An automated mechanism for invoking C routines from R

The Rffi, RGCCTranslationUnit and Rllvm packages

Duncan Temple Lang, University of California at Davis

Table of Contents

We describe an approach to calling arbitrary C routines from R. This is an alternative mechanism from writing wrapper routines via the .C or .Call interfaces. This approach involves dynamically describing the signature of the routine - the types of inputs and its outputs - of a routine. We then use this description without any supporting C code to invoke the existing routine. This approach can handle scalar data types as well as complex structures. We also illustrate a mechanism for automating this dynamic interface by programatically reading the existing C code and generating the signatures and data structures descriptions. We also suggest an approach to making R more efficient using compiled code.

High-level interpreted languages such as R, MATLAB and Python, typically provide a mechanism to invoke compiled, native code. One can pass values from those languages across the interface to C/C++ and FORTRAN routines. The values are marshaled to the corresponding representation in C/C++ or FORTRAN and control is passed to the native routine. The results are then marshaled back to an R representation. In R, this foreign function interface (FFI) is available via the `.C()`, `.Call()`, `.FORTRAN()` and `.External()` functions. The typical sequence of steps to interfacing to a routine is

1. identify the routine and its signature - types of its input parameters and its return type
2. determine whether to invoke the routine via the `.C()` or `.Call()` interface
3. write a wrapper routine that conforms to the specific types allowed by the chosen interface (.C or .Call)
4. compile the new wrapper routine, and link as a shared library with the compiled target routine
5. dynamically load the DSO containing the wrapper routine
6. invoke the routine, coercing the R values to the relevant R data types to conform with the expected types of the wrapper routine
7. debug using a native debugger such as gdb

This is often a time consuming task and some challenging for those less familiar with writing this type of code. One is working in two languages simultaneously and good design decisions are not initially obvious.

In this paper, we present an alternative mechanism for invoking native C routines from within R. It removes the need to determine the appropriate R interface (.C or .Call) (step 2), write a wrapper routine in C, compile and link and load that compiled wrapper routine (steps 3, 4, and 5). And because the user doesn't have to write C code, there is no new native code to debug (step 7). So we have reduced many of the steps. We have also removed the need for the user to have tools such as a compiler installed.

The key to the mechanism we describe here is libffi. This is a compiled library that provides data structures and routines via the C language which enable us to invoke routines by simply describing the types of the inputs and output. libffi takes care of making the call and marshaling the raw inputs to and from the routine being called. It does this in a portable manner across different operating systems, allowing us to have a single piece of code for R that works on all systems. The Rffi package provides an interface to libffi and supporting functionality to convert between R and C data types in a general manner.

This libffi-based mechanism for interfacing to native routines and data structure is simpler in many ways than the regular `.C()` or `.Call()` interfaces. Importantly, it allows us to invoke arbitrary routines directly without having to write a wrapper routine. However, it does require understanding a new model. Furthermore, the descriptions must be available and instantiated at run-time. This potentially makes the mechanism slightly slower than the `.C()` or `.Call()` interfaces. This is insignificant

Note

Measure

and so Rffi potential offers benefits in both flexibility and simplicity

The paper is divided into four sections. First we describe the Rffi package and how we can manually create descriptions of routines and data structures in C and use these to invoke routines from R without any compilation of code (after the Rffi package is installed). In the second part of the paper, we describe how we can use a tool such as RCIndex or RGCCTranslationUnit to programmatically generate the routine and data structure descriptions by automating the reading of the C code. In the third part of the paper, we show how we can pass function pointers in C as arguments in calls from R and also how we can use R functions as implementations of (This part I might not bother to do.) We end the paper with a brief discussion about different enhancements that others might want to pursue.

Need for support routines at run-time, e.g. to convert arrays in C to R vectors when the length of the array is not known when R is converting the result in Rffi, but just sees a pointer.

