Integration of R and Scala Using rscala

David B. Dahl
Brigham Young University

Abstract

The **rscala** software is a simple, two-way bridge between R and Scala that allows users to leverage the unique strengths of both languages in a single project. Scala classes can be instantiated from R and Scala methods can be called. Arbitrary Scala code can be executed on-the-fly from within R and callbacks to R are supported. R packages can be developed based on Scala. Conversely, **rscala** also enables R code to be embedded within a Scala application. The **rscala** package is available on CRAN and has no dependencies beyond base R and the Scala standard library.

Keywords: Java virtual machine (JVM), language bridges, R, Scala.

1. Introduction

This paper introduces **rscala** (Dahl 2018c), software that provides a bridge between R (R Core Team 2018) and Scala (Odersky *et al.* 2004). The goal of **rscala** is to allow users to leverage the unique strengths of Scala and R in a single program. For example, R packages can implement computationally intensive algorithms in Scala and, conversely, Scala applications can take advantage of the vast array of statistical packages in R. Callbacks from embedded Scala into R are supported. The **rscala** package is available on the Comprehensive R Archive Network (CRAN). Also, R can be embedded within a Scala application by adding a one-line dependency declaration in Scala Build Tool (SBT).

Scala is a general-purpose programming language that strikes a balance between execution speed and programmer productivity. Scala programs run on the Java virtual machine (JVM) at speeds comparable to Java. Scala features object-oriented, functional, and imperative programming paradigms, affording developers flexibility in application design. Scala code can be concise, thanks in part to: type inference, higher-order functions, multiple inheritance through traits, and a large collection of libraries. Scala also supports pattern matching, operator overloading, optional and named parameters, and string interpolation. Scala encourages immutable data types and pure functions (i.e., functions without side-effects) to simplify parallel processing and unit testing. In short, the Scala language implements many of the most productive ideas in modern computing. To learn more about Scala, we suggest *Programming in Scala* (Odersky *et al.* 2016) as an excellent general reference.

Because Scala is flexible, concise, and quick to execute, it is emerging as an important tool for scientific computing. For example, Spark (Zaharia *et al.* 2016) is a cluster-computing framework for massive datasets written in Scala. Several books have been published recently on using Scala for data science (Bugnion 2016), scientific computing (Jancauskas 2016), machine learning (Nicolas 2014; Karim and Alla 2017), and probabilistic programming (Pfeffer 2016). We believe that Scala deserves consideration when looking for an efficient and convenient general-purpose programming language to complement R.

R is a scripting language and environment developed by statisticians for statistical computing and graphics. Like Scala, R supports a functional programming style and provides immutable data types. Scala programmers who learn R will find many familiar concepts, despite the syntactical differences. R has a large user base and over 13,000 actively maintained packages on CRAN. Hence, the Scala community has a lot to gain from an integration with R.

R code can be very concise and expressive, but may run significantly slower than compiled languages. In fact, computationally intensive algorithms in R are typically implemented in compiled languages such as C, C++, Fortran, and Java. The **rscala** package adds Scala to this list of high-performance languages that can be used to write R extensions. The **rscala** package is similar in concept to **Rcpp** (Eddelbuettel and François 2011), an R integration for C and C++, and **rJava** (Urbanek 2018), an R integration for Java. Though the **rscala** integration is not as comprehensive as **Rcpp** and **rJava**, it provides the following important features to blend R and Scala. First, **rscala** allows arbitrary Scala snippets to be included within an R script and Scala objects can be created and referenced directly within R code. These features allow users to integrate Scala solutions in an existing R workflow. Second, **rscala** supports callbacks to R from Scala, which allow developers to implement general, high-performance algorithms in Scala (e.g., root finding methods) based on user-supplied R functions. Third, **rscala** supports developing R packages based on Scala which allows Scala developers to make their work available to the R community. Finally, the **rscala** software makes it easy to incorporate R in a Scala application without even having to install the R package. In sum, **rscala**'s feature-set makes it easy to exploit the strengths of R and Scala in a single project.

We now discuss the implementation of **rscala** and some existing work. Since Scala code compiles to Java byte code and runs on the JVM, one could access Scala from R via **rJava** and then benefit from the speed of shared memory. We originally implemented our Scala bridge using this technique, but later moved to a custom TCP/IP protocol for the following reasons. First, **rJava** and Scala both use custom class loaders which, in our experience, conflict with each other in some cases. Second, since **rJava** links to a single instance of the JVM, one **rJava**-based package can configure the JVM in a manner that is not compatible with a second **rJava**-based package. The **rscala** package creates a new instance of the JVM for each bridge to avoid such conflicts. Third, the simplicity of no dependencies beyond Scala's standard library and base R is appealing from a user's perspective. Finally, callbacks in **rJava** are provided by the optional JRI component, which is only available if R is built as a shared library. While this is the case on many platforms, it is not universal and therefore callbacks could not be a guaranteed feature of **rscala** software if it were based on **rJava**'s JRI.

The discussion of the design of **rscala** has so far focused on accessing Scala from R. The **rscala** software also supports accessing R from Scala using the same TCP/IP protocol. This ability is an offshoot of the callback functionality. Since Scala can call Java libraries, those who are interested in accessing R from Scala should also consider the Java libraries **Rserve** (Urbanek 2013) and **RCaller** (Satman 2014). **Rserve** is also "a TCP/IP server which allows other programs to use facilities of R" (`http://www.rforge.net/Rserve`). **Rserve** clients are available for many languages including Java. **Rserve** is fast and provides a much richer API than **rscala**. Like **rJava**, however, **Rserve** also requires that R be compiled as a shared library. Also, Windows has some limitations such that **Rserve** users are advised not to "use Windows unless you really have to" (`http://www.rforge.net/Rserve/doc.html`).

The paper is organized as follows. Section 2 describes using Scala from R. Some of the more important topics presented there include the data types supported by **rscala**, embedding Scala snippets in an R script, executing methods of Scala references, and calling back into R from Scala. We also discuss how to develop R packages based on Scala. Section 3 describes using R from Scala. In both Sections 2 and 3, concise examples are provided to help describe the software's functionality. Section 4 provides a case study to show how Scala can easily be embedded in R to significantly reduce computation time for a simulation study. We conclude in Section 5 with potential features for future work.

