User-Defined Tables in the R Search Path

Duncan Temple Lang

December 4, 2001

Abstract

We present a mechanism for allowing the use of general C and R code for binding name-value pairs in the R search path. This allows users to attach different data sources and treat remote objects as local R variables. Examples include relationa database tables; the CORBA naming service; variable bindings and tables in Java, Perl, Python, XSLT; archive files using zip and tar; URIs; directories with regular R objects or other formats such as XML, S-Plus, SAS, Stata, etc. The model allows new table types to be programmed using R code or native C code.

1 Motivation

Version 4 of S introduced the ability to attach user-defined "databases" to the search path and for general use. The idea is quite simple, yet very powerful. Each "database" is a table, binding names to values or objects. When such a table is attached to the search path, the names become variables available to any S code and the underlying computation model. When the S engine is searching for a variable on the search path, it asks each of the databases on the search path whether it has that variable. It does this using the *exists()*[1]. The first database to "admit" that it has that variable is then asked to retrieve the value. S uses the *get()* function to do this.

This simple lookup arrangement manages to gloss over the details of the details of how the contents of the database are determined or how a valid S object is retrieved based on the name. That is something the database itself knows how to do. This allows entirely different types of databases to be used in exactly the same way and present a common model for the S programmer and user. For example, here are brief descriptions of some useful name-value collections that might be used in R.

Relational Databases One database in the search path might contain the names of the columns of a table in a relational database. When the variable is retrieved as an S object, the records from the table may be returned via an SQL query to the database. Alternatively, the data transfer may be delayed until the values are needed and instead, a reference to the variable within the table can be returned to the S code as the value of the variable. Sophisticated aliasing based on these references can be then implemented, deferring computations to the relational database server rather than transferring enormous volumes of data to R to have similar computations be performed locally.

CORBA Remote objects in CORBA[2] can be registered into hierarchical folders and made available to the different agents by name. Each of these folders can be attached and objects within them used directly within S computations as if they were local objects.

Java, Perl, Python, XSL Interfaces to other languages such as Perl, Python, Java, JavaScript have been made possible as part of the Omegahat project so that the R interpreter can be running concurrently within the same process as the interpreters for these other languages. In this way, there are two groups of variables and it is convenient for the programmer/user to be able to treat variables in Python as local variables in R, and vice-versa. We can attach hashtables and basic environments/bindings from these languages to the R search path to effect this local view of the remote variables.

[1] The details are a more complicated due to efficiency concerns, etc. but this is the conceptual model.

[2] The Common Object Request Broker Architecture

Directories One can define object tables in R that read and write their contents from and to files within a directory. The objects can be stored in different formats, including the native R binary format and also other formats such as XML, SAS, S-Plus, Octave, Matlab, etc. This reduces the need to maintain separate copies of the same data and manage the synchronization and updating between them.

Archives One can combine variables into a single file or object as an archive and yet read individual elements without loading all of the objects. Tar, zip, hdf5 formats are easy to add to R's object table mechanism using R code only. More efficient versions can gradually be added by implementing the details in C code.

URIs One can provide access to directories within FTP and HTTP servers and make the entries available as variables. The particular format of the objects can be encoded dynamically in the R code that reads these objects, behaving like a dynamic filter.

Type Databases We can assign create databsaes that stors types for variables and ensure that when values are assigned to that variable, they have the correct type. This form of type-checking extends the S language and simplifies certain types of software development. (See **TypedListTable.S** in the **examples/** directory.)

2 The Interface

There are 5 basic "methods" defining how the S interpreter interacts with a user defined table. These are `exists()`, `get()` `assign()`, `remove()` and `objects()`. Those familiar with S will understand what each of these does as they mimic the global, system functions but apply only to the particular table.

exists returns a logical value indicating whether the table has a binding for the specified name.

get retrieve the value associated with the given name.

assign bind the specified value to the given name.

remove discard the binding for the given name, removing it from the table. This means that a subsequent call to `exists()` before a call to `assign()` with the same name will return **F**.

objects return a vector of the names of the bindings in the table, i.e. all the variable names for that table.

Having identified just 5 methods by which a table can provide sufficient information to R to be used on the search path, we can implement such a table in native code by specifying C routines for each of these methods. The mechanism to create such an object and pass it to R is quite simple and typically follows these basic steps: create the table, specify the 5 methods, and attach to the search path.