The Basics of Rffi

In this section of the paper we present the basic facilities provided by the Rffi package for directly and dynamically interfacing to existing compiled routines. This approach allows us to describe the details of a routine and the layout or definition of complex data structures within R. We then use these generic descriptions to invoke specific functions corresponding to that interface or that particular definition. The essential ideas are that we create a description of a routine to specify the different parameter and return types of the routine. This then allows us to invoke that native routine from R, passing R objects as the arguments to the routine and obtaining both the returned value and any mutable inputs that might have been changed. The types of the inputs and outputs of a routine can be simple primitive C-level data types such as int, double, short, long, char * (for character strings) and so on. We can also define complex data structures (structs) by defining the collection of field names and types. The Rffi package also handles marshaling such struct types to and from R. In this section we illustrate how to invoke different native routines from R.

We start with a very simple sample routine that passes an integer value to a C routine and returns an integer. The routine *intCall* in the Rffi package provides an example. In order to be able to invoke such a routine, we need to describe a generic routine with this signature - inputs are a single integer and outputs are a double. A signature is the both the collection of input types - the parameters - and the return type. We create a call interface object - a CIF - to describe the class of routines with the same signature. The R function `CIF()` allows us to define such an interface that we can use for any C routine with this particular signature. For our routine that accepts an integer and returns a C `double` we create the CIF with the R code

```
cif = CIF(doubleType, list(sint32Type))
```

The first argument is the return type. If the routine has input parameters, we specify their types via a list. The list can have names to give symbolic names to the parameters, but these are not used at present. There are 15 built-in data type identifiers available as R variables in the Rffi package. These are described in table ???

Table 1. Standard data type identifiers

R variable	C data type
`doubleType`	double
`floatType`	float
`longdoubleType`	long double
`pointerType`	pointer to any type of object/value
`sint16Type`	signed 16-bit integer
`sint32Type`	signed 32-bit integer, corresponding to int
`sint64Type`	signed 64-bit integer
`sint8Type`	signed 8-bit integer
`stringType`	a char *, a pointer to a string. This is introduced in R to identify strings rather than generic pointers.
`structType`	a struct description containing information about the types of the fields
`uint16Type`	unsigned 16-bit integer
`uint32Type`	unsigned 32-bit integer
`uint64Type`	unsigned 64-bit integer
`uint8Type`	unsigned 8-bit integer
`voidType`	the void type

signed integers with n bits can represent integer values between (-2)
<sup>n-1</sup>
and (2)
<sup>n-1</sup>
inclusive. Unsigned integers can take values This collection of R variables allows us to describe almost any type we need to identify in C[1].

---

[1] Bit-fields are not feasible using these.

What happens to data types R doesn't have? When converting from a C data type to R, we map the value to the type in R that is guaranteed to fit the value. For an unsigned 32- or 64- bit integer, we use a numeric vector as this is guaranteed not to loose information. An unsigned 16-bit integer however can fit into a regular integer vector in R without loss of information and so we would return an integer vector of length 1 in R. R does not have an explicit, separate representation for a float. Therefore, we convert a C-level float to an R numeric vector of length 1. `pointerType` identifies a generic pointer, but not necessarily the type of object to which it points. It can be used to identify an array or a regular pointer as it is the type for a memory address. `stringType` is a special pointer because of its common usage. This identifies a pointer to a sequence of char values. There is no information about character encoding. `voidType` is used to identify the void type `structType` is the one type that doesn't simply identify a scalar type. Instead, it represents a struct type that is an aggregation of one or more fields. It is similar to an S3 or S4 class in R, and requires information about the types of the individual fields. We will see an example of using the *`structType`*() function to create a description of a struct data type.

