Contents

1 Other sections to be added

- Omegahat and evaluator functionality.
- C routines for creating and assigning references in Java - anonymous and named.
- Adding converters
- Converters (in C) for R/S objects to Java.

2 The R-Java Interface

This is a description of what elements are used to make this system work. It is a high-level description for users who want to modify the system or simply understand how it works. It is written to remind the authors of these details.
The first thing is to compile the library *libRSNativeJava.so*. This is in **“Env –OMEGA˙HOME˝/Interfaces/Java/**. This contains routines to manage the embedding of the JVM in an arbitrary C application with many convenience routines for dealing with the JNI facilities. Also, it provides routines that are shared by both R and S implementations. This is then installed in **“Env –OMEGA˙HOME˝/lib/** and the header files used by both the R and S packages that use this library are copied to **“Env –OMEGA˙HOME˝/include/**.
The next thing to do is compile the *R.so* shared library in this package.
Ideally, both of these shared libraries will have been compiled with the -rpath or -R linker flags so that they contain the location of the files against which they link.
Using JDK1.2 on Linux (installed in **/home/duncan/jdk1.2.2/**), the dependencies are

```
[*]
 % ldd R.so
 libRSNativeJava.so => /home/duncan/Projects/org/omegahat/lib/libRSNative Java.so (0x40007
 libc.so.6 => /lib/libc.so.6 (0x40014000)
 libjvm.so => /home/duncan/jdk1.2.2/jre/lib/i386/classic/libjvm.so (0x401 07000)
 /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x80000000)
 libm.so.6 => /lib/libm.so.6 (0x40176000)
 libnsl.so.1 => /lib/libnsl.so.1 (0x40192000)
 libdl.so.2 => /lib/libdl.so.2 (0x401a8000)
 libhpi.so => /home/duncan/jdk1.2.2/jre/lib/i386/native_threads/libhpi.so  (0x401ac000)
 libpthread.so.0 => /lib/libpthread.so.0 (0x401b6000)
```

If the libraries are not found using **`ldd`**, set the environment variable `LD_LIBRARY_PATH` to include the directories in which they are contained. For example, in the case above,

```
[*]
setenv LD_LIBRARY_PATH /home/duncan/Projects/org/omegahat/lib:/home/duncan/jdk1.2.2/jre/li
```

At this point, we can test the validity of the shared libraries. Run R and load the library.

```
[*]
 % R

 dyn.load("R.so")
```

This should give no error messages. If it does, it is likely that the dynamic loading is failing because of the `LD_LIBRARY_PATH` setting. If it complains about a missing symbol, this is more serious. Most of the symbols we use in this package have a prefix `RS_JAVA()`. If it complains about any of these, the compilation is likely to have been corrupted. Finally, one can use **`nm`** to examine which symbols are undefined.

```
[*]
 nm R.so | grep ' U '
```

Next, we want to make use of some of the functionality provided by the embedded JVM. We start by initializing the JVM. The function *.JavaInit()* performs this task.
The *.JavaInit()* must provide the necessary arguments to the JVM initialization routine to set the classpath so that it can locate the Omegahat classes. Additionally, it can set any system level properties that one would usually pass to the **`java`** executable via the D argument. The pair of classpath and properties are then passed to the C routine `s_start_VM()` which initializes the JVM and starts the Omegahat evaluator which will be used to process subsequent commands from R.
Specifying the correct classpath can be a tedious and delicate process. The function *javaConfig()* is used to simplify this and return a suitable value. It has access to a default classpath that is created during the configuration of the R-Java package.
javaConfig() returns a list with 3 elements. The first is the classpath. It merges any user-specified values with the defaults in the approrpriate order. The second element is a character vector of the properties. Again, the user specified properties are merged with the defaults. And finally, the library path for shared libraries is constructed. By default, this is empty. One specifies values for it when the additional Java classes that one wishes to access during the session use native code in shared libraries. More on this later. (See 15)
The properties are specifed as a named character vector. For example, to specify the equivalent of the Java invocation

```
[*]
java  -DHOME=/home/duncan -DOMEGA_HOME=/home/duncan/Projects/org/omegahat  -DreadSerialize
DOmegaInit=OmegaInit -DUserScript=/home/duncan/.omegahatrc
```

we would use the following R commands

```
[*]
 javaConfig(properties=c(HOME="/home/duncan", "OMEGA_HOME"="/home/duncan/Projects/org/omega
SerializedClassLists=F, OmegaInit="OmegaInit", UserScript="/home/duncan/.omegahatrc"))
```