The first thing to do is to create the table. This usually involves a call to a C routine using the *.Call()* interface. The routine first processes its arguments and constructs any data it needs for the life of the table (and to store across calls by the R engine to the different methods). Then it allocates memory for a C structure named `R_ObjectTable`, which is defined by including **R˙ext/RObjectTables.h**. It is this structure that contains the information about the methods R is supposed to call for each piece of information it needs from the table.

Currently, the `R_ObjectTable` is defined as

```
typedef struct _R_ObjectTable R_ObjectTable;

struct  _R_ObjectTable{
  Rboolean  active;

  Rdb_exists   exists;
  Rdb_get      get;
  Rdb_remove   remove;
  Rdb_assign   assign;
  Rdb_objects  objects;
  Rdb_canCache canCache;
```

```
  Rdb_onDetach onDetach;
  Rdb_onAttach onAttach;

  void     *privateData;
}
```

The fields of interest are all but the first one. The last field is a place to store any data that is needed by any of the methods. This might be the name of the directory associated with the table, a connection to a relational database server, a file descriptor to an open archive file, the IOR (inter-operable object reference) for the CORBA naming server folder, or even a collection of R functions which implement the different methods for the table. This table-specific data is available to each of the method routines due to the fact that each of these routines is called with the `R_ObjectTable` instance associated with that table. The methods are expected to dereference and cast the `privateData` field appropriately.

The other 8 fields are holders for the native (*C*) routines. The first 5 implement the 5 basic methods for the table. The `canCache()` field provides a way to allow R to gain some efficiency by caching variables and avoiding repeated, potentially expensive variable lookups. This is described below. And the additional two methods are called each time the table is either attached or detached from the R search path.

exists `Rboolean Rdb_exists(const char *const name, Rboolean *canCache, R_ObjectTable*);` This method is responsible for returning either **T** or **F** depending on whether the table currently has a binding for the variable identified by `name`.

get `SEXP Rdb_get(const char *const name, Rboolean *canCache, R_ObjectTable *);` This returns the (current) value associated with the variable in the table identified by the name `name`.

remove `int Rdb_remove(const char * const name, R_ObjectTable *);` This discards the binding in this table for the specified variable identified by `name`. The return value should be non-zero if there was a binding for this variable in this table. Otherwise, it should be 0.

assign `SEXP Rdb_assign(const char * const name, SEXP value, R_ObjectTable *);` This binds the object `value` to the variable name `name` in this table. The return value should be the argument `value` so that assignments can be serialized, as in

```
x <- y <- 10
```

objects `SEXP Rdb_objects(R_ObjectTable *);` This returns a character vector with the names of each of the variables currently stored in the table. The only argument to the routine is the `R_ObjectTable` itself.

canCache `Rboolean Rdb_canCache(const char *const name, R_ObjectTable*);` If this is non-NULL, R will call this each time it explicitly retrieves an object from this table. The method can then control whether R is entitled to cache that value in its global cache of variables or whether it must perform the lookup for this variable each time. The primary reason for not caching the value of a variable is that the contents of the table may be externally modified, for example by another user or process. For example, if the table's contents are read from a directory, the user can add or remove an object from that directory using basic shell commands rather than explicit R commands to *assign()* and *remove()* values. Similarly, a new CORBA server may be registered with the naming service using an existing name and we would not see the new one if the value were cached. In such cases, we cannot use R's cached value as it will not necessarily know of such external changes. This method will usually respond with **T**or **F**for all variables in the table. However, the way in which it is called allows the table to permit caching for some variables and not others. Due to the potential performance gains from caching, this ability to identify certain variables as being essentially invariant from context may be important, while volatile or frequently updated variables would not be cached.

In the future, we may require this method to respond to a **NULL** argument to determine whether we can cache all the values. R can use this to perform the caching itself to make lookups more efficient and centralize the code for all table types.

attach `void Rdb_onAttach(R_ObjectTable *);` This routine is called each time the table is attached to the search path. This can be used to create and initialize data for use in the other methods, such as work space, cacheing the names of variables, ... For example, we might create a temporary directory into which files are extracted from archives, downloaded from HTTP servers; or obtain a file descriptor by opening an archive file.

detach `void Rdb_onDetach(R_ObjectTable *);` This routine is called each time the table is removed (or detached) from the search path. This can do any cleanup operations it wants such as releasing memory in the `privateData` field, removing temporary directories used to cache values, ...

These 8 different methods provide sufficient control for the table to handle very different types of remote name-value containers. They encapsulate the details of how variables are stored in the tables, yet provide sufficient information for R to access their contents.

