Reflectance from C/C++ code

Duncan Temple Lang
Statistics and Data Mining Research,
Bell Labs, Lucent Technologies

January 15, 2003

1 Introduction

We have developed numerous inter-system interfaces over the past few years between the S language and other interpreted languages, systems and applications. For example, we have a bi-directional interface between Java, Python, Perl, Octave, Objective-C (almost complete). We also have dynamic facilities for communicating with CORBA and (D)COM objects in a distributed environment. Fundamental to each of these is a mechanism to dynamically discover information about objects, classes and methods that we have access to in one system from the other. This discovery mechanism is called reflectance and allows us to find out the number and types of arguments to methods, the class of an object, etc. Given this reflectance capability, we need only write a small, but technical, engine that can handle arbitrary code at run time to implement a seamless inter-system interface.

The C language does not provide such reflectance. There is no information available at run time that can be queried and use to implement an interface dynamically. Instead, we must explicitly compile interfaces and manage them manually. So powerful is the notion of reflectance and what one can do with the information, be it at run-time or once at "compile" time, it is useful to think how we can easily (i.e. with little manual effort) access the large body of existing C/*C++* libraries. In this document and accompanying software, we provide a way to get compile-time reflectance for C/*C++*.

In this module, we are attempting to read information from C source code about the routines and data structures defined within it. We can use this information to, for example, programmatically generate bindings to some of the routines from S, or copy the data structures from C/C++ to S and vice-versa. We might define S classes to mirror the C data structures or C++ classes by enumerating the different fields of the data structure and creating corresponding slots. We can also use this reflectance information to identify global variables and where they are used, and how they can be eliminated. This is useful when considering using the code in threaded applications. In short, we are seeking to generate reflectance information from source code that is often available at run time in other languages and systems such as Java, Python, Objective-C, etc.

This has been done in a variety of different ways for other languages. SWIG (C++), SIP (Qt), Kalyptus and Smoke (Qt) and defs files (Gnome), doxygen for documentation are different approaches to reading information. These rely on variious means to read the information. Some rely on a rich set of regular expressions, others rely on special markup within the source code or modified versions of it. We want to avoid people maintaining two version of the source, and so want to work directly from the original and unique source code. We also want to ensure that the results are "correct" and not intelligent guesses. Thus we must honor all pre-processing inherent in the source language. Nor do we want to rely on difficult to maintain heuristic parsers based on regular expressions that must track extensions to the target language (e.g. GNU extensions to C, extensions to Linux/GNU header files).

Rather than using an external parser and semantic analyzer, we use gcc, the GNU compiler, directly and have it emit information about the contents of the source code it has read. This is now possible via the *-fdump-translation-unit* option for gcc/g++. This option causes the compiler to write its semantic information about the data structures, C routines and call graph to a file. The module we are describing here is a higher-level interface to that information.

The approach we use is to write the translation unit from gcc to a file. Then we process that file using Python and collect the different nodes. Then, within that Python module, we aggregate the low-level nodes into higher-level information such as function declarations, type definitions, variable declarations, etc. All of this processing is done in Python. The `RSPython` package provides a way to invoke these computations from within R as well as get the results as R objects.

By using gcc as the source code processing engine, we are guaranteed to get the same behaviour as the run-time code if that is used as the compiler. Since gcc is widely used, especially for R packages containing native code, the connection to gcc is desirable and not a constraint. As extensions are added to the compiler, we will not need to update our tools to process the source code. At most, we may have to add facilities for processing new node types in the translation unit output. Even this should be rare as the primitives are likely to be sufficient for expressing high-level extensions.

Originally, we adapted another compiler – lcc, the Little C Compiler – to be embedded within S. This was harder to maintain as our changes would need to be applied to subsequent releases of the core distribution. And as header files changed on Linux, the compiler was not usable.

2 Using the Tools

The tools currently provided by this package are sufficient for generating higher-level representations of native code. Depending on the desired task, one will want to operate on these intermediate outputs in different ways. Regardless of the purpose, however, the steps to obtain the higher-level descriptions are the same.

