A New Serialization Mechanism For R

Luke Tierney
Department of Statistics and Actuarial Science
University of Iowa

June 25, 2003

This note describes a new saved workspace format and serialization mechanism that has been added to R. This format is now the default save format in the development version of R. The `save` function now takes a `version` argument; specifying `version = 2` causes the new format to be used and `version = 1` requests the format used through 1.3.1.

A PostScript version of this document[1] is also available.

1 Background

The main reason a new format is needed is to support name spaces. While the details of name spaces are not yet complete, the following seem clear at this point:

- Name spaces will be represented as environments. Registered name spaces will have some identifying information, such as the package they are associated with and perhaps some information to help locate the package for loading. This information can be encoded in a character vector.
- When `save` encounters a registered name space while saving a node graph, it should not save the contents of the name space. Instead, it should record the identifying information in the saved work space.
- When `load` finds a reference to a name space in a work space it is loading, it can use this identifying information to locate the work space, either within a table of loaded name spaces or by searching for and then loading an appropriate package.

The `save` mechanism needs to be modified to support this handling of name spaces.

Since a revision of the save format seems necessary to support name spaces, there are some other issues that can be addressed at the same time:

- String `NA` values should be preserved on saving; currently they are not.

[1] See URL `http://www.stat.uiowa.edu/~luke/R/serialize/serialize.ps`.

- Currently strings with embedded null characters are truncated when saved. A lot of internal code assumes that `strlen` gives the correct string length, so strings with embedded nulls are not of much use, but it would be nice to preserve them correctly in case this changes.

- External pointer and weak reference objects are currently not treated as references. That is, if `x` and `y` refer to the same external pointer object and are saved, then `x` and `y` will refer to different external pointer objects after loading them from a saved work space. This is not a major issue for standard uses, since both external pointers and weak references are only meaningful within a session. But it may cause problems for non-standard uses.

- The current save format is quite inefficient in its use of space when saving a node: a number of items that are usually `NULL` are always written out, and several items that need only one or a few bits are each written as full integers. This is not an issue for saving data, which usually involves few nodes and large vector data segments. But it is a problem for work spaces containing mostly code, such as those created by installing packages with a `--save` option.

- Currently when the environment frame of a package is saved it is saved as an ordinary environment. For an attached package environment this means all package environments in the search list after it are also saved. This is clearly not desirable, and is one reason why the `library` code tries to fix up the environments of closures. Despite this it is possible, though very unlikely, that a package environment can get saved. It might be useful to prevent this in the save code by handling packages along similar lines as name spaces.

- User code might be able to benefit from having a mechanism for customizing the serialization mechanism, in particular for reference objects. For example, it might be useful to be able to arrange for an external pointer to try to reinitialize itself when it is restored from a work space.

- Currently the `save` function is responsible both for serializing an object, i.e. turning it into a stream of bytes, and writing the serialized stream to a file. It would be useful to separate the two. This would allow sending serialized objects over a socket connection, saving to a compressed file, or using some other form of external storage such as a data base of some sort.

- When long dotted pair lists, such as those in an environment, are written to and read from a work space there is the possibility of a stack overflow since recursion is used. Recursion for loading is hard to avoid with the current format since the `ATTRIB` field is written after the `CDR` field. By writing the `CDR` last it is easier to arrange for a non-recursive algorithm to be used for both reading and writing.

The new serialization approach attempts to address some of these issues. In looking at the new code there are two major questions that need to be answered:

1. Is the serialization format adequate to do the things we need in the sort term at least?

2. Is the code that uses the format adequate?

Resolving the first question is more important than the second. For example, exactly how to restore a name space or a package based on the information provided can be resolved later as long as the format allows sufficient information to be stored.

To take full advantage of the options offered it would be useful to have a way for users to customize the action of `save.image` and the way a work space is saved on exit. This would allow users to request, for example, that their work space be compressed or perhaps that it be stored in a data base. This has not been addressed yet.