2. Accessing Scala in R

This section provides a guide to accessing Scala from R. Those interested in the reverse — accessing R from Scala — will also benefit from understanding the ideas presented here.

2.1. Installation

The **rscala** package is available on the Comprehensive R Archive Network (CRAN) and can be installed by executing the following R expression.

```
install.packages("rscala")
```

The **rscala** package requires Scala, which itself requires Java. System administrators can install Scala and Java using their operating system's software management system (e.g., "`sudo apt install scala`" on Ubuntu based systems). Administrators and users can also do a manual installation. To get the currently supported major versions of Scala, use:

```
names(rscala::scalaVersionJARs())

## [1] "2.11" "2.12" "2.13"
```

The simplest way to satisfy these dependencies, however, is with the `scalaConfig` function:

```
rscala::scalaConfig()
```

This function tries to find Scala and Java on the user's computer and, if needed, downloads and installs Scala and Java in the user's `~/.rscala` directory. Because this is a user-level installation, administrator privileges are not required.

2.2. Instantiating a Scala bridge

Load and attach the **rscala** package in an R session with the `library` function:

```
library("rscala")
```

Create a Scala bridge using the `scala` function:

```
s <- scala()
```

The `scala` function takes several arguments to control how Scala is run, including options to add JAR files to the classpath and control the memory usage. Details on this and all other functions are provided in the R documentation for the package (e.g., `help(scala)`).

A Scala session is only valid during the R session in which it is created and cannot be saved and restored through, for example, the `save` and `load` functions. Multiple Scala bridges can be created in the same R session. Each Scala bridge runs independently with its own memory and classpath. A Scala bridge cannot be shared across multiple R processes/threads.

2.3. Evaluating Scala snippets

Snippets of Scala code can be compiled and executed within an R session using several operators. The most basic operator is the `+` operator which runs code in Scala's global namespace and always returns `NULL`. Consider, for example, computing the binomial coefficient $\binom{n}{k} = \prod_{i=1}^{k}(n-i+1)/i$. The code below uses Scala's `def` statement to define the function. The expression `1 to k` creates a range and the higher-order `map` method of the range applies the expression `(n-i+1) / i.toDouble` to each element `i` in the range. Finally, the results are multiplied together by the `product` method.

```
s + '
  def binomialCoefficient(n: Int, k: Int) = {
    ( 1 to k ).map( i => ( n - i + 1 ) / i.toDouble ).product.toInt
  }
'

## NULL
```

This definition is available in subsequent Scala expressions:

```
s + 'println("10 choose 3 is " + binomialCoefficient(10, 3) + ".")'

## 10 choose 3 is 120.
## NULL
```

Notice the side effect of printing `120` to the console. The behavior for console printing is controlled by arguments of the `scala` function. Default values are set such that console output is displayed in typical environments.

Scala snippets can also be evaluated with the `*` operator. Whereas the `+` operator evaluates in Scala's global namespace and returns `NULL`, the `*` operator evaluates in a local block and always returns the result of the last expression:

```
choose(10, 3) == s * 'binomialCoefficient(10, 3)'

## [1] TRUE
```

2.4. Scalar and copyable types

A Scala result of type `Byte`, `Int`, `Double`, `Boolean`, or `String` is passed back to R as a length-one vector of raw, integer, double, logical, or character, respectively. We refer to these as the scalar types supported by the **rscala** package. Further, Scala arrays and rectangular arrays of arrays of the scalar types are passed to R as vectors and matrices of the equivalent R types. We call copyable types those types that are scalar types, arrays of scalar types, and rectangular arrays of arrays of the scalar types. The name emphasizes the fact that these data structures are serialized and copied between Scala and R. This may be costly for large data. Table 1 shows the mapping of Scala and R types using code examples. The example below shows how the Scala and R expressions produce the same result.

```
fromScala <- s * 'Array(Array(1, 2, 3), Array(4, 5, 6))'
fromR     <- matrix(1:6, nrow = 2, byrow = TRUE)
identical(fromScala, fromR)

## [1] TRUE
```

2.5. Passing data to Scala

It was shown previously that data of copyable types is returned to R when evaluating Scala snippets using the `*` operator. Conversely, data of copyable types can be passed to Scala snippets. A Scala bridge is represented in R as a function. Arguments passed to a Scala bridge are made available to the associated Scala snippet:

```
s(name = "Hannah") * 'name.toUpperCase == name.toUpperCase.reverse'

## [1] TRUE
```

The previous example demonstrates using a single named argument, but any number of named or unnamed arguments can be used:

```
names <- c("Hannah", "David", "Reinier")
s(names, convertToUpperCase = TRUE) * '
  val x = if ( convertToUpperCase ) names.map(_.toUpperCase) else names
  x.map { y => y == y.reverse }
'

## [1]  TRUE FALSE  TRUE
```

Scalar	Vectors / Arrays	Matrices / Rectangular arrays of arrays

`as.raw(3)`
`3.toByte`
	`as.raw(c(1, 2))`
	`Array(1.toByte, 2.toByte)`
		`matrix(as.raw(c(1, 2)), nrow = 2)`
		`Array(Array(1.toByte), Array(2.toByte))`

`TRUE`
`true`
	`c(TRUE, FALSE)`
	`Array(true, false)`
		`matrix(c(TRUE, FALSE), nrow = 2)`
		`Array(Array(true), Array(false))`

`1L`
`1`
	`c(1L, 2L, 3L)`
	`Array(1, 2, 3)`
		`matrix(c(1L, 2L), nrow = 2)`
		`Array(Array(1), Array(2))`

`1.0`
`1.0`
	`c(1.0, 2.0, 3.0)`
	`Array(1.0, 2.0, 3.0)`
		`matrix(c(1.0, 2.0), nrow = 2)`
		`Array(Array(1.0), Array(2.0))`

`"a"`
`"a"`
	`c("a", "b", "c")`
	`Array("a", "b", "c")`
		`matrix(c("a", "b"), nrow = 2)`
		`Array(Array("a"), Array("b"))`

Table 1: Scala values of type `Byte`, `Int`, `Double`, `Boolean`, or `String` (labeled "scalar"), as well as arrays and rectangular arrays of arrays of these types, are copied from Scala to R as length-one vectors, vectors, and matrices of the equivalent R types. These are called copyable types. Each cell in the table contains two lines: an R expression (top) and the equivalent Scala expression (bottom).

