Accessing the GeoIP library from R with Rffi and RGCCTranslationUnit

Duncan Temple Lang, University of California at Davis

Table of Contents

The GeoIP library

The GeoIP library from MaxMind is an Open Source library that maps Internet Protocol addresses to estimated latitude and longitude, city and county values. (MaxMind also provides a more accurate commercial version of the database.) This ability to take an IP address and get its location is useful in many data analysis applications. One example is doing intrusion detection on network traffic packets. Another is drawing a map of visitors to a Web site by analyzing web logs.

Before we can use the GeoIP C library, we need to both understand the basic programming model/interface and then we need to build the bindings from R to the C code. We will use the RGCCTranslationUnit package to programmatically obtain a description of the C-level API. We will use Rffi to invoke the routines. So all that remains before getting started is to get an understanding of the programming model. We can pass an IP address either as a number, a "dotted-quad" or by host name such as "cran.r-project.org". For each operation in the C library, there are 3 routines corresponding to these different input types. The names of the routines end in _ipnum, _addr and _name, respectively. We can use IP V6 or regular IP V4 addresses and the IPV6 routines have a _v6 suffix (following the _ipnum, _addr or _name). So to get the name of an IP address's country, we can use *GeoIP_country_code_by_ipnum*, *GeoIP_country_code_by_addr* or *GeoIP_country_code_by_name*

Each routine takes a pointer to the GeoIP "database" or agent and then the IP address in the appropriate form. To be able to use any of these routines, we must first create a GeoIP instance and obtain a reference to it. We do this with *GeoIP_new*. When we are finished with this, we can use *GeoIP_delete* to clean up the memory associated with it. There are other routines that give us more flexibility, but these are all we need for now.

The following is pseudo, heuristic, informal sample C code that we might use to interact with the library as if we were writing interactively without declarations.

```
db = GeoIP_new(GEOIP_STANDARD)
r = GeoIP_record_by_addr(db, "169.237.46.32")
r->latitude
```

```
r->longitude
r = GeoIP_record_by_name(db, "www.omegahat.org")
r->city
r->postal_code
r->country_code
```

The *GeoIP_record_by_addr* routine returns a pointer to a C struct with 12 different fields identifying the location of the IP address.

Manually Interfacing to the C library with Rffi

We'll start by manually creating the FFI-based interface to the routines in the pseudo C code above. First, we need to define a call interface (CIF) for *GeoIP_new*. This takes an integer and returns a pointer to a *GeoIP* object. We can treat that as an opaque data type and we don't need to know about its contents. We just pass it to the other routines we will invoke. So we can define the CIF as

```
library(Rffi)
GeoIP_new_cif = CIF(pointerType, list(sint32Type))
```

Next we need to know the appropriate value to pass as the integer argument. By reading the documentation or examples, we see that it is a combination of one or more options from the GeoIPOptions enumeration. We'll just use the GEOPI_STANDARD value which is 0. We'll talk later about how we can use the symbolic names for these enumerations to make the code clearer and more robust. So now we are ready to call the routine. The first thing to do is load the GeoIP library. The details will differ on your machine but on a Mac OS X box, we can use

```
dyn.load("/usr/local/lib/libGeoIP.dylib")
```

(We can use

```
.Platform$dylib.ext
```

to find the appropriate extension. /usr/local/lib is a reasonable place to expect the library to be located, but it is not guaranteed.)

So now we have loaded the routines, we can invoke *GeoIP_new* with

```
db = callCIF(GeoIP_new_cif, "GeoIP_new", 0L)
```

db is a simple external pointer object in R. This is very "raw" as it stands and we probably want to create a class to represent that this is a GeoIP pointer object and create a function to hide the details of calling GeoIP_new. But we'll come back to this niceties later.