2.1 Generating the Translation Unit

First, we generate the translation unit. We do this by invoking `gcc` or `g++` with the *-fdump-translation-unit* flag. This step cannot be reliably done by this generic module since the pre-processor and other compiler flags will be project-specific and must be specified via a makefile, manually or via some other mechanism. It is easy to add a general rule to a makefile to generate the translation unit file:

```
%.c.tu: %.c
 gcc $(CPPFLAGS) $(CFLAGS) -fdump-translation-unit  -c $<

%.cc.tu: %.cc
 g++ $(CPPFLAGS) $(CXXFLAGS) -fdump-translation-unit  -c $<

%.cpp.tu: %.cpp
 g++ $(CPPFLAGS) $(CXXFLAGS) -fdump-translation-unit  -c $<
```

This can then be used with commands of the form

```
gmake globals.tr
```

The contents of the resulting .tu file will be something like

2.1.1 Nodes

Each node in the translation unit starts with a `@n` at the beginning of a line where n is the number of the node[1]. Each node as a "type", given in the second field of the node entry, such as function_decl for function declaration, identifier_node giving a name of a symbol, and parm_decl identifying a parameter in a function definition. There are numerous other types.

The remaining elements of the node entry are typically name-value pairs given as name: value. The srcp entry identifies the name of the source code file from which the node entry was generated. Even in the case of a single source file, not all srcp entries will be the same. Instead, some will refer to header files that are included within the original source file by the pre-processor. Other nodes will be generated by `gcc`itself and be identified as `<internal>`.

[1] The nodes are not necessarily ordered consecutively, but merely monotonically.

```
@1      function_decl    name: @2       type: @3       srcp: globals.c:23
                         chan: @4       args: @5       extern
@2      identifier_node  strg: branch   lngt: 6
@3      function_type    size: @6       algn: 64       retn: @7
                         prms: @8
@4      function_decl    name: @9       type: @10      srcp: globals.c:17
                         chan: @11      extern
@5      parm_decl        name: @12      type: @13      scpe: @1
                         srcp: globals.c:22            argt: @13
                         size: @14      algn: 32       used: 1
```

Figure 1: An example of the output from a translation unit dump

Not all elements within a node are name-value pairs; some are simply qualifiers or flags. For example, the first entry in figure 1 has the value extern.

2.1.2 Connections between Nodes: The Graph

Many of the entries for a node have values of the form `@n`. These refer to another node within the translation unit. For example, in node *1*, the name element has a value `@2`. To interpret this, we lookup node *2* and we see that this is an identifier_node and its strg element contains that literal giving the name of the function in node *1*. Similarly, the args element in the first node has a value `@5` which refers to node *5*, a *param_decl*. This is the first element of the parameter list for the function. It has a chan entry which points to the next parameter entry (node *14*).

These examples illustrate that the nodes form a graph and we traverse different paths to resolve information such as names, types, definitions, etc. Much of the module for processing the translation unit deals with resolving the nodes to create higher level data structures that aggregate the information into Python class that represent information about an entire symbol. The code is responsible for resolving the nodes and traversing the graph to construct these objects describing the higher-level concepts within the native code.

2.2 Processing the Nodes

The next step is to read the nodes into Python and make them available for further higher-level computations.

3a ⟨ 3a⟩≡

```
% python2
■> import translation
<module 'translation' from 'translation.py'>
■> g = translation.GccTranslationParser("globals.c.tu")
```

Next we read the file and convert each node as we encounter it to a Python object.

3b ⟨ 3a⟩+≡

```
■> g.process()
```

At this point, we have just the nodes in the graph. We have not resolved them in any way and connected them to the different pieces to which they refer. So we now are ready to resolve these. How we do this depends on what information we are interested in.

If we want function definitions so that we can register them with R or build interfaces to them from S, we can resolve the *function_decl* elements that are explicitly defined within the source file (e.g. `globals.c`). The python method *resolveFunctions()* does this for us. Since this is a common task, we have arranged that the nodes which define such functions are stored in the GccTranslationParser instance as it iterates over the nodes.

