An Extensible Conversion Mechanism for *.C()* Calls

Duncan Temple Lang

May 15, 2001

Abstract

This is a note on an addition to the *.C()* interface. The idea is to generalize the way we convert S objects to C-level pointers that are passed to the C routine being invoked by the caller of the *.C()*. The mechanism is aimed at handling references to external data such as shared memory, hdf5 files, database result sets, and data in other languages such as Python, Perl, Java, JavaScript, etc. Rather than copying the data reference in these objects into S objects and then passing them to C, we can pass these directly by converting the reference object to data values appropriate for the C routine. This has the potential to avoid the amount of data duplication and make certain memory-bound computations feasible in R.

This allows for package/library developers to register C level routines which can convert an S object to a C value. These return a *void ** value which is passed to the C routine. If no suitable converter is found, the current mechanism is used for that argument. Currently, this conversion mechanism is only invoked a) if there registered converters, and b) if the object has a class.

A similar mechanism is used throughout the Omegahat inter-system interface packages to handle objects of arbitrary types which can only be handled with user-level contextual information.

This is prompted by discussions with Robert, John, Luke. An example of how it can be used is provided in the SCConverters package.

The change extends the routines `RObjToPtr()` and the component of `do_dotCode()` that calls `CPtrToRObj()`. These changes are in some ways orthogonal to the changes we are proposing for symbol lookup. However, that mechanism of registering/exporting C routines allows one to provide additional information that can identify the converters for the R arguments and for the return value(s).

1 Motivation

The example is motivated by Robert's needs to handle data stored in hdf5 and the general inter-system interface mechanism that we have developed in Omegahat for sharing data across different languages and interpreters. Consider the following scenario. We create an object of class *hdf5Reference*.

```
x <- hdf5(filename)
```

This is an external pointer type which contains the address of a C level object which knows about its own data type and content. For our example, let's assume it is an array of real values. The C level structure might look something like

```
type length | value_0 value_1 ... value_n
```

It is reasonably easy to overload the different functions that access the data. For example, we can provide a method for *length()* and *[()*. We can provide coercion method such as *as.integer()*, etc. which create a new reference to the *same* data and add information about the mode of how the data is to be used.

One of the problems with these overloaded, symbolic methods arises when we pass the object to a C routine via the *.C()*. In our example, we will call the C routine `foo()` which expects two arguments. The first is an array of *Rint*s and the second is a scalar giving the length of this array. This is also passed to `foo()` in the form of a pointer to a *Rint*. So the declaration of `foo()` is given as

```
void foo(Rint *val, Rint *len);
```

If we pass the hdf5 reference object directly to the C routine, the current `RObjToCPtr()` will not convert it to the appropriate array of integers. Instead it will pass it as a list or simply discard the class, depending on the exact format of the *hdf5* class. What we would like to have happen is the following. A simple S call of the form

```
.C("foo", as.integer(x), length(as.integer(x)))
```

should retrieve the relevant data from the hdf5 object. If the underlying data within the hdf5 object is an array of *Sint*s, then we want to be able to pass a pointer to that single copy of the values. Otherwise, we will create the necessary array and copy the values to it. Since we have assumed in our example that the underlying data is an array of reals, we will need to create this array of *Rint* and populate it with the real values. However, if the particular instance of the hdf5 object contained *Rint* elements, we would just point to the data segment of the object. Similarly, we could avoid copying the data if the routine expected real values. Additionally, we are in control of the copy of the data and will not incur the expense of duplicate copies created by R.

So how will this simple call work? Firstly, the call to *as.integer()* will create a new S-level object. This contains a refinement of the original *hdf5Reference*. This new object will contain embedded information about the expected type of the data. This might be via a *type* slot or by changing the class to be *hdf5IntegerReference*. Either approach conveys sufficient information.

Secondly, the *length()* method for this type of object will compute the appropriate length – in our case the number of elements in the array. The current mechanism for converting S objects to arguments to C routines called via the *.C()* function will handle this second argument – the length of the array. However, it will not correctly deal with the first argument which is not a primitive type, but an object of class *hdf5Reference* or *hdf5IntegerReference*. The extension to the conversion mechanism allows the developer of the hdf5 package to register C routines that will convert these meta-objects into the appropriate C level pointers. This extensibility is necessary since we cannot know how to perform the conversion for an arbitrary S object.