Next we want to invoke *GeoIP_record_by_addr*. So again we need create a CIF for this signature. This takes a pointer (to a GeoIP object) and a string giving the dotted-quad IP address. The routine returns a pointer to a GeoIPRecord structure. This is defined in the GeoIPCity.h header file as

```
typedef struct GeoIPRecordTag {
 char *country_code;
 char *country_code3;
 char *country_name;
 char *region;
 char *city;
 char *postal_code;
 float latitude;
 float longitude;
 union {
   int metro_code; /* metro_code is a alias for dma_code */
   int dma_code;
        };
 int area_code;
 int charset;
 char *continent_code;
} GeoIPRecord;
```

We don't have to deal with this stuct quite yet. Instead, we need only define the CIF as returning a pointer. So our CIF is created as

```
GeoIP_record_by_addr_cif = CIF(pointerType, list(db = pointerType, addr = stringTy
```

The name of the R variable used to store the CIF is irrelevant, of course. I am using the name of the routine with a _cif suffix for clarity so we know what the value is for. But of course, this same CIF could be used with any routine with the same signature.

Now we can call the routine

```
r = callCIF(GeoIP_record_by_addr_cif, "GeoIP_record_by_addr", db, "169.237.46.32")
```

With this, we find we get an error from GeoIP

```
Invalid database type GeoIP Country Edition, expected GeoIP City Edition, Rev 1
```

This is because I am using the city-level API on the country database. I need to install the city database. We'll take a slight detour and show how to make a call to find the country of an IP address and then return to dealing with the city database.

In the GeoIP.h header file, there is a routine *GeoIP_country_code_by_addr*. This has a very similar signature as *GeoIP_record_by_addr* but returns the name of the country in which the IP address is located. So we can create a new CIF and invoke this routine:

```
cif = CIF(stringType, list(db = pointerType, addr = stringType))
country = callCIF(cif, "GeoIP_country_code_by_addr", db, "169.237.46.32")
```

If we look at the R object `country`, we see it is a list with two elements named "value" and "inputs".

```
country$value
```

is the return value and gives us the name of the country - "US". The second element - "inputs" - is the `db` argument to the routine. Since that is a pointer, the routine may have modified its contents. So *`callCIF()`* returns that. We know that any updates will be see in the R variable `db` and also we think that it is actually not modified in the routine. So when creating the CIF, we can say that this is not mutable. Alternatively, in the call to *`callCIF()`*, we can tell it not to return the inputs. So

```
country = callCIF(cif, "GeoIP_country_code_by_addr", db, "169.237.46.32", returnIn
```

returns just the country name as a string. And

```
cif = CIF(stringType, list(db = pointerType, addr = stringType), mutable = c(FALSE
country = callCIF(cif, "GeoIP_country_code_by_addr", db, "169.237.46.32")
```

also returns just the string. Setting the mutability of the parameters in the CIF is a stronger statement as this applies to all routines we call with the same CIF.

Now let's return to the issue of the city database. You can download this from http://www.maxmind.com/app/geolitecity. Follow the download and installation instructions there. We need to tell our GeoIP "server" which database to use. *GeoIP_new* is a little too simple as we cannot control the choice of database. Looking in the GeoIP.h header file, we see three potentially promising routines:

```
GEOIP_API int GeoIP_db_avail(int type);
GEOIP_API GeoIP* GeoIP_open_type (int type, int flags);
GEOIP_API GeoIP* GeoIP_open(const char * filename, int flags);
```

If type identifies which database to open, we can check whether the database is available and use that with GeoIP_open_type. If we have the database file in a non-standard location, we can use GeoIP_open to provide the full file name.

The type should be one of the GeoIPDBTypes enumerated values and we can try GEOIP_CITY_EDITION_REV1 (with a value of 2) as the possible option. So let's create a CIF for calling GeoIP_db_avail and call that:

```
callCIF(CIF(sint32Type, list(sint32Type)), "GeoIP_db_avail", 2L)
```

This returns 0, i.e. not available for me. We can try the different options, but GeoIP seems to only see the country database. So let's try to load the file by name.

