A new save and restore file format for R

Honours project
December 1999

Chris K. Young

Table of Contents

Abstract

This paper describes a new file format that is used in R to save and restore data objects. It works by employing recursion, and while not at all compatible with the previous format, it is in the author's opinion a more suitable format for reflecting the recursive nature of R data objects. Environment dependency loops and multiple dependencies (state sharing) are also addressed.

The format described here has been implemented in code, which is also discussed in this document. A modified version of this code has been merged with the mainstream R code for testing.

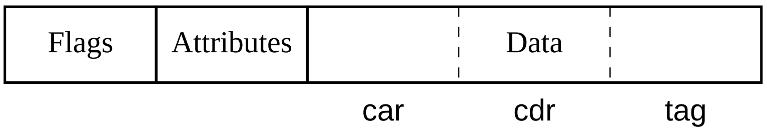
1 Introduction

1.1 A cursory glance at R data structures

The first section takes a casual look at *what* exactly is being saved and restored; readers familiar with R's data structures should not hesitate to skip to the next section (see Section 1.2 [The project mission], page 6) which discusses *why* exactly the project described in this paper exists.

1.1.1 The SEXPREC structure

The atomic data unit in R is called an SEXPREC—its naming roughly corresponds to a Lisp "S-expression". *Every* R object, ultimately, is represented as an SEXPREC.



Each rectangular block in the figure can be seen as a "word". The first word contains various information about the object, notably its type (explained shortly). The second word points to the attributes list of the object, which the user can access via `attributes(`*object*`)`. The last three words contain the actual data; their use varies with the object type, but are typically referred to as `car`, `cdr`, and `tag`.

Pointers to SEXPREC structures are so often used within the R *kernel* code (the code that makes up the basic R core) that such pointers are given their own type, SEXP.

1.1.2 The `nil` object

Perhaps the simplest object in R is the `nil` object; one can see it as an object holding no data. To an R user, this object is called NULL and often represents a list with zero items (in other words, an empty list), though it has many other uses. In R kernel code, the `nil` object is referred to as `R_NilValue`.

1.1.3 Simple objects

Simple objects are typically understood to be those that can exist on their own, and do not refer to other (non-`nil`) objects.

1.1.3.1 Simple vectors

Vectors of logical values (FALSE and TRUE), integers, real numbers, and complex numbers are considered simple objects. Some readers may now be wondering why this would be the case. After all, an integer vector is made up of some number of integers, each of which surely must be more simple than the vector that contains them.

In R, even a single integer is an integer vector (containing one item); similarly for the other types above.

1.1.3.2 Strings

A string itself is also a simple object, being a vector of its component characters. However, since an R user can only get at vectors of strings (even if one such vector contains only the one string), string vectors (see Section 1.1.4.3 [Composite vectors], page 6) are not simple objects.

1.1.3.3 Primitive functions

Primitive functions are functions defined within the R kernel. Such functions provide basic functionality “necessary” for R to be useful, and cannot be implemented as R library code. For example, the arithmetic operators (+, -, *, /, ^, &c) are primitive functions.

References to primitive functions are simple objects. In R kernel code, these are used as offsets into the R function table (see Section 3.1.1 [Calling primitive functions], page 11). From version 0.90 onwards, user-level code can use `.Primitive(`*name*`)` to make such references.

1.1.4 Composite objects

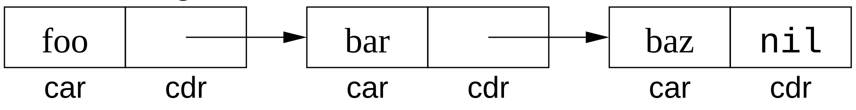
Composite objects are larger objects built together from a group of simple objects. A selection of composite object types are discussed briefly below, among an abundant range of types available in R.

1.1.4.1 Lists

Lists in R are singly-linked. Each element of a list, except the last, links to the one after it. These elements are known as *cons cells* from the way they are constructed in Lisp (using the `cons` function). A cons cell has two elements, historically known as `car` and `cdr`.

Conventionally, the `car` contains the first item in the list, and the `cdr` points to a list containing the other items. Since the `nil` object (see Section 1.1.2 [The `nil` object], page 3) represents an empty list, a one-item list has one cons cell: the `car` contains that one item, and the `cdr` contains `nil`.

A three-item list, containing ‘foo’, ‘bar’, and ‘baz’, would look like



A cons cell is sometimes called a “dotted pair”, because a cons cell can be made in Lisp using a dot: the following expressions are equivalent:

```
(cons 1 2)
'(1 . 2)
```

In R, cons cells have a `tag` slot, which can be used to name the list item by pointing that slot to a symbol (see Section 1.1.4.4 [Symbols], page 6). Lists that use the `tag` slots are called *tagged lists*, and have uses in certain applications (such as in environments; see Section 1.1.4.5 [Environments], page 6).

