From R to Java: the TypeInfo and RWebServices paradigm

Nianhuaf Li*, Martin T. Morgan, Seth Falcon,
Robert Gentleman, Duncan Temple Lang†

14 July, 2006

Abstract

Web services are most effective on statically typed objects exposed in a well-developed infrastructure. This document summarizes our approach to exposing R objects and functionality in a Java class hierarchy of statically typed methods. The approach is to use R's formal (S4) class system to strongly type R functions using TypeInfo. We then convert strongly typed functions to Java objects and methods for exposure as Java-based web services.

Exposing and implementing the web service in Java involves the package SJava. Documentation for these steps will be provided later.

1 Introduction

Exposing R objects and functions as web services poses several challenges. First, R has both informal 'classes' and a formal (S4) class system, whereas web services are most effective with well-defined objects. Second R functions are not strongly typed, whereas web services deploy statically typed functions. Finally, well-developed infrastructure supports Java-based web services, whereas web services client and server functionality for R requires substantial *de novo* development. TypeInfo and RWebServices are packages that combine to provide a paradigm for exposing R functions as effective web services in a Java-based web services context.

Here we document the paradigm of using TypeInfo and RWebServices for type mapping between R and Java.

*Fred Hutchinson Cancer Research Center, 1100 Fairview Ave. N., PO Box 19024 Seattle, WA 98109

†Department of Statistics, 4210 Mathematical Sciences Building, One Shield Avenue, Davis, CA 95616

2 Steps to describing R objects in Java

2.1 Adding TypeInfo to R functions

The main purpose of TypeInfo is to provide type specification for function arguments and return values. By 'type specification' we mean definition of argument and return types in terms of defined R objects. The named objects are defined in R, and objects and function definitions are translated to equivalent Java objects and methods using RWebServices.

To illustrate, the following defines and invokes a hypothetical R function square taking an un-typed argument x and returning an untyped return value.

```
> square <- function(x) {
+     return(x^2)
+ }
> square(10)

[1] 100
```

The function evaluates correctly when provide a numeric argument; non-numeric arguments result in a run-time error. Importantly, there is no way to query the function to determine its argument or return type.

Type specification is applied by loading the TypeInfo package and annotating the definition of square:

```
> library(TypeInfo)
> STS <- SimultaneousTypeSpecification
> TS <- TypedSignature
> typeInfo(square) <- STS(TS(x = "numeric"),
+     returnType = "numeric")
```

(The symbols STS and TS are defined for convenience to be synonyms for the longer function names from the TypeInfo library).

Applying TypeInfo provides two important changes to the behavior of square, without altering the body of the function. First, the argument x and return type *must* be objects of type numeric (approximately, double[] in Java). Attempts to invoke square with non-numeric arguments result in an error. Programming errors returning non-numeric values also cause an error.

The second important consequence of applying TypeInfo is to allow functions annotated in this way to be queried for their argument and return types:

```
> typeInfo(square)

[SimultaneousTypeSpecification]
  [TypedSignature]
    x: is(x, c('numeric'))  [InheritsTypeTest]
  returnType: is(returnType, c('numeric'))  [InheritsTypeTest]
```

This information can be readily extracted and transformed programmatically.

R functionality is usually organized into *packages*. The intention is that package authors, or individuals responsible for exposing R functionality as web services, apply TypeInfo to functions in the package. Thus type-specified functions are defined within packages.

Full documentation of TypeInfo is available with the package. Entering `library(help=TypeInfo)` at the R prompt provides a synopsis of available commands. Documentation of each command is available by typing `?typeInfo` at the R prompt. Additional illustration of TypeInfo, written for a general audience, is distributed with the packages as a PDF file TypeInfoNews.

2.2 Using RWebServices to create Java mappings

The main purpose of RWebServices is to translate R object and function definitions into equivalent Java class definitions. Note that there are two components to translation. The focus here is on *describing* R objects in Java. The process of moving data from R to Java and vice-versa is implicit in this description, but the software for performing this translation (SJava) is not part of the paradigm being described here.

RWebServices operate on type-specified functions. RWebServices extracts information about argument and return types. It determines the underlying structure of potentially complicated R objects specified in the type definition. Based on this information, RWebServices produces Java class hierarchies reflecting data objects, and composes Java method signatures appropriate for the functions.

From the R perspective, the process of producing web services templates for a function, e.g., caAffy with TypeInfo applied in the package CaAffy is straightforward:

```
> library(CaAffy)
> RJavaSignature(c(caAffy))
```

RJavaSignature queries caAffy for its argument types. It then uses standard S4 object type definition specified in CaAffy (or other R packages), and function definitions in CaAffy to construct Java signatures. RJavaSignature then produces documented Java beans representing the R data objects and functions, organized in a hierarchy reflecting the package structure. Suppose caAffy takes arguments magePlaceholder and caAffyTuningParam of class *MagePlaceholder* and *CaAffyTuningParam*, and returns an object of *MagePlaceholder*. The Java beans and methods are packaged as described below.