We need to create a CIF for GeoIP_open and invoke it:

```
cif = CIF(pointerType, list(stringType, sint32Type))
db_city = callCIF(cif, "GeoIP_open", "/usr/local/share/GeoIP/GeoLiteCity.dat", 0L)
```

This successfully returns a pointer. So now we can return to obtaining a record:

```
r = callCIF(GeoIP_record_by_addr_cif, "GeoIP_record_by_addr", db_city, "169.237.46
```

And this also returns successfully. We probably don't want the mutable inputs so we can change our call

```
r = callCIF(GeoIP_record_by_addr_cif, "GeoIP_record_by_addr", db_city, "169.237.46
```

Now we have the record which is a pointer to the 12-element struct displayed above. Since we want to access fields in that struct, we need to create a description of the types of elements in the struct. We can do this programmatically, but for now we will do it manually.

We need to create a new type similar to sint32Type, stringType, etc. but for describing this particular GeoIPRecord structure. We use the function `structType()` to do this. We pass it an ordered list of the types of the fields in the struct. So for this struct, we would use R code something like

```
GeoIPRecord_type =
            structType(list(stringType, # country_code
                             stringType, # country_code3
                             stringType, # country_name
                             stringType, # region
                             stringType, # city
                             stringType, # postal_code
                             floatType,  # latitude
                             floatType,  # longitude
                             sint32Type, # the union element
                             sint32Type, # area_code
                             sint32Type, # charset
                             stringType # continent_code
                            ))
```

Now that we have this description, we can use it to have Rffi retrieve individual fields from the pointer to the struct or copy all of the fields to an R list. The two functions to do this are `getStructField()` and `getStructValue()`. We will access the latitude and longitude fields. Unfortunately, we didn't put names on the elements of the struct description so R doesn't know how to match latitude to the 7th element. So we can use the index of the element using

```
getStructField(r, 7, GeoIPRecord_type)
```

and we get our value. Similarly, we can get the city name with

```
getStructField(r, 5, GeoIPRecord_type)
```

Using indices is not a good idea. So we should put names on the elements in the list we pass to `structType()`:

```
GeoIPRecord_type =
           structType(list(country_code = stringType,
                           country_code3 = stringType,
                           country_name = stringType,
                           region = stringType,
                           city = stringType,
                           postal_code = stringType,
                           latitude = floatType,
                           longitude = floatType,
                           "?" = sint32Type,  # the union element
                           area_code = sint32Type,
                           charset = sint32Type,
                           continent_code = stringType
                          ))
```

Now we can use a name to access the field:

```
getStructField(r, "latitude", GeoIPRecord_type)
getStructField(r, "city", GeoIPRecord_type)
```

We can even get multiple values in a single call:

```
getStructField(r, c("latitude", "city", "region"), GeoIPRecord_type)
```

There are lots of other things we can look at in the API that are more complex, e.g. GeoIP_next_record, GeoIPRegion struct and fixed length arrays, the union, etc., but we will return to those later. For now we have the information we want from GeoIP and we are working with simple/built-in data types and a single struct. If we wanted to access the fields in the GeoIP object, we would describe its structure in the same way as for the GeoIPRecord. This would give us access to the name of the file, when it was last updated, etc.

We will take a brief look at how we can access global variables. libGeoIP has several global variables which are strings or arrays of strings or pointers to strings. For example, GeoIPDBFileName is a pointer to a string, GeoIPDBFileName is a string and GeoIP_country_code is an array of 253 strings of length 3. Once we describe the type of the variabe, we can use `getNativeValue()` to retrieve the value of the object. This will convert the C value to an R object using the same conversion mechanism as used in converting the return value via `callCIF()`.

So let's find the value of GeoIPDBFileName:

```
getNativeValue("GeoIPDBFileName", stringType)
```

Note

Come back to this and accessing arrays in R by indexing, etc.

