Using the RGCCTranslationUnit package.

Duncan Temple Lang

Table of Contents

Installing the package 1
Creating the translation unit file 2
.... 2
Resolving the types 5
    Duplication and the DefinitionContainer 7
Creating Interfaces to Native Code 8
Manipulating the Source Code 11
    Call Graphs 11
    Determining output parameters 12
Global Variables 13
    Non-constant Global Variables 14
    Other "constants" - enums, defines 16
Customizing the Code Generation: Type Maps 16
Dependencies Files 19

This document provides an introduction to using the RGCCTranslationUnit package. It is a work in progress and the API will probably change slightly. However, the basic facilities will remain approximately the same. Most likely, higher level facilities will be added that combine the calls to some of the different facilities into a single action.

Installing the package

Firstly, you will have to install the RGCCTranslationUnit package. And if you want to read a translation unit file, you will need RSPerl. One can use the package without needing Perl if you are working with objects that were stored from a previous R session on a different machine. That is why the package does not explicitly depend on RSPerl. When installing RSPerl, one should instruct it to provide dynamic loading support for the Perl modules named Socket, IO, Fcntl, POSIX and Storable. Do this with

```
                                                                  R
R CMD INSTALL --configure-args-'IO Socket Fcntl POSIX Storable' RSPerl
```

When you load RGCCTranslationUnit and try to parse a tu file using parseTU, you may get errors or even an entire crash of R (which relates to not having the dynamic loading for the modules mentioned above). See the FAQ.html file on the Web site or in the package for some suggested.

We have done almost all our work and testing with .tu files generated by GCC version 3.2.2. Using tu files created with version 4.1.0 of GCC will generate lots of output when using parseTU(). These are warnings about particular elements not processed by the Perl parser module. These will be fixed soon. The output for the tu files is different from the versions of GCC, but when we process them using our tools to higher level objects, things seem to be the same for the code we have tested. More details are necessary. One thing to note is that version 3.2.2 generates files named foo.c.tu and foo.cpp.tu, i.e. by appending the .tu to the name of input source file. Version 4.1.0 appends .t00.tu, e.g. foo.c.t00.tu.

Creating the translation unit file

If you want to read the contents of source code from C/C++ or header files, you will need to generate the translation unit from those files. You can generate the .tu file from each .c/.cpp/.cc/.h file or, if you want, generate one large tu file for all of them. To do the latter, create a new C/C++ file whose content simply consists of

R
```
#include "firstFile.c"
#include "secondFile.c"
```
where you use the file names of interest to you.

Now that we have the relevant source file, say foo.c, we generate the .tu file using

R
```
gcc -fdump-translation-unit  -c foo.c -o /dev/null
```
Note that I have told the compiler to write the object file (.o) to /dev/null, i.e. not to create it. It doesn't matter if you do. You will need to add any additional compilation flags such as include directories with -I, and defines with -D. I typically put a rule in a GNUmakefile to generate the .tu file as these flags are already present for doing the regular compiling.

Even if you get an error from the compiler about the content of the source code, the compiler may still have generated the .tu file. If the error is a syntax problem, the .tu file will not be generated. But if the compiler has read the entire file and is only giving errors about the validity of the meaning of the C/C++ code, then it will have dumped the .tu file before it reached this processing stage.

Having created the .tu file, we only need to revisit this step if the source code changes. And if we have the appropriate dependencies in our makefile, the .tu file will only be generated when these have changed and you call make.

Now that we have the .tu file for input to our processing, we start an R session and load the RGCCTranslationUnit package.

R
```
library(RGCCTranslation)
```
Next, we use *`parseTU`*() to read the .tu file:

R
```
p = parseTU("foo.c.tu")
```
or foo.c.t00.tu if using a more recent version of GCC.

The value in **p** is a strange thing. It is an R object that actually identifies a Perl object. If you call *`class`*() on this object, you will see that it is a reference to a Perl object, in fact a Perl array (ordered list) and, most specifically, is a *GCC::TranslationUnit::Parser*

R
```
class(p)
```

```
[1] "GCC::TranslationUnit::Parser" "PerlArrayReference"
[3] "PerlReference"
```

The elements in this Perl array are the nodes in the tu graph. And there will be lots of them:

R
```
.PerlLength(p)
```

You can access the elements by position, e.g.

R
```
p[2]
```

Note that the first one is a "dummy" node and of no interest to us.

For those of you who are unfortunate enough to be in any way familiar with the format of the .tu file, you can use the node ids (i.e. the number after the @123) to identify the node

R
```
p[["123"]]
```

Note that all the nodes are also R objects that refer to Perl objects. And all of them have classes indicating the type of node in the tree, e.g. GCC::Node::namespace_decl, GCC::Node::record_type, GCC::Node::function_decl. Most of the nodes are actual hash tables in Perl, i.e. like named lists in R. You can find out what elements a node contains using, e.g.

R
```
names(p[[3]])
```

If it helps to know, the names come from fields in the .tu file for that node. Those "loose" values such as const or volatile are accessible through method calls on the node, e.g `p[[3]]$quals()` or via the R function *`getQualifiers`*() . If you are not familiar with the .tu file format, don't worry about all the details; you can just think about what they might mean. For the most part, the higher-level functions in the R package will remove the need for you to know anything about the nodes.