4a ⟨ 3a⟩+≡
```
■> g.functionDefs
['1', '4', '11']
```

So we know there are three functions/routines defined locally within this source file. We can now create objects that describe these by resolving each of the specified nodes.

4b ⟨ 3a⟩+≡
```
■> f = g.resolveNode('1')
```

The class FunctionDef is used to represent such a (resolved) node.

4c ⟨ 3a⟩+≡
```
■> f
<translation.FunctionDef object at 0x81559d4>
```

We have a stringification method to convert into a human-readable format.

4d ⟨ 3a⟩+≡
```
■> print f
void branch(int a, double * z)
```

Additionally, we can access different pieces of information about the function such as its name or any of the other elements we store.

4e ⟨ 3a⟩+≡
```
■> f.name
'branch'
■> f.__dict__.keys()
['returnType', 'name', 'parameters', 'qualifiers']
```

We can compute the number and type of the parameters from the *parameters* field.

4f ⟨ 3a⟩+≡
```
■> len(f.parameters)
2
■> f.parameters[0].name
'a'
■> print f.parameters[0].type.typeName
'int'
```

Note that we have not resolved the body of the function. For our common purposes, we are more interested in just the signature so that we can interface to it.
We can resolve all the functions in a single call using the convenience function *resolveFunctions()*

4g ⟨ 3a⟩+≡
```
■> g.resolveFunctions()
```

This returns the dictionary of resolved functions. It is also cached within the GccTranslationParser and accessible via the *functions*.

5a ⟨ 3a⟩+≡
```
■> g.functions()
```

3 Other Computations

The un-resolved nodes are stored in the GccTranslationParser and can be retrieved via the *nodes* field. We can iterate over these and perform arbitrary computations.

3.1 Enumerations

For example, suppose we wanted to find and process all enumeration defintions. (We will ignore the fact that we would typically expect these to be within a type definition of the form

5b ⟨ 3a⟩+≡
```
typedef enum {A, B, C} foo;
```

Instead, we will just look at the enumeration definition itself and ignore the mapping to a new type.)

We will use the file `enum.c` in the `examples/` directory as our example code for this.

5c ⟨ 3a⟩+≡
```
g = translation.GccTranslationParser("examples/enum.c.tu")
g.process()
```

We can find all the nodes that define an enumeration using *filter()* and a simple test function. We iterate over the values in the dictionary of *nodes* in the GccTranslationParser and apply a simple function that checks whether the element is an instance of the EnumerationNode class.

5d ⟨ 3a⟩+≡
```
enumNodes =  filter(lambda x: isinstance(x, translation.EnumerationNode), g.nodes.values())
```

Given this collection of enumeration nodes, we need only resolve them. We can use *map()* for this and a simple function that merely resolves each node with respect to the nodes in the GccTranslationParser.

5e ⟨ 3a⟩+≡
```
defs = map(lambda x: x.resolve(g.nodes, g), enumNodes)
```

Now, we can look each of these and extract the elements

5f ⟨ 3a⟩+≡
```
■> print defs[0]
{'Blue': 4, 'Green': 3, 'Red': 1}
■> defs[0].elements.keys()
['Blue', 'Green', 'Red']
■> defs[0].elements['Red']
1
```

3.2 Non-Local Variables

People are quite prone to using non-local or "global" variables in native code to simplify communication between different routines or maintaining state across different calls. For example, some S programmers writing native C code to be accessed via the *.C()* interface compute the result and then return the number of elements it occupies. A second routine is then called from S with the suitably allocated object into which the result is inserted. Generally, global variables are bad. They make code harder to understand as the operations are not local and have side-effects both relative to the position of the code and across time. They prohibit that code from being used within a threaded application, and are generally a nuisance.

We can use our facilities for processin native code to identify global variables. We could use this to provide an interface to retrieve their values into an interpreted language and make the native variable effectively accessible directly from within S or the like. We can also identify where global variables are used and provide guidance for how to remove them. At the extreme of the spectrum, we might be able to rewrite the C code to remove those variables!