1.1 Inter-Language Interfaces

A second example in which this facility may be useful is in the Omegahat inter-system interface packages. Consider the case where we have a reference to a Java or Python array of real values. If the goal is to use these in a C routine called by R, we may wish to avoid first copying the values from the foreign system (Java or Python) to R and then from R to C. Instead, we can go directly from Python to C using a converter that recognizes a foreign reference to an array of doubles and converts this to the appropriate C level value. In certain cases, the internal representation of the foreign data may be the same as expected by the C level code and no copying is necessary. Any changes made by the C routine to the data (by writing on that memory) will be immediately visible in the other system.

1.2 Return Values

Routines called via the *.C()* do not have an explicit return value. However, the *.C()* call returns the values it was given, containing any changes made by the C routine to these values. To be generally useful, we must be able to handle these return values in a manner that is consistent both with the S view of the *.C()* functions return values, and also the foreign system's view of the data. To do this, we need a *reverse* converter which takes the output from the *.C()* corresponding to an argument and creates a) an appropriate S object, and b) updates the foreign reference's data appropriately.

1.3 Managing Converters

The S user can query the list of converters using some simple functions. She can obtain the number of converters via *getNumCConverters()*. Brief descriptions of each converter (as specified by its author) are available from the function *getCConverterDescriptions()*. One can also determine which converters are active using *getCConverterStatus()*.

The user can modify individual converters by making it active or inactive. An inactive converter is ignored when searching for a potential converter for an object. This allows one to temporarily remove its effect without having to subsequently restore it at the C level. Additionally the function *removeCConverter()* explicitly removes the specified element from the list. Both of these functions allow the user to identify the element of interest in the converter list either by index or by its description string (returned from *getCConverterDescriptions()*).

At present, the user cannot directly add a converter to the list via S commands. Instead, she must use C code and specifically the routine `R_addToCConverter()`. The reasons for this restriction are simple. Firstly, the routines

and user-level data are C level symbols and specifying them in R can be more convoluted than simply writing the C code. Secondly, the converters are written in C and typically operate on C level values, so we assume the author is familiar with C. An S-language interface for adding less general converters will be added soon.

2 Generic Converters

This entire mechanism may appear complex and somewhat awkward. However, it is intended to be a general framework on which classes of simpler, more generic converters can be added. For example, a common idiom is to have a predicate routine that checks whether the object being converted has a particular class. The routine `R_matchClass()` (provided in the R binary) can be used to do this. When the developer registers the converter, she specifies `R_matchClass()` as the predicate and the name of the class which the converter is prepared to handle. When the predicate is called, it compares this name to each of the names in the class attribute of the object being converted. An example of this is contained in the SCConverters example.

Also, one can use a convention for locating the actual C routine to do the conversion. We can take the name of the target class stored in the user-level data field and use it to look for a C symbol with a related name. For example, suppose the converter can handle objects of class *hdf5*. Then, we might look for a C symbol named `R_hdf5_to_C()`. If we find it, we invoke it, passing it the object to be converted. This generic converter routine is relatively easy to define. Then other developers registering converters need only specify this routine, the `R_matchClass()` and the name of the class (e.g. *hdf5*).

One can also develop predicate and converter routines that use S functions. This allows developers to avoid programming in C while still converting the argument to a value to be directly used in the C routine.

Part of the complexity of the interface comes from the reliance on meta or symbolic computation in that the reference objects must contain information about the desired type. We have the data in one system and a reference to it in R. Coercion to different types doesn't force the data to be modified, but simply the reference to it (or more precisely a new reference to the same data). One needs to have a class and methods to support such coercion (e.g. *as.integer()*), computing dimensions (e.g. *nrow()*, *dim()*) and lengths (e.g. *length()*), etc. This extra effort does provide more structure to the software and so is of some value in its own right. More importantly however is the potential for this mechanism to make certain computations feasible. By avoiding the numerous copies that the S language must make to operate on data in a way that preserves the pass-by-value semantics, this direct access can greatly reduce the memory requirements for certain computations. If such computations could not be done in R, but now can be with direct access conversion, the benefits may be worth the extra work.