Note that name OMEGA˙HOME must be quoted as R treats the underscore specially.
The properties can be conveniently converted to command line form for java via the R expression (where *p()* refers to the properties)

```
[*]
 paste("-D", paste(names(p), p,sep="="),sep="")
```

In fact, the function *mergeProperties()* is used to do this same task and also allowing the user to specify additional properties. This is done automaticall in *.JavaInit()*.
The first call below shows how one can add more properties to the JVM initialization. The second call shows how to replace the default configuration properties values. The third call shows how to override all the values.

```
[*]
.JavaInit(list(classpath=, properties=c(HOME="/home/duncan", "OMEGA_HOME"="/home/duncan/Pro
SerializedClassLists=F, OmegaInit="OmegaInit", UserScript="/home/duncan/.omegahatrc"),libra

.JavaInit(default=javaConfig(classpath=.javaConfig$classpath, properties=c(HOME="/home/dunc
SerializedClassLists=F, OmegaInit="OmegaInit", UserScript="/home/duncan/.omegahatrc"),libra

.JavaInit(default=javaConfig())
```

It is often convenient to specify the property `java.compiler` with the value `NONE` when initializing the JVM. This can be done permanently in the *.javaConfig()*. This turns off the Just-in-time compilation in the virtual machine and means that stack traces displayed when an exception is displayed contain the line numbers of the Java classes. This is "useful" when debugging.
The call to *.JavaInit()* passes the arguments to a C routine which calls another routine `create_Java_vm()` in the library shared by both R and S. This initializes the virtual machine and then creates the evaluation manager which will broker the requests from R to the Java classes. When *.JavaInit()* returns, the virtual machine is alive and we are ready to issue Java commands.
By default, the evaluation manager created during the initialization of the R-Java connection is an instance of class *OmegaInterfaceManager*. However, a different class can be used. The fully qualified name of a class (in internal Java notation i.e. separating package qualifiers with a / rather than a .) can be specified as the value of the property *InterfaceManagerClass*.

3 Initialization Errors

Most of the initialization errors arise from a mis-specified classpath. Alternatively, if one specifies a different class for the evaluator manager, it may not be found in the classpath or does not provide a constructor with no arguments.

4 Accessing Java Classes

The following R commands exercise the basic functionality of the interface.

```
[*]
.JavaConstructor("java.util.Vector", as.integer(10), .name="vv")
.Java("vv","add", "A string")
.Java("vv","addElement", 1)
.Java("vv","addElement", as.integer(10))
x <- rnorm(10)
.Java("vv","addElement", x)
.Java("vv","elementAt", as.integer(0))
.Java("vv","size")
```

More interesting and gratifying examples can be generated via the GUI facilities.

```
[*]
.Java(NULL, "help")
 b <- .JavaConstructor("javax.swing.JButton","Click me")
 .JavaConstructor("GenericFrame", b, T)
```

The heart of the *.Java()* function is the C routine `RS_JAVA_genericJavaCall()` which connects the R engine with the JVM via the JNI. This routine is quite simple (since it relies on other routines to perform the sub-tasks.) The steps it takes are as follows.

- Convert the specification of the object on which the method is to be invoked to a Java object. This handles the special case **NULL** which identifies the evaluator manager as the object. Also, R objects of class Omegahat references are converted to their counterpart in Java.
- The R arguments are converted to an array of Java `Object`s. They are converted to Java objects as they are added using the same conversion as for the qualifier above.
- The names of the R arguments (specified via the ... mechanism in R) are converted to an array of `String` values. These are used to create permanent references to the converted values in Java for future use. (Unnamed arguments are removed after the call.)
- The value specified via the *.name* argument is converted to a Java `String`.
- At this point, we can call the method `genericCallMethod()` with the signature

  ```
  public Object genericCallMethod(Object qualifier,
                                  String methodName,
                                   Object[] args, String[] names,
                                     String returnName)
  ```

  The method name is specified expclicitly as the second argument to the *.Java()* function and it is converted to a Java `String`.
- Finally, the value returned from this call is converted to an R object. This uses the registered converters for Java objects to R objects and the built-in converters for primitives (and references).