We use the `globals.c` for our example. It contains 3 non-local variables: `A`, `x` and `y`. The first two are generally accessible from other code, whereas `y` is accessible only within this file due to its static qualifier. [2]

6a ⟨ 3a⟩+≡
```
varNodes = filter(lambda x: isinstance(x, translation.VarDeclNode), g.nodes.values())
vars = map(lambda x: x.resolve(g.nodes, g), varNodes)
```

To make this simpler and return a named collection of variables, we have provided the method *findGlobalVariableDefs()*.

6b ⟨ 3a⟩+≡
```
vars = g.findGlobalVariableDefs()
```

This returns a dictionary with elements corresponding to the names of the variables.

6c ⟨ 3a⟩+≡
```
■> vars.keys()
['y', 'A', 'x']

■> vars['y'].qualifiers
['static']
```

We can find the initialized value of the variable declarations by following the init element of the *var_decl* element.

4 Mapping Values to S

Suppose we are interested in accessing global variables or return values of routines from C routines directly in S. In other words, we want to be able to transfer C values back to S. If we can do this, we can make C variables directly accessible from the S language (via dynamic variables or object tables).

6d ⟨ 3a⟩+≡
```
p = translation.GccTranslationParser("examples/Smap.c.tu"); p.process()
foo = p.resolveFunctions()['foo']
```

[2] For some reason, this qualifier is not available to us. Perhaps there is an error in the way the translation unit is being generated in `gcc`. If we use `g++`, we get the correct value. Also, this generates information about the body of functions, etc. So in general it is still desirable to enclose C code within an `extern "C" { include "filename.c"}` block and then run `g++`or simply compile with `g++`always and ignore the name mangling. The translation unit information contains both the regular and the mangled name. See `http://gcc.gnu.org/ml/gcc-patches/2002-11/msg00636.html`.

The routine we want to generate can be written as:

7a ⟨ 3a⟩+≡
```
SEXP
R_foo(SEXP s_x, SEXP s_y, SEXP s_i)
{
  SEXP ans;
  Out o;

  int    x = INTEGER_DATA(s_x)[0];
  double y = NUMERIC_DATA(s_y)[0];
  In     i = asIn(s_i);

  o = foo(x, y, i);

  ans = asROut(&o);

  return(ans);
}
```

The mapping of the names is relatively easy to see. The name of the C routine is given as `R_` concatenated with the name of the actual routine (`foo`). Similarly, the names of the parameters for this wrapper routine are the concatenation of `s_` and the names of the parameters of the actual routine.

We declare the return value (`ans`) as a `SEXP` regardless of the return type of the underlying routine. We also declare a variable to store the value of this underlying routine.

The next stanza of the code dereferences the S-values given as arguments and converts them to the C-level values. This dereferences the values using macros from `RSCommon.h` for built-in types and by invocations of specialized routines for mapping S objects to C structures (e.g. `asIn()`).

Next, we invoke the underlying routine and assign the result to the low-level C value. Finally, we convert this value to an S object and return it.

To invoke this function, we provide an S function which a) coerces the arguments to the target data types, and b) invokes the corresponding C wrapper routine.

7b ⟨ 3a⟩+≡
```
foo <-
function(x, y, i)
{
  x <- as.integer(x)
  y <- as.numeric(y)

  .Call("R_foo", x, y, i)
}
```

To generate these two code segments (the C routine and S function), we provide Python functions to generate them. These must know about the built-in types, and the corresponding S coercion types and C accessors.

Given the *FunctionDef* object, *foo*

7c ⟨ 3a⟩+≡
```
foo = p.resolveFunctions()['foo']
```

we can generate the necessary code. We start by defining the name of the S function, which is a simple copy of the C routine, *foo()*. The names of the parameters are also given by the names for the parameters to the C routine.

8a ⟨ 3a⟩+≡
```
str = foo.name + " <-\nfunction("
for p in foo.parameters:
  str += p.name
  if not isLast(p):
    str += ", "

str += ")\n{"

for p in foo.parameters:
 # coerce to the appropriate type if possible.

str += ".Call('R_" + foo.name + "'"
for p in foo.parameters:
  str += ", " + p.name
str += ")\n\n}"
```

The method *asSFunction()* does this. We use the file `SMap.c` in the `examples` directory to illustrate this.