Before we proceed, we should note that if you are working with C code and not C++ code, but chose wisely to use the g++ to create the .tu file, then you should tell the parser that it is really C code. We do this in R with

R
```
p = setLanguage(p, "C")
```

or alternatively, when reading the TU file we can specify the language

R
```
p = parseTU("foo.c.tu", "C")
```

This does not affect the Perl object, just the R-side of things in further processing.

These node references are intelligible, but still somewhat low-level concepts. And certainly we don't want to be dealing with the nodes of the graph and following the paths on this graph to work with routines, global variables, data structure definitions and so on. So we use the higher-level R functions that do this for us.

Typically, we will be interested in the routines[1]defined within the native source code. We can use the function *`getRoutines`*() to get a list of the nodes that are native routine declarations.

[1]I use the term routine to refer to what some call a function in native code. We use function to refer to R language functions. Unfortunately, the Perl parser and tu file refer to GCC::Node::function_decl to confuse matters.

R
```
routines = getRoutines(p)
```

getRoutines() provides a brief description about the routine declarations. It identifies the node by it index and name, and provides the identity of the nodes for the return type and the parameters of the routine.

We do have the names of the routines accessible via

R
```
names(routines)
```

If you do this on your file, you will notice that there a lot more than you might expect. And you may not recognize all the names, but some will be familiar to you if you are a C programmer. Names like fprintf, memcpy, strchr and so on are "system" routines, i.e. those defined in header files provided by your operating system or compiler. They are present because of included header files, e.g. `#include "filename"` in your original source code. The compiler has dumped everything it could see at the code level so that we can make sense of it.

We usually don't want to deal with all of these routines but rather limit ourselves to those in our source code files. *getRoutines*() has a *files* parameter that can be given "names" of files which are used to filter the function declaration nodes returned. Only the function declarations whose source attribute corresponds to an entry in this vector will be returned.

So

R
```
routines = getRoutines(p, "msa.c")
```

will give those routines defined in msa.c. And it will (currently) return more quickly than with no filter.

There is a problem with giving the file names on which to filter. Specifically, the compiler only gives us the file name and not the entire path to the file. Thus, it is impossible for us to distinguish between files with the same name but in different directories. For example, suppose we have a local source code file named time.h. Unfortunately, there is a system header file in sys/ also named time.h. We include time.h directly using

R
```
#include "time.h"
```

and it is found locally by the compiler. Even though we do not include the system time.h and there is no ambiguity, another system header file might have a line

R
```
#include <sys/time.h>
```

and then we have introduced two different files named time.h. And only the file name will appear in the code. So we sometimes have to then filter the set of nodes returned by *getRoutines*() further. But we can do this in R using the usual subsetting operators on the returned list since we have names on the elements.

Example 1.

There are tools in the package to help differentiate between code from files with the same name. See Dependencies Files

By the way, note that there is a difference between a declaration and a definition. The declaration node may have a reference to the actual definition of the routine and on to its body. But if you used the regular C compiler (gcc) and not g++ or if you are working from header files rather than the complete source code, then you won't have the definition but just the declaration. All the nodes returned by *getRoutines*() are function declarations, and some may have a path to the body.

There are functions for finding other types of nodes. To find nodes corresponding to "global" variables within the source code, we have *getGlobalVariables*() . To find enumerated constants (enum) definitions, we use *getEnumerations*() . And to find declarations of data structures, e.g. structs, unions, typedefs, the function *getDataStructures*() searches the entire collection of nodes. For working with C++ code, we can find the class definitions using *getClassNodes*() . (This is named as such to avoid any idea that it is similar to *getClass*() in R.) Like the *getRoutines*() function, each of these accepts a *files* argument to filter the nodes based on the source attribute.

These functions do not return as much information directly as *getRoutines*() . *getEnumerations*() and *getGlobalVariables*() return just the index of each enumeration declaration node. *getClassNodes*() returns the name/id and the index of the nodes. *getDataStructures*() returns the node object directly, i.e. the *PerlReference* object. Regardless of precisely what these functions return, they each give us information about how to identify the nodes of interest. And what we want to do is move from dealing with nodes to R-level data structures that describe high-level entities within our native code. To do this, we only need the starting or top-level node for that entity, and so essentially all of these functions are returning us that information, i.e. the index of the node of interest. We go from this node to the R-level data structure by resolving the node and following all its references through the graph.

Resolving the types

So far, we have seen how to read the node graph into R and find the nodes corresponding to different high-level entities in the source code, such as routines, global variables, data structures and classes. Each of the functions returns the identity of the nodes corresponding to the entity in the source code. The identity is given by the node id or name, e.g. the "123" in "@123", and also its position in the Perl array. (The id and position are related as the position is the integer value of the id less 1.)

Having these node identifiers is fine, but it is not what we want to work with. We want an R object that describes a routine completely or gives us the fields in a struct. To do this for a given node, we need to traverse the graph starting at that node and collect all the different pieces. You can do this yourself, but it is tedious and somewhat complicated. The package provides a function *resolveType*() which attempts to do this for you.