3 The Basic Idea

We maintain a linked list of converter elements. This list is used for converting R objects to C pointers. Each element in the list contains two essential ingredients

1. a predicate function called to determine if this element can successfully convert the R object, and
2. a converter function which performs the actual conversion of the R object to a C pointer.

A third, but potentially optional, routine is one that converts the C-level value passed to the C routine back to R. This reverses the conversion and allows the author to process any changes the C routine makes to the input arguments.

Additionally, the converter element has information about whether the converter is active and user-level data that is available to the predicate, converter and reverse converter routines. This is used to parameterize these routines.

Finally, each converter should have a description string that allows users to understand what converters are in effect.

To register a converter, a developer will call the routine `R_addToCConverter()`. This is (currently) declared in the header file **R˙ext/RConverters.h**. It takes five arguments. The first three are the predicate/match routine, the converter routine and the reverse converter. The reverse converter can be specified as **NULL**in which case the corresponding position in the list returned by *.C()* will be `NULL`.

Each of these three routines has a very particular signature. Again, these are specified in **R˙ext/RConveters.h** as *R˙ToCPredicate*, *R˙ToCConverter* and *R˙FromCConverter* respectively.

The remaining arguments consist of data to be used by the different routines and a description which is used by end users to identify the converter. The data argument is an arbitrary C value (specified as a *void **). It is stored in the converter element and accessible to each of the three routines so that these can be customized for different classes, etc. The description is a string which is copied so that it can be stored in the converter element.

The following code snippet contains two examples taken from the SSConverters package used to test this mechanism.

```
R_addToCConverter(matchClass, convertSeq, NULL,
                      "VirtualIntegerSequence",
                      "VirtualIntegerSequence to integer");

R_addToCConverter(matchClass, convertDoubleSeq, reverseDoubleScalar,
                      "VirtualDoubleSequence",
                      "VirtualDoubleSequence to numeric");
```

In the first case, we discard the return value by specifying **NULL** for the reverse converter argument.

3.1 Using the Converters

When converting an R object to a C argument, the sequence of actions is as follows:

1. iterate over the list of converters
   (a) invoke the predicate routine
   (b) if it returns **TRUE**,
   (c) invoke the converter
2. use the default conversion mechanism currently in `RObjToCPtr()`.

In the inter-system packages we have two lists which convert from the foreign system (i.e. C, in this case) to R. In the case of *.C()* calls, there is no return type. However, we do have to handle converting the arguments to R objects. When converting the arguments to the C routine back to R objects, we use the `reverse` field of the converter element used to convert that argument. This is stored when we convert the R object to C before the call to the C routine. If the `reverse` field of the converter element is **NULL**, we return `R_NilValue`. Otherwise, if it is non-null, we invoke it. We pass it the original argument, the C value to which that was converted by the converter itself. We also pass it auxiliary information defining the context in which the conversion is being made. This includes the name of the function, all of the arguments – both C and R values – , and the index of the argument being converted. Finally, we also pass to the `reverse()` routine the converter element itself. This allows it to retrieve the `userData` that can be used to parameterize the `matcher()`, `converter()` and `reverse()` routines. As one might expect, if the value of the R argument was not handled by one of these user-registered converters, we use the regular mechanism to convert the C value back to an R object.

4 Extensions

The change to the code is relatively minimal and hopefully the performance impact will also be negligible in the case that there are no registered converters.

We are developing a tool that can parse C code and determine the return type of a routine. In that case, we can potentially convert the return value in a semi-automatic fashion.

At present, the C routine is given no information about the type expected by the C routine. We can use the new `R_registerRoutines()` mechanism for specifying information about the parameters of a C routine accessible from S. The type of the parameter to which the R object is being converted can then be optionally specified and can be made available to the converter routine.

5 An Example: Virtual Sequences

This is a very simple example of using this facility and is merely a description of the code in the SCConverters package. It is an intentionally artificial example so as to avoid requiring other packages such as hdf5, Python, Java, etc.