2 Interfaces

2.1 R Interface

The only direct R interface provided in the core distribution for now is through the `save` function: calling it with `version=2` produces a work space in the new format.

An experimental interface is available as a package `serialize`[2] with two functions, `serialize` and `unserialize`. The usage is

3a ⟨*serialize package usage* 3a⟩≡
```
serialize(object, connection, ascii = FALSE, refhook = NULL)
unserialize(connection, refhook = NULL)
```

The connection argument to `serialize` can be a open connection or `NULL`. If it is `NULL`, then `object` is serialized to a string and that string is returned as the value of `serialize`. Otherwise `object` is serialized to the connection and `NULL` is returned. The `connection` argument to `unserialize` can be an open connection or a string.

This interface may need some adjustment. It would be nice if we could use only proper connections, but the current version of text connections, in particular text output connections, doesn't seem quite adequate.

In terms of this interface,

3b ⟨*R session* 3b⟩≡ 8a▷
```
save(list=mylist, file="myRData", version = 2)
```

is roughly equivalent to

3c ⟨*save equivalent using serialize package* 3c⟩≡
```
conn<-file("myRData","wb")
writeChar("RDX2\n", conn, eos=NULL)
serialize(mylist,conn)
close(conn)
```

and loading this work space with `load` corresponds to

3d ⟨*load equivalent using serialize package* 3d⟩≡
```
conn<-file("myRData", "rb")
if (readChar(conn,5) == "RDX2\n") {
    val <- unserialize(conn)
    names <- names(val)
    for (i in seq(along=names))
```

[2] See URL `http://www.stat.uiowa.edu/~luke/R/serialize/serialize.tar.gz`.

```
        assign(names[i], val[[i]], envir = envir)
}
else
    ...
close(conn)
```

The `refhook` functions can be used to customize the handling of non-system reference objects (all external pointers, all weak references, and all environments except `.GlobalEnv`, name spaces, and package environments). If `serialize` is provided with a `refhook` function, then that function is called with each reference object before that object is written. If `refhook` returns `NULL` then the object is written in the usual way. Otherwise, `refhook` must return a character vector, and the character strings are saved (only the stings, no names or other attributes). If a serialization contains a value produced by a `refhook`, then it must be unserialized with a corresponding `refhook`. The `unserialize` hook is called with the reconstructed character vector as its argument, and should return whatever object the character vector indicates. Some examples of using this mechanism are given in Section 3.

2.2 C Interface

The `pstream` portion of names used in this interface is meant to be short for *persistent stream*.

The C interface uses structures to represent input and output serialization streams.

4a ⟨*serialization stream declarations* 4a⟩≡ 4b▷

```
typedef struct R_outpstream_st {
    ...
} *R_outpstream_t;
typedef struct R_inpstream_st {
    ...
} *R_inpstream_t;
```

The structure declarations are in `Rinternals.h`, and user code must provide storage for the structure, but it should not be used directly as its contents might change. Instead, it should be initialized by one of the initialization functions.

Two sets of higher level initialization functions are provided. One allows writing to an open `FILE *` pointer; this is used by the internal `save` code:

4b ⟨*serialization stream declarations* 4a⟩+≡ ◁4a 4c▷

```
void R_InitFileInPStream(R_inpstream_t stream, FILE *fp,
                         R_pstream_format_t type,
                         SEXP (*phook)(SEXP, SEXP), SEXP pdata);
void R_InitFileOutPStream(R_outpstream_t stream, FILE *fp,
                          R_pstream_format_t type, int version,
                          SEXP (*phook)(SEXP, SEXP), SEXP pdata);
```

The format types are

4c ⟨*serialization stream declarations* 4a⟩+≡ ◁4b 5a▷

```
typedef enum {
    R_pstream_any_format,
    R_pstream_ascii_format,
```

```
    R_pstream_binary_format,
    R_pstream_xdr_format
} R_pstream_format_t;
```

An explicit format must be provided for output. For input, `R_pstream_any_format` can be used to indicate that any format is acceptable; if an explicit format is provided on input then an error is raised if the actual stream format does not match the specification.