Although linked lists are heavily used within the R kernel code, all user-level lists are converted to generic vectors (see Section 1.1.4.3 [Composite vectors], page 6). The difference is that lists use up cons cells whereas vectors fill the vector heap (both of these have fixed-size allocation); read the “Memory” section in R help, as well as Section 3.1.2 [Storage areas], page 11, for a discussion of R’s memory management as it stands at the time of writing. Since lauguage objects (see Section 1.1.4.2 [Language objects], page 4) are lists, and the R library defines many complicated functions, cons cells are worth conserving.

1.1.4.2 Language objects

A *language object* is R’s notion of separating code and data: unlike Lisp, it will not treat data as code, nor vice versa. A language object is structured like a linked list, except it is used to hold code, whereas a linked list is used to hold data.

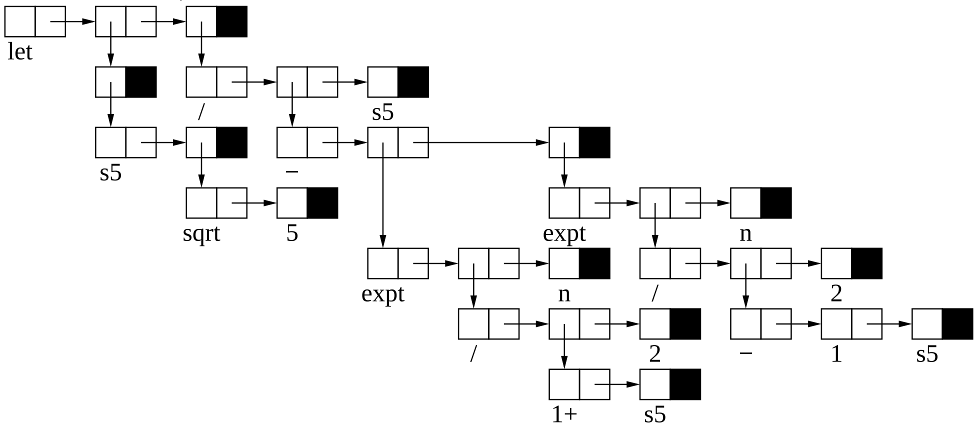
Arbitrarily complicated functions can be made out of cons cells. For example, the *n*-th fibonacci number can be calcuated in Lisp using the expression

```
(let ((s5 (sqrt 5))) (/ (- (expt (/ (1+ s5) 2) n)
                           (expt (/ (- 1 s5) 2) n)) s5))
```

which is roughly identical to

$$\frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

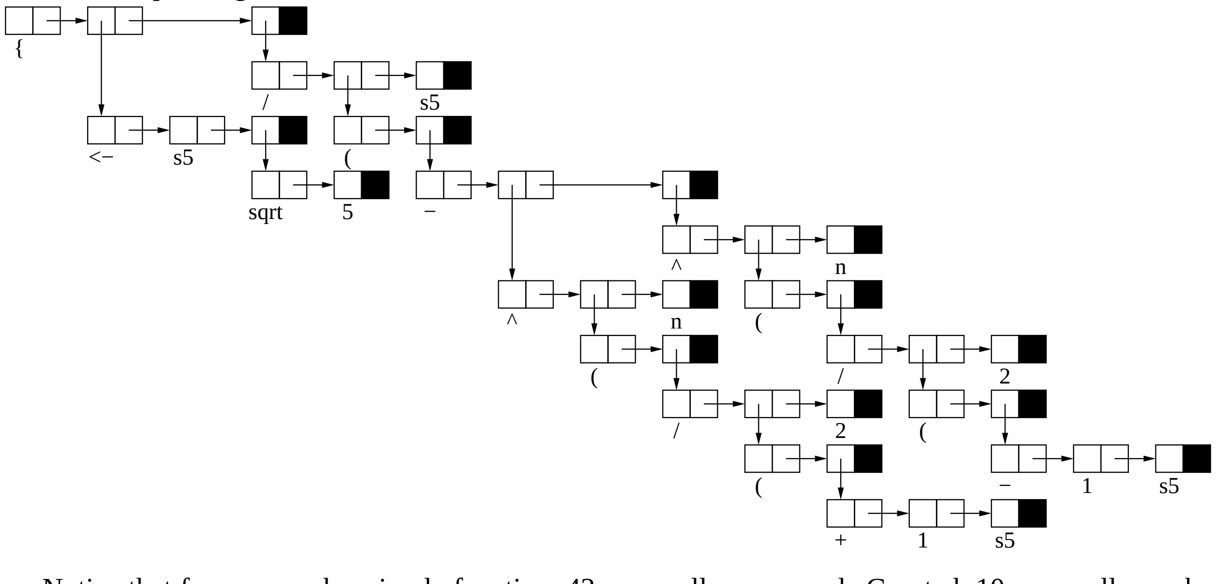
As a tree of cons cells, this would look like



In R, such a function would be written as

```
{
    s5 <- sqrt(5);
    (((1 + s5) / 2)^n - ((1 - s5) / 2)^n) / s5;
}
```

with a corresponding tree of



Notice that for even such a simple function, 42 cons cells were used. Granted, 10 cons cells can be saved by removing the `(` objects, which are purely for cosmetic purposes, but then the code will print as

```
{
    s5 <- sqrt(5)
    1 + s5/2^n - 1 - s5/2^n/s5
}
```

Despite that not being very human-readable, it does work correctly.