The idea is very basic. We provide an S class for representing sequences symbolically. Objects of this virtual sequence do not actually store or compute their elements but merely the symbolic information that define these elements. In our case, we will look at sequences that require knowing only the length. This is because they are intended to represent sequences of the form $1, 2, \ldots, length$.

5.1 Constructor Function

The constructor for these objects is as follows.

```
vseq <-
function(len)
{
 len <- as.integer(len)
 class(len) <- "VirtualSequence"
 len
}
```

5.2 Coercion Function

We provide coercion methods *as.integer()* and *as.double()* to convert instances of the virtual sequence class to virtual sequence objects with data of that class. Again, the elements of the sequence will not be computed at this time. Instead, the result of these coercion functions is a new object with a different class. These classes are *VirtualIntegerSequence* and *VirtualDoubleSequence*, and are prepended to the vector of S3-style class names.

```
as.integer.VirtualSequence <-
function(x)
{
  class(x) <- c("VirtualIntegerSequence", class(x))
  x
}

as.double.VirtualSequence <-
function(x)
{
  class(x) <- c("VirtualDoubleSequence", class(x))
  x
}
```

5.3 Length Function

We make *length()* generic and define a method for *VirtualSequence* objects.

```
length.VirtualSequence <-
function(x)
{
 as.integer(unclass(x))
}
```

5.4 C Routines

So far these objects don't do much. We can provide methods for *[()*, etc. but this is not our focus here. Instead, we want to pass instances of the *VirtualSequence* to C routines that expect actual values and not a symbolic version. We define a simple C routine that expects an array of *Sint* values and a second argument giving the length of this array. This routine just computes the sum of the elements of the array and returns this via the second argument. This is defined easily as

```
void myTest(const Sint *vals, Sint *len)
{
    int total = 0, i;

    for(i = 0; i < *len; i++) {
        total += vals[i];
    }
    *len = total;
}
```

5.5 The Converter

In our example, we want to call this routine and pass it a *VirtualSequence* object as its first argument.

```
  v <- vseq(10)
  .C("myTest", as.integer(v), x = length(v))$x
```

To do this, we will need to convert the sequence into actual values and this is where the converters enter the picture. We want to have the internal implementation of the *.C()* function recognize the first argument as being special and call a routine that we provide to create the appropriate array of *Sint* objects. The second argument will be converter in the usual manner using the default conversion mechanism.

In order to have the *.C()* code call our converter routine, we must first write it and then register it. We first provide the predicate function that indicates whether it can handle the argument to be converted. In our case, we want to handle any object which inherits from the class *VirtualIntegerSequence* (i.e. the value returned from *as.integer()* applied to a *VirtualSequence* object). Since this involves matching the class name of the object against the string `"VirtualIntegerSequence"`, we can use the predicate routine `R_matchClass()`.

The converter routine is quite simple. It needs to allocate an array of *Sint* values and fill them in with the values of the array. The length of this array is taken from the value contained in the *VirtualSequence* object being converted. In our simple setup, this is obtained by coercing the object to an integer. The code for this converter can be written as follows.

```
void *convertSeq(SEXP obj, R_CConvertInfo *inf, R_toCConverter *el)
{
 Sint *ans;
 int i, n;

 n = asInteger(obj);
 ans = (Sint *)R_alloc(n, sizeof(Sint));
 for(i = 0; i < n ; i++) {
     ans[i] = i+1;
 }

 return(ans);
}
```

5.6 Reverse Converter

In our example, we are not interested in converting the results back to R. The `myTest()` routine returns the answer via the second argument. So we can discard the array we create in the converter knowing that `myTest()` has not

changed its values, etc. Since we don't need a reverse converter, we simply specify this as **NULL** when registering the converter.

We now have all the components in order to register the converter. We do this with the call

```
  R_addToCConverter(matchClass, convertSeq, NULL,
                        "VirtualIntegerSequence", "VirtualIntegerSequence to integer");
```

We give it the predicate function and the user-level data it needs, that is the name of the class we are prepared to handle with this converter – `"VirtualIntegerSequence"`. We also specify the convert routine `convertSeq()`. And finally, we give a description indicating that this converter handles turning a *VirtualIntegerSequence* to an array of integers.