The second higher level interface is for connections:

5a ⟨*serialization stream declarations* 4a⟩+≡ ◁4c 5b▷

```
void R_InitConnOutPStream(R_outpstream_t stream, Rconnection con,
                          R_pstream_format_t type, int version,
                          SEXP (*phook)(SEXP, SEXP), SEXP pdata);
void R_InitConnInPStream(R_inpstream_t stream,  Rconnection con,
                         R_pstream_format_t type,
                         SEXP (*phook)(SEXP, SEXP), SEXP pdata);
```

The connection must be open in the appropriate direction, and must be binary for binary streams.

The hook mechanism is analogous to the `refhook` mechanism in the R interface. For output, `phook` is called with the reference object and `pdata` as its arguments, and should return `R_NilValue` or an `STRSXP` of length at least one. For input, an `STRSXP` is supplied and the appropriate object should be returned.

A lower level initialization mechanism can be used to build higher level ones. The lower level mechanism requires two routines, one for handling single characters, used mostly for ascii streams and for writing header information, and one for handling blocks of bytes.

5b ⟨*serialization stream declarations* 4a⟩+≡ ◁5a 5c▷

```
void R_InitInPStream(R_inpstream_t stream, R_pstream_data_t data,
                     R_pstream_format_t type,
                     int (*inchar)(R_inpstream_t),
                     void (*inbytes)(R_inpstream_t, void *, int),
                     SEXP (*phook)(SEXP, SEXP), SEXP pdata);
void R_InitOutPStream(R_outpstream_t stream, R_pstream_data_t data,
                      R_pstream_format_t type, int version,
                      void (*outchar)(R_outpstream_t, int),
                      void (*outbytes)(R_outpstream_t, void *, int),
                      SEXP (*phook)(SEXP, SEXP), SEXP pdata);
```

Once a stream is initialized, it is read and written by

5c ⟨*serialization stream declarations* 4a⟩+≡ ◁5b

```
void R_Serialize(SEXP s, R_outpstream_t ops);
SEXP R_Unserialize(R_inpstream_t ips);
```

3 Examples

Some of these examples use an `assert` function that can be defines as

5d ⟨*assert function* 5d⟩≡
```
assert <- function(expr)
    if (! expr)
        stop(paste("assertion failed:", deparse(substitute(expr))))
```

3.1 Serialization to Files

The following writes an object to a file and reads it back in:

6a ⟨*file examples* 6a⟩≡ 6b▷
```
x<-list(1,2,3)
f<-file("sertmp", open="wb")
serialize(x,f)
close(f)
f<-file("sertmp", open="rb")
y<-unserialize(f)
close(f)
assert(identical(x, y))
```

By using the right magic number header, we can create a saved work space that `load` can read:

6b ⟨*file examples* 6a⟩+≡ ◁6a 6c▷
```
f<-file("sertmp","wb")
writeChar("RDX2\n", f, eos=NULL)
serialize(list(x=1,y="2"),f)
close(f)
load("sertmp")
assert(x == 1 && y == 2)
```

Similarly, we can read a file written by `save` with `version = 2`:

6c ⟨*file examples* 6a⟩+≡ ◁6b
```
x<-list(1,2,3)
save("x", file="sertmp",version=2)
f<-file("sertmp", "rb")
readChar(f,5)
y<-unserialize(f)
close(f)
assert(identical(x, y$x))
```

3.2 Serialization to Strings

When called with `NULL` for the connection argument `serialize` will serialize to a string. The string is likely to contain embedded null characters unless `ascii=TRUE` is specified. `unserialize` can handle this properly, but since other aspects of R can't it might be worth considering an alternate form of return value for binary serializations to memory.