Cleaning up the CIF interfaces

Before we turn our attention to how we might automate a lot of the work we did above, we will think about how we might like to have the R interface to these routines and data structures. Firstly, we want to hide the create of the CIFs and the calls to `callCIF()` and provide R functions that are direct proxies for the C routines with all these details inside the R functions. Secondly, we would like to define classes for the pointers that are returned so we can identify the type of object to which they point. We also would like to have `$`() and `[[`() operators for accessing fields in a struct rather than having to cal `getStructField()`. Additionally, if we are making repeated calls to the same routine, we don't want to incur the penalty of having to find the address of the routine in each call. So our functions should cache that on the first call or when they are defined. We can do this with closures or a global variable.

Automating the Interface Generation

```
invisible(sapply(c("const.R", "createRFunc.R", "defClasses.R", "tuTORType.R", "tuT
```

We now turn our attention to how we can programmatically obtain descriptions of C routines and data structures and then use these to create CIFs and struct types so that we don't have to manually build the interface between R and C code. We use the RGCCTranslationUnit package to read the C code. (We could use RCIndex, but that is still a work in progress.) We first create a simple C file that merely includes the GeoIP.h and GeoIPCity header file.

```
#include <GeoIP.h>
#include <GeoIPCity.h>
```

Then we generate the TU output from gcc with

```
gcc -fdump-translation-unit -o /dev/null -c Rgeoip.c
```

Now we can read this translation unit into R:

```
library(RGCCTranslationUnit)
tu = parseTU("~/Projects/org/omegahat/R/GeoIP/inst/doc/Rgeoip.c.001t.tu")
```

We are interested in the routines, so let's find all of them and then get the subset for the GeoIP interface:

```
funcs = getRoutines(tu)
funcs = funcs[ grep("^GeoIP", names(funcs), value = TRUE) ]
```

We will also be interested in the different enumerations used in the code, so let's find those also.

```
enums = getEnumerations(tu)
names(enums)
```

From the work we did manually, we are interested in the routines *GeoIP_open* and *GeoIP_record_by_addr*. Because we want to access the fields in the struct returned by *GeoIP_record_by_addr*, we will be interested in the GeoIPRecord structure. We'll come by that passively when finding out about the signature for *GeoIP_record_by_addr*.

Let's start with *GeoIP_open*. We have the node in the translation unit graph that corresponds to this routine's declaration. We now want to pull together all the information about the pieces of this routine and we use *resolveType()* for this. We pass it the node and the translation unit so it can follow all the relevant links:

```
ip.open = resolveType(funcs[["GeoIP_open"]], tu)
ip.open
```

Note that the names of the parameters aren't available in the TU from GCC and we use simple numbers!

We can see the types of the parameters printed on the console, but now we want to programmatically access them and map to the corresponding FFI types. We can loop over the parameters, fetch the type of each and pass this to *gccTUTypeToFFI()*. As the name suggests, the function maps a GCC-TU type to the corresponding FFI type. So our code to get the inputs for creating the CIF for the routine is

```
parmTypes = lapply(ip.open$parameters, function(x) gccTUTypeToFFI(x$type))
rt = gccTUTypeToFFI(ip.open$returnType)
```

Now we can create the CIF:

```
ip.open.cif = CIF(rt, parmTypes)
```

We may want to add information about whether the parameters are mutable or not by looking for const declarations.