Note that, for unnamed arguments, the identifiers (e.g., `names` in the previous example) are used as Scala variable names. Since Scala has different rules for variable names than does R, only the intersection of valid variable names in both Scala and R can be used. For example, `use.upper` and `_useUpper` would be invalid arguments to a Scala bridge, the first being an invalid identifier in Scala and the second being invalid in R.

The previous example also illustrates that vectors are typically passed to Scala as arrays (e.g., `names` in the previous example), except vectors of length one are passed not as arrays but as scalars (e.g., in the previous example, `convertToUpperCase` is a scalar). If the user wants to ensure that a vector is passed as an array, R's "as-is" function `I` is used. In the example below, the length of `x` is random but the Scala code is valid because `x` is wrapped in `I` to guarantee that it is passed as an array.

```
x <- letters[sample(length(letters), rbinom(1, size = 2, prob = 0.5))]
s(x = I(x)) * 'x.map(_.toUpperCase).mkString'

## [1] "B"
```

2.6. Scala references

If the result of a Scala expression is not a copyable type, the `*` operator returns a reference to a Scala object that can be used in subsequent evaluations. If a Scala reference is desired, even when working with copyable types, use the `^` operator.

In the next example, an instance of the class `scala.util.Random` is created and, because the result is not a copyable type, a Scala reference is returned. Second, a Scala reference to an array of integers is returned — despite the fact that this is a copyable type — because the ˆ operator is used.

```
rng <- s * 'new scala.util.Random()'
rng

## rscala reference of type scala.util.Random

oneToTenReference <- s ^ 'Array.range(1, 11)'
oneToTenReference

## rscala reference of type Array[Int]
```

Scala references can also be passed as arguments to a Scala bridge:

```
s(rng, len = 15L) * 'rng.alphanumeric.take(len).mkString'

## [1] "W46ZoCnDkKWDI73"
```

2.7. The $ operator

Accessing methods and variables of Scala objects

Taking inspiration from **rJava**'s high-level $ operator, methods associated with Scala references can be called directly using the $ operator:

```
rng$setSeed(24234L)
rng$nextInt(10L)

## [1] 4

oneToTenReference$sum()

## [1] 55
```

As with arguments to a Scala bridge, variables of copyable types and Scala references may be used as arguments when employing the $ operator. If the result of a method call on a Scala reference is not a copyable type, then a Scala reference is returned. If a Scala reference is desired even when working with copyable types, add a dot immediately after the $ operator:

```
rng$.nextInt(10L)

## rscala reference of type Int
```

The value of an instance variable may be accessed as if there was a method of the same name taking no arguments. For example, the value `self` in an instance of `scala.util.Random` is accessed as:

```
rng$self()

## rscala reference of type java.util.Random
```

In an interactive R session, the **rscala** package provides rudimentary tab-completion for method names of Scala references.

Other uses of the `$` *operator*

There are several other uses of the `$` operator. In the next example, the following are generated with the `$` operator: an instance of the class `scala.util.Random`, an instance of a mutable hash map, and a null reference of type `String`.

```
seed <- 123L
rng <- s$.new_java.util.Random(seed)
map <- s$".new_scala.collection.mutable.HashMap[String, Double]"()
nullString <- s$.null_String()
```

Note the use of quotes for the hash map in the previous example. Scala has type parameterization which is similar to (but arguably more advanced than) generics in Java and templates in C++. In many instances, the Scala compiler infers the type parameter, but the user may need or want to explicitly provide it. When using the `$` operator, quoting may be needed since the type involves characters that are not allowed in R identifiers (e.g., `[` and `]`). Likewise, names of Scala methods may not be valid identifiers in R and may also need to be quoted to avoid parsing errors in R. For example, note that the `List`'s append method `:+` is quoted here:

```
myList <- s$List(1L, 2L, 3L)
augmentedList <- myList$':+'(100L)
paste0(augmentedList$toString(), " now contains 100.")

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

The next example shows usage of the `$` operator to access a previously defined function (e.g., `binomialCoefficient`), a method of a companion object (e.g., `Array`'s `range` method), a factory method of a companion object (e.g., `List`'s implied `apply` method), and a method of a singleton object (e.g., `scala.util.Properties`'s `versionNumberString` method).

```
s$binomialCoefficient(10L, 3L) == choose(10, 3)

## [1] TRUE
```

```
oneToTenReference <- s$.Array.range(1L, 11L)
myScalaList <- s$List(1, 2, 3, 4)
s$scala.util.Properties.versionNumberString()

## [1] "2.13.1"
```

2.8. Interfacing with Java

Scala runs on the JVM and since it supports instantiating Java classes and calling object and static methods, the **rscala** package automatically provides this support as well. For example, we can find the system's time zone through a chain of calls using the standard Java library:

```
s$java.util.TimeZone.getDefault()$getDisplayName()

## [1] "Mountain Standard Time"
```

2.9. Callbacks to R from Scala

When the `scala` function creates a Scala bridge, an instance of `org.ddahl.rscala.RClient` is bound to the identifier `R` within Scala. It is through this instance that callbacks to the R interpreter are possible. The `RClient` class is thread-safe. Its source code and Scaladoc are located on GitHub: https://github.com/dbdahl/rscala/.

All of the evaluation methods of this class take the same arguments. The first argument is a template for an R expression, where `%-` is a placeholder for items that are provided as variable arguments. The result type is indicated by the suffix of the method name `evalXY`, where $X \in \{$`R`, `I`, `D`, `L`, `S`$\}$ and $Y \in \{$`0`, `1`, `2`$\}$. The value of X indicates whether the result from R should be interpreted as raw, integer, double, logical, or character, respectively. The value of Y indicates whether the result should be interpreted as a scalar, an array, or a rectangular array of arrays, respectively. The method `evalObject` returns a Scala reference to an arbitrary R object which can be passed as an argument to another evaluation method. Several examples are below.