1.1.4.3 Composite vectors

There are three types of vectors which hold objects. A string vector holds zero or more strings (see Section 1.1.3.2 [Strings], page 3), an expression vector holds language objects (see Section 1.1.4.2 [Language objects], page 4), and a generic vector holds objects of any type (and is suitable to serve as a replacement for lists, to conserve cons cell usage—see Section 1.1.4.1 [Lists], page 4).

1.1.4.4 Symbols

A *symbol* is an object holding a name. Whenever you define a variable, its name is used to construct a symbol object, which is then put into R's internal symbol table (see Section 3.1.3 [The symbol table], page 11).

1.1.4.5 Environments

An *environment* consists of a list of variables, and their values. In R it is implemented as a tagged dotted pairs list: the `tag` slot of a list item is used to hold the name of the variables, and the `car` slot the value.

Each environment has a parent environment, which is searched if a sought variable cannot be found in the current environment "frame". Tracing environment "ancestry" will ultimately lead to the global environment, which is its own parent. The global environment holds most of a user's data, and is accessible at the user level as `.GlobalEnv`, and in R's kernel code as `R_GlobalEnv`.

1.1.4.6 Closures

An R user level function is stored internally as a so-called *closure*; apart from containing the function body (as a language object), it holds information on which formal arguments will be passed in to the function, as well as which environment to begin variable searches in.

1.2 The project mission

To explain the problems with the old save/load file format, and possible advantages of a new one, the former will be briefly described. See Appendix A [The old format], page 19, for a more detailed description.

1.2.1 Saving data, the old way

The current R saving mechanism works by marking the given list of objects to be saved, and all objects referenced by the same, by recursing into composite objects until simple objects are reached.

The number of symbol and non-symbol objects marked as above are counted, and recorded in the file. Each such marked object is assigned a sequence number, starting from zero, up to one less than the total number of marked objects. Thus, each object has a different sequence number.

For each symbol, its sequence number, relative address (relative to the start of the vector heap; see Section 3.1.2 [Storage areas], page 11), and name are written out. This set of data will collectively be referred to in this section as the "symbol list".

The sequence numbers and relative addresses of the non-symbol objects are then written out—the "forwarding-address list". When the file is loaded back, the relative addresses saved here are translated (or "forwarded") to newly-allocated object spaces.

Then the actual data contained in each object referred to in the forwarding-address list are written out—any references to an object, such as in composite objects, use the relative address.

Finally, the relative address of the given list of objects to save, is itself recorded.

1.2.2 Loading data, the old way

Remember that in saving the objects, the number of symbols and non-symbols were recorded. When loading, an `SEXPREC` is allocated to hold each of the saved objects, and a translation table is set up to map the relative addresses (as saved in the symbol and forwarding-address lists) to the newly-allocated `SEXPREC` structures. This translation table is indexed by the sequence numbers of the saved objects.

In practice, the symbol list functions equivalently to the forwarding-address list, only that the former is listed first so that all the symbols can be "installed" (see Section 3.1.3 [The symbol table], page 11) first before loading the data objects. The sole purpose of the forwarding-address list is to set up the translation table.

During the loading of actual data, any reference to a relative address is resolved to the new location, as per the translation table.

1.2.3 The joys of using the old format

There are some reasons why a new format is in order; these are listed in no particular order, and totally subjective:

- The use of relative addresses in the old format ties it down to the memory system, which is long overdue for change. It is difficult (but not impossible—see Section 4.2 [Backward-compatibility], page 17) to hack the save/load code to free it of such ties, but since the notion of relative addresses is so utterly arbitrary, why bother?
- The old format treats all `SEXPREC` structures as disjoint, making it hard for human readers (if people actually read the saved files manually) to make links between structures that make up a composite object.

2 Design

2.1 A brief look at R environment semantics

The design of the new file format had much influence from the way environments work in R; this is described by Ross Ihaka in his article, "RFC: A Partial Redesign of R Internals" (`http://www.stat.auckland.ac.nz/~ihaka/R/Redesign.html`).

2.1.1 Scoping

R uses a variant of lexical scoping; Kurt Hornik explains this extensively in "The R FAQ" (`http://www.ci.tuwien.ac.at/~hornik/R/`), with some really useful examples (including object orientation, which I will get into; see Section 2.1.2 [State sharing], page 9).

The bottom line is that all functions are closures, with attached environments where variables not found in the local environment frame (which is created per invocation of a function) are looked up. Which environment gets attached to a closure depends on where the latter is created: a typical function entered by the user interactively is most likely to have the global environment as its environment, whereas a function defined within a function will use the enclosing function's environment.