We start with a node, say the first one from

R
```
routines = getRoutines(p, "msa")
```

We can call *resolveType*() giving it the first routine to resolve along with the graph or array of nodes which will be used to follow references to other nodes. So

R
```
type = resolveType(routines[[1]], p)
```

gives us an R object of class *ResolvedNativeRoutine*. This is different from the *NativeRoutineDescription* we had in `routines[[1]]` because this now contains all the information about the return and parameter types. And importantly, these type objects are entirely within R and do not need to refer back to the graph/tu parser. [2] So we have resolved the type into a single, self-contained R object without references to other nodes and types defined elsewhere. This is the computational model in R - no references. And this is convenient but can also make some computations on graphs complicated.

Let's look at the `ResolvedNativeRoutine` object. It (currently) is an S3-style object. It has the same elements as the `NativeRoutineDescription`, i.e. the name, returnType, parameters and the INDEX of the defining node should we want to return to it. The name is

R
```
type$name

[1] "msa_reverse_data_segment"
```

The names of the parameters are

R
```
names(type$parameters)

[1] "data"  "start" "end"
```

So far this is the same as if we hadn't resolved the `routines[[1]]` object. However, looking at the returnType field we see

R
```
class(type$returnType)

[1] "voidType"
attr(,"package")
[1] "RGCCTranslationUnit"
```

This tells us that there is no return value from this routine, i.e. it is declared as a void. Note, in the `NativeRoutineDescription`, this was the PerlReference object to the Perl GCC::Node::void_type object. If there were a non-trivial return type, this would have been resolved and we would have a description of that type, e.g. an unsigned integer, a struct, etc..

What about the parameters? Each parameter is also an S3-style list object that has fields giving the name of the parameter (id), the type and the default value, and the node in the parse graph where it was defined. We can find the class of the type of each parameter with

R
```
sapply(type$parameters, function(x) class(x$type))

         data         start          end
"PointerType"     "intType"     "intType"
```

This tells us that the first parameter is a pointer and the two others (start and end) are integers. We have to look further to see what type the pointer points to and whether the start and end parameters are simple int

[2] It is possible that there are PendingType objects within these so. These arise in recursive definitions when one type is being defined and refers to itself or another type that is also being defined. Such objects know how to resolve themselves when used, at least within the same R session. But you will not have to return to the tu parser and resolve them explicitly.

values or more complex such as unsigned int, etc. All this information is in the R objects within the type field of the parameters. It is available to you to do whatever you want with it. However, there are functions in the package that will do some of the things you might want to do, such as creating an R interface to the routine (see *createMethodBinding*()).

We can resolve any node with *resolveType*() and indeed *resolveType*() does this itself as it processes the nodes recursively. For example, we can find all the anonymous or unnamed/typedef'ed enumeration types by looking for those enumerations that have strange, invalid C names such as "._digitdigit..." which is how gcc/g++ identifies them to us. So we can do this using R functions such as *grep*() to find the enumerations with these odd names, such as

R
```
ee = getEnumerations(p)
grep("\\._[0-9]+", names(ee), value = TRUE)
```

Then we can resolve these and find the mapping between the symbolic names and integer values. We might be more interested in the named enumerations in our files

R
```
e = getEnumerations(p, c("msa", "hmm"))
hmm_mode = resolveType(ee[["hmm_mode"]], p)
```

This is an object of (S4) class *EnumerationDefinition*. The key slot is **values** which contains a named integer vector giving the definition of the enumeration elements. In our **hmm_mode**, we have

R
```
hmm_mode@values

 VITERBI  FORWARD BACKWARD
       0        1        2
```

And we can do the same for the other nodes from the data structures and classes. The key thing is that we give a node or node identifier (node name or position) along with the parser/graph/node array itself. For C++ classes, resolving the class finds the fields and the name of the class as well as the usual qualifiers such as scope. It also includes the names of the base class(es) and their node identifiers and the resolved methods for this class alone. It does not include the inherited methods, but these can be retrieved by resolving the ancestor class nodes.

resolveType() uses S4 methods to know what to do for each type of node and will create an R object that describes what it collects on its traverse of the graph. This makes it extensible. Hopefully, the objects that are returned contain all the information we need when processing the code in R. If not, please let me know. And also, it is typically possible to recover more specific information using the object returned as the identities of the sub-nodes are typically included in the result.

Duplication and the DefinitionContainer

We can use *lapply*() to resolve a list of routines, e.g. returned from *getRoutines*() . If you think about this, there is likely to be a lot of redundant computations involved. If two routines refer to the same type, the *resolveType*() will end up visiting that type node twice (at least). By design, we will end up with separate copies of the resolved type in each resolved routine, but we do not want to do the processing multiple times. To avoid this, we pass a third argument to *resolveType*() and provide a *DefinitionContainer* object which is used to manage the resolved types. For the most part, you will never be concerned with how this works. It is a good idea to create one and assign it to an R variable and then pass it to *resolveType*() whenever you call it. But it is important to remember that one cannot use a *DefinitionContainer* with a different tu parser/graph/node array. The *DefinitionContainer* uses the ids of the nodes in the tu graph to manage the types. So if it is used with a different set of nodes, there will be a great deal of confusion!

