TemperatureF", "Date")]
    p <- ggplot(d, aes(x=Date)) + coord_cartesian(ylim = c(-10, 110))
    p <- p + labs(x="", y=expression(paste("Temperature (", degree, "F)")))
    p <- p + geom_line(aes(y=Max_TemperatureF))
    p <- p + stat_smooth(se=FALSE, aes(y=Max_TemperatureF), method="loess", span=0.3)
    p <- p + geom_line(aes(y=Min_TemperatureF))
    p <- p + stat_smooth(se=FALSE, aes(y=Min_TemperatureF), method="loess", span=0.3)
    p <- p + geom_ribbon(aes(ymin=Min_TemperatureF, ymax=Max_TemperatureF),
                         fill = "tomato", alpha = 0.4)

    fn <- normalizePath(paste0(tempdir(), .Platform$file.sep, key, ".svg"))
    svg(fn, width=6, height=4)
    print(p)
    dev.off()

    fmt <- "%B %e, %Y"
    result <- list(filename=fn, minValue=minT[1, 1], minDate=format(minT[1, 2], fmt),
                                maxValue=maxT[1, 1], maxDate=format(maxT[1, 2], fmt))
    assign(key, result, envir=.cache)
    result
  }
}
```

The task for the developer now becomes how to serve the plot and information in the **Play Framework**. Two possible solutions are to use Rscript or **Rserve**, as discussed in Section 4.4.3. Moving away from the **Play Framework**, one might consider setting up a web server using RApache (Horner 2013), CGIwithR (Firth 2003), or Shiny (Chang *et al.* 2015). In this case study, however, we assume that one has already invested resources in setting up and maintaining a website based on the **Play Framework**. We demonstrate how easy it is to add the desired functionality to the existing infrastructure using **rscala**.

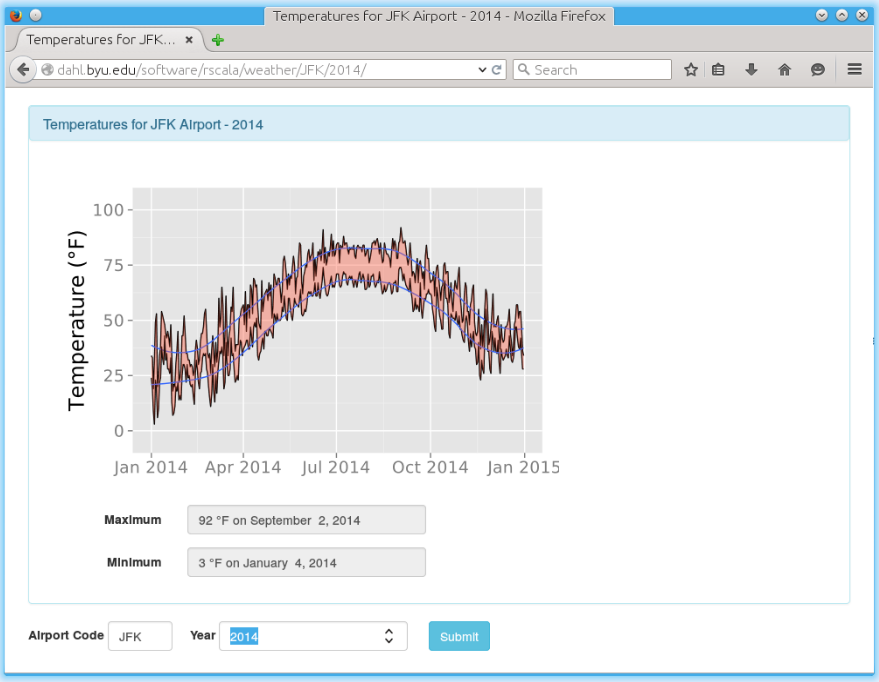
Entire books have been written on the **Play Framework** (e.g., Hilton, Bakker, and Canedo (2013), Petrella (2013)), so here we only highlight issues specific to using **rscala** in the **Play Framework**. First, the **rscala** JAR should be copied to the `lib` directory of the web application. The location of this JAR is available from R using `rscala::rscalaJar('2.11')`. In the *controller* for the web application, we instantiate a Scala interpreter using