Next we will create an R function named GeoIP_open and have it accept 2 parameters and call the corresponding C routine. Our function can be built from the description of the routine `ip.open`. The function needs to know the name of the R variable containing the relevant CIF, i.e. oip.open.cif, or alternatively create the CIF itself, either once or each time the function is called. The function then merely invokes *callCIF()*, passing the 2 arguments it receives. So the function would be

```
GeoIP_open = function(x1, x2, ..., .cif = ip.open.cif) {
   callCIF(.cif, ip.open.cif, "GeoIP_open", x1, x2, ...)
}
```

(The fact that the parameter names are lost in the translation unit is a pity. We are working on an approach where they are not.) Note that we have allowed the caller to specify a different CIF should she want. We have also provided a ... argument to allow arguments to be passed on to *callCIF()*, i.e. `returnInputs` and any others we add. Instead of referring to a variable that contains the CIF, we could provide an expression to create the CIF as the default value for `.cif`. To do this, we would get not the CIF types for the parameters and return type, but rather then names of these variables and create the call to *CIF()* with the elements explicitly articulated, i.e.

```
GeoIP_open = function(x1, x2, ...,
                       .cif = .cif = CIF(pointerType, list(stringType, sint32Type)
   .........
}
```

Our function should also convert the result to an explicit reference to a GeoIP object. We should define a class, say `GeoIP_ref`, which is a sub-class of `RCReference`, and turn the pointer into an instance of this class. We should define methods for the $ and [[operators for this reference class to access individual fields using *getStructField()*. Since a GeoIP object is a struct in C, we should also define a class in R that mirrors that struct and has the same slots. We should also provide a coercion method to transfer a C reference to a GeoIP object to the R type by calling *getStructValue()*. So mapping this routine to an R function involves a lot of peripheral functions that are done just once for each "complex" data structure.

We want to write a function that will take a GCC-TU description of a routine and create the R code to call that routine, i.e. the routine above.

Of course, in creating our CIF and function and methods, we have had to do some programming using the translation unit code. The amount of effort is at least as much as when we did it manually by reading the header file ourselves, although we have a much cleaner and more comprehensive interface. The important thing to recognize, however, is that we can automate this. While we did go through all the steps in detail, these can be put into a function to create the CIF for an arbitrary routine. And since we have information for all the routines, we can generate CIFs for the entire GeoIP library, or any C library for that matter.

The RGCCTranslationUnit package provides the functions to create the R code described above for arbitrary routines and structure definitions. We can use them as follows to create an interface to GeoIP_open and GeoIP_record_by_addr:

```
ip.open = resolveType(funcs[["GeoIP_open"]], tu)
define(createRFunc(ip.open))
define(defStructClass(ip.open$returnType@type))
record = resolveType(funcs[["GeoIP_record_by_addr"]], tu)
define(createRFunc(record))
define(defStructClass(record$returnType@type))
```

We can find out what classses and functions were defined with

```
getClasses(globalenv())
ls(pattern = "^GeoIP")
```

We used closures, so all of the CIF and struct type definitions were made locally within the

The next thing we might want to do is write the generated code to a file so that we can use it in other R sessions. Instead of calling `define()`, we can store the generated code and then use `cat()` to write it to a file. So we generate it as follows:

```
code= list(createRFunc(ip.open),
           defStructClass(ip.open$returnType@type) ,
           createRFunc(record),
           defStructClass(record$returnType@type))
```

Then we can write it with

```
cat("library(Rffi)",
    "dyn.load('/usr/local/lib/libGeoIP.dylib')",
    unlist(code), sep = "\n", file = "geoIP.R")
```

Now that everything is defined, we can load the DSO and call the functions

```
dyn.load("/usr/local/lib/libGeoIP.dylib")
db = GeoIP_open("/usr/local/share/GeoIP/GeoLiteCity.dat", 0L, FALSE)
class(db)
r = GeoIP_record_by_addr(db, "169.237.46.32", FALSE)
class(r)
names(r)
r$city
r$area_code
r$latitude
r$longitude
```

The following code takes 1000 IP addresses from a web log and gets the city for each:

```
load("ip1000.rda")
o = sapply(ip, function(i) GeoIP_record_by_addr(db, i, returnInputs = FALSE)$city)
```