Examples (assume these are entered interactively):

```
foo <- function()
{
    bar;
}
```

`foo` uses the global environment. When run, since its local frame does not contain `bar`, it will return the value of `bar` in the global environment if it exists (or signal an error otherwise).

```
baz <- (function()
{
    bar <- 3;
    function()
    {
        bar;
    }
})();
```

`baz` uses the local environment of the anonymous outer function. When run, since its own local environment does not contain `bar`, it looks one level up, into the environment of the anonymous function. There, `bar` is 3, so it returns that; but if that environment does not contain `bar`, then the search will ascend one level into the global environment.

2.1.2 State sharing

R can be used in a quasi-object-oriented fashion. A simple example (a huge simplification of the `demo("scoping")` material):

```
ctor <- function(x) list(get=function() x, set=function(i) x <<- i);
```

`ctor` is essentially a constructor for a very simple class (containing just one object). It would work like this:

```
> instance <- ctor("hello world");
> instance$get();
[1] "hello world"
> instance$set(ctor);
> instance$get();
function (x)
list(get = function() x, set = function(i) x <<- i)
> instance$set(NULL);
> instance$get();
NULL
```

. . . and so on. `instance` holds one object, of any type. Notice that looking at `instance` itself, we see

```
> instance;
$get
function () x
<environment: 0x584bc0>

$set
function (i) x <<- i
<environment: 0x584bc0>
```

Of course the environment address will be different for every different instance, but the important thing here is that both functions use the same environment, which contains the state information both the `get` and `set` methods share.

2.1.3 Environment loops

In a rather contrived example, an environment can be made to contain items that refer to itself (the environment). For example

```
foo <- function() bar <- function() bar;
```

When you run `foo`, a function is returned that uses an environment that contains a function. This latter function is bound to that same environment, which contains the same function. And so on and on.

2.2 The design criteria

From the previous chapter, and the above, we see that:

1. The general strategy for saving a data object is to recurse through composite objects, until we come to a simple object, writing each object out as it is traversed. This preserves the links between all the otherwise-separate SEXPREC structures.
2. But, we can't just write out environments entirely every time we come across one—with an example like the one in Section 2.1.3 [Environment loops], page 10, the environment will never be completely saved. There's also the state sharing business to deal with: if we write out separate environments for each of the methods described in Section 2.1.2 [State sharing], page 9, they will have disjoint states when they are loaded back, and `get` will not refer to the same object as `set`.

Thankfully, the other object types in R don't have such complications. So, if we keep tabs on which environments have been seen, and write out just references (rather than the whole environment) when we encounter them again, then we have a way to tell apart, multiple instances of the same environment, from multiple identical (in content) but different environments. This solves the problem with state sharing, and with environment loops too.

With one other change, that's the new format in a nutshell: all environments encountered have their contents written out the first time, and a reference written out subsequent times, and every other composite object is recursed into and written out in full. The one other change is an idea inherited from the old format; we keep tabs on symbols too, and write out references for symbols encountered just as for environments. The motivation for that is given in Section 4.3 [The symbol list], page 18.

The new format, in detail, is described in Appendix B [The new format], page 21.

3 Implementation

3.1 A quick peek at R kernel code

The code used to save and load in the new file format (see Appendix C [Complete code listing], page 23), is implemented in R's kernel. The first section hopefully allows people, who have never tinkered with the kernel, to understand the code better.

3.1.1 Calling primitive functions

R's primitive functions are listed in `R_FunTab`, an array of `FUNTAB` structures. Each `FUNTAB` represents one primitive function, and includes such information as its name (as accessed in R user code), the kernel function that handles it, how many arguments it accepts, and so on. `R_FunTab` is defined in '`src/main/names.c`'.

For example, the user-level functions to save a list of objects to a file, and to load objects from a file, are respectively `"save"` and `"load"`. In `R_FunTab` one can find

```
{"save",        do_save,        0,      111,    3,      PP_FUNCALL},
{"load",        do_load,        0,      111,    2,      PP_FUNCALL},
```

In this case, the kernel functions to save and load data objects are `do_save` and `do_load` respectively. `do_save` expects 3 arguments; `do_load` expects 2.

The `StrToInternal` function converts a primitive function name into an index into `R_FunTab`. Internally, references to primitive functions are stored as this index.

3.1.2 Storage areas

(It is my understanding that the memory system will be changing soon. This section serves only to describe certain aspects of the memory system that the old format used.)

Under the current memory management system, there are two main areas of memory that R utilises—they do not grow, but their sizes can be set when starting R:

1. One area of memory is used specifically to store cons cells. All cons cell allocation comes from this pool. Things that use cons cells are lists (see Section 1.1.4.1 [Lists], page 4) and language objects (see Section 1.1.4.2 [Language objects], page 4). R's cons cell usage is quite heavy: after loading the standard R library, about half of the default allocation of 250000 cells are used.