```
s * '
  R.eval("primes <- %-", Array(2, 3, 5, 7, 11, 13, 17, 19, 23, 29))
  val rFunction = R.evalObject("function(x) x * primes")
  val primesTimesTwo = R.evalI1("%-(2)", rFunction)
  R.evalI2("matrix(%-, nrow = %-)", primesTimesTwo, 2)
'

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    4   10   22   34   46
## [2,]    6   14   26   38   58

exists("primes")

## [1] TRUE
```

A more interesting use case is calling a user-supplied R function from Scala. First, consider an R function that computes $f(n, \alpha)$, the expectation of the Ewens(n, α) distribution, i.e., the expected number of clusters when sampling n observations from a discrete random measure obtained from a Dirichlet process with mass parameter α.

```
f <- function(n, alpha) sapply(alpha, function(a) sum(a / (1:n + a - 1)))
f(100, 1.0)

## [1] 5.187378
```

In a Bayesian analysis, the Ewens distribution is a prior distribution in random partition models and α is a hyperparameter. In the prior elicitation process, practitioners may want to find the value of α that corresponds to the expert's anticipated number of clusters. Thus, the task is to numerically solve $f(n, \alpha) = \mu$ for α, given fixed values for n and μ. To be specific, suppose $n = 1000$ and $\mu = 10$. The value α can be obtained using root finding methods. Here, we demonstrate the bisection method implemented in Scala. Note that the function's first argument, `func`, is a user-defined R function.

```
bisection <- function(func, lower = 1.0, upper = 1.0, epsilon = 0.0000001) {
  s(lower, upper, epsilon) * '
    def g(x: Double) = R.evalD0("func(%-)", x)
    val (fLower, fUpper) = (g(lower), g(upper))
    if ( fLower * fUpper > 0 ) sys.error("Root is not straddled.")
    type D = Double
    @scala.annotation.tailrec
    def engine(l: D, u: D, fLower: D, fUpper: D): Double = {
      if ( math.abs( l - u ) <= epsilon ) ( l + u ) / 2
      else {
        val c = ( l + u ) / 2
        val fCenter = g(c)
        if ( fLower * fCenter < 0 ) engine(l, c, fLower, fCenter)
        else engine(c, u, fCenter, fUpper)
      }
    }
    engine(lower, upper, fLower, fUpper)
  '
}
bisection(function(a) f(1000, a) - 10, 0.1, 20)

## [1] 1.443818
```

The most important aspect of the previous example is in the first line of the Scala snippet, where the `evalD0` method calls the `R` function `func` and returns the result as a `Double`.

The **rscala** package supports infinite recursion (subject to available resources) between R and Scala. For example, the `recursive.sum` function below repeatedly calls itself from Scala to compute $0 + 1 + 2 + \ldots + n$:

```
recursive.sum <- function(n) s(n) * '
  if ( n <= 0 ) 0L else n + R.evalI0("recursive.sum(%-)", n - 1)
'
recursive.sum(10)

## [1] 55
```

2.10. Speed considerations

Section 4 considers the speed and ease of implementating a simulation study in R, C++ via **Rcpp**, and Scala via **rscala**. It is not a comprehensive comparison of the performance of these languages. For that, we refer readers to benchmarks available on the web. Here we simply highlight performance characteristics of **rscala** itself.

All calls into Scala require compilation before invocation. Subsequent uses of the same code skip the time-consuming compilation due to caching. Consider, for example, two calls to the method `nextGaussian` of an instance of `java.util.Random`:

```
rng_rscala <- s$.new_java.util.Random()
first  <- system.time( rng_rscala$nextGaussian() )['elapsed']
second <- system.time( rng_rscala$nextGaussian() )['elapsed']
c(first = first, second = second, ratio = first / second)

##  first.elapsed second.elapsed  ratio.elapsed
##          0.146          0.001        146.000
```

By way of comparison, **rJava** provides two means to call the `nextGaussian` method. Suppose that `rngRJava` is the result of instantiating an object of class `scala.util.Random` using **rJava**. The high-level `$` operator of **rJava** can call this method using `rngRJava$nextGaussian()`. Alternatively, the **rJava**'s low-level interface provides the `.jcall` function. The next example and Table 2 compare the speed of **rscala**'s `rng$nextGaussian()` and **rJava**'s two ways of calling the same method.

```
library("rJava")
rJava::.jinit()
rng_rJava          <- .jnew("java.util.Random")
rng_rJava_LowLevel <- function() .jcall(rng_rJava, "D", "nextGaussian")
microbenchmark::microbenchmark(times = 1000, rng_rJava_LowLevel(),
  rng_rscala$nextGaussian(), rng_rJava$nextGaussian())
```

The results in Table 2 indicate that **rJava**'s low-level `.jcall` interface is much faster than the other techniques, but **rscala**'s implementation of the `$` operator is itself much faster than that of **rJava**. We recommend that R users avoid calling short-lived Scala code in tight inner loops where microsecond delays can add up.

2.11. Developing packages based on rscala

The **rscala** package enables developers to use Scala in their own R packages to implement

Expression	Package	Q1	Mean	Median	Q3
`rng_rJava_LowLevel()`	**rJava**	22.37	32.42	24.23	34.68
`rng_rscala$nextGaussian()`	**rscala**	163.25	204.98	187.06	217.12
`rng_rJava$nextGaussian()`	**rJava**	789.27	880.52	814.63	853.13

Table 2: Comparison of execution time of various ways to call the `nextGaussian` method of an instance of the `java.util.Random` class. Since the method itself is relatively fast, the timings here are an indication of the overhead involved with the various techniques. Each expression was evaluated 1000 times and the results are in microseconds.

computationally intensive algorithms. For example, the **shallot** (Dahl 2018b) and **bamboo** (Dahl 2018a) packages on CRAN use Scala via **rscala** to implement statistical methodology of their associated journal articles (Dahl *et al.* 2017; Li *et al.* 2014). Readers are encouraged to study those examples in addition to our description here.

An R package based on **rscala** should include `rscala` in the `Imports` field of the package's `DESCRIPTION` file. Also, add `import(rscala)` to the `NAMESPACE` file. Typically a package based on **rscala** will instantiate a Scala bridge in the package's `.onLoad` function. To make the bridge available to the other functions in the package, the author should assign the bridge to the package environment. The `.onLoad` function may be as simple as:

```
.onLoad <- function(libname, pkgname) {
  assign("s", scala(), envir = parent.env(environment()))
}
```

If the package is to access precompiled code from a JAR file, we suggest cross compiling against the major versions supplied by:

```
names(rscala::scalaVersionJARs())

## [1] "2.11" "2.12" "2.13"
```

This is done in part by adding a line to SBT's `build.sbt` file, like:

```
crossScalaVersions := Seq("2.11.12", "2.12.8", "2.13.0")
```

The JAR files should be copied to directories `inst/java/scala-X.XX` relative to the package root, where `X.XX` represent a major version of Scala (e.g., `2.13`). The cross compiling and copying of JAR files is automated by the `rscala::scalaSBT` function. If JAR files of compiled Java code are to be included in the package, they should be placed directly in the `inst/java` directory of the source package. As another aid, the `rscala::scalaDevelDownloadJARs` function is meant to be called from bare code of a package that depends on **rscala** in a script such as `zzz.R`. When called during package installation, it downloads JAR files to the appropriate directories and avoids the need to distribute some JAR files in the source package.

To make the JAR file available to the package's R functions, the name of the package should be passed as the first argument to the `scala` function, e.g.:

```
.onLoad <- function(libname, pkgname) {
  assign("s", scala(pkgname), envir = parent.env(environment()))
}
```

It is common in the `.onLoad` function to define global imports, classes, objects, and functions using the `+` operator. We recommend, however, that this be accomplished through the `scalaLazy` function to delay the evaluation until necessary. This gives the Scala bridge the chance to start up without blocking R's read-eval-print loop. For example, the `.onLoad` function of the **bamboo** package is:

```
.onLoad <- function(libname, pkgname) {
  s <- scala(pkgname)
  scalaLazy(function(s) s + 'import org.ddahl.bamboo._')
  assign("s", s, envir = parent.env(environment()))
}
```

Since packages should not leave external processes (in this case, Scala) running when the package is unloaded, the package should close the Scala bridge in the `.onUnload` function, e.g.:

```
.onUnload <- function(libpath) {
  close(s)
}
```

Finally, a package can piggy-back on another package by using its Scala bridge. For example, consider two fictious packages: **pkg1** and **pkg2**. The **pkg2** package can use the Scala bridge from the **pkg1** package and, assuming **pkg2** is installed when **pkg1** is loaded, the additional JAR files of **pkg2** will already available. That is, the `.onLoad` function for the **pkg2** package might be:

```
.onLoad <- function(libname, pkgname) {
  s <- pkg1:::s
  assign("s", envir = parent.env(environment()))
}
```

In this case, since the **pkg2** package is not the original owner of the Scala bridge, the **pkg2** package should not call `close(s)` in an `.onUnload` function.

3. Accessing R in Scala

So far we have demonstrated accessing Scala from R. Conversely, **rscala** can also embed an R interpreter in a Scala application via the `org.ddahl.rscala.RClient` class. In this case, however, there is not an existing instance of the R interpreter. The R client spawns an R instance, immediately starts the embedded R server, and connects R to Scala.

The `RClient` class is thread-safe. Source code and Scaladoc are located on GitHub: `https://github.com/dbdahl/rscala/`. As a convenience, **rscala**'s JAR file is available in standard repositories for use by dependency management systems. To use `RClient` in a Scala application, simply add the following line to SBT's `build.sbt` file:

```
libraryDependencies += "org.ddahl" %% "rscala" % "(VERSION)"
```

where `(VERSION)` is replaced with the current package version. Note that, since the necessary R code is bundled in the JAR file, the **rscala** package does not need to be installed in R. An embedded R interpreter is instantiated as follows:

```
scala> val R = org.ddahl.rscala.RClient()
```

This assumes that the registry keys option was not disabled during the R installation on Windows. On other operating systems, R is assumed to be in the search path. If these assumptions are not met or a particular installation of R is desired, the path to the R executable may be specified explicitly (e.g., `org.ddahl.rscala.RClient("/path/to/R_HOME/bin/R")`). Console output from R is not automatically serialized back to Scala.

The **rscala** package can be an easy and convenient way to access statistical functions, facilitate calculations, manage data, and produce plots in a Scala application. Consider, for example, wrapping R's `qnorm` function to define a method in Scala by the same name:

```
scala> val R = org.ddahl.rscala.RClient()
+  type D = Double
+  def qnorm(x: D, mean: D = 0, sd: D = 1, lowerTail: Boolean = true) = {
+    R.evalD0("qnorm(%-, %-, %-, lower.tail = %-)", x, mean, sd, lowerTail)
+  }
+  val alpha = 0.05
+  println(s"Pr( Z >= ${qnorm(alpha, lowerTail = false)} ) = $alpha.")

Pr( Z >= 1.6448536269514726 ) = 0.05.
```

The next example uses R's dataset `eurodist` to compute the European city that is closest, on average, to all other European cities. While this statistical calculation is easily implemented in R, one can imagine a Scala application that needs to perform a more taxing calculation that leverages R's rich data-processing functions.

```
scala> val R = org.ddahl.rscala.RClient()
+  val distances = R.evalD2("as.matrix(eurodist)")
+  val cities = R.evalS1("attr(eurodist, 'Labels')")
+  val centralCity = distances.map(_.sum).zip(cities).minBy(_._1)._2
+  println(s"Europe's central city is $centralCity.")