Full documentation of RWebServices is available with the library. Entering `library(help=RWebServices)` at the R prompt provides a synopsis of available commands. Documentation of each command is available by typing `?RJavaSignature` at the R prompt. Although the RWebServices package depends on SJava for performing web services, the functionality described here does not use the facilities of SJava.

3 Understanding Java representations of R objects and functions

RWebServices has two main functionalities. First, RWebServices generates Java representations of R functions and data objects. Second, RWebServices allows R functions to be evaluated from within Java, including Java-based web or analytic services. This section describes in detail the functioning of RWebServices as it generates Java representations.

A central purpose of RWebServices is to generate Java representations of R data and functions. The main interface to RWebServices is provided through the R function RJavaSignature. Starting with a list (provided by the user or programmatically extracted from the package) of TypeInfo-annotated functions, RWebServices parses the functions for data types, and creates Java representations of each data type and method. The Java representation of methods and parsed data types are then collated into Java packages with a layout consistent with the R package structure. RWebServices also generates Java service APIs and adapters for the R functions. Internally, the function RWebServices:::generateFunctionMap is responsible for these steps.

The Java data and method representations are written to disk as a file hierarchy reflecting the structure of the corresponding R objects, including the libraries in which the R data types and methods were defined. Details are provided below, but a simple example is:

```
package / CaAffy / data (Java data objects)
                 / functions (Java methods for R functions)
        / CaPROcess / data
                    / functions
        / CaDNAcopy / data
                    / functions
service / bioconductor (Java service API)
```

The R packages in this example include CaAffy, CaPROcess, and CaDNAcopy.

3.1 Java representations of R data objects

The responsibility for generating Java representations of R data objects is in the internally defined function RWebServices:::generateDataMap. This function operates by creating a hash of R data types used in the R functions. The function then creates Java class definitions representing the R data types (limitations concerning multiple inheritance are described below). The representations reflect underlying R data type structure, for instance, capturing slots present in S4 classes. Part of this process is to identify functions required for low-level data conversion (e.g., R numeric to Java RDouble); details of the low-level conversion process are presented below. R class names are mangled to reflect Java conventions (e.g., R class.name becomes Java className) and to avoid Java keyword conflicts.

The Java representations are written to disk in a folder data contained inside the corresponding package folder, e.g., biocJavaMap/CaAffy/data.

3.2 Generating Java representations of R function signatures

RWebServices:::generateDataMap uses the R function signature to generate Java class methods. Methods are constructed by looking up input and output R data types with their corresponding Java representation. Argument input names are mangled to be consistent with Java convention. Javamethod names correspond to R function names, except when several R functions have the same name but different return types. In this case simple aliases (e.g., foo_1, foo_2) are created in the Java representation.

The Java representations are written to disk in a folder function containing a single class with methods corresponding to all R functions defined in the R package.

3.3 Generating the Java API and adapters

RWebServices creates an API that represents the main entry to invoke R functionality from Java. In its simplest form, the API consists of a single Java class (e.g., service.bioconductor.java) with a method for each R function. Each method in the API invokes the corresponding method in the individual Java packages. For example, the affy method in the main service API might invoke biocJavaMap.CaAffy.function.caAffy(). Multiple web services can also be defined, with each service API dispatching to one or several Java packages encapsulating R methods.

RWebServices also creates a naive client interface to be used during testing, and an adapter to implement the web service interface generated by Axis or other web service facilities.

The Java API, client, and adapters are written to disk in the folder service / bioconductor (or as defined by the user).

4 Understanding how Java invokes R functions

Invoking R functions from Java relies on the SJava package. There are two main tasks. The first is conversion of data types between Java and R. The second is to evaluate the R functions, using an R session embedded in the Java virtual machine.

4.1 Data types and conversions

SJava allows C code to interface between native Java types (accessible through JNI) and native R types (R native types are C data structures that define S-expressions, or SEXPs). Each data type conversion is performed by converter

functions, written in C or R. Converters for basic data types are provided by SJava. Additional converters can extend or override the basic converters, and can be registered with SJava for dynamic dispatch.

4.1.1 Data models

RWebServices uses the flexible infrastructure of SJava to convert basic R types to Java primary types (integer, double or classes (e.g., Integer[], Double[], etc.), and to convert the structured S4 R objects to corresponding Java classes. This basic mapping provides sufficient flexibility for data transfer between languages, while promoting interoperability through reuse of common data types. RWebServices also supports a richer object model, capturing the use of R *attributes* to convey object information, e.g., about dimensions or missing values. This richer model is not exposed in caBig.