The *DefinitionContainer* acts as a broker for *resolveType*() . When *resolveType*() is asked to resolve a node, it asks the container whether it has already been resolved. If so, it retrieves that and simply returns the previously computed value. If not, it asks whether that node is currently being resolved, i.e. by a higher-level (in the call stack sens) call to *resolveType*() . If so, it returns the *PendingType* for that node which is essentially a promise to give you the type later. Otherwise, it goes ahead and resolves the node and then tells the container the result so that it will be available for future calls.

The *DefinitionContainer* is simply an R hash table or environment and uses hashing to make lookups fast since there are typically a lot of nodes.

Creating Interfaces to Native Code

One of the purporses of the RGCCTranslationUnit package is to be able to create interfaces to C/C++ code for R. This involves creating R functions that call C/C++ routines and methods which in turn call the routines and methods in the original C/C++ code. So we create both R and C/C++ wrapper code. We also create R classes to hold references to C and C++ structures and classes. And we also create R classes that mirror the definitions of C structures and unions so that we can copy objects to and from C and R. This allows us to have objects that persist across R sessions and also allows us to copy objects which may be deleted in the C code. Since we are creating new C/C++ routines that we call from R, we also need to create registration information for these and add them to a NAMESPACE file so we can access them from within R properly. And finally, for each C++ class, we want to define a new C++ class that allows the R user to extend that class and implement methods using R functions.

There are various functions in the RGCCTranslationUnit package to achieve these different aspects of generating code. Let's start with the case that we have some C code, e.g. the msa.c file. We assume that we have created and read the tu file describing the code into R.

Now, we will focus on the routines. We get the routines defined in the msa.c file with

R
```
r = getRoutines(p, "msa.c")
```
The next step is to resolve these with

R
```
types = DefinitionContainer(p)
msaRoutines = resolveType(r, p, types)
```
So the next step is to generate code for these in both R and C. *createMethodBinding*() takes one of these resolved routines and generates R and C wrapper code for it and provides additional information that can be used to register the code. We loop over the routines and call this function with

R
```
bindings = lapply(msaRoutines, createMethodBinding)
```

And now we can write the R and native code to different files. We use *writeCode*() to output the code for either language as this understands the structure of the **bindings** object. To write the R code, we give the command

```
R
writeCode(bindings, "R", file = "msa.R")
```

(The "R" here indicates that we want the R code, not the native C/C++ code.) We can pass a connection object as the *file* argument rather than a simple file name. This allows us to pass an already open connection so that the content can be appended to that file.

We use the same approach to create the C source code file. However, in this case, we have to arrange to add top-level material that includes the appropriate header files. We can do this in two ways. The first is to open the file and write the material ourselves and then pass *writeCode*() the already open connection.

```
R
con = file("/tmp/Rmsa.c", "w")
  writeIncludes(c("<msa.h>", "<Rinternals.h>", "<Rdefines.h>", '"RConverters.h"'),
  writeCode(bindings, "native", file = con)
close(con)
```

For this case, we only have include files to specify and we can pass this vector to *writeCode*() via the includes argument of **...**:

```
R
writeCode(bindings, "native", file = con,
             includes = c("<msa.h>", "<Rinternals.h>", "<Rdefines.h>", '"RConverte
```

At this point, you should hopefully be able to read the R code back into R using *source*() . Additionally, you might be able to compile the C code with `R CMD COMPILE Rmsa.c` assuming you have the relevant compilation flags set in the Makevars file or in the PKG_CPPFLAGS environment variable. And you can create a DLL/Shared Object with `R CMD SHLIB Rmsa.c` and then load that into R with *dyn.load*() . Of course, we can put this code directly into a package structure and INSTALL and then load it using *library*() .

Unfortuantely, the code is not likely to usable without some additional steps. The C code may not compile cleanly or link at all because of references in the generated code to routines that don't exist. Firstly, we need to make certain we have a copy of the RConverters.h file that we added as an include when generating the C code. And we will want its sibling file RConverters.c also. In the future, we will simply add a dependency in the DESCRIPTION file of our new package to the *RAutoGenRunTime* package. But for now, simply copy those files into the same directory as the newly generated Rmsa.c file. You can find these fines in the *RAutoGenRunTime* source distribution.

But we have further issues that are specific to the code we generated. For example, the wrapper for the routine *msa_new_from_file* will have a call to the routine *R_copy_MSA_to_R*. This performs a "deep" copy of the MSA reference to create an R object. Firstly, we have to create that routine and add it to Rmsa.c or another file that we will compile and link with Rmsa.o. Additionally, we have to create the R class definitions for MSA and MSARef. Of course, we don't have to do this manually, but rather the RGCCTranslationUnit package has code to do this. We get the definition for the MSA data structure. We can do this using *getDataStructures*() and finding the node associated with its definition and then resolving that node:

```
R
```

```
dataStructs = getDataStructures(p, "msa")
MSA = resolveType(dataStructs[["MSA"]], p)
```

Of course, we don't need to do this as it must have already been resolved when we resolved the routines. After all, the msa_new_from_file routine must have resolved it as it is (part of) the return type. If we look in the DefintionContainer that we used to manage the resolved types, we will find it there:

R
```
types$MSA
```

and it is fully resolved.