Enumerations

We have looked at working with routines and different data structures. We now return to enumerations and symbolic constants. We saw the GeoIPOptions data type which are values we pass to *GeoIP_open* or *GeoIP_new*. The possible values are GEOIP_STANDARD, GEOIP_MEMORY_CACHE, GEOIP_CHECK_CACHE, GEOIP_INDEX_CACHE and GEOIP_MMAP_CACHE. These labels correspond to the values 0, 1, 2, 4, and 8, respectively, but of course the names are far more suggestive of the meaning. As a result, we want to use the symbolic names in R. So we can define R variables with the same names and values:

```
GEOIP_STANDARD = 0L
GEOIP_MEMORY_CACHE = 1L
GEOIP_CHECK_CACHE = 2L
GEOIP_INDEX_CACHE = 4L
GEOIP_MMAP_CACHE = 8L
```

Note that we have explicitly made these integers.

There are three aspects of this we want to consider. The first is somewhat cosmetic and a matter of convenience. All of these values start with the string "GEOIP_". It would be more convenient to optionally refer to the values without this common prefix, e.g. STANDARD, MEMORY_CACHE, etc. Secondly, while the GeoIP API declares the flag parameter type as *int*, we should use explicit enumeration types when this is what is expected. This helps to catch errors in the code and is also clearer. If a routine is declared with an enumeration parameter, we must be able to validate the value as being from the set of possible values. We can achieve this by defining a class for the enumeration type in R and defining coercion methods from integers or names to the actual value. Furthermore, we can use the symbolic names on the integer values to make the values more intelligible for humans to read. The third issue is that not all enumerations are simple mutually exclusive options. Some enumerations are intended to be combined together as bitwise-AND values and then compared to the individual possible values via bitwise OR operations. This is typically done using powers of 2 for the values to set individual bits. This might be done with an enumeration in which the values for the individual elements are explicitly specified, e.g.

```
enum {
 CURLPROTO_HTTP = 1,
 CURLPROTO_HTTPS = 2,
 CURLPROTO_FTP = 4,
 CURLPROTO_FTPS = 8,
  ...
};
```

or with a collection of pre-processor #define directives, e.g.,

```
#define CURLPROTO_HTTP   (1<<0)
#define CURLPROTO_HTTPS  (1<<1)
#define CURLPROTO_FTP    (1<<2)
#define CURLPROTO_FTPS   (1<<3)
 ...
```

. These come from libcurl and relate to the different protocols.

A critical different between these and enumerations is that we combine two of these values via an OR operation and we obtain a new and permissible value. For example, we can query whether the protocol to be used includes HTTP or HTTPS with

```
protocol & (CURLPROTO_HTTP | CURLPROTO_HTTPS)
```

This is very different from an enumeration. We also want the object to identify its original components, so we add combine the names of the individual elements as the name of the integer value. So to allow both HTTP and HTTPS, we might use

```
CURLPROTO_HTTP | CURLPROTO_HTTPS
```

and we want the result to appear in R as

```
CURLPROTO_HTTP,CURLPROTO_HTTPS
                             3
```

To address these issues, we define two basic classes - *EnumValue* and *BitwiseValue*. The first is used as the root or base class for any new enumeration type we want to define. This is used for sets of simple symbolic constants represented as enumerations. *BitwiseValue* is used for values that we can combine together into a single value as in the curl protocol example above. To represent the different values of the GeoIPDBTypes enumeration, we would use *EnumValue* to define a *GeoIPDBTypes* class in R:

```
setClass('GeoIPDBTypes', contains = 'EnumValue')
```

We create R variables for each of the values using the corresponding name in C code, e.g.,

```
`GEOIP_COUNTRY_EDITION` <- GenericEnumValue('GEOIP_COUNTRY_EDITION', 1, 'GeoIPDBTy
`GEOIP_REGION_EDITION_REV0` <- GenericEnumValue('GEOIP_REGION_EDITION_REV0', 7, 'G
```