   The number of cons cells allocated can be changed using '`--nsize`'; for example, to increase to 400000 cons cells, use *`R --nsize 400000`*.
2. Another area of memory is used for everything else; this area is known as the vector heap, as most of this space is used to store vectors.

   6 megabytes are allocated towards the vector heap, by default; this amount can be changed using '`--vsize`'; for example, *`R --vsize 8388608`* will cause R to allocate 8 megabytes.

3.1.3 The symbol table

R has a symbol table, `R_SymbolTable`, that every symbol (see Section 1.1.4.4 [Symbols], page 6) is a member of. Symbols are typically created using the `install` function, which returns the symbol for the given name if it exists in the symbol table, or makes a new symbol (and puts it in the symbol table) otherwise. In effect, every name corresponds to one, and just one, symbol.

Example: `install("foo")` returns the (new or existent) symbol associated with `"foo"`.

3.1.4 SEXPTYPE

`SEXPTYPE` is a field in the `SEXPREC` structure which determines the type of the object described. Commonly-used values are:

`NILSXP` (0)
: the `nil` object (`R_NilValue` and `NULL`—see Section 1.1.2 [The `nil` object], page 3); used for no other object.

`SYMSXP` (1)
: symbols (see Section 1.1.4.4 [Symbols], page 6)

`LISTSXP` (2)
: cons cell lists (see Section 1.1.4.1 [Lists], page 4)

`CLOSXP` (3)
: closures (see Section 1.1.4.6 [Closures], page 6)

`ENVSXP` (4)
: environments (see Section 1.1.4.5 [Environments], page 6)

`PROMSXP` (5)
: promises (lazily evaluated closures; may or may not have been evaluated)

`LANGSXP` (6)
: language objects (see Section 1.1.4.2 [Language objects], page 4)

`SPECIALSXP` (7)
: special forms (primitive functions whose arguments are not evaluated before passing in—`expression`, the primitive used to make expression vectors, is an example of one)

`BUILTINSXP` (8)
: primitive functions—like `SPECIALSXP`, this type contains only an index into `R_FunTab` (see Section 3.1.1 [Calling primitive functions], page 11), not the actual primitive function, obviously

`CHARSXP` (9)
: single strings (see Section 1.1.3.2 [Strings], page 3); users cannot access this type directly

`LGLSXP` (10)
: logical vectors (containing `FALSE` and `TRUE` values; see Section 1.1.3.1 [Simple vectors], page 3)

`INTSXP` (13)
: integer vectors (see Section 1.1.3.1 [Simple vectors], page 3)—types 11 and 12 are deprecated factor types that are now converted to this type

`REALSXP` (14)
: real number vectors (see Section 1.1.3.1 [Simple vectors], page 3)

`CPLXSXP` (15)
: complex number vectors (see Section 1.1.3.1 [Simple vectors], page 3)

`STRSXP` (16)
: string vectors (containing `CHARSXP` objects; see Section 1.1.4.3 [Composite vectors], page 6)

`DOTSXP` (17)
: ‘`...`’ objects used in function argument lists, and sometimes function calls

`VECSXP` (19)
: generic vectors (see Section 1.1.4.3 [Composite vectors], page 6)

`EXPRSXP` (20)
: expression vectors (see Section 1.1.4.3 [Composite vectors], page 6)

3.2 Code commentary

For maximum enjoyment of this section, please refer regularly to Appendix C [Complete code listing], page 23. Functions will be covered in a top-down fashion.

3.2.1 Helper functions

3.2.1.1 `R_assert`

In C programming, `assert` is a simple but useful bug tracking device; it is used to detect “impossible” situations, resulting from internal errors. My code was littered all around with `assert` calls, until frequent crashes became annoying.

`R_assert` is an attempt to make a “friendlier” `assert`, by virtue of the `longjmp`-like behaviour of `R`’s `error` function—instead of crashing it returns program control to the “top level”. It is invoked exactly the same way as `assert`, and can be disabled by defining `NDEBUG`.

3.2.1.2 `Out*`, `In*`

In the old save/load code, `do_save` set up the `OutInit`, `OutInteger`, `OutReal`, `OutComplex`, `OutString`, `OutSpace`, `OutNewline`, and `OutTerm` function pointers to the versions appropriate for the format to save as (text, binary, or XDR), prior to calling `DataSave` (which does the grunt work saving the objects).

Similarly for `do_load` in regard to `InInit`, `InInteger`, `InReal`, `InComplex`, `InString`, and `InTerm`, except they are used for loading. The `OutSpace` and `OutNewline` functions are there purely for cosmetic reasons when saving in text format, and so there exist no functions to explicitly read them in.