6d ⟨*string examples* 6d⟩≡ 7a▷
```
x<-list(1,2,3)
y<-unserialize(serialize(list(1,2,3),NULL))
assert(identical(x, y))
```

Sharing of environments is preserved within a serialization, but identity is not preserved by serialization:

7a ⟨*string examples* 6d⟩+≡ ◁6d 7b▷

```
e1 <- new.env()
e2 <- new.env()
y<-unserialize(serialize(list(e1,e2,e1,e2),NULL))
assert(identical(y[[1]],y[[3]]))
assert(! identical(y[[1]],e1))
```

We can use the `refhook` mechanism to attempt to preserve identity as well (but just in this artificial setting where we are saving from and loading into the same process—in general this is of course impossible):

7b ⟨*string examples* 6d⟩+≡ ◁7a

```
outhook <- function(e) {
    if (identical(e,e1)) "e1"
    else if (identical(e,e2)) "e2"
    else NULL
}
inhook <- function(n) get(n)
y<-unserialize(serialize(list(e1,e2,e1,e2),NULL,refhook=outhook),
               refhook=inhook)
assert(identical(y[[1]],e1))
```

3.3 A GDBM interface

For the remaining examples we need a very simple (minded) interface to the GNU `dbm` library. This library implements a simple key/value data base. Unlike the original `dbm` library, GNU `dbm` does not limit the size of keys or values.

The interface is quite inefficient since it uses a fresh connection for each operation, but this should be adequate for simple illustrative purposes. The interface, available as the `gdbm` package[3], is:

7c ⟨*gdbm interface usage* 7c⟩≡

```
gdbmNew(name)
gdbmInsert(name, key, value)
gdbmFetch(name, key)
gdbmExists(name, key)
gdbmDelete(name, key)
gdbmList(name)
```

Some examples:

7d ⟨*gdbm examples* 7d⟩≡

```
gdbmNew("mydb")
gdbmInsert("mydb","bob","dog")
gdbmInsert("mydb","fred","cat")
gdbmList("mydb")
```

[3]See URL `http://www.stat.uiowa.edu/~luke/R/serialize/gdbm.tar.gz`.

```
gdbmExists("mydb","bob")
gdbmExists("mydb","joe")
gdbmFetch("mydb","bob")
gdbmDelete("mydb","bob")
gdbmExists("mydb","joe")
```

3.4 Storing Base in a GDBM Data Base

Currently R reads the whole base library on startup. This takes time and uses a significant amount of memory. For embedded applications that might use just a small fraction of R in a given session, it might be useful to avoid this overhead by storing at lest the functions in the base environment in a data base and arranging for the code that finds symbol values to search the data base if a definition is not in memory. In low memory settings, purging functions in base would also be an option.

Implementing this idea would require surgery on `envir.c`. But we can partially simulate it by replacing closures in base by promises that load the closure from a data base. To measure the effect, we start with a regular R session and look at the memory usage:

8a ⟨*R session* 3b⟩+≡ ◁3b 9b▷

```
> gc()
         used (Mb) gc trigger (Mb)
Ncells 196188  5.3     407500 10.9
Vcells  37757  0.3     786432  6.0
```

To start, we need to store the closures in base in a data base. Since this simple approach cannot deal with shared environments, only the closures with `.BaseNamespaceEnv` as their environment are stored.

8b ⟨*storing base in a gdbm data base* 8b⟩≡ 9a▷

```
# create the data base
gdbmNew("base")
# fill it in
for (name in ls(NULL, all=TRUE)) {
    val <- get(name, env=NULL)
    if (typeof(val) == "closure" &&
        identical(environment(val), .BaseNamespaceEnv))
        gdbmInsert("base", name, serialize(val, NULL))
}
# check it
for (name in ls(NULL, all=TRUE)) {
    val <- get(name, env=NULL)
    if (typeof(val) == "closure" &&
        identical(environment(val), .BaseNamespaceEnv)) {
        if (! gdbmExists("base", name) ||
            ! identical(val, unserialize(gdbmFetch("base",name))))
            stop(name)
    }
}
```