Now that we have the defintion of this type, we can generate code to define R classes and copy it to and from C, and provide element-wise accessor to the fields. We use the function *generateStructInterface*() to do all of these things. This calls *defineStructClass*() , *createCopyStruct*() , and *createRFieldAccessors*() for each of the different tasks. We pass only the resolved type to the *generateStructInterface*() as in

R
```
msa_iface = generateStructInterface(types$MSA)
```

This is an object of class *CStructInterface* and has information about the class definitions (in classDefs), the generic functions for the $ and $<- methods for the reference class, i.e. MSARef, coercion methods for converting between C and R versions of the type, i.e from MSA to MSARef and from MSARef to MSA. And of course it has the C routines that perform the copying and provide access to the fields.

We can write the different elements of the *CStructInterface*() to a file (or any connection) using *writeCode*() . There are methods for this generic function for handling this class and its sub-elements. So

R
```
writeCode(msa_iface, "r", "/tmp/msa.R")
writeCode(msa_iface, "native", "/tmp/Rmsa.c")
```

will create the two files with the generated code.

This defines two classes, a reference class that holds a pointer to an object in C/C++ and an equivalent R-only class which has slots parallel to the fields. There is a constructor function for creating a reference object, e.g.

R
```
ref = new_MSA()
```

We have access to fields in a reference object using the $ method, e.g. `ref$nseqs` and of course to the slots in the R-based object `obj@nseqs`. *names*() called with a reference object gives back the field names. And we can set fields in the reference object with

R
```
ref$nseqs = 100
```

Note that the coercion is done for us. So in this way, the reference appears like a list in R, but it is important to remember that changes to the underlying C-level object will be seen in subsequent computations in R. We can use *as*() to copy from one representation to the other with

R
```
obj = as(ref, "MSA")
as(obj, "MSARef")
```

How do we know which data types are being used in the routines and therfore for which we need to generate these class definitions and routines? The simplest method is to resolve the routines you are interested in and then look at all the data structure elements in the `DefinitionContainer` you passed to the *resolveType*() call. This will contain *all* the types that were encountered and needed to be resolved. Therefore, these are the types you will need to have code to support.

And last, but not least, we need to generate code to handle global variables and enumerated constants.

Manipulating the Source Code

In addition to being able to generate bindings, the RGCCTranslationUnit package provides programmatic access to manipulating very detailed information about C or C++ code. If you have generated the tu files using the C++ compiler, g++, and are working with the complete source code (i.e. .c) rather than the header file and the declarations of routines, the function declarations in the tu file will have a reference to their bodies. This allows us to process the code in the routine and not just its declaration about parameters and return type. There are numerous things we can do with this information.

Call Graphs

The simplest is to find out what routines are called from within a particular routine. In other words, starting with a routine, say msa_read_fasta, we can find out what routines are called in that code. And we can do this for numerous routines and build a graph of what routine calls what other routines. This is a very helpful tool for understanding code and also for doing statistics on the structure of the software. This is the static call graph as many of the calls to other routines may never occur as they might be enclosed within conditional statements. The run-time call graph is observed when the code is run and tells us about what routines were actually called by another. It is interesting to compare these two graphs.

The function *getCallGraph*() computes the call graph for a given routine. We give it ae routine description returned from *getRoutines*() and it will recursively follow all the nodes in the body of the routine and find all calls to other routines. Note that it will not be able to handle calls via routine/"function" pointers. Returning to our msa.c code again, we can use *getCallGraph*() with the following code.

R
```
p = parseTU("msa.c.tu", language = "C")
routines = getRoutines(p, "msa.c")

calls = getCallGraph(p, routines$msa_new_from_file)
```
The result stored in **`calls`** is very simple and is merely an integer vector identifying the function_decl nodes in our parser/graph that were called within the body of the routine msa_new_from_file. The names of the elements in the vector **`calls`** are the names of the routines, and these are more interesting to the human. However, we do often want to be able to follow those routines and so having the node identifiers is convenient. So

R
```
names(calls)

 [1] "str_new"                "msa_read_fasta"         "la_to_msa"
 [4] "la_read_lav"            "ss_read"                "fscanf"
 [7] "die"                    "msa_new"                "smalloc"
[10] "smalloc"                "msa_alph_has_lowercase" "smalloc"
[13] "smalloc"                "str_readline"           "str_trim"
[16] "strcpy"                 "fscanf"                 "fgets"
```

```
[19] "isspace"                "strcpy"                 "fgets"
[22] "isspace"                "toupper"                "isalpha"
[25] "die"                    "die"                    "str_free"
```

gives the names of all the routines that were called. Note that there are duplicates, e.g. die and smalloc. And the order is often suggestive of the order the routines are mentioned in the code, but of course we branch down different if and while statements and so the order is not necessarily meaningful. Rather than looking at the raw names, we can see how often each routine is called using

R
```
sort(table(names(calls)), decreasing = TRUE)

              smalloc                   die                 fgets
                    4                     3                     2
               fscanf               isspace                strcpy
                    2                     2                     2
              isalpha           la_read_lav             la_to_msa
                    1                     1                     1
msa_alph_has_lowercase              msa_new        msa_read_fasta
                    1                     1                     1
              ss_read              str_free               str_new
                    1                     1                     1
          str_readline              str_trim               toupper
                    1                     1                     1
```

Determining output parameters

As we mentioned above (

Note
not yet!