5 Creating Arrays

Arrays can be returned from a Java method and R vectors are converted to arrays as needed. However, we cannot use the *.JavaConstructor()* and *.Java()* methods directly to create and manipulate arrays.
We want to be able to create arrays in the following manner

```
[]
 .JavaArray("String", dim=n)
 .JavaArray("String", attributes(eurodist)$Labels[1:4])

 .JavaArray("util.Vector", dim=n)
```

Creating nested arrays can be handled in the same manner. Initializing them is slightly tricky. We specify a vector as the length of t

```
[]
 .JavaArray("String", dim=c(3,4,))
```

Finally, we want to be able to treat array objects as if they were vectors in S.

```
[]
 a <- .JavaArrayConstruct("String", letters[1:6])
 a[1:3]
 a[1:3] <-
```

6 Data Conversion

The "innovative" aspect of this (and the CORBA) interface is the way we handle the transfer of data between the two systems - Java and R. The idea is quite simple. We leave all non-primitive objects (where primtive objects are ints, doubles, long, short, characters, void, float, strings and booleans) in the system in which they were created. Instead of copying their contents recursively or generating some ad hoc general mechanism for transferring them, we store them in their native system and pass a reference to the object to the other system. That reference can be used by the foreign system to invoke methods and access fields in that object. It is this approach that makes the interface facilities rich. It can be termed "lazy transfer" and avoids copying data unecessarily. It also allows us to deal with all data types in the same manner, whether they were known at compile time or not.
There are classes for which it does make sense to convert a Java value/instance to an R object. An obvious example of this is the reference classes themselves. Also, property tables, reflectance objects (e.g. Method and Constructor instances), and so on are convenient to explcitly transfer. Basically, these are all known at compile time and we have a natural way to represent them in R. Additionally, they are important classes which we want to efficiently copy between systems.
When Java returns an object/value as the result of a call, it determines whether its is considered convertible. The evaluator manager does this by asking its *ConvertibleClassifier* object whether the object is convertible. If this returns false, a reference to the object is returned. Otherwise, it is assumed that the C code will convert the object appropriately. The basic convertible classifiers used maintain a list or vector of classes that are deemed convertible. If the class of the object is contained in this list, the object is deemed convertible and passed to the JNI code that connects R and Java as is.
Note that if the R call to a Java method or constructor specifies a name for the return value (via the *.name* argument), no attempt to copy the object is made by the interface manager. Instead, a (named) reference is automatically returned.
The default *ConvertibleClassifierInt* object is constructed when the interface manager is created (during the call to the R function *.JavaInit()*). The manager allows the user to specify which class to instantiate via the property *ForeignConvertibleClassifierClass*. It then attempts to create an instance of this by calling the constructor that takes the evaluator as its only argument. If this fails, an instance of *BasicConvertibleClassifier* is used.
The user can (dynamically) manage which classes are considered convertible. The manager provides convenience methods for adding and removing classes from the list of types that are convertible. The *R* function *setConvertible.java()* takes the name of a class, a logical value and optionally an intege. This calls the corresponding method in the interface manager which passes it on to the *ConvertibleClassifierInt.*
(See the Javadoc comments for a more detailed description of what each argument does.) One must ensure that the C routines registered (see below) to perform the conversions are added to the internal conversion list at the same time as a