Now we can replace all closures in base by promises that load them as needed:

9a ⟨*storing base in a gdbm data base* 8b⟩+≡ ◁8b

```
wrap<-function(name) {
   name <- name # need to force evaluation!
   delay(unserialize(gdbmFetch("base",name)),env=environment())
}
for (i in gdbmList("base")) assign(i, wrap(i), env=NULL)
```

To see the effect, we can again run a `gc`:

9b ⟨*R session* 3b⟩+≡ ◁8a 10c▷

```
> gc()
         used (Mb) gc trigger (Mb)
Ncells 41995  1.2     350000  9.4
Vcells 18345  0.2     786432  6.0
```

Memory usage has dropped from 5.6Mb to 1.4Mb. The promises do take up some space, but even that could be eliminated by making a modification in `envir.c`.

Using a data base for persistent storage of R code seems like an idea worth exploring in more depth. GDBM is one option for the data base. GDBM ports are available for almost all, if not all, platform where R runs, including Windows and classic Mac OS, so this may be a good default choice. Other data bases may work just as well and may in some cases be more suitable, so allowing a mechanism for choosing the data base is probably a good idea.

3.5 Preserving Shared Environments in GDBM Serialization

Allmost all the closures in base have `.BaseNamespaceEnv` as their environment, but a few do not. The simple approach of the previous section would fail if two closures shared a non-global environment since separate serializations would not preserve that sharing. The `refhook` mechanism can be used to overcome this problem. This section provides a simple illustration of how this can be done. The code presented here is available as package `shelf`[4]. The name is taken from a similar facility available in Python (though Python's facility does *not* try to preserve sharing across entries, just within entries, if I understand it correctly).

3.5.1 Some Utilities

To start off we need a few utilities. The first takes an environment and returns a named list of the name/values:

9c ⟨*shelf utilities* 9c⟩≡ (14c) 10a▷

```
envlist <- function(e) {
    names <- ls(e, all=TRUE)
    list <- lapply(names, get, env=e, inherits=FALSE)
    names(list) <- names
    list
}
```

[4] See URL `http://www.stat.uiowa.edu/~luke/R/serialize/shelf.tar.gz`.

The second is sort of an inverse—given a named list and an environment, it adds the contents of the list to the environment.

10a ⟨*shelf utilities* 9c⟩+≡ (14c) ◁9c 10b▷

```
listIntoEnv <- function(list, e) {
    names <- names(list)
    for (i in seq(along = names))
        assign(names[i], list[[i]], env = e)
}
```

Next, we need a means of creating unique names for a set of environments—a little data base of sorts. Given an environment, we need to be able to ask for the name of the environment if it is already in the data base. If it is not, we need to insert it and generate a name for it. The generated names are of the form `env::<index>`. (The `getenv` function is not actually needed.)

10b ⟨*shelf utilities* 9c⟩+≡ (14c) ◁10a 11a▷

```
envtable <- function() {
    idx <- 0
    envs <- NULL
    enames <- character(0)
    find <- function(v, keys, vals)
        for (i in seq(along=keys))
            if (identical(v, keys[[i]]))
                return(vals[i])
    getname <- function(e) find(e, envs, enames)
    getenv <- function(n) find(n, enames, envs)
    insert <- function(e) {
        idx <<- idx + 1
        name <- paste("env", idx, sep="::")
        envs <<- c(e, envs)
        enames <<- c(name, enames)
        name
    }
    list(insert = insert, getenv = getenv, getname = getname)
}
```

Some examples:

10c ⟨*R session* 3b⟩+≡ ◁9b 13b▷

```
> et<-envtable()
> e<-new.env()
> et$getname(e)
> et$insert(e)
[1] "env::1"
> et$getname(e)
[1] "env::1"
```