), a routine can have parameters that are meant to return information and not act simply as inputs. These parameters will be pointers or possibly references in C++. But of course, not all pointers or references are actually used to return information. The *`paramStyle`* parameter of

R
```
createMethodBinding
```

allows us to indicate which parameters are to be treated as inout- or out-style parameters, and the values are included in the return value to R. Unfortunately, a human has to specify which parameters are inout and out. There is no convenient, portable way for the author of the code to indicate this directly in the code although she might document it and then it becomes easy for a human to find this information. In the absence of that however, one can read the code and follow the logic to find out what parameters are modified and so presumably an out value. If we can do this by eye, we can come up with at least ad hoc approaches to doing programmatically from the detail code description of the body. The function *`getInOutArgs`*() is an initial attempt to do this.

We start by resolving the routine of interest, e.g.

R
```
p = parseTU("inout.c.tu")
r = getRoutines(p, "inout")
foo = resolveType(r$foo, p)
```

Then, we pass this routine object to *getInOutArgs*() along with the parser nodes **p** and *getInOutArgs*() recursively processes all the code and (attempts to) determine if the parameter or any of its fields was assigned a value in the code. That would make it an out argument. It is an inout argument if a value was accessed in the parameter not as the left hand side of an assignment operation. This can become complicated and so the code needs to be trained and made more particular. But it does work on some trivial and some real example code.

The result from the call

R
```
getInOutArgs(foo, p)
```

is a list containing the parameter descriptions from the routine of those mutable parameters that appear to be modified in the code of the routine.

This currently does not handle calls to other routines to determine whether they are modified.

Global Variables

In addition to data structures and routines, we also want access to global variables defined within the native library. Global variables are bad, but constant ones are okay. They act as symbolic constants. These const values are relatively easy to deal with in R. For such variables that correspond to basic types in R, e.g. integer, logical, character, numeric, it is obvious that we can merely create parallel versions in R bound to an R variable with an equivalent name as in the native code. Since the variables in the native code are constant, they will not change and there is no need to synchronize the value of the corresponding R variable. All that we need to do is compute the values and assign them to R. We can do this when we load the package, or preferably when we are installing the package as this computation only needs to be done once, not each time the package is used.[3]

We identify all the constant primitive global variables in the files of interest and then create C/C++ code that is compiled into an executable. This executable can be run when the R package is installed and it generates `var <- value` R expressions which can be put in a file as part of the R source code for the package. We'll use the wxWidgets library (http://www.wxWidgets.org) as an example. We have the translation unit nodes in **tu** and the names of the files containing the code in **targetFiles** so that we can filter only those global variables and not the ones from the system files. We call

R
```
globalConstants = computeGlobalConstants(tu, files = targetFiles)
writeCode.ComputeConstants(globalConstants, file = "RwxConstants.cpp"),
                              includes = '<wx/wx.h>')
```

Next, we compile this code

R
```
gcc -o wxConstants  `wx-config --cflags --libs` wxConstants.cpp
```

[3] Of course, if the original native code changes between R sessions, we will have to recompute these values, but we will have to regenerate the entire set of bindings, so our claim is still true.

adding the additional compilation and linker flags we need. Next, we run the executable

R
```
./wxConstants > wxConstants.R
```

and this creates the file wxConstants.R that looks something like

```
wxDefaultTimeSpanFormat <- "%H:%M:%S"
wxDefaultDateTimeFormat <- "%c"
wxListCtrlNameStr <- "listCtrl"
wxEVT_COMMAND_LIST_ITEM_FOCUSED <- as.integer(10065)
wxEVT_COMMAND_LIST_COL_END_DRAG <- as.integer(10061)
wxEVT_COMMAND_LIST_COL_DRAGGING <- as.integer(10060)
wxEVT_COMMAND_LIST_COL_BEGIN_DRAG <- as.integer(10059)
```

And so we can load this into R with,

R
```
source("wxConstants.R")
```

or simply add it to the R/ directory of the R package you are creating. We run the executable in the configuration script of the package to put the

Non-constant Global Variables

For non constant variables, we need to be able to ensure that we get the current value of that variable. In the phast code, we have only two global variables, both of which relate to regular expressions. These are re_syntax_options and re_max_failures. The former has type re_syntax_t and the latter is an int. Neither are const and they can change within the code[4].

The "simplest" thing we do is to create an R object that is a reference to the native variable, i.e. contains the address of that variable. We can ask for the value of that object using the valueOf() function in R and that derefences the address and returns the current value (either as a regular R object or as an external pointer). This allows us to get and use the current value in arbitrary calls. For example, we can get the value, call a routine which will have a side effect of updating the value and then get the new value with code something like:

R
```
valueOf(re_max_failures)
foo() # which changes the value of re_max_failures
valueOf(re_max_failures)
```

We could also use

R
```
as(re_max_failures, "integer")
```

or

[4]It appears that re_max_failures is never modified within the code, but it may be from code outside of this library that links to it!

```r
as(re_max_failures, "numeric")
```

and this will call valueOf() and then coerce to the final target type. Unfortunately, we are not provided the target type in the coercion method, i.e. the to value. So one has to use

```r
as(valueOf(re_max_failures), "numeric")
```

We could introduce a hideous dereferencing syntax such as x$'$' that is trivial to implement, but...