Europe's central city is Lyons.
```

Spark, a cluster-computing framework for massive datasets, is another example of a Scala application that might benefit from access to R. Spark provides an application programming interface to Scala, Java, R, and Python. R users who are not already familiar with Scala would be best served by accessing Spark from R using a dedicated package such as **sparklyr** or **sparkr**. Scala developers, however, might prefer to program directly with Spark's machine learning library (MLlib) in Scala and to supplement its functionality with R through **rscala**. Recall that every `RClient` has its own workspace, so several instances can be used to overcome the single-threaded nature of R. One could, for example, use software to manage a pool of `RClient` objects on each worker node. One potential limitation is the cost of pushing large datasets over the TCP/IP bridge.

4. Case study: Simulation study accelerated with rscala

While the previously mentioned **shallot** and **bamboo** packages demonstrate the ability to develop packages based on **rscala**, we demonstrate in this section the ease with which computationally intensive statistical procedures can be implemented by embedding Scala code in an R script. The algorithm is embarrassingly parallel and we consider two means of parallelization: one using Scala's `Future` class and the other using R's **parallel** package. By way of comparison, we include a pure R implementation of the same algorithm, and also an implementation that uses inline C++ code via the **Rcpp** package. All four implementations define a function that takes an arbitrary R function for sampling.

We investigate a simulation study of the coverage probability of a bootstrap confidence interval procedure. Consider a population parameter β_1/β_2, where β_1 and β_2 are population quantiles associated with probabilities p_1 and p_2, respectively. Based on a sample of n observations, a point estimator of the parameter is the ratio of the corresponding sample quantiles, and the following bootstrap procedure can be used to find a confidence interval when the population distribution is unspecified. The sample estimate is recorded for each of `nSamples` bootstrap samples. A bootstrap confidence interval is given by (l, u), where l and u are quantiles of the bootstrap sampling distribution associated with $\alpha/2$ and $1-\alpha/2$, respectively. Although the nominal coverage is $1-\alpha$, interest lies in computing the actual coverage probability of this bootstrap confidence interval procedure using a Monte Carlo simulation study. `nIntervals` samples from the population are obtained from a user-supplied sampling function. Although the code is general, we sample $n = 100$ observations from the standard normal distribution and set $p_1 = 0.75$ and $p_2 = 0.35$, making $\beta_1/\beta_2 \approx -1.75$. We use `nIntervals` $=$ 10,000 Monte Carlo replicates, each having `nSamples` $=$ 10,000 bootstrap samples.

The four implementations are listed in Appendix A. (The code is also available in the package: `system.file("doc/bootstrap-coverage.R",package="rscala")`). The R implementation is the shortest and the **rscala** implementations are somewhat more concise than that of **Rcpp**. The **Rcpp** implementation is written in a C style. All but one implementation use the **parallel** package to harness all available cores; the first **rscala** implementation uses Scala's `Future` class for parallelism and, when sampling the data, a single instance of `RClient` is used by multiple JVM threads to call back to R. On machines with many cores, having each thread wait its turn to access the one R instance will likely slow down the execution. In the second **rscala** implementation, each CPU core has a separate R instance with a corresponding `RClient`.

We tested on machines running Ubuntu 16.04 with 4 and 56 cores, Mac High Sierra with 8

Machine	Implementation	Min.	Q_1	Mean	Median	Q_3	Max.
Ubuntu 4 cores	Pure R						
	Rcpp	255.0	257.6	259.6	259.4	262.7	263.4
	rscala #1	171.1	174.6	174.8	175.1	175.6	177.0
	rscala #2	187.2	191.8	192.4	193.0	193.7	196.3
Ubuntu 56 cores	Pure R	324.8	325.8	327.9	328.3	329.4	332.2
	Rcpp	16.0	16.0	16.5	16.3	17.1	17.6
	rscala #1	17.2	17.3	18.0	17.9	18.7	19.1
	rscala #2	13.6	13.7	15.0	13.8	14.0	26.0
Mac 8 cores	Pure R						
	Rcpp	133.2	134.2	135.9	136.3	136.5	142.1
	rscala #1	82.7	82.8	84.5	83.8	84.9	92.3
	rscala #2	87.0	88.6	90.2	89.8	90.8	98.2
Windows 8 cores	Pure R						
	Rcpp	116.4	116.6	117.0	116.7	116.9	119.0
	rscala #1	58.0	58.0	58.5	58.1	58.6	60.4
	rscala #2	65.0	65.2	67.6	66.3	69.7	74.5

Table 3: Elapsed time (in seconds) for the four implementations of the bootstrap simulation study, executed 10 times on four different machines. The **rscala** implementations had the fastest execution times.

cores, and Windows 10 with 8 cores. R was installed from CRAN binaries for all machines except the 4-core Ubuntu machine, where R was compiled from source. All machines used R 3.5.1, Scala 2.12, Java 8, **Rcpp** 0.12.19, and a pre-release version of **rscala** 3.2.1.

Elapsed times (in seconds) for 10 replications of the simulation study are found in Table 3. For the sake of time, the pure R implementation was only run on the 56-core Ubuntu machine. The pure R implementation ran about 23 times slower than the fastest implementation. The **Rcpp** implementation and the two **rscala** implementations were similar in terms of speed on the 56-core Ubuntu machine. The second **rscala** implementation (which uses the **parallel** package) was the fastest overall on the 56-core machine, and the first **rscala** implementation shows a performance penalty from sharing a single instance of `RClient` when many cores are present. On the machines with fewer cores, the first **rscala** implementation was the fastest and both **rscala** implementations were somewhat faster than the **Rcpp** implementation.

5. Conclusion

This paper introduced the **rscala** software to bridge R and Scala, which allows a user to leverage their skills in both languages and to exploit strengths in each language. For example, R users can implement computationally intensive algorithms in Scala, write R packages based on Scala, and access Scala libraries from R. Scala programmers can take advantage of R's tools for data analysis and graphics from within a Scala application.

We are exploring possible improvements for our software. First, we are exploring a mechanism to allow the R user to interrupt Scala computations without destroying the TCP/IP bridge. Second, we are exploring support for transcompiling a subset of R syntax into Scala code to avoid the overhead of callbacks from Scala to R. Experimental support has already been implemented. For example, `s ^ function(x = stD1) sd(x) / mean(x)` returns a Scala reference of type `Array[Double] => Double` which computes the coefficient of variation without calling back to R.

Acknowledgements

This paper was supported by NIH NIGMS R01 GM104972. The author thanks the CRAN maintainers and the following students: Floid Gilbert, Deepthi Uppalapati, Brandon Carter, Spencer Newcomb, Scott Ferguson, Richard Payne, and Devin Johnson.

References

Bugnion P (2016). *Scala for Data Science.* Packt Publishing - ebooks Account. ISBN 9781785281372.

R Core Team (2018). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. URL `https://www.R-project.org/`.

Dahl DB (2018a). ***bamboo**: Protein Secondary Structure Prediction Using the Bamboo Method.* R package version 0.9.23, URL `https://CRAN.R-project.org/package=bamboo`.

Dahl DB (2018b). ***shallot**: Random Partition Distribution Indexed by Pairwise Information.* R package version 0.4.4, URL `https://CRAN.R-project.org/package=shallot`.

Dahl DB (2018c). ***rscala**: Bridge Between R and Scala with Callbacks.* R package version 3.2.1, URL `https://CRAN.R-project.org/package=rscala`.

Dahl DB, Day R, Tsai JW (2017). "Random Partition Distribution Indexed by Pairwise Information." *Journal of the American Statistical Association*, **112**(518), 721–732. `doi:10.1080/01621459.2016.1165103`.

Eddelbuettel D, François R (2011). "**Rcpp**: Seamless R and C++ Integration." *Journal of Statistical Software*, **40**(8), 1–18. `doi:10.18637/jss.v040.i08`.

Jancauskas V (2016). *Scientific Computing with Scala.* Packt Publishing - ebooks Account. ISBN 1785886940.

Karim MR, Alla S (2017). *Scala and Spark for Big Data Analytics: Explore the Concepts of Functional Programming, Data Streaming, and Machine Learning.* Packt Publishing. ISBN 1785280848.

Li Q, Dahl DB, Vannucci M, Joo H, Tsai JW (2014). "Bayesian Model of Protein Primary Sequence for Secondary Structure Prediction." *PLOS ONE*, **9**(10), 1–12. `doi:10.1371/journal.pone.0109832`.

Nicolas PR (2014). *Scala for Machine Learning*. Packt Publishing - ebooks Account. ISBN 1783558741.

Odersky M, Spoon L, Venners B (2016). *Programming in Scala: Updated for Scala 2.12*. 3rd edition. Artima Press. ISBN 0981531687.

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

Pfeffer A (2016). *Practical Probabilistic Programming*. Manning Publications. ISBN 1617292338.

Satman MH (2014). "**RCaller**: A Software Library for Calling R from Java." *British Journal of Mathematics & Computer Science*, **4**(15), 2188–2196.

Urbanek S (2013). ***Rserve**: Binary R Server*. R package version 1.7-3, URL https://CRAN.R-project.org/package=Rserve.

Urbanek S (2018). ***rJava**: Low-Level R to Java Interface*. R package version 0.9-10, URL https://CRAN.R-project.org/package=rJava.

Zaharia M, Xin RS, Wendell P, Das T, Armbrust M, Dave A, Meng X, Rosen J, Venkataraman S, Franklin MJ, Ghodsi A, Gonzalez J, Shenker S, Stoica I (2016). "Apache Spark: A Unified Engine for Big Data Processing." *Commun. ACM*, **59**(11), 56–65. ISSN 0001-0782. doi:10.1145/2934664.

A. Code for case study in Section 4

Below is the code that was used in the simulation study for the bootstrap coverage in Section 4.