3.5.2 Creating Back End Connections

The mechanism described here can in principle be used with different back end key/value data bases. The default will be a GDBM data base, but other options are possible. For a GDBM data base we need a writer connection that provides an `insert` method,

11a ⟨*shelf utilities* 9c⟩+≡ (14c) ◁10b 11b▷

```
makeGdbmWriter <- function(file) {
     force(file)
     gdbmNew(file)
     list(insert = function(name, value) gdbmInsert(file, name, value))
}
```

and a reader connection that provides `list`, `fetch`, and `exists` methods:

11b ⟨*shelf utilities* 9c⟩+≡ (14c) ◁11a

```
makeGdbmReader <- function(file) {
     force(file)
     list(list = function() gdbmList(file),
          fetch = function(name) gdbmFetch(file, name),
          exists = function(name) gdbmExists(file, name))
}
```

3.5.3 Creating and Opening a Shelf

Now we can create a data base from a named list of values. The result is called a shelf, a term used by Python for a similar facility. The shelf stores its data in a gdbm data base but with slightly mangled keys. A variable `x` is stored under the key `var::x`, and environments referenced by values are stored under the generated environment keys with an `env::` prefix.

11c ⟨*creating a shelf* 11c⟩≡ (14c)

```
makeShelfWriter <- function(db,  ascii = TRUE) {
     if (is.character(db))
          db <- makeGdbmWriter(db)
     table <- envtable()
     ser <- function(val)
          serialize(val, NULL, ascii = ascii, refhook = envhook)
     envhook <- function(e) {
          if (is.environment(e)) {
               name <- table$getname(e)
               if (is.null(name)) {
                    name <- table$insert(e)
                    data <- list(bindings = envlist(e),
                                 enclos = parent.env(e))
                    db$insert(name, ser(data))
               }
               name
          }
     }
     putvar <- function(name, val, prefix="var", sep="::") {
          key <- paste("var", name, sep="::")
          db$insert(key, ser(val))
     }
     list(putvar = putvar)
```

```
}

makeShelf <- function(list, db, ascii = TRUE)
{
    names <- names(list)
    if (length(names) != length(list))
        stop("must provide a named list of values")
    s <- makeShelfWriter(db, ascii)
    for (i in seq(along=names))
        s$putvar(names[i], list[[i]])
}
```

To get a listing of the variables in a shelf, we can use

12a ⟨*listing a shelf* 12a⟩≡ (14c)

```
listShelf <- function(db) {
    if (is.character(db))
        db <- makeGdbmReader(db)
    prefpat <- "^var::"
    sub(prefpat, "", grep(prefpat,db$list(), value=TRUE))
}
```

We need a way of retrieving values from a shelf that insures that sharing of environments is maintained for values retrieved separately. We do need some connection between retrievals to allow this, and that connection is provided by a shelf connection object, which is created and returned by `openShelf`. Sharing is preserved for values retrieved from the same shelf connection.

12b ⟨*opening shelf* 12b⟩≡ (14c)

```
openShelf <- function(db) {
    if (is.character(db))
        db <- makeGdbmReader(db)
    envenv <- new.env(hash = TRUE)
    varkey <- function(name) paste("var", name, sep="::")
    fetch <- function(name)
        unserialize(db$fetch(name), refhook = envhook)
    envhook <- function(n) {
       if (exists(n, env = envenv, inherits = FALSE))
           get(n, env = envenv, inherits = FALSE)
       else {
           e <- new.env(hash = TRUE)
           assign(n, e, env = envenv) # MUST do this immediately
           data <- fetch(n)
           parent.env(e) <- data$enclos
           listIntoEnv(data$bindings, e)
           e
        }
    }
    list(getvar = function(name) fetch(varkey(name)),
         exists = function(name) db$exists(varkey(name)),
         list = function() listShelf(db))
```

```
}
```

As a side note, having `parent.env<-` available at the R level seems like a really bad idea because of the potential for real serious mischief, like clobbering the search list and totally confusing the internal global cache mechanism. But the facility it provides is essential in this case since the environment *must* be created and registered before its contents and parent are unserialized so that circular references to the environment are handled properly. Currently with `parent.env<-` available this can be handled in pure R code. But unless we find a good way of preventing inadvertent use, it would probably be good to get rid of this at the R level, and thus require a little bit of C code to implement this stuff.