We now return to our example of invoking *intCall*. We have created the call interface - CIF above. We can now use that interface description to invoke the routine via the function *`callCIF`*(). This takes the CIF object, the name of the routine or a pointer to the routine, and then the arguments to pass to the routine. We can do this with

```
callCIF(cif, "intCall", 4L)
```

This returns a regular R numeric vector with a single element - 5.3863. Note that we passed the argument as an explicit integer. If we had passed a numeric value, e.g.,

```
callCIF(cif, "intCall", 4.0)
```

the call would work correctly and the result would be the same. Since we know the target type, the R and marshaling code underlying the *`callCIF`*() function take care of handling simple conversions. We can even pass the number as a string

```
callCIF(cif, "intCall", "4")
```

and *`callCIF`*() will take care of the details.

Note

Mention **`NA`**s here. We don't deal with them because it is something the caller should figure out with respect to the C routine being called and its policies and handling of **`NA`**s. Calling intCall with a non-number, e.g.

```
callCIF(cif, "intCall", "abc")
```

results in a NaN as the NA value is passed across.

The *intCall* routine was already loaded as it is an example routine that is part of the Rffi package's DSO. *`getNativeSymbolInfo`*() was able to get the pointer to or address of the routine from its name. If the routine is in a different DSO, we have to load that having compiled it appropriately. As an example, let's assume we have installed the libsndfile library for working with various audio formats.

```
dyn.load("/usr/local/lib/libsndfile.dylib")
callCIF(CIF(stringType), "sf_version_string")
```

Three arguments

Struct return

Struct as an argument. Filling in a struct.

Pointer argument

Pointer return value

Arrays

Accessing Global Variables

The Rffi package also allows us to get the value of an accessible native variable using `getNativeValue()`. We use `getNativeSymbolInfo()` to obtain the address of the symbol and we provide a description of the symbol's type using the libffi data type identifiers. We pass both of these to `getNativeValue()` and the current value of the variable is converted to an R data type using the same rules as for converting the return type of a routine to an R object.

Let's look at an example. We'll suppose we have the GNU Scientific Library (libgsl) installed and we want to find out the version of that library. We could load the libgsl DSO and then access the variable

```
dyn.load("/usr/local/lib/libgsl.dylib")
getNativeValue(getNativeSymbolInfo("gsl_version")$address, stringType)
```

(The location and name of the libgsl DSO will be different on different machines. We can use

```
.Platform$dynlib.ext
```

to get the typical extension for a DSO on the current platform.)

Programmatically Obtaining & Using Interface Descriptions

Using Function Pointers

Consider numerical optimization. Suppose we have C code that takes a starting value/vector of the parameters and a function that is to be optimized and its derivative/gradient. The C routine takes care of the internal details of the optimization algorithm such as step halving, etc. As it tries to find the optimal value of the parameters, it invokes the function and gradient. We'll ignore the gradient for the moment. The C routine we want to invoke is declared as

```
int
optimize(double start, double (*fun)(double val, void *userData), void *userData);
```

We can create a CIF for this routine with

```
cif = CIF(doubleType, list(doubleType, pointerType, pointerType), rep(FALSE, 3))
```

Now we need to provide the routine that gives the value of the function to be optimized. If we have loaded the DSO with the routine that we want to pass to *optimize*, we can access it with `getNativeSymbolInfo()`, e.g.

```
myFunPtr = getNativeSymbolInfo("myFun")$address
```

This is an `externalptr` object that contains the memory address of the routine. It is a pointer to a C routine. We can pass this to *optimize* now using

```
opt = callCIF(cif, "optimize", 2.3, myFunPtr)
```

This the returns the value that optimizes this function, based on the algorithm in *optimize*.