In the near future, we may use the `canCache()` routine to determine if we can cache *all* variables for the table. In such cases, we will arrange for R to compute the "fixed" contents of the table when it is attached or the first time it is used and then work directly from the cached values.

The first field of the `R_ObjectTable` allows R to temporarily make the table inactive so that R will not ask it for objects even though it is an element of the search path. This is needed for technical purposes (to avoid infinite recursion - see **??**), but it may also prove useful in the future to avoid detaching and re-attaching.

To create a user-defined table, one allocates the `R_ObjectTable` instance and sets the fields to point to the appropriates routines. Then one returns a pointer to this `R_ObjectTable` as an R object. This is done using the external pointer mechanism, as in the following C code:

```
R_ObjectTable *rdb = (R_ObjectTable*) calloc(sizeof(R_ObjectTable));
 ...
return(R_MakeExternalPtr(rdb, Rf_install("UserDatabase"), NULL))
```

It is important that this object be an instance of the class *Database*. So when we return this object from the C code, we assign this class to it.

2.1 Attaching the Database

At this point we have a user-defined table but cannot do much with it unless we define methods to access its contents. The next step is to attach it to the search path. To do this, we can simply call *attach()*.

```
db <- createDB()
attach(db, name="MyDatabase")
```

What happens at this step involves the R internals. However, it is quite simple. Essentially, we allocate a new R object (a SEXP) which is of type **ENVSXP**, an environment holder. This has a reference or link to the next element in the search path so that R can follow these links to traverse the elements of the search path. Unlike the usual objects in the search path, user-defined tables do not have an associated hashtable or frame in which the name-value pairs are stored. Instead, the `hashtab` field in the environment object is used to the R object that contains the pointer to the `R_ObjectTable` that contains the pointers to the different routines implementing the user-defined table.

At this point, the usual operations are available to us. We can examine the contents of the database using the regular function *objects()*. Similary we can check whether a variable is available using *exists()* and retrieve a value with *get()*. We can assign to the database using *assign()* and similarly discard variables using *remove()*.

A simple example of a fixed database implemented entirely in C is provided in **SimpleTable.c** and an S-level function to create such an object is available in *SimpleTable.S()*.

```
> db <- newTable()
> attach(db, pos=3, name="simple")
attaching a simple table instance
> search()
[SimpleTable_get] Looking for search
[1] ".GlobalEnv"   "tb"           "simple"       "Autoloads"    "package:base"
> objects(3)
```

```
[1] "i"    "num"  "logi"
> i
[SimpleTable_get] Looking for i
[1] 0 0 0
> num
[SimpleTable_get] Looking for num
[1] 0 0
> logi
[SimpleTable_get] Looking for logi
[1] FALSE FALSE FALSE FALSE FALSE
> x
[1] 1 2 3 4
> find("logi")
[SimpleTable_get] Looking for find
[SimpleTable_get] Looking for gsub
[SimpleTable_get] Looking for logical
[SimpleTable_get] Looking for vector
[SimpleTable_get] Looking for ls
[SimpleTable_get] Looking for grep
[SimpleTable_get] Looking for which
[SimpleTable_get] Looking for dim
[SimpleTable_get] Looking for is.null
[1] "simple"
```

3 Implementing Tables with S Code

We have described the *C*-level mechanism by which any user can define their own table type. One need only provide routines to implement the different methods of interest. One can use the `privateData` field to store data that persists across these routines. This will satisty some developers who want access at this low-level in order to connect to 3^{rd}-party C code and to provide highly efficient access to variables. For the rest of us, we don't relish the opportunity to program in C, but prefer to use higher level languages such as R. After all, we are adding the functionality to R, so it is likely that we would like to use R to implement the methods for the table in R itself. And there are two basic styles to do this.

3.1 Functions & Closures

The basic user-defined table package provides a collection of *C* routines that implement a table by calling the corresponding function from a list provided to it and stored in the `privateData`. Typically, one provides a list of functions for each of the 8 different methods in the `R_ObjectTable`. Then one can attach the resulting database object. When R interacts with it, it calls these "generic" *C* routines that call the corresponding function. These functions are responsible for implementing the functionality of the table.