8b ⟨ 3a⟩+≡
```
p = translation.GccTranslationParser("examples/SMap.c.tu"); p.process()
p.resolveFunctions()

print p.functions['foo'].asSFunction()
```

8c ⟨ 3a⟩+≡
```
foo <-
function(x, y, i)
{
   x = x
   y = y
   i = i
   .Call('R_foo', x, y, i, PACKAGE=■)
}
```

The last step is to create the converters for the different data types involved in the routine. In this example, that is for `In` and `Out`.

8d ⟨ 3a⟩+≡
```
print p.types['Out'].toRRoutine()
```

or, equivalently,

8e ⟨ 3a⟩+≡
```
print foo.returnType.toRRoutine()
```

since the return type of `foo()` is `Out`. This gives the following code, corresponding to the different fields of the structure.

9a ⟨ 3a⟩+≡

```
SEXP
asROut(Out val)
{
   SEXP ans = R_NilValue, names;

   PROTECT(names = NEW_CHARACTER(6));
   PROTECT(ans = NEW_LIST(6));

   SET_VECTOR_ELT(ans, 0, asRint(i));
   SET_STRING_ELT(names, 0, COPY_TO_USER_STRING(i));

   SET_VECTOR_ELT(ans, 1, asRdouble(d));
   SET_STRING_ELT(names, 1, COPY_TO_USER_STRING(d));

   SET_VECTOR_ELT(ans, 2, asRlong(l));
   SET_STRING_ELT(names, 2, COPY_TO_USER_STRING(l));

   SET_VECTOR_ELT(ans, 3, asRshort(s));
   SET_STRING_ELT(names, 3, COPY_TO_USER_STRING(s));

   SET_VECTOR_ELT(ans, 4, asRfloat(f));
   SET_STRING_ELT(names, 4, COPY_TO_USER_STRING(f));

   SET_VECTOR_ELT(ans, 5, asRstring(string));
   SET_STRING_ELT(names, 5, COPY_TO_USER_STRING(string));


   SET_NAMES(ans, names);
   UNPROTECT(2);
   return(ans);

}
```

If we wanted to use S4 classes, we could have this routine assign the values of the C structure to the different slots of the S class. For example, the string slot would be set as

9b ⟨ 3a⟩+≡

```
SET_SLOT(ans, Rf_install("string"), COPY_TO_USER_STRING(string));
```

We would have to declare the class in S before it is used. Also, the routine would have to fetch the class and create an instance. This is done with the C code

9c ⟨ 3a⟩+≡

```
klass = MAKE_CLASS("Out");
ans = NEW(klass);
```

We provide the basic conversion routines `asRint()`, etc. since these are commonly used and relate to the know primitives in both C and S. Putting this all together, to interface to the `bar()` routine in `SMap.c`, we use the following code. In Python, we parse the translation unit file and generate the code for bar. This output code consists of an S function and two C routines, `asROut()` and `R_bar()`.

10a ⟨ 3a⟩+≡

```
p = translation.GccTranslationParser("examples/SMap.c.tu"); p.process()
p.resolveFunctions()

bar = p.functions['bar']

f = open("SMap_S.c", 'w')
f.write('#include "SConverters.h"\n')
f.write('#include "SMap.h"\n')

f.write(bar.asCRoutine())
f.write("\n")
f.write(bar.returnType.toRRoutine())
f.write("\n")
f.close()

f = open("SMap_S.S", 'w')
f.write(bar.asSFunction())
f.write("\n")
f.close()
```

Now we can compile this code along with the `SConverters.c` file [3].

10b ⟨ 3a⟩+≡

```
R CMD SHLIB -o SMap.so SMap.c SMap_S.c SConveters.c
```

For S-Plus, one would use a "chapter" and compile using the automatically generated makefile.

And now we are ready to run S and load this interface.