```
makeConfidenceInterval <- function(p, n) {
  me <- qnorm(0.975) * sqrt( p * ( 1 - p ) / n )
  c(estimate = p, lower = p - me, upper = p + me)
}

p1    <- 0.75
p2    <- 0.35
truth <- qnorm(p1) / qnorm(p2)
n     <- 100
alpha <- 0.05



cat("######## rscala implementation #1")

library(rscala)
s <- scala()

coverage.rscala1 <- function(f, n, truth, p1, p2, nSamples, a, nIntervals) {
  coverage <- s(n = as.integer(n[1]), truth = as.double(truth[1]),
                p1 = as.double(p1[1]), p2 = as.double(p2[1]),
                nSamples = as.integer(nSamples[1]), a = as.double(a[1]),
                nIntervals = as.integer(nIntervals[1])) * '
    import scala.util.Random
    import scala.concurrent.{Await, Future, duration}
    import scala.concurrent.ExecutionContext.Implicits.global

    def quantile(sorted: Array[Double], p: Double) = {
      val i = ((sorted.length - 1) * p).asInstanceOf[Int]
      val delta = (sorted.length-1) * p - i
      ( 1 - delta ) * sorted(i) + delta * sorted( i + 1 )
    }

    def statistic(x: Array[Double]) = {
      scala.util.Sorting.quickSort(x)
      quantile(x, p1) / quantile(x, p2)
    }

    def resample(x: Array[Double], rng: Random) = Array.fill(x.length) {
      x(rng.nextInt(x.length))
    }

    def ciContains(x: Array[Double], rng: Random) = {
      val bs = Array.fill(nSamples) { statistic(resample(x, rng)) }
      scala.util.Sorting.quickSort(bs)
      quantile(bs, a / 2) <= truth  &&  truth <= quantile(bs, 1 - a / 2)
    }

    Await.result( Future.sequence( List.fill(nIntervals) {
      val dataset = R.evalD1("f(%-)", n)
      val seed = R.evalI0("sample(c(-1, 1), 1) * sample.int(2 ^ 31 - 1, 1)")
      val r = new Random(seed)
      Future { ciContains(dataset, r) }
    }), duration.Duration.Inf).count(identity) / nIntervals.toDouble
  '
  makeConfidenceInterval(coverage, nIntervals)
}



cat("######## All the remaining implementation use the parallel package.")