Let's consider a simple example. We will create a user-defined table which has fixed contents consisting of 3 variables. We will provide the *objects()*, *get()* and *exists()* methods. We can omit the *remove()* and *assign()* methods since the database is fixed and read-only. For simplicity, we can also ignore the *onAttach()*, *onDetach()* and *canCache()* methods.

```
fixedTable <-
function()
{
  values <- list(x = 1:4, y = letters[1:5], cube = function(x) { x^3 } )

  objects <- function() {
    names(values)
```

```
  }

  get <-  function(name) {
    values[[name]]
  }

  exists <- function(name) {
    return(!is.na(match(name, names(values))))
  }

  return(list(assign=NULL,
              get = get,
              exists = exists,
              remove = NULL,
              objects = objects
              ))
}
```

(Note that the order is currently important.)

Now we can use this

```
> tb <- newRClosureTable(db = fixedTable())
> attach(tb, name="foo")
> objects(2)
[1] "x"    "y"    "cube"
> cube(3)
[1] 27
```

Using closures, as above, allows us to share variables locally between the different functions and to provide mutable state across calls to the different functions. To do this, all one needs to do is define the functions within a top-level function and return them as a list. To change the value of shared variable, use the global assign operator `<<-`.

For technical reasons, the internal C code for these types of databases will call the `exists()` function when looking for an object as it is impossible for an S function to return anything other than an S object. As a result, we cannot tell the difference between a variable not having a binding and having the value `NULL`.

3.2 S4 Methods

In the above setup, we specify the actual S functions to be called by the C routines implementing the table functionality. Another approach is to leave R to find the appropriate function when it needs it. We allow R to use its usual function lookup mechanism, namely by finding the appropriate method for the call, usually depending on the class of the database. In this approach, the developer of the user-defined doesn't have to do much to create the low-level internal `R_ObjectTable` object. She can do this by calling the C routine `newRFunctionTable()` using the *.Call()* mechanism. The developer creates an S object that represents the database. This is specified as the argument in the call to `newRFunctionTable()`.

At this point, we have an attachable object. When any of the 8 different methods are called, the underlying C code will call the associated R function. For compatability with S4, these functions are named

dbexists

dbread

dbwrite

dbremove

dbobjects

dbcanCache

dbattach

dbdetach

These are generic functions. The developer can define methods for these functions. The first argument to each of these is the database on which the operation is called. When the internal mechanism calls these functions, it does so by passing the database object used in the call to `newRFunctionTable()`.

The file **TarTable.S** provides (S3-style) methods for a table that use a tar file to store regular objects created using *save()*, and desrializes them using *load()* after extracting them from the tar file. One can create an example tar file using the command

```
make Data1.tar.gz
```

Then we can attach this archive file and access its entries as regular R variables.

```
> db <- tarObjectTable("Data.tar")
> db1 <- newRFunctionTable(db)
> attach(db1)
```

We can now use the usual tools to access the variables.

```
> objects(2)
 [1] "a" "b" "d"
> a
 [1]  1  2  3  4  5  6  7  8  9 10
> a
Cached value from tar file
 [1]  1  2  3  4  5  6  7  8  9 10
> get("b")
 [1] "a" "b" "c" "d"
> get("d", pos=2)
 [1] -1.12010933 -0.32923339  0.27270609  0.47075837 -2.96734931 -0.04350143
 [7] -0.23723288  2.38727637  0.59666549  1.10805966
```

3.3 Restrictions on using S code

The ability to use S functions (or closures) to implement a user-defined table type makes it easy to experiment with different interfaces and approaches. There is however one minor restriction for tables implement with S functions. Specifically, the S code used to implement the different methods cannot refer to objects – either functions or any other type of data – that are located in that table. In practice, this is almost never an issue and so is not a practical restriction.

What is that we mean by saying that code used to implement a table cannot refer to objects on that table? Let's look at a potential example. Suppose we implement the *get()* method for the table with code something like

```
tarGet <-
function(name, db)
{
  getTarFile(name, db$fileName)
}
```

and *getTarFile()* is a function that extracts the specified entry from the target archive file. Then when we evaluate the call

```
  get("foo", tarDB)
```

R will eventually call the *tarGet()* function above. When it evaluates the body of that function, it will search for the function *getTarFile()*. If that function is a variable in the table *tarDB*, R will attempt to retrieve it using the `get()` method for the user-defined table. Hence, we will end up calling *tarGet()* and we will start the cycle again, looking for *getTarFile()*, and so on.

What we do to solve this problem is to temporarily make the user-defined table inactive during the evaluate of the internal `get()` method. In this way, when we look for the *getTarFile()* function, we will not find it in the user-defined table and nor will we look for it there. Hence we will avoid the infinite recursion.