To extend the analogy, I’ve put in functions `OutCHARSXP` and `InCHARSXP`, which deal with `CHARSXP` objects (single strings, see Section 1.1.3.2 [Strings], page 3). There are also macros `OutVec` and `InVec`, which generically handle all the simple vector types (including string vectors, with the addition of `OutCHARSXP` and `InCHARSXP`). You will see them in action in `cky_write_vec` and `cky_read_vec`.

3.2.1.3 `cky_make_lists`

This function builds the symbol and environment lists. Each object to be saved is recursed into, and each symbol/environment encountered is added to the respective list if unique. Currently this check for uniqueness is done via a linear search.

3.2.1.4 `cky_*_special_hook`

Some objects need to be special-cased. These are checked for first when saving or loading any object, by calling the appropriate special-object hook (`cky_save_special_hook` and `cky_load_special_hook` respectively).

`R_NilValue` (−1)
: The `nil` object probably occurs the most frequently of all the data types; making it a special case will make the saved file much smaller.

`R_GlobalEnv` (−2)
: The global environment is huge, and its contents largely depend on the `R` (user-level) libraries in use, which are dependent on the `R` version. Not saving the global environment not only reduces the size of the saved file, but also makes the file more portable.

`R_UnboundValue` (−3)
`R_MissingArg` (−4)
: These are special objects that would lose their semantic significance if they are saved by writing the objects out in full (they are differentiated by their locations in memory, not by a field in the `SEXPREC` structure—while the old format could very well have exploited that, as they will have a fixed relative address, the new format cannot hope to).

3.2.1.5 `cky_*_auxiliary`

Objects contain certain other attributes that were stored in the old format; these are also used in the new format. `cky_write_auxiliary` and `cky_read_auxiliary` handle these, and can be extended as necessary.

`ATTRIB` The user-settable attributes associated with the object; these are accessible to the user via `attributes(`*obj*`)`.

`LEVELS` This is an attribute that has many uses, and often overloaded. Rather than describe all the uses, I shall leave the reader to “Use the Source, Luke”.

`OBJECT` This attribute is accessible to the user via `is.object(`*obj*`)`, which is all I can say about it.

3.2.2 Saving/loading functions

The `do_save` and `do_load` functions call `DataSave` and `DataLoad` (respectively) at the end, so the latter are the points my code patch into.

3.2.2.1 `cky_DataSave`, `cky_DataLoad`

These functions are responsible for writing out/reading in the header information (the numbers of symbols and environments saved, and the symbols and environments themselves). `cky_DataSave` then writes the objects that the user asked to save; `cky_DataLoad` reads them back in.

As part of writing out the header, `cky_DataSave` calls `cky_make_lists` (see Section 3.2.1.3 [`cky_make_lists`], page 13) to construct the symbol and environment lists.

For each item to write, `cky_DataSave` calls `cky_write_item`; for each item to read, `cky_DataLoad` calls `cky_read_item`.

3.2.2.2 `cky_*_item`

These functions (`cky_write_item`, `cky_read_item`) handle writing/loading one object of any type.

Several objects are not written out in full (see Section 3.2.1.4 [`cky_*_special_hook`], page 13); these are the same objects special-cased by the old format.

Two types are also special-cased, the symbol and environment types. An index to the respective list is written out/read in instead of the full object.

Vectors are handled in `cky_*_vec` (see Section 3.2.2.3 [`cky_*_vec`], page 15), primitive functions (and special forms) have their names written out/read in, and everything else is recursed into (since they are composite objects).

Certain “meta-information” are also written/read via the functions `cky_*_auxiliary` (see Section 3.2.1.5 [`cky_*_auxiliary`], page 14).

3.2.2.3 `cky_*_vec`

These functions (`cky_write_vec`, `cky_read_vec`) handle writing/loading one vector (whether simple or composite) object.

Simple vectors (this includes string vectors here, because of the `OutCHARSXP` and `InCHARSXP` functions) are written/read using the `OutVec`/`InVec` function. See Section 3.2.1.2 [`Out*`/`In*`], page 13.

Composite vectors (generic vectors and expression vectors) have each of their items processed by `cky_*_item` (see Section 3.2.2.2 [`cky_*_item`], page 14)—in essence, a recursion.

4 Postscript

The code, as described in Chapter 3 [Implementation], page 11, and presented in Appendix C [Complete code listing], page 23, was my work as of 15 August 1999; that was the version that I have submitted to Ross Ihaka for inclusion into `R`. I have not touched the code any further, and since that time I have become aware of several issues. These, and others I've thought about back when I wrote the code, are presented here.

4.1 The use of sentinels

. . . was a big mistake. Two types of sentinels are used in the code, and both cause problems.

1. The use of `CAR(`*obj*`) = TAG(`*obj*`) = R_NilValue` as the last item of a list was far from sensible, as it was possible to construct lists (or at least language objects) that contain `nil` in the middle. Consider a function returning just `nil`:

   ```
   function()
   {
       NULL;
   }
   ```

   That function will save under the new format, but it won't load back properly. It appears that the only sane way to determine where the end of a list is, is to see where `CDR(`*obj*`) == R_NilValue`. (The amended file format, as used in the mainstream `R` source tree, has fixed this problem.)