Here are some examples. First create some environments with some ordinary variables and some references to each other, and place these in a shelf:

13a ⟨*simple shelf example* 13a⟩≡

```
e1<-new.env()
e2<-new.env(parent = e1)
listIntoEnv(list(x=2,y=3, v=3,ee=e2),e1)
listIntoEnv(list(x=e1,y=e2), e2)
makeShelf(list(x=e1,y=e1, z = 3), "mydb")
```

The `listShelf` and `gdbmList` functions show the variables in the shelf and the actual keys in the data base, respectively:

13b ⟨*R session* 3b⟩+≡ ◁10c 13c▷

```
> listShelf("mydb")
[1] "z" "y" "x"
> gdbmList("mydb")
[1] "env::2" "env::1" "var::z" "var::y" "var::x"
```

Now we can open the shelf and examine its contents. The pointer values in the environments show that sharing is being handled properly.

13c ⟨*R session* 3b⟩+≡ ◁13b

```
> s<-openShelf("mydb")
> s$list()
[1] "z" "y" "x"
> s$getvar("x")
<environment: 0x8f89ba4>
> s$getvar("y")
<environment: 0x8f89ba4>
> ls(s$getvar("x"))
[1] "ee" "v"  "x"  "y"
> get("ee", s$getvar("x"))
<environment: 0x8f89048>
> parent.env(get("ee",s$getvar("y")))
<environment: 0x8f89ba4>
```

3.5.4 Lazily Loading and Attaching a Shelf

Finally, we can combine the promise idea used earlier for closures in base to produce lazy load and attach functions for a shelf. The `loadShelf` functions adds bindings for all variables in a shelf to the specified environment. The bindings are promises that load the values on demand using a common connection created when the shelf is loaded.

14a ⟨*loading a shelf* 14a⟩≡ (14c)

```
loadShelf <- function(db, envir = parent.frame()) {
    s <- openShelf(db)
    wrap<-function(name) {
        name <- name # need to force evaluation!
        delay(s$getvar(name), env=environment())
    }
    for (n in listShelf(db))
        assign(n, wrap(n), envir = envir)
}
```

`attachShelf` creates an environment on the search list and fills it using `loadShelf`.

14b ⟨*attaching a shelf* 14b⟩≡ (14c)

```
attachShelf <- function(db, pos = 2, name) {
    if (missing(name)) {
        if (is.character(db))
            name <- paste("shelf", db, sep = ":")
        else name = "shelf:<no name>"
    }
    env <- attach(NULL, pos = pos, name = name)
    on.exit(detach(pos))
    loadShelf(db, env)
    on.exit()
}
```

3.5.5 Limitations

The shelf system presented here creates the shelf and then provides only read access. This is useful for code libraries—it could for example be used for packages that are installed with the `--save` option.

This simple shelf system does not allow values to be deleted or new or changed values to be inserted. Something along these lines would be useful of course. It does however raise a number of complicating issues that would need to be addressed. Some are quite standard, such as dealing with managing coherency for different shelf connections within a given process or even from separate processes. Others are more specific to this approach, such as garbage collection. If variables are deleted then there may be environment entries that are no longer needed. Some mechanism for removing these would be needed.