R functions as pointers to C routines

What if we want to implement the function to be optimized as R code. This is certainly more convenient to our work flow as we do not have to step away from R and write the code in C, compile and debug it and then load it. We might have our likelihood function as something like

```
lik = function(theta)
        sum(log(dexp(data, theta)))
```

or using a closure to avoid the global variable `data`

```
likGenerator =
 function(data)
    function(theta)
        sum(log(dexp(data, theta)))
```

and the

```
myLik = likGenerator(myData)
```

We want to pass this as the `userData` argument to *optimize* and arrange for the C routine we pass as the C routine pointer argument - `fun` - to take the candidate value and the user data (the R function) and to turn this into a call to the R function which will evaluate the likelihood. To do this, we need to create this C routine to call our function. If we were to write it by hand, it might look something like the following:

```
double
R_myFun(double val, void *data)
{
    SEXP call, ans;
    PROTECT( call = allocVector(LANGSXP, 2));
    SETCAR(call, (SEXP) data);
    SETCAR(CDR(call), ScalarReal(val));
    ans = Rf_eval(call, R_GlobalEnv);
    UNPROTECT(1);
    return(asReal(ans));
}
```

This creates a call to a function and inserts the function and the candidate value as R objects. Then it evaluates the call and converts the result to a numeric scalar and returns the value.

We can use this in our call to *optimize* as

```
fun.ptr =  getNativeSymbolInfo("R_myFun")$address
callCIF(cif, "optimize", 2.3, fun.ptr, myLik)
```

The *optimize* routine has no knowledge that R was called to evaluate the function. Instead it has merely invoked our routine *R_myFun*.
e

Note that the routine *R_myFun* was created to have the same signature as the function pointer expected by *optimize*. Furthermore, the body of the routine would be different and more involved if there were additional parameters of different types in the routine that in turn had to be passed to the R function. So what we would like is a mechanism to automate the create of such a wrapper routine. We can do this with a description of the signature and data types in the function pointer. But we cannot arrange to have the routine called unless we compile and load it.

We can generate the code in R from the CIF types or from the translation unit description of the function pointer. After we do this and write it to a file, we then compile and load the code and continue as if we had written the code manually, as above. So let's now focus on how we generate this routine using functionality in RGCCTranslationUnit.

See funPtrTU.R We will write a function that takes an description of a function pointer from the translation unit and generates a wrapper routine that calls an R function. The function needs the description of the function pointer, the name of the new routine to create and may require the caller to specify which parameter refers to the R function, i.e. the user data passed to the routine. The function generates code very similar to that in the *R_myFun* code above. It creates an R data structure to represent the call with one less arguments than the number of parameters passed to the actual routine. This is because we don't pass the user data to R since this is the R function object. Our routine then adds the corresponding R value for each of the parameters in the routine to the call. Then it invokes the call with *Rf_eval*. The final step is to convert the R object returned from the R function to the appropriate C representation. Within the code, there are declarations of local variables and simple R memory management.

```
# This is for runFunPtr in ../../src/test.c - not the optimize() in the paper.

createCallRFunctionWrapper =
function(funPtr,    # the description of the function pointer - class FunctionPoin
         funcName,  # name of the routine we create
                    # index of parameter that is the user data containing the R fu
         userDataParam = findUserDataParam(funPtr),
                    # names to use for the parameters of the routine
         paramNames = names(funPtr@parmeters))
{
        # So we have an explicit representation for a FunctionPointer
        # We have the return type and the parameters. From these we can
        # define our wrapper routine
  if(length(names(funPtr@parameters)) == 0)  #?  NULL since the TU loses them - ye
      names(funPtr@parameters) = paste("x", seq(along = funPtr@parameters), sep =


     # Now create the body of the routine first.
   params = funPtr@parameters
   body = c(
                  # local variables
             "SEXP call, ans, ptr;",

                  # create the call
             sprintf("PROTECT(ptr = call = allocVector(LANGSXP, %d));", length(par

                  # put the function into the first element of the call
             sprintf("SETCAR(ptr, (SEXP) %s); ptr = CDR(ptr);", names(params)[ use
                  # add each of the parameters, except the userDataParam
             mapply(function(id, parm)
                       sprintf("SETCAR(ptr, %s); ptr = CDR(ptr);", convertValueToR
                    names(params)[ - userDataParam], params[ - userDataParam]),

                  # invoke the call
             "ans = Rf_eval(call, R_GlobalEnv);",

             "UNPROTECT(1);",
                  # conver the result back to a C value
             sprintf("return(%s);", gsub(";$", "", convertRValue("",  "ans", funPt
            )

       # Now we have to get its declaration or signature to add to the top of the
   ret = getNativeDeclaration("", funPtr@returnType, addSemiColon = FALSE)
   decl = mapply(getNativeDeclaration, names(params),
                                       lapply(params, function(x) x@type),
                  MoreArgs = list(addSemiColon = FALSE))


       # put the pieces of the code together into a character vector
  c(ret,
      sprintf("%s(%s)", funcName, paste(decl, collapse = ", ")),
      "{",
      paste("    ", body),
      "}")

}
```