class is registered as being convertible with the *ConvertibleClassifierInt* object. One need not remove converters from the internal C conversion list when a class is removed from the classifier. This is because it will never be utilized since an object of that class will be converted to an anonymous reference before the classifier is asked about its conversion.
While the basic implementations of the *ConvertibleClassifierInt* work on classes, more general versions can examine the particular characteristics of an object (such as its length, etc.) rather than simply considering its type.
A list of all classes that are considered convertible by the *ConvertibleClassifierInt* can be retrieved from interface manager via the R function *getConvertibleClasses()()*.
As objects are passed from an R function call to a Java method, they are converted into a form suitable for Java. The examples above illustrate how primitive values can be transformed automatically. The primitives are vectors of characters, logicals, integers, and numerics. These are mapped to the corresponding Java scalar if the R vector is of length 1, and to a Java array whose elements are the corresponding Java type for $n > 1$. Similarly, the return value of Java methods are converted in the opposite direction, but symmetrically.
The transfer of non-primitive objects has always been the difficulty in inter-system interface design and implementation. We describe here a novel approach and illustrate its benefits. Consider the return value of a Java method which is not a primitive.
The function *.Java()* calls the *Java*^{TM} method `genericMethod()`. This is done via the JNI in C code. The JNI invocation of the Java method is returned a *Java*^{TM} object. At this point, it converts it to an R object. It is here that the primitives are converted. However, if the object is not a primitive or a reference, a more generic mechanism is used. Firstly, we look through a table of converters. This searches each of the keys in the table for a match with the class of the object. (This is a little tricky and discussed below). The first of these that matches is then used to perform the conversion to an R object.
The matching of a converter to an object is a little complicated. We have a general, if inefficient, method. The user provides a routine which can test not only the type of the actual object to be converted but also its contents or values to determine if the conversion routine associated with this converter element is appropriate.
One might imagine that we can simply look at the class of the object being converted and compare it to the class expected by the converter. However, consider the issue of extended or derived classes. Suppose we have a class A and a class B that is derived from A. If an object of class B is to be converted, the matching routine can determine whether a match is appropriate in any of the following ways

```
[]
  obj instanceof A
  obj.getClass().equals(A)
  A.isAssignableFrom(obj.getClass())
```

We provide C routines to implement each of these strategies. They are `SimpleExactClassMatch()`, `InstanceOfFromClassMatch()` and `AssignableFromClassMatch()`. To use these, one normally registers a converter element using `addFromJavaConverter()` or `addFromJavaConverterInfo()`. We use the `userData` field of the converter element to store a reference to the Java class we know how to convert from. Then, we retrieve this and compare the class of the object being converted and the one we are capable of handling.
One can determine the number of converters registered for each direction of conversion using the function *getNumConverters.java()*. Additionally, one can retrieve a description for each of the converters. The description must be specified when the converter is registered and is as good as the person specifying it. In other words, it is currently not automated.
The structure below is what is used internally to store the converters.

```
[*]
typedef struct {

   /* Determines whether this element is to be used for
      converting the specified object. If this returns
      false, it is skipped. Otherwise, it is used to
      perform the conversion.
```

```
    */
 boolean (*match)(jobject obj, JNIEnv *env, RStoJavaConverter *converter);

   /* The method that performs the computation
      to convert the Java object to an R/S object.
    */
 USER_OBJECT_ (*convert)(jobject obj, JNIEnv *env, RStoJavaConverter *converter);


} RStoJavaConverter;
```

When a converter realizes it cannot actually convert a Java object, it can elect to do the default operation of returning an anoymous reference object. This can be done in C code by calling the evaluation manager's Java method. Then the associated R object must be created from the return value of that call. This can be done in a single call to `anonymousAssign()`.

7 Reflectance

One of the convenient facilities in both R/S and Java is that we can use the reflectance provided by those languages to examine elements of the language. For example, we can determine the number and types of arguments a Java method expects. We can use this to perform automatic conversions of R arguments to Java methods. Additionally, we can provide feedback to the user in the case of ambiguities due to polymorhpism. We can even automatically create wrappers for Java methods, compare the number of arguments of R functions with those of the corresponding Java methods, etc.
In order to facilitate this style of meta-programming, we provide special converters for certain Java classes and methods within the interface manager for accessing the methods, constructors and fields of a class.
A Java method description in R has the following fields/slots.

name	
Class	
signature	a vector of class names
modifier	whether it is public, protected, private; static or not