```
private val R = org.ddahl.rscala.callback.RClient()
```

Note that the **Play Framework** is highly parallelized, but recall that the bridge between R and Scala is not thread-safe. A pool of instances of the `RClient` class could be employed but, for simplicity, here we simply wrap any access to the single `R` object using the `R.synchronized` method. The R code listed above is evaluated in the *controller* using the `eval` method of the `R` object. Setting up the *model*, *view*, and other aspects of the *controller* are developed as usual in the **Play Framework**. Figure 1 shows a screenshot of the web application and the web application is hosted here:

http://dahl.byu.edu/software/rscala/temperature/

We provide the complete source and instructions for setting up the application in the online supplement.


Temperatures for JFK Airport - 2014 - Mozilla Firefox
Temperatures for JFK...
dahl.byu.edu/software/rscala/weather/JFK/2014/
Search
Temperatures for JFK Airport - 2014
Temperature (°F)
100
75
50
25
0
Jan 2014
Apr 2014
Jul 2014
Oct 2014
Jan 2015
Submit

Maximum: 92 °F on September 2, 2014

Minimum: 3 °F on January 4, 2014

Airport Code: JFK

Year: 2014

Figure 1: Screenshot of the temperatures web application developed with the **Play Framework** using **rscala**.

6. Conclusion

This paper introduced the **rscala** package that allows users to seamlessly incorporate R and Scala in one program, exploiting each language's respective strengths. We believe that the **rscala** package is stable and complete. In the future, we would like to support more data structures in R (e.g., lists and data frames). While callbacks from Scala to the original R interpreter are supported, we would also like to support callbacks from R to Scala as well. We also suspect that our client/server protocol could be extended to support other JVM-based languages such as Jython (Jython developers 2015) and jRuby (JRuby team 2015).

Acknowledgements

The author's work on this paper is supported by NIH NIGMS R01 GM104972. The author thanks the following students: Richard Payne, Deepthi Uppalapati, Scott K. Ferguson, and Floid Gilbert.

References

Bell ET (1938). "The iterated exponential integers." *Annals of Mathematics*, **39**, 539–557. URL http://www.jstor.org/stable/1968633?seq=1#page_scan_tab_contents.

Briere CE, Lucas R, McGrath JM, Lussier M, Brownell E (2015). "Establishing Breastfeeding with the Late Preterm Infant in the NICU." *Journal of Obstetric, Gynecologic and Neonatal Nursing*, **44**, 102–113.

Chang W, *et al.* (2015). ***shiny***. R package version 0.11.1, URL `http://shiny.rstudio.com/`.

Dahl DB (2015). ***rscala**: Bi-Directional Interface Between R and Scala with Callbacks.* R package version 1.0.6, URL `http://CRAN.R-project.org/package=rscala`.

Eddelbuettel D, François R (2011). "**Rcpp**: Seamless R and C++ Integration." *Journal of Statistical Software*, **40**(8), 1–18. URL `http://www.jstatsoft.org/v40/i08/`.

Firth D (2003). "**CGIwithR**: Facilities for processing web forms using R." *Journal of Statistical Software*, **8**, 1–8. R package version 0.50, URL `http://www.omegahat.org/CGIwithR/`.

Hilton P, Bakker E, Canedo F (2013). *Play for Scala: Covers Play 2*. 1st edition. Manning Publications. ISBN 1617290793.

Horner J (2013). ***rApache**: Web application development with R and Apache*. URL `http://www.rapache.net/`.

Mersmann O, Beleites C, Hurling R, Friedman A (2014). ***microbenchmark**: Accurate Timing Functions*. R package version 1.4-2, URL `http://CRAN.R-project.org/package=microbenchmark`.

Narasimhan R (2014). ***weatherData**: Get Weather Data from the Web*. R package version 0.4.1, URL `http://ram-n.github.io/weatherData/`.

Odersky M, *et al.* (2004). "An Overview of the Scala Programming Language." *Technical Report IC/2004/64*, EPFL, Lausanne, Switzerland.

Oracle (2015). *Java*. Redwood Shores, CA. URL `http://www.java.com/`.

Petrella A (2013). *Learning Play! Framework 2*. Packt Publishing. ISBN 1782160124.

Plummer M, Stukalov A, Denwood M (2015). ***rjags**: Bayesian Graphical Models using MCMC*. R package version 3-15, URL `http://mcmc-jags.sourceforge.net`.

Premraj R (2015). ***mailr**: Interface to Apache Commons Email to send emails from R*. R package version 0.4.1, URL `https://github.com/rpremraj/mailR`.

JRuby team (2015). *JRuby: The Ruby Programming Language on the JVM*. URL `http://jruby.org/`.

Jython developers (2015). *Jython: Python for the Java Platform*. URL `http://www.jython.org/`.

R Core Team (2015). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL `http://www.R-project.org/`.

Stan Development Team (2014). *Stan Modeling Language Users Guide and Reference Manual, Version 2.5.0*. URL `http://mc-stan.org/`.

Stuebe A (Fall 2009). "The Risk of Not Breastfeeding for Mothers and Infants." *Reviews in Obstetrics and Gynecology*, **2**, 222–231. URL http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2812877/.

Sturtz S, Ligges U, Gelman A (2005a). "R2WinBUGS: A Package for Running WinBUGS from R." *Journal of Statistical Software*, **12**(3), 1–16. URL http://www.jstatsoft.org.

Sturtz S, Ligges U, Gelman A (2005b). "R2WinBUGS: A Package for Running WinBUGS from R." *Journal of Statistical Software*, **12**(3), 1–16. URL http://www.jstatsoft.org.

Su YS, Yajima M (2015). ***R2jags**: Using R to Run 'JAGS'*. R package version 0.5-6, URL http://CRAN.r-project.org/package=R2jags.

Thomas A, O'Hara B, Ligges U, Sturtz S (2006). "Making BUGS Open." *R News*, **6**(1), 12–17. URL http://cran.r-project.org/doc/Rnews/.

Typesafe (2015). "Play Framework." URL https://www.playframework.com/.

Urbanek S (2013a). ***rJava**: Low-Level R to Java Interface*. R package version 0.9-4, URL http://CRAN.R-project.org/package=rJava.

Urbanek S (2013b). ***Rserve**: Binary R server*. R package version 0.6-8.1, URL http://CRAN.R-project.org/package=Rserve.

Wickham H (2009). *ggplot2: elegant graphics for data analysis*. Springer New York. ISBN 978-0-387-98140-6. URL http://had.co.nz/ggplot2/book.

Yan J, Prates M (2013). ***rbugs**: Fusing R and OpenBugs and Beyond*. R package version 0.5-9, URL http://CRAN.r-project.org/package=rbugs.

Affiliation:

David B. Dahl
Associate Professor
Department of Statistics
Brigham Young University
223 TMCB
Provo, UT 84602
E-mail: dahl@stat.byu.edu
URL: http://dahl.byu.edu

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/

Volume VV, Issue II *Submitted:* 2015-05-15
MMMMMM YYYY *Accepted:* yyyy-mm-dd