We can now call this code for our particular example and then test it.

```
library(RGCCTranslationUnit)
 # read the TU file
tu = parseTU("../TU/test.c.001t.tu")
 # find only the routines in files that start with test
r = getRoutines(tu, "test")
 # get the routine of interest
f = r$runFunPtr
 # resolve all the data types referenced in the routine
f = resolveType(f, tu)

 # Verify the types interactively
sapply(f$parameters, class)
sapply(f$parameters, function(x) class(x$type))

 # Get the function pointer type parameter
funPtr = f$parameters[["fun"]]$type
class(funPtr) == "FunctionPointer"

  # Now we generate the wrapper routine, sourceing the function to do this
  # and then invoking it with the relevant arguments.
source("funPtrTU.R")
```

```
code = createCallRFunctionWrapper(funPtr, "R_myFun2", 2, paramNames = c("value", "
 # Write the generated code along with the necessary C header files
cat("#include <Rdefines.h>",
  code,
  sep = "\n", file = "foo.c")


  # Compile the foo.c into a DSO
 system(sprintf("%s/bin/R CMD SHLIB foo.c", R.home()))

  # load the resulting DSO
 dyn.load("foo.so")
  # get a reference to the newly generated routine which we
  # can pass as the function pointer argument.
 f = getNativeSymbolInfo("R_myFun2")$address

   # Define the R function that will be called each iteration
 myFun = function(val)
       val + 1

  # Now we can invoke the original C routine and pass our function pointer - R_m
  # and our R function - myFun. We create the CIF first and then invoke it.
 library(Rffi)
 cif = CIF(doubleType, list(sint32Type, doubleType, pointerType, pointerType), re

 ans = callCIF(cif, "runFunPtr", 3, pi, f, myFun)
 print(ans)
```

Compiling directly with Rllvm

We want to illustrate a different approach than generating C code via RGCCTranslationUnit. Instead of assuming the machine has a compiler such as gcc and having to invoke it and get the compilation and linker flags set appropriately or call the SHLIB command in R, we want to compile and load the code directly within in R. The Rllvm package allows us to do this.

Rllvm is an interface to the low-level virtual machine software that can portably generate machine code on different computer architectures. Within R, we can create descriptions of the definitions (not just declarations) of native routines and have llvm generate the machine code to implement those within the R process. So we can generate the implementation of of *R_myFun* as we would write it above and then use the routine directly within R without calls to a compiler, etc. And we can do this for arbitrary function pointer signatures.

Using the C code above as a template, we can generate the Rllvm code to define the equivalent routine.

There is no doubt that this is a complex task. We are working with the instruction set of a low-level virtual machine. This makes writing C code seem terse, let alone R code. The point of this example is not to suggest this is how R users should consider writing wrapper routines for passing R functions as C function pointers.

Instead, the aim is to illustrate that this can be automated using RGCCTranslationUnit and Rllvm and can be used via Rllvm. My hope is that advanced users will be able to explore additional uses of these powerful underlying technologies and the R interfaces to them.