We have added some high-level access methods to the Interface Manager class (*OmegaInterfaceManager* which allows one to get the methods and constructors for a class. This can then be used to drill down recursively through a class definition and determine sufficient information to check the number and potential type of arguments to a method, constructor and so on.
The conversion is done directly in C code for efficiency and simplicity. See Conversion below.
These methods can be access as in the examples below.

```
[]
.Java(NULL, "getMethod", "util.Vector", "add")
.Java(NULL, "getMethods", "java.awt.event.ActionListener")

.Java(NULL, "getConstructors", "util.Vector")
```

8 The Omegahat Evaluator

The heart of the link between R and Java is the Omegahat evaluator. This brokers the method invocation requests from Java, manages the Java objects that are exported to R as results from method invocations, and also performs many tasks that greatly simplify the interaction with Java from R.

The evaluator is a Java object and one can invoke its methods quite simply. By specifying the value **NULL** or the name ¨*Evaluator()* for the qualifier argument in the *.Java()* function, one identifies the evaluator (or an Omegahat function). All the methods this object offers can then be invoked directly. For example, to get a list of the objects it currently has in its default databases (the named references) we can issue the command

```
[*]
> .Java(NULL, "objects")
[1] "__Manager"   "__Evaluator" "x"
```

Additionally, we can ask the evaluator to resolve a partially qualified class name for us.

```
[*]
> k <- .Java(NULL, "findClass", "JFrame")
> k
$key
[1] "1"

$className
[1] "java.lang.Class"

attr(,"class")
[1] "AnonymousOmegahatReference"
> .Java(k, "getName")
[1] "javax.swing.JFrame"
```

Or alternatively, we can inline the two calls as

```
[*]
> .Java(.Java(NULL, "findClass", "JFrame"),"getName")
[1] "javax.swing.JFrame"
```

As mentioned below, we can use the evaluator to query how things will happen and get meta-information about the system. For example, we can ask it what Java classes are considered convertible to R.

```
[*]
> .Java(NULL, "getConvertibleClasses")
[1] "org.omegahat.Interfaces.NativeInterface.ForeignReferenceInt"
[2] "java.lang.reflect.Method"
[3] "java.lang.reflect.Constructor"
[4] "java.util.Properties"
```

We can compare the result against the classes for which the low-level R internals (the C code) has converters.

```
[*]
> getJavaConverterDescriptions(F)
[[1]]
[1] "Converts any Java InterfaceReference"
[2] "class == java.lang.reflect.Method"
[3] "class == java.lang.reflect.Constructor"
[4] "instanceof java.util.Properties"
```

9 Where to Compute

One of the difficulties posed by interfaces between different languages is the question of where to implement code. This is an embarassment of riches that we should be happy to have. Consider the case where we want to use Java to read a file on the Web. We can use sockets in R to do this. Alternatively, we can use shell commands accessed from R. For example, **`wget`** performs this task admirably. Alternatively, we can use Java's network capabilities. The $\widehat{\Omega}$ class *StatDataURL* provides a convenient class to use from R or S for this purpose. We instantiate an object of that class by calling its constructor that expects the a URL as a string. Then we call the `getContents()` method that returns an array of `String` objects. This is automatically converted from Java to an R character vector and we are done.

```
[*]
u <- .JavaConstructor("StatDataURL", "http://www.omegahat.org/index.html")
txt <-  .Java(u, "getContents")
```

Finally, we should free the resources associated with

```
[]
.Java(NULL, "clearReference", u)
```

10 Bugs, Anomalies, and errors

Quitting the R session may not behave correctly. Sometimes one has to quit, interrupt and then quit again. We will find out what is causing this and fix it. Most likely it is the presence of multiple threads in the JVM.