library(parallel)
cluster <- makeCluster(detectCores())
```

```
cat("######## rscala implementation #2")

clusterEvalQ(cluster, {
  library(rscala)
  s <- scala()
  ciContains.rscala2 <- function(f, n, truth, p1, p2, nSamples, a) {
    s(n = as.integer(n[1]), truth = as.double(truth[1]),
      p1 = as.double(p1[1]), p2 = as.double(p2[1]),
      nSamples = as.integer(nSamples[1]), a = as.double(a[1])) * '
      import scala.util.Random

      def quantile(sorted: Array[Double], p: Double) = {
        val i = (( sorted.length - 1 ) * p).asInstanceOf[Int]
        val delta = ( sorted.length - 1 ) * p - i
        ( 1 - delta ) * sorted(i) + delta * sorted( i + 1 )
      }

      def statistic(x: Array[Double]) = {
        scala.util.Sorting.quickSort(x)
        quantile(x, p1) / quantile(x, p2)
      }

      def resample(x: Array[Double], rng: Random) = Array.fill(x.length) {
        x(rng.nextInt(x.length))
      }

      val x = R.evalD1("f(%-)", n)
      val seed = R.evalI0("sample(c(-1, 1), 1) * sample.int(2 ^ 31 - 1, 1)")
      val r = new Random(seed)
      val bs = Array.fill(nSamples) { statistic(resample(x, r)) }
      scala.util.Sorting.quickSort(bs)
      quantile(bs, a / 2) <= truth  &&  truth <= quantile(bs, 1 - a / 2)
    '
  }
})

coverage.rscala2 <- function(f, n, truth, p1, p2, nSamples, a, nIntervals) {
  clusterExport(cluster, c("f", "n", "truth", "p1", "p2", "nSamples", "a"),
    envir = environment())
  coverage <- mean(parSapply(cluster, 1:nIntervals, function(i) {
    ciContains.rscala2(f, n, truth, p1, p2, nSamples, a)
  }))
  makeConfidenceInterval(coverage, nIntervals)
}
```

```
cat("######## Pure R implementation")

coverage.pureR <- function(f, n, truth, p1, p2, nSamples, a, nIntervals) {
  statistic <- function(x) {
    q <- quantile(x, probs = c(p1, p2))
    q[1] / q[2]
  }
  ciContains.pureR <- function(x) {
    samples <- numeric(nSamples)
    for ( i in seq_along(samples) ) {
      samples[i] <- statistic(sample(x, replace = TRUE))
    }
    ci <- quantile(samples, probs = c(a / 2, 1 - a / 2))
    ( ci[1] <= truth ) && ( truth <= ci[2] )
  }
  clusterExport(cluster, c("f", "n", "truth", "p1", "p2", "nSamples", "a"),
    envir = environment())
  coverage <- mean(parSapply(cluster, 1:nIntervals, function(i) {
    ciContains.pureR(f(n))
  }))
  makeConfidenceInterval(coverage, nIntervals)
}



cat("######## Rcpp implementation")

clusterEvalQ(cluster, {
  library(Rcpp)
  sourceCpp(code = "
    #include <Rcpp.h>
    using namespace Rcpp;

    double quantile(double *sorted, int length, double p) {
      int i = (int) (( length - 1 ) * p);
      double delta = ( length - 1 ) * p - i;
      return ( 1 - delta ) * sorted[i] + delta * sorted[ i + 1 ];
    }

    int compare_double(const void* a, const void* b) {
      double aa = *(double*) a;
      double bb = *(double*) b;
      if ( aa == bb ) return 0;
      return aa < bb ? -1 : 1;
    }
    double statistic(double *x, int length, double p1, double p2) {
      qsort(x, length, sizeof(double), compare_double);
      return quantile(x, length, p1) / quantile(x, length, p2);
    }

    double *resample(double *x, int length) {
      double *y = (double*) malloc( length * sizeof(double) );
      for ( int i = 0;  i < length; i++ ) {
        y[i] = x[ (int) (Rf_runif(0, 1) * length) ];
      }
      return y;
    }

    // [[Rcpp::export]]
    bool ciContains(NumericVector data, double truth,
                    double p1, double p2, int nSamples, double a) {
      double *y = (double*) malloc( nSamples * sizeof(double) );
      for ( int i = 0; i < nSamples; i++ ) {
        int length = data.size();
        double *z = resample(data.begin(), length);
        y[i] = statistic(z, length, p1, p2);
        free(z);
      }
      qsort(y, nSamples, sizeof(double), compare_double);
      bool result =  ( quantile(y, nSamples,     a / 2) <= truth ) &&
                     ( quantile(y, nSamples, 1 - a / 2) >= truth );
      free(y);
      return result;
    }
  ")
})

coverage.Rcpp <- function(f, n, truth, p1, p2, nSamples, a, nIntervals) {
  clusterExport(cluster, c("f", "n", "truth", "p1", "p2", "nSamples", "a"),
    envir = environment())
  coverage <- mean(parSapply(cluster, 1:nIntervals, function(i) {
    ciContains(f(n), truth, p1, p2, nSamples, a)
  }))
  makeConfidenceInterval(coverage, nIntervals)
}



cat("######## Benchmarks")

system2("hostname")
sessionInfo()
```

```
library(microbenchmark)
engine <- function(nSamples, nIntervals, times) microbenchmark(
  pureR   = coverage.pureR(
              rnorm, n, truth, p1, p2, nSamples, alpha, nIntervals),
  Rcpp    = coverage.Rcpp(
              rnorm, n, truth, p1, p2, nSamples, alpha, nIntervals),
  rscala1 = coverage.rscala1(
              rnorm, n, truth, p1, p2, nSamples, alpha, nIntervals),
  rscala2 = coverage.rscala2(
              rnorm, n, truth, p1, p2, nSamples, alpha, nIntervals),
  times = times)

engine(nSamples = 10000L, nIntervals = 10000L, times = 10)
```

Affiliation:

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