We would then define a global variable that represents the entire enumeration definition, containing the names of the elements and their values. This is used when performing conversions from numeric and string values and validating values. And we finish the R code to handle such an enumeration by defining coercion methods for converting values specified by integer or numeric or by the symbolic name as a string. These allow us to have calls of the form

```
as("GEOIP_ORG_EDITION", "GeoIPDBTypes")
as(5, "GeoIPDBTypes")
```

For a bitwise enumeration, we define the same variables. We inherit the `|`() method for combining two values of this type, e.g.,

```
GEOIP_CHECK_CACHE | GEOIP_MEMORY_CACHE
```

which yields

```
GEOIP_CHECK_CACHE | GEOIP_MEMORY_CACHE
GeoIPOptions                                        3
```

showing the value resulting from combining of the two elements, along with the name of the composition and the class of the object.

Using the RGCCTranslationUnit package, we can resolve the enumeration definition nodes we find in the data structures. The package attempts to determine which enumerations are really bitwise enumerations. In the case of GeoIPOptions, it recognizes that all values are a power of 2 and so determines that it is a bitwise enumeration and the resolved object is of class *BitwiseEnumerationDefinition*. For the GeoIPDBTypes, the values do not suggest a bitwise enumeration and so we get an object of class *EnumerationDefiniton*. The code that does this is

```
tu = parseTU("~/Projects/org/omegahat/R/GeoIP/inst/doc/Rgeoip.c.001t.tu")
```

```
enums = getEnumerations(tu)
TUOptions(checkBitwiseAtResolve = TRUE)
ip.opts = resolveType(enums$GeoIPOptions, tu)
ip.db.types = resolveType(enums$GeoIPDBTypes, tu)
```

We can now turn these descriptions into R code with `genCode()`

Processing all routines and data structures

```
library(Rffi); library(RGCCTranslationUnit)
RGCCTranslationUnit:::TUOptions(checkBitwiseAtResolve = TRUE)
tu = parseTU("~/Projects/org/omegahat/R/GeoIP/inst/doc/Rgeoip.c.001t.tu")
funcs = getRoutines(tu)
funcs = funcs[grepl("^GeoIP", names(funcs))]

sapply(paste("../../R/", c("createRFunc.R", "tuToRType.R", "tuToFFI.R", "genCode.R

funcs.code = lapply(funcs, function(x) createRFunc(resolveType(x, tu)))

ds = getDataStructures(tu)
ds = ds[ grepl("^GeoIP", names(ds))]
rds = lapply(ds, resolveType, tu)
ds.code = lapply(rds, genCode)
```

If we want to source the code into an existing session, we can use

```
library(RGCCTUFFI)
code = genTUInterface("inst/doc/Rgeoip.c.001t.tu", pattern = "^GeoIP")
```

If we want the code so that we can put it in a package, we have to be

```
code = genTUInterface("inst/doc/Rgeoip.c.001t.tu", pattern = "^GeoIP",
                       useClosure = TRUE,
                       useGlobalCIF = TRUE,
                       useGlobalFFIType = TRUE,
                       putGlobalsInLoad = TRUE)
```

```
cat( "library(Rffi)",
     "library(RAutoGenRunTime)",
     ".onLoad = function(...) dyn.load('/usr/local/lib/libGeoIP.dylib')",
     unlist(code), sep = "\n\n",
    file = "R/RGeoIP.R")
```

```
db = GeoIP_open("/usr/local/share/GeoIP/GeoLiteCity.dat", GEOIP_STANDARD, FALSE)
r = GeoIP_record_by_name(db, "www.omegahat.org", FALSE)
r[]
r[c("latitude", "longitude")]
r$lat
r[["lat"]]
```

```
r = GeoIP_record_by_addr(db, "169.237.46.32", FALSE)
r[]
```

```
r[c("latitude", "longitude")]
```

```
ll = readLines(gzfile("~/omegahat.log.gz"), n = 7000)
addr = gsub("([^ ]+) .*", "\\1", ll)
dup = duplicated(add)
ip = addr[!dup]
#ip = unique()
```

```
library(Rffi) # for isNilPointer()
pos = sapply(ip, function(h) {
                   r = GeoIP_record_by_addr(db, h, returnInputs = FALSE)
                   if(isNilPointer(r))
                     c(NA, NA)
                   else
                     r[c("longitude", "latitude")]
                  })
```