11 Callbacks

While it is convenient to be able to call Java methods from R, copying data to Java quickly becomes an expensive operation. Instead, we want to use a reference approach in which an R object is passed to Java by reference rather than copying all its contents. The benefits of this are quite extensive. It allows one to defer computation until needed. An object is not "evaluated" until necessary.
The following is an example of how we use an R function (actually a closure) to implement a Java *interface*. Consider the case where we want to associate an R function with a Java method so that when the Java method is called, it invokes the R function. A simple example is an event handler for clicking on a button. Suppose we create a Swing `JButton`. When the user clicks on it, we want to print out the total number of times this button has been clicked. (This is just an example, and more interesting scenarios are discussed below!). We use an R closure to store the count of the number of clicks and also provide functions to both increment and query that value. The *handler()* below is such a closure generator. We invoke it to get a function with an environment that contains an instance of *n()*.
The function (*actionPerformed()*) also prints out its only argument.

```
[*]
 handler <- function() {
   n <- 0
   actionPerformed <- function(event) {
     n
<<- n + 1
     print(event)
     print(n)
   }
   return(list(actionPerformed=actionPerformed))
 }
 handler()
```

As described, this has nothing to do with Java. However, we will use an object generated by a call to this function to respond to a user clicking on a button.
We first create the button, and put it into a window. (We use a convenience class provided by Omegahat for creating the window.)

```
[]
 b <- .JavaConstructor("javax.swing.JButton","Click me")
 win <- .JavaConstructor("GenericFrame", b, T)
```

Now, we want to register this function as a listener for events on the button.

```
[]
 .Java(b, "addActionListener", handler())
```

A little thought about how the appropriate Java method within the class of *b()* illustrates a potential problem. There is only one method `addActionListener()` in the class `JButton`. It takes one argument and this is of type `java.awt.event.ActionListener`. Clearly, there is no connection between an `ActionListener` and an R closure/function.
We solve this difficulty by using a symmetric approach to the concept of remote or foreign references from Omegahat. Firstly, keep the function in R and pass a reference identify the R object to Omegahat. The reference can be used to access the R object when it is needed. In this case, that is when the `actionPerformed()` method of the argument to `addActionListener()` is invoked. We will discuss how the reference system works later. We can assume that the reference is converted to a Java object and that there is a mechanism to invert this procedure.
Now, a more compelling aspect is how we create an object which implements the `ActionListener()` and calls the R function with the event object. We will see that this can be done automatically by Omegahat. However, it is useful to consider how we might do this. (The mechanism is almost identical to the mechanism we use to implement the embedded CORBA in R and S, functions in Omegahat as Java interfaces, and several other places we have inter-system callbacks).
Firstly, we need to create a new class, say *RFunctionActionListener*s which implements `ActionListener`. Since this interface contains just one method, we only have to implement it. The body of the method must call C code which invokes the associated R function. For this, we can use a **native** Java method.

```
[]
 native void actionPerformed(ActionEvent ev);
```

Now, we must implement this in C. The C routine receives a reference to **this**, the Java object whose native method is being called. Also, it receives a reference to the single argument, the event. (Also, it receives a reference to the JNI environment.) It must convert the event argument to an R object of the suitable type. This, by default, would be an anonymous reference. Then, it must retrieve the reference to the R object and resolve it. The reference needs to be stored in the **this** object being invoked. Thus, the Java class must store this when it is created and provide a constructor which can supply the reference.

```
[]
{
 jobject jref, eventRef;
 USER_OBJECT_ rref, e;
    jref = getJavaReference(This,  env)
    rref = getRReference(jref, env);

    PROTECT(eventRef = anonymousAssign(env, ev, NULL_JAVA_OBJECT));
    e = allocVector(LANGSXP, 4);
    PROTECT(e);
    CAR(e) = rref;
    CAR(CDR(e)) = eventRef;

    eval(e);

    UNPROTECT(2);
   /* Ignore the return value of R. */
}
```