10c ⟨ 3a⟩+≡

```
dyn.load("SMap.so")
source("SMap_S.S")
```

[3] In the future, we will allow the shared library/DLL in one package to link against code in another using *frameworks*. The code in `SConverters.c` will then be available in its own framework.

And finally, we can call this routine from within S

11a ⟨ 3a⟩+≡

```
bar(10,11)
$i
[1] 11

$d
[1] 13

$l
[1] 3

$s
[1] -1

$f
[1] 7.3

$string
[1] "xyz"
```

5 Working from S: RSPython

Of course, since the generation of the code and writing to a file is a common and repetitive task, we can put this into a function. There are a variety of conventions we can use which can greatly simplify the process. Given the compiler flags (CFLAGS), we can generate the .tu file from a given C/C++ file. Similarly, we can compute the list of files simply by listing the contents of the specified directory. Then we can apply the basic procedure to each file. The user may want to exclude certain routines from the generated interfaces, or explicitly select which ones to process. Additionally, the inclusion of header files (e.g. `SMap.h`) in the output file (`SMap_S.c`) can be inferred/guessed from the translation unit since the location of the definitions of the structures, etc. is given in the nodes. If this inference is not sufficient, the user may need to supply their own "preamble" for the C file.

Since we can provide sensible defaults for most situations, but want to allow the user override these in dynamic and programmable ways, we will want to make this as easy for the user as possible. While this module is language neutral, we are focussed on S. And so we will want to allow the S user to easily customize the computations. To do this, We can do this by using the `RSPython` package. We can initiate the processing from R by creating an instance of the GccTranslationParser class using *.PythonNew()*. Then, we can invoke the same sequence of methods on this object from within R as we did in Python. When we have generated the relevant information (e.g. the FunctionDef objects via the *resolveFunctions()* method), we can transfer the objects back to R and work with them there.

First we ensure that we can find the `translation.py` code. If this is installed as a regular Python module, all is well. Otherwise, ensure that the directory in which it resides is included in the PYTHONPATH variable.

11b ⟨ 3a⟩+≡

```
setenv PYTHONPATH "."
```

Now we enter R and create an instance of the GccTranslationParser with the `SMap.c.tu` to process.

12a ⟨ 3a⟩+≡
```
importPythonModule("translation")
p = .PythonNew("GccTranslationParser", "examples/SMap.c.tu", .module="translation")
```

Then we process the nodes and resolve the functions.

12b ⟨ 3a⟩+≡
```
p$process()
funs = p$resolveFunctions()
#XXX
```

At this point, we have the function definitions in R and can operate on them as we please. The Python parser is still around and we can perform additional computations there if we want, or do them in R. We can control how code is output using S functions or regular S commands.

6 References to C Structures and Classes

In some cases, we may want to hold a reference to a C-level object rather than copying its contents. If we know that the C object will continue to exist and be valid for the lifetime of the S object used to refer to it, then having a reference allows us to share the object with other code, both S and C.

Given an S reference to a structure or class instance, we will then want to be able to access its fields. We can do this by overloading the $ or @ operator in S.

12c ⟨ 3a⟩+≡
```
ref = getInRef()
ref$i
ref$d
```

We can implement this style of functionality in a variety of different ways. The critical aspect is that we must know the names of the available fields and their types. We can write a method for these operators given the names of these fields that first checks that the request is for a valid field. Then, it invokes a C routine that dereferences the C-level object, extracts the relevant field and converts it to R.

12d ⟨ 3a⟩+≡
```
"$.InRef" <-
  function(x, name)
{
  idx <- match(name, c("i", "d"))
  if(is.na(idx))
   stop("No such field in In structure")

  .Call("R_getInField", x, idx)
}
```

The C routine `R_getInField()` then looks something like the following.