3.5.6 The Shelf Code

14c ⟨*shelf.R* 14c⟩≡

```
require(serialize)
require(gdbm)
⟨shelf utilities 9c⟩
⟨creating a shelf 11c⟩
```

⟨*listing a shelf* 12a⟩
⟨*opening shelf* 12b⟩
⟨*loading a shelf* 14a⟩
⟨*attaching a shelf* 14b⟩

3.6 Distributed Scope

Serialization can be used with sockets or with the Rpvm library to allow code and data to be transferred between processes or machines for distributed computing. One issue that arises is how to handle environments. The Obliq[5] system uses a notion of *distributed scope*: environments remain where they were created. This seems to provide for a nice high level model for distributed computation. It should be possible to use the `refhook` interface together with the active value ideas recently added to R to implement (a part of) this sort of thing, but I have not had a chance to try this yet.

4 The Serialization Algorithm

This section gives a bit more information on the serialization algorithm.

The algorithm uses a single pass over the node tree to be serialized. Sharing of reference objects is preserved, but sharing among other objects is ignored. The first time a reference object is encountered it is entered in a hash table; the value stored with the object is the index in the sequence of reference objects (1 for first reference object, 2 for second, etc.). When an object is seen again, i.e. it is already in the hash table, a reference marker along with the index is written out. The unserialize code does not know in advance how many reference objects it will see, so it starts with an initial array of some reasonable size and doubles it each time space runs out. Reference objects are entered as they are encountered.

This means the serialize and unserialize code needs to agree on what is a reference object. Making a non-reference object into a reference object requires a version change in the format. An alternate design would be to precede each reference object with a marker that says the next thing is a possibly shared object and needs to be entered into the reference table.

Adding new `SXP` types is easy, whether they are reference objects or not. The unserialize code will signal an error if it sees a type value it does not know. It is of course better to increment the serialization format number when a new `SXP` is added, but if that `SXP` is unlikely to be saved by users then it may be simpler to keep the version number and let the error handling code deal with it.

The output format for dotted pairs writes the `ATTRIB` value first rather than last. This allows `CDR`s to be processed by iterative tail calls to avoid recursion stack overflows when processing long lists. The writing code does take advantage of this, but the reading code does not. It hasn't been a big issue so far—the only case where it has come up is in saving a large unhashed environment where saving succeeds but loading fails because the `PROTECT` stack overflows. With the ability to create hashed environments at the user level this is likely to be even less of an issue now. But if

[5]See URL `http://www.luca.demon.co.uk/Slides.html#Distributed␣Mobile Computation␣in␣Obliq`.

we do need to deal with it we can do so without a change in the serialization format—just rewrite ReadItem to pass the place to store the CDR it reads. (It's a bit of a pain to do, that is why it is being deferred until it is clearly needed.)

CHARSXP are now handled in a way that preserves both embedded null characters and NA_STRING values.

The XDR save format now only uses the in-memory XDR facility for converting integers and doubles to a portable format.

The output format packs the type flag and other flags into a single integer. This produces more compact output for code; it has little effect on data.

Environments recognized as package or name space environments are not saved directly. Instead, a STRSXP is saved that is then used to attempt to find the package/name space when unserialized. The exact mechanism for choosing the name and finding the package/name space from the name still has to be developed, but the serialization format should be able to accommodate any reasonable mechanism.

A mechanism is provided to allow special handling of non-system reference objects (all weak references and external pointers, and all environments other than package environments, name space environments, and the global environment). The hook function consists of a function pointer and a data value. The serialization function pointer is called with the reference object and the data value as arguments. It should return R_NilValue for standard handling and an STRSXP for special handling. In an STRSXP is returned, then a special handing mark is written followed by the strings in the STRSXP (attributes are ignored). On unserializing, any specially marked entry causes a call to the hook function with the reconstructed STRSXP and data value as arguments. This should return the value to use for the reference object. A reasonable convention on how to use this mechanism is need, but again the format should be compatible with any reasonable convention.

Eventually it may be useful to use these hooks to allow objects with a class to have a class-specific serialization mechanism. The serialization format should support this. It is trickier than in Java and other reference based languages where creation and initialization can be separated–we don't really have that option at the R level.