An alternative to passing the event to the R function is to decompose it and pass the relevant information it contains. This information includes the source of the event (the `JButton`) and the action string contained in the event.
The reader might well consider this to be an excessive amount of work. Not only is it a complex task, but it must be done before the R session is started so that we can create the C code and load it into R. (Of course, we could develop it while the session is running, but the point is that it must be done externally.) But the good news is that there is a better way.
Omegahat can automatically compile a class which performs the necessary calls to invoke an R function with the Java arguments converted to R objects. This can be done for any interface and can be performed at any point during the session or outside of the session ahead of time. Specifically, it can be done on demand when a new class is needed by Omegahat to satisfy the constraints on an parameter type.
The code is generated via the *ForeignReferenceClassGenerator*. This creates code for each of the methods in the interfaces passed to it which packages the arguments into a *List* and calls the `eval()` method inherited by the base class each of the newly generated classes extends. This `eval()` is also given information about the the *Java*^{TM} method that is calling it. This information consists of the name of the method, the return type and the class type of each of the arguments. All of these are provided as strings, and the parameter information is given as an array of strings.
The `eval()` method simply passes on the information provided to it in addition to the name/identifier of the reference to another form of the `eval()`. This latter method is a native routine implemented in *C*. This converts the auxilary information describing the real Java method being invoked, i.e. the one that initiated the call to `eval()` into R character vectors and adds them to a an R list. Then the actual arguments are converted to R objects. This uses the basic conversion mechanism, looking for converter elements that match.
At this point, we are ready to call the appropriate R function. We still have yet to determine which this is. The default strategy is to call the registered Java handler. This is a function within a closure that has access to the reference objects. This function resolves the specified reference. It then looks at this value and determines how to dispatch the call. If it is a simple function, then we call it. If it is a list, we look for an element whose name matches the method name. If this is a function, we call it passing the converted Java arguments. Finally, if this is a list, then we have polymorphic function, and we search for a match. It is at this point that we try to use the Java parameter types to identify which of the functions to invoke. How this can be done simply and effectively remains to be seen.
One might think that all the layers of indirection will cause severe performance difficulties. Indeed, this may well be true. However, the generailty of the setup allows one to override this at numerous different points in the process. At worst, one can use a specialized C routine (like we did in the example above using the `actionPerformed()` method). To do this, one can create a new class using the *ForeignReferenceClassGenerator*, but specifying a different base or super class.

```
[]
gen = new ForeignReferenceClassGenerator("java.awt.event.ActionListener","RActionListener",
gen.superClassName("SpecializedRForeignReference");
gen.make();
```

Then one provides a different implementation of the native `eval()` method. All the routines we use to convert the arguments, resolve references, etc. are available to this routine. Additionally, the standard Java utilities (via JNI) can

be used, as can all the facilities.
A more reasonable approach is to specify a different central dispatch function. This is the function that is called with the reference name, method name, arguments and signature elements. One can provide a different function that takes advantage of a specialized context and implements the dispatching more efficiently. To set a new dispatch handler, one can use the *getJavaHandler()* and *setJavaHandler()*. These retrieve and set the current setting of the handler. These ensure that the C routines can see the object and protect it from being deleted by the memory manager. When replacing the handler, one can use the *javaHandlerGenerator()* to create a new instance. One should also copy the references in the existing handler to the new one.

```
[*]
 old <- getJavaHandler()
 old$handler <- function(...) {
                  # do things differently
                }
 setJavaHandler(old)


[*]
 old <- getJavaHandler()
 newHandler <- ....
 for(i in old$references()) {
  newHandler$addReference(i)
 }
```

This approach can be complicated by the presence of multiple references that can be called asynchronously from Java. If these need different dispatching handlers, one cannot replace the default handler without providing a general version that handles *all* references.
We can support a more generic approach by having classes or attributes for references and handlers. A handler (the dispatch function and the reference table) can be given an attribute to identify it as being of a particulr type. This can be understood by the C code that implements the *Java*^{TM} `eval()` method. For example, we might check for the existence of a “NoSignature” attribute on the reference value, or check if the reference is of class “DirectReference”. Then we can dispatch differently according the presence or absence of this type of attribute.
Note that it is reasonably straightforward to experiment with different native routines that call R references to implement *Java*^{TM} methods. The *C*routines `getJavaReference()`, `getRReference()`, `anonymousAssign()` allow one to manipulate the references in the two languages. Creating and evaluating a call to an R function call is relatively simple to copy from our examples (see `createCall()`).