The Java representation of complex R objects (e.g., S4 objects) are programmatically generated using R language reflection to identify object structure (R slots) in terms of basic R types. Limitations to this approach are indicated below. Additional R class structures can also be represented in Java. For instance, class unions are an R concept where members of the class union form a single class, even though they are otherwise unrelated.

```
> setClass("A", "logical")

[1] "A"

> setClass("B", "character")

[1] "B"

> setClassUnion("C", c("A", "B"))

[1] "C"
```

An instance of class C can be assigned either logical or character values. This pattern of inheritance cannot be represented as a single Java object, but RWebServices implements Java representations of class unions using inspiration from the Abstract Factory pattern.

4.1.2 Converters

A converter handles conversion between a specific pair of R and Java objects. There are two components to RWebServices converters. A 'match' function (e.g., RWebServices:::cvtIntegerToJava) is used for dynamic dispatch. A convert function (e.g., RWebServices:::cvtIntegerToJava) in RWebServices is written in R; converters rely on calls to underlying C code (e.g., RIntegerVector_JavaIntArray) or on SJava functionality to copy data types between R SEXPs and Java native representations.

Converters for each complex R object is programmatically generated by recursively visiting the object slots (corresponding to Java fields) until basic R types are encountered. Converters are included in the data output directory, e.g., biocJavaMaps/CaAffy/data/TypeConverter.R) and loaded in the embedded R.

4.1.3 Limitations

There are several limitations to the object model and conversion process outlined here. R objects can have arbitrary attributes, but the RWebServices implementation only recognizes attributes essential for representing data structures to web or analytic services (e.g., dim to describe RArray dimensions). The main reason for restricting RWebServices in this way is that the resulting Java representation is likely not to be used often. The implementation is flexible enough that future extensions are possible.

Classes from the the informal 'S3' object system of R do not contain sufficient information about class structure for programmatic transformation between R and Java; these objects can be defined more formally as S4 objects, and the S4 objects used with TypeInfo to specify argument and return types.

S4 classes consist of slots specific to the class, and relationships to other classes; the class system is similar to but richer than that in Java, allowing multiple inheritance, class unions, etc. RWebServices captures the entire data representation of S4 objects, but does not contain information about class relations. For instance, in the following example

```
> setClass("A", representation = representation(x = "numeric"))

[1] "A"

> setClass("B", contains = "A", representation = representation(y = "numeric"))

[1] "B"
```

An R instance of class B has two slots x, y; information about the inheritance of x is contained in the class definition of B, but the structure of instances of B does not include this information. The Java representation of class B created by RWebServices has two fields x and y, but no knowledge of the class hierarchy that these slots represent in R because Java requires single inheritance. This is a satisfactory solution for present purposes, since the data contained in the Java instance is sufficient for data transformation. A development might more fully leverage single inheritance in Java to represent classes with only single inheritance in R.

RWebServices allows R objects to be represented in Java, but does not provide facilities for automatically representing Java objects as R classes. This is satisfactory for the goal of exposing R functions and data object as web or analytic services.

4.2 Function invocation

Invocation of R functions is initiated in the Java API created by RWebServices (e.g., `service / bioconductor`). This API initializes and uses SJava facilities. SJava embeds R in the Java virtual machine as a shared library. SJava mediates interactions with the embedded R through instances of the Java classes *ROmegahatInterpreter* and *REvaluator*. The Java API uses *REvaluator* to establish the environment for R function evaluation, including loading R packages required for function evaluation and installing converter functions. The Java virtual machine is now able to invoke R functions.

The interface to R functions starts at the main API. The main API invokes the package-level (e.g., CaAffy) Java representations of the R function. The package-level representation invokes `REvaluator.call()`. This method takes as arguments a character string representing the R function name and a Java *Object[]* containing Java representations of input parameter, and returns a Java *Object*. `REvaluator.call` invokes necessary data translators for data transfer to and from R, and arranges for R function evaluation of appropriate arguments. Input parameters and return types of `REvaluator.call()` are generic; type coercion takes place in the package-level Java representations.

Error handling facilities are available. Errors triggering the exception handling system in R during function evaluation or type conversion are propagated as Java exceptions, and returned to the Java virtual machine. Serious R faults (e.g.,segmentation faults) trigger Java exceptions that are also propagated.

The implementation has several limitations. Callbacks to Java from R are not yet tested. SJava implements the concept of foreign language references, where functions in one language operate on references to complex data types in the other language, rather than on the data itself. The RWebServices implementation has not yet taken advantage of this feature.

Finally, R is not thread safe, so that each Java virtual machine can have at most one instance of R. This requires that evaluation of several functions must occur sequentially. One solution is to use multiple Java processes in a coordinated fashion, e.g., using the Java Message Service.

5 Next steps: Exposing R as web and analytic services

The forgoing sections have described how R data types and functions are exposed to Java applications. There are well-established mechanisms to facilitate the transformation of stand-alone Java applications to web or analytic services. For example, Apache Axis tools generate WSDL from stand-alone applications, and web services layers from WSDL. Likewise, the caGrid tool Introduce coupled with caDSR tools for semantic annotation allow generation of analytic services from stand-alone Java applications.