If we process all of the lines in omegahat.log.gz, we end up with 265782 unique IP addresses. If we time the sapply() loop below to get the locations, this takes

```
user   system  elapsed
1262.574   17.274 1425.822
```

to process them all or 0.005 seconds per IP address. There are 21 that cannot be matched.

```
i = which(is.na(pos[1,]))
ip[i]
```

We can use the *Rffi* package to create the CIF object describing this type of routine and then invoke it.

```
library(Rffi)
GeoIP_new = CIF(pointerType, list(sint32Type))
```

```
ip.opts = resolveType(enums$GeoIPOptions, tu)
```

Let's skip over the details of the bitwise enumeration for the present and we'll use the value of GEOIP_STANDARD:

```
GEOIP_STANDARD = ip.opts@values["GEOIP_STANDARD"]
```

```
dyn.load("/usr/local/lib/libGeoIP.dylib")
```

So now we can call GeoIP_new

```
ipDB = callCIF(GeoIP_new, "GeoIP_new", GEOIP_STANDARD)
```

We probably want to define a class "GeoIP" and identify this pointer as an instance of this class

```
setClass("RCReference", representation(ref = "externalptr"))
setClass("GeoIP", contains = "RCReference")
```

```
ipDB = new("GeoIP", ref = ipDB)
```

Let's interface to the GeoIP_database_info and GeoIP_database_edition routines to check things are okay.

```
GeoIP_database_info = CIF(stringType, list(pointerType), FALSE)
GeoIP_database_edition = CIF(uint32Type, list(pointerType), FALSE)
```

```
callCIF(GeoIP_database_info, "GeoIP_database_info", ipDB@ref)
callCIF(GeoIP_database_edition, "GeoIP_database_edition", ipDB@ref)
```

Now let's see about calling GeoIP_id_by_addr or GeoIP_id_by_name. These have the same signature so we can use the same CIF.

```
int.GeoIP_String = CIF(sint32Type, list(pointerType, stringType), c(FALSE, FALSE))
```

Then we can call this with

```
callCIF(int.GeoIP_String, "GeoIP_id_by_addr", ipDB@ref, "74.125.45.100")
```

Now let's look at the routine GeoIP_country_name_by_addr. This takes a GeoIP pointer and a string and returns a string. So we define the CIF as

```
GeoIPString = CIF(stringType, list(pointerType, stringType))
```

Now we can use this to call the routine

```
callCIF(GeoIPString, "GeoIP_country_name_by_addr", ipDB@ref, "74.125.45.100")$valu
```

Data Structures

Let's look at GeoIP_region_by_addr. This returns a pointer to a GeoIPRegion object. This is a struct containing two elements, both strings of a fixed length 3, i.e. char [3]. We can create the CIF as

```
GeoIPRegion.GeoIP_string = CIF(pointerType, list(pointerType, stringType))
```

This will allow us to call the routine and get the pointer to the GeoIPRegion.

```
ans = callCIF(GeoIPRegion.GeoIP_string, "GeoIP_region_by_addr", ipDB@ref, "74.125.
```

Now we have to be able to identify the fields in the pointer to the structure. We can manually examine the fields or we can use RGCCTranslationUnit to identify them.

```
ds = getDataStructures(tu)
geoDS = ds[ grep("^GeoIP", names(ds), value = TRUE) ]
reg = resolveType(resolveType(geoDS[["GeoIPRegion"]], tu))
```

We can extract the fields and their types. The names are obtained via

```
names(reg@fields)
```

The type of the first element is

```
reg@fields[[1]]@type
```

and this is an ArrayType. It contains the length and element type of the array.