We should also be able to assign a value to it. This is syntactically slight akward. One might think that

```r
re_max_failures = 100
```

should do the job. But of course, that will overwrite the current value of the R object with the R value 100. It will not assign the value to the native variable. We could arrange for this to work using the RObjectTables package and having assignments to these mirrored variables be assigned to the native variable. This is a nice application of the RObjectTable mechanism and we may pursue it in the future as time admits. In the mean time,

```r
valueOf(re_max_failures) = 100
```

could be used. We can also use *RObjectTables* to dereference the value of one of these VariableReference objects when it is accessed. The RObjectTable instance would then act as a broker for accessing these values and pass the requests to access or set the native variable as if it were an R variable.

If we want to make use of the value of the variable in a call to a function, we can use *valueOf*() , e.g.

```r
bar(valueOf(re_max_failures))
```

However, it would be more convenient and natural to write

```r
bar(re_max_failures)
```

And when bar is routine for which we have generated bindings, it is more efficient and simple to pass the R value of re_max_failures directly as a reference and have it be dereferenced in the C code rather than copy it from C to R and back. This is somewhat tricky to handle portably with variables that are not themselves pointers but rather actual literal data types, e.g. int structs, ... So in the meantime, we will arrange that there is a coercion method for the different types via a call to valueOf. This converts a VariableReference to its value. Unfortunately, it is currently difficult to then ensure that that is the correct type.

So, we arrange for the dereferencing of a VariableReference object to attempt to perform the derferencing appropriately in the C code since we know what the target type is in the general dereferencing, i.e. via R_GET_REF_TYPE (macro) calls in C.

That's the general strategy and interface. There are some details that need to be handled (e.g. how to set global variables, and the case of a parameter being a struct object rather than a reference, but these don't pertain specifically to global variables.) So let's see how we can use the functions in the *RGCCTranslationUnit* package to generate the interface to the global variables. We'll work with the globals examples in the package which has 3 global variables (a, aref and i), one static variable (dummy), and a constant double named x. The tu file is in globals.c.tu.

```r
p = parseTU("globals.c.tu")
```

```
gg = generateGlobalVariableCode(p, "globals.c")

writeCode(gg$consts, "c", "globalConstants.c", includes = '"globals.h"')
```

Note the quotes around the value of the *includes* value.

R
```
writeCode(gg, "r", "globals.R")
writeCode(gg, "c", "Rglobals.c", includes = c('"globals.h"'. '"Rglobals.h"'))
```

Other "constants" - enums, defines

In addition to regular top-level/global variables, we also have other types of variables which are enumerations and preprocessor defines. We can find all enumeration definitions within the code using *getEnumerations*() .

Customizing the Code Generation: Type Maps

For any particular piece of code, you may know some specific information about how to convert an C-level structure to R or vice versa. The generic treatment as a reference may not be sufficient. So you will want to be able to influence how the code is generated and the data types marshalled to and from R. The concept of a "type map" is used for this. This is merely a table, i.e. a list, in R with elements that identify the particular type to which it refers and provide information about how the different aspects of the marshalling process are done. At present, there are three different steps in marshalling. The first is in the R function that is called by the R user and involves coercing the input value to the appropriate type. For example, if the C routine requires an integer for the parameter x, there is a line in the R wrapper function of the form

R
```
x = as(x, "integer")
```

This ensures that the value has the appropriate representation when passed to the native routine. Importantly, it also allows the user to pass a value which is not an integer to the function without worrying about the particular type. And it also allows us or the user to provide methods for coercing objects of different types to an integer. This makes the mechanism extensible to us and others.

The second step in marshalling is when the R value is converted to its equivalent C type. For instance, when we pass the integer object from R, this is converted to an int in C with the code

R
```
int x;
 x = INTEGER(r_x)[0];
```

This is built in to the bindings to access an R object which is expected to be an integer object and map it to an int in this way. However, we may want to change how this is done.

And the last step in the marshalling is to convert a value from C back to R, e.g. the return value of a routine or the value of a field in a `struct`. Again there is some C code that does this and returns the appropriate value or leaves it in a suitable variable.

The type map provides a mechanism to optionally control any of these marshalling steps for a given native type. One specifies the target type either by a simple C-like declaration, or via a more complete and formal *TypeDefinition* object. The other elements named coerceR, convertRValue, and convertValueToR relate to the three steps respectively. A type map element in R is specified as a list with a target element and any or all of these three actions. The target is a string or *TypeDefinition* object and is compared to each of the types to which we are trying find a match in the type map table. Each of the three action elements can be either a simple string giving the name of an R function or C routine (as appropriate), or an R function. If a string is given, this is simply used in a call to convert the given R or C variable to the appropriate value. All that the code generation mechanism does is take that function name and create an invocation string with the given argument. So if the string were "as.integer", we would see `as.integer(x)`. However, if we want a more interesting call such as `open(x, "r")`, simply specifying a single string will not suffice. So it is most general to provide a function which will generate the relevant code string. Such a function takes three arguments: the name of the variable being processed, a named list of the parameter types, and the type map object to be used in recursive processing. The function is expected to return a string which can be inserted into the generated code. If it is a simple string, some of the code generation functions will append additional code such as assignments to local variables. One can avoid this by returning, e.g. an object of class RCode, to by-pass any default processing.