Why is this restriction not important? In general, we will store data in these tables and we can think of the tables as providing access to different locations. In this view, the functionality to implement the table is not an element in the table, but provided by a package that is attached earlier in the R session. For example, suppose we have two instances of the "tar" table and attach them both. Why would we have the function *getTarFile()* in both tables rather than a single version of it in a package. This aids the separation of the functionality to manipulate the contents of an archive from the ability to attach it to R's search path.

Relying on a function being available on the database implemented using that function can be very dangerous. If the order in which the contents are accessed is not exactly as the developer predicts, it is possible that one will not be able to use the table and access its contents. So such dependencies are typically not part of a good design model.

There are occassions where it is convenient to separate functionality used by two or more of the methods used to implement a table's methods and not to create a separate package. In these cases, one might naturally want to have the function in the table itself. However, there is a natural way around this in R. One can use closures to share variables, including functions, between different functions. Let's consider an example in which the *get()* and *remove()* method both call a filter to transform the S name of a variable to an internal form (such as mapping it to a file name or column name in an SQL database).

```
dbget <- function(name, db) {
   load(filter(name, db))
}

dbremove <- function(name, db) {
   file.remove(filter(name, db))
}
```

Where is the filter function defined? By defining these *dbget()* and *dbremove()* functions within an other function, the bodies of these functions have access to other variables defined within that outer function. In other words, we can define the following function generator.

```
filteredDBHandlers <- function() {
   filter <- function(name, db) {
     gsub("\\\.", "_", name)
   }

   dbget <- function(name, db) {
      load(filter(name, db))
   }

   dbremove <- function(name, db) {
      file.remove(filter(name, db))
   }

   return(list(get=dbget, remove=dbremove, ..... ))
}
```

(where the . . . indicate the other methods for a table). Then, we can use the different functions returned as elements in the list from this function to specify the different table methods. For example,

```
db <- newRClosureTable(filteredDBHandlers)
class(db) <- c("FilteredDatabase", "Database")
```

When R performs the lookup for the *filter()* function when evaluating the bodies of either the *dbget()* or *dbremove()* functions, it does this in a slightly different way. It looks in the nested environments of the function being called (i.e. *dbget()* and *dbremove()*) and searches these environments for the variables. Hence it will find the *filter()* within the containing (or parent) environment of each of these functions. This will avoid the need to do the lookup in the user-defined table itself.

In general, closures are very useful for user-defined tables. First, they allow the hiding of utility functions. More importantly, they provide a mechanism for storing data that changes across calls to the different methods of a user-defined table. In other words, they offer mutability. See the example in **RClosureTable.S** for how this can be used.

4 Extensions

- Speed is an issue when one changes the way that variables are accessed by the interpreter. There is a significant danger of degrading performance even if the user-level tables are not activated. Simple measurements taken from running '`make check`' for R suggest that there is no difference in run-time.

- Caching is used in R. When a table is attached, we discard any value in the central cache that has a binding in the newly attached table. For fixed tables, this leads to a well-defined and correct operation. For a table whose contents are dynamic and asynchronously modified, the semantics are not the same. A simple example shows the problem. Suppose we have cached the variable *x* in the global cache. Then we attach a table that does not have such an entry. Then we look for the variable *x*. We will find the cached value. However, in the intervening time, the contents of the database might have changed and may have an entry for *x*. So in this case, we will get the wrong value. To be absolutely exact, we may have to flush the entire cache. Alternatively, we might want to allow the code for the table itself to provide a list of all possible variables and flush these if they exist. Such a list will not be meaningful in most situations.

- In the future, we would like to add the ability to look for objects of a certain type. For example, if we are looking for a function named *c()*, then we want to allow tables which are known not to contain functions to decline without any expense. This arises for example if we had a database table attached with a column named *c*. We would perform the remote query to get the values, convert them to an R object, and then return it only to discover that it was not of interest.

- It will be difficult but desirable to allow user-defined tables to be attached at position 1 of the search path. The semantics of existing computations may not permit this, but it is worth looking at. The primary motivation is to allow the default assignments go to a directory-based table so that top-level assignments are committed immediately to a persistent form. This would be helpful to avoid losing data when crashes occur. This might be feasible by changing the default assignment database rather than requiring the more general attachment at position 1.

- John Chambers and David James have expressed interest in having metadata associated with certain types of databases. By constructing this extensible table mechanism, we hope that these can be implemented at the user level rather than requiring changes to the R internals.