12 Generating Interface methods in R

We can use Java’s reflectance to create a skeleton or collection of stubs for an R closure that implements the Java interface. For example, consider the idea of implementing a *DataFrameInt* with an *data.frame*. We must create a closure with the appropriate methods. We can discover the methods that a *DataFrameInt* must provide using the command

```
[]
 .Java(NULL,"getMethods","DataFrameInt")
```

Having got this information in R, we can then generate a closure with template implementations of these methods. The function *interfaceGenerator()* performs this functionality.

```
[]
```

When we transfer a non-primitive object from R to Java, there is a potential one-to-many mapping to the Java type. Primitives als have a one-to-many mapping, but the set of possible target types is small. However, for non-primitives, the number of target types is essentially the set of all known classes. We can get information from the Java method to be invoked. However, at present, the caller in R must specify the target type.
In our example, rather than simply print the number of times the handler has been invoked, we can use it to hide the window. We can either store a reference to the window in the closure, or compute it within the *actionPerformed()* function. The former can be done as follows. (We inline the definition within the call to the closure generator.)

```
[*]
win <- .JavaConstructor("GenericFrame", b, T)
(function(window)
{
 actionPerformed <- function(ev) {
  .Java(window,"setVisible", F)
 }
 return(actionPerformed)
})(win)
```

The second approach involves performing additional computations in the *actionPerformed()*. In Java, we would use the information in the event to get the source of the event. This is the button. Then we would ask for its parent (taking advantange of the fact that we know the window is the parent container of the button). Then we can simply hide that parent window. In R/S, this involves two calls to the *.Java()* function.
Note that we do not need a closure here as there is no call-specific data to be stored with the function's environment.

```
[]
 actionPerformed <- function(ev) {
   src <- .Java(ev, "getSource")
   window <- .Java(src, "getParent")

  .Java(window,"setVisible", F)
 }
```

Note that this does simplify the memory management as there is no longer a reference to the window in the Omegahat databases. As a result, the window object can be released. Otherwise we would have to explicitly release the R reference (via the Omegahat database) to the window. That can be done passively by registering a listener for the destruction of the window. (This is now what is intended here as the window is simply being hidden and not destroyed and we can display it again (assuming we have a reference to it) without recreating it.
We can use our example above to show how to mix a command line and event driven programming model. When we click on the button, the window is hidden as a result of the callback. However, we still have a reference to the window object. We can call the Java `setVisible()` method for the window and cause it to be displayed.

```
[]
 .Java(win, "setVisible", T)
```

Synchronization is of course an issue here. However, this illustrates an important advantage of systems such as R and S, Omegahat, Matlab, XLisp, and so on. The presence of a interactive, command line, interpreted language underlying a graphical interface provides significantly greater flexibility than the GUI by itself. Additionally, all GUIs should be designed and implemented so that the individual actions accessible from the GUI can be performed programmatically from within code, be it compiled or interactive as in R. This essentially means that the event handlers should call routines which perform the desired action. These routines should update the state of the GUI display (e.g. checkboxes, radio buttons, selected elements with a list or text widget) rather than assuming this has been done as an integral part of the event.

What if you forget to store a reference to a foreign object? For example, suppose we create the button and window as in the example above but omit to save a reference to the window.

```
[]
   b <- .JavaConstructor("javax.swing.JButton","Click me")
  .JavaConstructor("GenericFrame", b, T)
```

Now, suppose we want to hide the window. (Of course, we can "compute" a reference to the window from the object *b()*.) Well, the reference is still available from the Omegahat manager. We can ask it to give us a list of all the references it is managing and this can be done for either or both of the named and anonymous databases. The R function *getForeignReferences.java()*. (This is similar to *getNumConverters.java()* and *getConverterDescriptions.java()*.) It takes a vector of logical values indicating which type of references in which we are interested. This defaults to both named and anonymous.

13 Errors in R functions

When there is an error in the function being invoked, a RuntimeException is thrown by the C code and encountered when the Java code is handed control. (Needs to be implemented.)

14 Debugging

Debugging the R.so and associated code is difficult using gdb. The support for threading in gdb is limited, although there is a patch.

15 JNI and Java Classes