A function is most general, but often just as with the `open(x, "r")` we just want have the variable name inserted into the first argument. We can do this by providing not a string, but a character vector. And there are two ways to do this. One is to provide a character vector with NA values where one wants the variable name inserted. For example, if we wanted the result to be the string `foo(x, length(x))`, we could provide as our type map converter a function of the form

R
```
function(name, parameters, typeMap)
{
  paste("foo(", name, ", length(", name, ")")
}
```

Or alternatively, we could provide the slightly simpler

R
```
c("foo", NA, ", length(", NA, ")")
```

and avoid the function definition. Since this is simply text substitution and there is no dynamic computation involved, this is probably simpler.

And if we have a very simple case such as wanting to produce `open(x, "r")` which only involves a single insertion, we can provide a character vector of the form

R
```
c('open(', ', "r")')
```

This is a character vector with just two elements. The conversion code that uses the type map element will insert the variable name between the two elements and glue them together to form the desired string.

We have given the dry, "formal" description of these type maps but they are easier to understand with an example. We will consider the case where a C routine takes a parameter

R
```
FILE *
```

. This is a simple pointer to a FILE instance which is obtained from opening a file in some format. There is no analogous type in R that we can use. [5] There are two straightforward strategies to map these from R types to C when going through the R and C wrapper routines. The two step process provides us degrees of freedom to consider how this can be achieved. The first approach is to convert the argument to the R

[5]Connections are analogous, but there is no API that allows us to access the underlying FILE instance.

function to the name of a valid file. Then in the C code, we call a routine with the R object giving the file name and return the desired FILE * instance. We can do this with a type map as

R
```
typeMap( "FILE *" = list(target = "FILE *",
                             # Can be a string, e.g. asFILE, but then couldn't get
                           coerceRValue = function(name, ...) paste("asFILE(", nam
                           convertRValue = "R_openFile"
                          ))
```
Note that we specify the target type for which this element matches in its C declaration format, "FILE *". We can give this as either the name of the type map element or explicitly as the target entry within the type map element. Then we specify the mechanism to coerce the R value to a file name which is a call to a function *asFILE*() which we will have to write. This checks the file exists and then expands the file name to expand ~, etc. which accepts for filenames, but C does not.

The conversion from this R type to the C-level type FILE * is given by the convertRValue element. This is merely a call to a hand-written routine that takes the R string and dereferences it and calls the *open* routine to create and return the FILE *. The `typemap/` example contained in the package illustrates this in the context of a simple routine *getLine*. The directory contains all the relevant code to generate the bindings and the results and is "runnable". For the routine declared as

R
```
char *getLine(FILE *)
```
we end up with the R wrapper function

R
```
getLine =
function( f ) {
     f = asFILE( f , 'r')
    .Call('R_getLine', f)
}
```
and C routine

R
```
SEXP
R_getLine(SEXP r_f)
{

    SEXP r_ans = R_NilValue;
   FILE * f ;
     char * ans ;

        f  =  R_openFile ( r_f ) ;

    ans =   getLine ( f ) ;
    r_ans =  mkString( ans  ?  ans : "") ;

    return(r_ans);
}
```
Note the calls to *asFILE*() and *R_openFile*() that come from our type map.

So we need to define these two entities (see utils.R and Rutils.c respectively.)

R

```
asFILE =
function(filename)
{
  if(!file.exists(filename))
    stop("No such file ", filename)

  path.expand(filename)
}
```

R

```
SEXP
R_asFILERef(SEXP r_f, SEXP r_mode)
{
    FILE *f;
    char *fileName;

    fileName = CHAR(STRING_ELT(r_f, 0));
    f = fopen(fileName,  CHAR(STRING_ELT(r_mode, 0)));
    if(!f) {
 PROBLEM "cannot open file %s", fileName
        ERROR;
    }

    return(R_MAKE_REF_TYPE(f, FILERef));
}
```

There are two things to note. If we didn't want to define a new function named `asFILE()` , we could inline its contents, i.e. the call to file.exists and path.expand. We would specify this in the typemap as

R

```
coerceRValue = function(name, parameters, typeMape)
                 RCode(paste("if(!file.exists(", name, "))"),
                       "stop('no such file ', name)",
                        paste(name,"= path.expand(", name, ")")
                       )
```

Creating an RCode object means that the assignment is not prefixed by the code that uses this, i.e. "f = "

Alternatively, we could specify this using NAs for the locations to insert the name.

R

```
structure(c("if(!file.exists(", NA, "))\n\tstop('no such file'," NA, ")\n", NA, "=
          class = "RCode")
```

Dependencies Files