   So, it was back to a recursive style of saving lists, just like the other object types. With big lists, this might cause the stack to get unnecessarily big, but two factors suggest that this is not a serious concern:

   a. in recent versions of `R`, lists are automatically converted to vectors
   b. if the `cky_write_auxiliary` (and by extension also `cky_read_auxiliary`) function is called at the beginning instead of the end of `cky_write_item`, and the `OutNewline(fp)` at the very end is removed, *tail recursion* can occur (i.e., when recursing into the final item, use the same recursion level rather than descend one level, on the premise that there are no more items on the current level to process)

2. The use of null-terminated strings was also far from sensible. In `R`, a string is represented internally as a `CHARSXP`, which specifies a size (just like all other vector types). The most evil code are the `OutString` and `InString` functions, which make big assumptions that strings are null-terminated.

   At the moment, my `OutCHARSXP` and `InCHARSXP` functions use the said functions. With little modification, namely by not using those evil functions, they can be made to handle null characters properly. Parties considering implementing such a change should remember to exclude '`<string.h>`' in the process.

4.2 Backward-compatibility

I *had* thought about making the functions use the old format, complete with a dummy forwarding table and all. However, it'd be difficult to write code that can handle the old format and still not depend on the memory structure.

(Not impossible though: the `OffsetToNode` function, used in restoring, is independent of the memory system; the fake forwarding table can be constructed without depending on the memory location of objects too—after all the relative addresses are just serial numbers, as far as the file format is concerned. Using that approach however, an *object* list has to be built, as opposed to just symbol or environment lists which are not expected to get very large.)

Having said that: the new format has been designed to contain all the information contained in the old format. It should be possible to convert one format to the other with no loss of information.

4.3 The symbol list

The symbol list was an idea copied from the old format. It's not a strictly necessary "feature", but it does improve loading speed if at least some of the symbols occur in high frequencies.

Because quite often most or all of the symbols occur only once, especially for small data sets, it would be beneficial to have a variation on the new format: if the symbol count (in the header) is -1, then a type 1 object (`SYMSXP`—see Section 3.1.4 [`SEXPTYPE`], page 12) contains the name of the symbol (represented as a string) rather than the location of the symbol in the symbol list.

That is, in addition to than using

```
1 0
"foo"
-1
2
1 1 -1 0 0
16 1 "bar" -1 0 0
-1 -1 -1 0 0
-1
```

to represent a symbol `foo` (with value `"bar"`), this would also be possible:

```
-1 0
-1
2
1 "foo" -1 0 0
16 1 "bar" -1 0 0
-1 -1 -1 0 0
-1
```

This idea has not been incorporated into code, but would be trivial to implement. Instead of having the program decide when to make a symbol list and when not to, this choice should be left to the user.

4.4 Previous approaches

Currently, the `cky_make_lists` function builds both the symbol and environment lists at once. Previously, a functional approach was attempted where each list was saved separately.

Unfortunately, environments do loop at times (see Section 2.1.3 [Environment loops], page 10), and it would do no good for a symbol scan to not also be able to tell which environments have already been traversed. Having both lists updated in the same function makes this possible. It also reduces one recursive scan of the data.

I also found out along the way that the code looked a lot more readable when I removed the hacks needed to make the original list building functions (called `cky_env_list` and `cky_sym_list`—no longer in the final code obviously) use functional strategies.

Appendix A The old format

file format

number of symbols
: integer

number of non-symbols
: integer

number of vector cells
: integer

symbol list
: symbol list item × *number of symbols*

forwarding-address list
: forwarding entry × *number of non-symbols*

object list
: object × *number of non-symbols*

address of list of objects to save
: relative address[1] → `LISTSXP`

symbol list item

sequence number[2]
: integer

symbol address
: relative address → `SYMSXP`

printing name
: string

forwarding entry

sequence number
: integer

object address
: relative address

object

sequence number
: integer

object type
: integer[3]

OBJECT *attribute*
: integer

LEVELS *attribute*
: integer

user-settable attributes
: relative address → `LISTSXP` or `R_NilValue`

---

1 The relative address is the address of the object relative to the start of the vector heap (see Section 3.1.2 [Storage areas], page 11), unless that object is special-cased (see Section 3.2.1.4 [`cky_*_special_hook`], page 13). Any reference to an object in this file format will be via a relative address.

2 A sequence number is assigned to each object; this goes from 0 to (*number of symbols* `+` *number of non-symbols*). The sequence numbers used in *forwarding address list* and *object list* are the same, and do not include the sequence numbers allocated for the symbol objects.