13a ⟨ 3a⟩+≡
```
SEXP
R_getInField(SEXP ref, SEXP fieldIndex)
{
 SEXP ans;
 In *in;
 in = (In *) DEREFERENCE(ref);

 switch(fieldIndex) {
   case 1: /* i */
     ans = asRint(in->i);
   break;
   case 2: /* d */
     ans = asRdouble(in->i);
   break;
   default:
      PROBLEM "This is not to be called directly and must have an argument between 1 and 2"
      ERROR;
 }

 return(ans);
}
```

Clearly, we can generate this code from the information in the StructDef object. And we can similarly define assignment methods by overloading the $<- operator. We can use S4 or S3 methods. If we use S4 classes, we might define the InRef class something like

13b ⟨ 3a⟩+≡
```
setClass("InRef", representation("CReference"))
```

We would define CReference as a general class that holds references to C values. This might also have a field indicating the number of indirections to the actual object, i.e. the number of $*$ in the declaration. The reference class can then be used to refer to a pointer to an object, the object itself, or a pointer to a pointer to an object of this type, and so on. The `DEREFERENCE()` macro would be able to understand how to interpret the memory address held in the reference object based on this count.

We can use class/static fields to store information about the particular structure or class being referenced. For example, we might add its name (In), its size (number of bytes), the name of the source file or dynamically loadable library in which its definition can be found. The size is useful if we want to allocate an instance of the type directly. For example, we might have a generic allocation routine that takes the number of bytes and returns a CReference object which can be used to construct a more specific reference.

13c ⟨ 3a⟩+≡
```
new("InRef", .Call("R_mallocRef", as.integer(InRef@size)))
```

We could alternatively create a regular constructor function which would take the fields as arguments and both allocate and initialize the structure.

13d ⟨ 3a⟩+≡
```
In <-
function(i = 0, d = 0)
{
 .Call("R_newInRef", as.integer(i), as.numeric(d))
}
```

And the corresponding C routine would be of the form

14a ⟨ 3a⟩+≡
```
SEXP
R_newInRef(SEXP s_i, SEXP s_d)
{
  SEXP klass, ans;

  In *in;
  in = (In *) malloc(sizeof(In));

  in->i = INTEGER_DATA(s_i)[0];
  in->d = NUMERIC_DATA(s_d)[0];

  klass = MAKE_CLASS(Rf_install("InRef"));
  PROTECT(ans = NEW(klass));
  SET_SLOT(ans, "ref", R_MakeExternalPtr((void *) in, Rf_install("In"), R_NilValue);
  UNPROTECT(1);
  return(ans);
}
```

Again, we see that we can easily generate this code programmatically as we did when creating an interface to a C routine that involved a structure.

7 Registration Tables

R provides a facility by which one can explicitly identify C routines in a DLL/shared library that can be called from R via the *.C()*, *.Fortran()*, *.Call()* and *.External()* interfaces. This provides a safer and more controlled way to locate symbols (allowing also for user-level aliases), and also allows additional information to be specified and made available to R and the user about the number and type of arguments that are expected. This information is important if we wish to make R secure when run as an embedded interpreter in a server.

The additional benefits registration provides are coupled with extra work the developer of the native code must perform to use it. She must create the table describing the different routines. This is a task that we would like to automate. It involves identifying the routines that are or can be called and then determining the number and types of arguments. Given our ability to process C code, we can gather this information relatively easily for C and *C++*(i.e. non-Fortran code).

The file `Rregistration.py` in the distribution provides a script that one can invoke to create the registration information. First we create the .tu files.

14b ⟨ 3a⟩+≡
```
cd examples
make CFLAGS=-I‘R RHOME‘/include Scode.c.tu Scode2.c.tu
```

Then we pass these two files as arguments to the script and have it create two files: `init.h` and `init.c`. These contain the declarations of the routines and the registration information respectively.

14c ⟨ 3a⟩+≡
```
Rregistration.py -p MyPackage examples/Scode.c.tu examples/Scode2.c.tu
```

These files (`init.*`) can then be compiled into the package and the initialization routine will be automatically invoked by R when the library is loaded and will register the routines.