We can register the converter at any time. It is often convenient to do this when the library is loaded. This can be done by calling a C routine from the package's *.First.lib()*. Alternatively, one can use the new initialization mechanism that calls the C routine `R_init_<lib name>()`, if it exists. We do this by defining the following routine and registering the converter(s) within it.

```
void R_init_SCConverters(DllInfo *info)
{
  R_addToCConverter(matchClass, convertSeq, NULL,
                        "VirtualIntegerSequence", "VirtualIntegerSequence to integer");
}
```

5.7 Querying the Converters

Before invoking the `myTest()` routine, we can examine what converters are registered, and which are active.

```
> getCConverterDescriptions()
[1] "VirtualIntegerSequence to integer"
> getCConverterStatus()
VirtualIntegerSequence to integer
                             TRUE
```

We could de-activate this converter and check it has taken effect.

```
> setCConverterStatus(1, FALSE)
[1] TRUE
> getCConverterStatus()
VirtualIntegerSequence to integer
                            FALSE
```

And we can re-activate it easily with the command

```
> setCConverterStatus(1, TRUE)
```

At this point, we want to call the `myTest()` routine and check the converter.

```
> v <- vseq(10)
> .C("myTest", as.integer(v), length(v))
[[1]]
NULL

[[2]]
[1] 55
```

Note that the first element of the returned list is `NULL`. This happens because we did not specify a reverse converter routine for this particular conversion element. The second value contains the sum of the elements in the sequence, which is the same as `sum(1:10)`

5.8 Reverse Converters

A more interesting example involves providing a reverse conversion routine. In this example, we will pass the sequence to a routine that expects an array of *double*s.

```
void myTest1(double *other, Sint *olen)
{
    int i;
    double sum = 0.0;

    for(i = 0; i < *olen; i++)
        sum += other[i];

    other[0] = sum;
}
```

This routine returns the sum of the elements as the value of the first element passed to it. We must retrieve that value and convert it to an R object.

First, we must define a new converter to handle *VirtualDoubleSequence* objects. This is very similar to the converter for integers, but allocates an array of *double*s rather than *Sint* elements.

```
void *convertDoubleSeq(SEXP obj, R_CConvertInfo *inf, R_toCConverter *el)
{
 double *ans;
 int i, n;

 n = asInteger(obj);
 ans = (double *)R_alloc(n, sizeof(double));
 for(i = 0; i < n ; i++) {
     ans[i] = i + 1;
 }

 return(ans);
}
```

We can use the `R_matchClass()` as the predicate, giving the class name *VirtualDoubleSequence*. What remains is writing the reverse converter to take the array of *double*s passed to `myTest1()` and extracting the answer. In our case, we know that we only want the first value. Therefore, we can use the following routine.

```
SEXP reverseDoubleScalar(void *value, SEXP arg, R_FromCConvertInfo *info,
                         R_toCConverter *el)
{
 SEXP ans =  NEW_NUMERIC(1);
 NUMERIC_DATA(ans)[0] = ((double*) value)[0];

 return(ans);
}
```

We register this converter using `R_addToCConverter()` in the following manner:

```
  R_addToCConverter(matchClass, convertDoubleSeq, reverseDoubleScalar,
                      "VirtualDoubleSequence",
                      "VirtualDoubleSequence to numeric");
```

And now we are ready to invoke the `myTest1()` routine and indirectly the converter routines.

```
> .C("myTest1", x=as.numeric(v), length(v))$x
[1] 55
```

The reverse converter has access to contextual information via the argument `info()`. This is of type *R˙FromCConvertInfo*. It contains the name of the C routine being invoked and all the arguments passed from R and to the C routine itself. In the future, it will also contain information about the C symbol as stored in the *DllInfo*. As we develop the tools, this may contain information about the different parameters of the C routine, such as their type, whether it can be modified by the routine, etc. This will provide useful information that allow us omit converting values that will not be used.