15a ⟨ 3a⟩+≡
```
make tus
<wherever>/Rregistration.py *.tu
```

We can change the location where the code gets written using the -o and -h arguments. For example, suppose we do not want the header file and we want the registration tables and initialization code in a file named `MyPackage_init.c`, we would give the command

15b ⟨ 3a⟩+≡
```
<wherever>/Rregistration.py -o MyPackage_init.c -h /dev/null *.tu
```

If we run this script in the `ctest/` directory in the `examples/` directory, we will end up registering 2 routines that are not of interest. (This is due to the fact that we allow `floats` to be used.) `nscor1()` and `nscor2()` are not intended to be called directly from S. Some of them are intended for internal use only but happen to have signatures that make them valid for invocation from R.

We have four approaches for removing these from the registration table. The manual mechanism is of course to edit the resulting file. Each time it is regenerated, one has to apply these changes. The other mechanism is to run the code within the python script interactively. After using the GccRNativeDeclarations class to process the routines and put them into the different categories, one can the edit this table and discard the ones that are not to be registered. To make this easier, we allow a command line argument to specify the names of the routines to omit or skip. This is -s and given as a comma-separated list of routine names, e.g.

15c ⟨ 3a⟩+≡
```
python <wherever>/Rregistration.py -s "nscor1, nscor2" *.tu
```

The spaces between the names are optional.

As well as specifying which routines to skip, we also allow the user to specify a list of those to include. The -i flag is used for this in the same way as the -s flag.

The fourth approach is to examine the S code in a package and determine which routines are actually called within that. This assumes that the routines are not to be called directly by the user (e.g. via a *.C()* or *.Call()*) which is very much the advisable approach since arguments must be coerced to the expected types. The S function *getRoutineRefs()* (based on Kurt Hornik's *checkFF()*) in the `Slcc` package does exactly this. It iterates over all the code in a package or file and cumulates the names of the routines called by the different interfaces (C, Call, Fortran, External). This information can then be used to compute the names of the routines that are to be included by the registration table generator, GccRNativeDeclaration, and specified by the -s flag.

8 Call Graph

If we are given or find source code, we are typically interested in how we can use it and generally understanding which are the top-level entry points and how the routines are related. There are a variety of tools for visualizing the relationships between the routines and which calls which. We can assist in this endeavor by constructing the call graphs using this information from `gcc/g++`. We can also use graphviz[4] to visualize the results of the call graph.

No code currently for this.

9 Free Variables

As we mentioned earlier, there are occassions where we are interested in identifying the non-local variables that are used within routines. We might try to group these to identify which ones are used in common routines. Given this information, we can partition the global variables into groups based on the routines in which they are shared. Then we can remove them by aggregating them into a structure and passing a point to this structure as an argument to these routines. The author can then remove the remaining global variables in a context-specific manner.

We identify the non-local variables first. These are identified by node index. Then we look through the bodies of the different functions to determine whether each is referenced by that function.

No code currently for this.

10 Alternatives

SWIG is the most well-known of the tools that construct bindings or interfaces to native code. Developed originally by David Beazley, is an interface generator that uses its own C/C++ parser to understand the native code. It is extensible and could be enhanced to generate bindings for the S language and its dialects.

CPPX[5] is another implementation of this, using GXL as the output dialect for representing the translation unit nodes. GCC-XML[6] is an extension to `gcc` that outputs the translation unit in a different and higher-level XML dialect. In both cases, we can use the `XML` package to read the information into S and then process it locally. Both of these require users to apply patches to the `gcc`source and re-compile from source. This makes it unattractive to the casual user.

SIP is a tool for generating bindings for *C++* classes. It was originally written to It operates on marked-up header files. This requires modifying the original source code and so some investment of time by the user. As a result, SIP is not very convenient for rapid and casual integration of code, but better suited for use when one has decided to create customized bindings to a particular library.

11 References

1. GCC Translation Unit Information `http://www.sunsite.ualberta.ca/Documentation/Gnu/gcc-3.0.2/html_node/gcc_187.html`
2. SWIG `http://www.swig.org`
3. SIP `http://www.riverbankcomputing.co.uk/sip/index.php`

⟨ 3a⟩

[4]See URL `www.graphviz.org`.
[5]See URL `http://swag.uwaterloo.ca/ cppx/`.
[6]See URL `http://www.gccxml.org`.