3 See Section 3.1.4 [`SEXPTYPE`], page 12.

object data	generic object (`LISTSXP`, `LANGSXP`, `CLOSXP`, `PROMSXP`, `ENVSXP`) string with length (`SPECIALSXP`, `BUILTINSXP`, `CHARSXP`) simple vector (`REALSXP`, `CPLXSXP`, `INTSXP`, `LGLSXP`) composite vector (`STRSXP`, `VECSXP`, `EXPRSXP`)

generic object

car	relative address
cdr	relative address
tag	relative address

string with length

string length	integer
string	string

simple vector

number of items	integer
items	relevant type $\times$ *number of items*

composite vector

number of items	integer
items	relative address $\times$ *number of items*

Appendix B The (tentative) new format

file format

number of symbols
: integer

number of environments
: integer

symbol name list
: string × *number of symbols*

environment list
: full environment × (*number of environments* + 1)[1]

full environment

parent environment
: object → `ENVSXP`

frame
: object → `LISTSXP`

tag
: object

object[2]

object type
: integer[3]

object data
: integer[4] (`SYMSXP`)
: linked list × *variable*[5] (`LISTSXP`, `LANGSXP`)
: integer[6] (`ENVSXP`)
: generic object (`CLOSXP`, `PROMSXP`)
: string (`SPECIALSXP`, `BUILTINSXP`)
: string with length (`CHARSXP`)
: simple vector (`LGLSXP`, `INTSXP`, `REALSXP`, `STRSXP`)
: composite vector (`VECSXP`, `EXPRSXP`)

user-settable attributes
: object → `LISTSXP`

LEVELS *attribute*
: integer

OBJECT *attribute*
: integer

linked list

tag
: object

car
: object

[1] The list of objects to save is masqueraded as the final environment to save, which is why (*number of environments* + 1) environments are saved.

[2] The usual special-casing rules (see Section 3.2.1.4 [`cky_*_special_hook`], page 13) apply. If the object is a special item, the relevant negative number is used, and the rest of this structure is omitted.

[3] See Section 3.1.4 [`SEXPTYPE`], page 12.

[4] A symbol reference, specifying the one-based index into *symbol name list*.

[5] The list ends when both *tag* and *car* point to `R_NilValue`. As mentioned in Section 4.1 [The use of sentinels], page 17, this is *really* broken.

[6] An environment reference, specifying the one-based index into *environment list*.

generic object		
	car	object
	cdr	object
	tag	object
string with length		
	string length	integer
	string	string
simple vector		
	number of items	integer
	items	relevant type × *number of items*
composite vector		
	number of items	integer
	items	object × *number of items*

Appendix C Complete code listing

```
/*
 * x-saveload.c—experimental save/load procedures for R.
 * Chris K. Young <cky@pobox.com>, June 1999.
 *
 * This file is to be licensed under the same conditions as the rest of R.
 *
 * Under the current experimental stage, all functions are prefixed with
 * ‘cky_’... feel free to rename them when they come in production use.
 *
 * I’m not entirely sure when to PROTECT(), due to a lack of understanding
 * of how the GC system works. So I might have overlooked many places.
 */

/*
 * Right now, in order to avoid altering ‘saveload.c’, I’ll have this
 * file include ‘saveload.c’ at the top. In production, this file should
 * be included at the bottom of ‘saveload.c’.
 */
#include "saveload.c"

#include "Defn.h"
#include <string.h>

/* assert function which doesn’t crash the program. */
#ifdef NDEBUG
#define R_assert(e) ((void) 0)
#else
/* The line below requires an ANSI C preprocessor (stringify operator) */
#define R_assert(e) ((e) ? (void) 0 : \
    error("assertion ‘%s’ failed: file ‘%s’, line %d\n", #e, __FILE__, __LINE__))
#endif

static void cky_write_item (SEXP s, SEXP sym_list, SEXP env_list, FILE *fp);
static SEXP cky_read_item (SEXP sym_table, SEXP env_table, FILE *fp);

/*
 * This function determines if the item is ‘special’ (R_NilValue,
 * R_GlobalEnv, R_UnboundValue, R_MissingArg). Returns a non-zero
 * value if it is.
 */
static int
cky_save_special_hook (SEXP item)
{
    if (item == R_NilValue) return -1;
    if (item == R_GlobalEnv) return -2;
    if (item == R_UnboundValue) return -3;
    if (item == R_MissingArg) return -4;
    return 0;
}
```

```
static SEXP
cky_load_special_hook (SEXPTYPE type)
{
    switch (type) {
    case -1: return R_NilValue;
    case -2: return R_GlobalEnv;
    case -3: return R_UnboundValue;
    case -4: return R_MissingArg;
    }
    return (SEXP) 0;    /* not strictly legal. . . . */
}

/*
 * There could be a lot of wheel-reinvention here, because I haven't had
 * a hard-enough look at the rest of the code to see whether all this had
 * been covered elsewhere.
 *
 * The lookup code certainly can do with improvement, because I doubt a
 * linear search works best here.
 */
static int
cky_lookup (SEXP item, SEXP list)
{
    SEXP iterator = list;
    int count;

    if ((count = cky_save_special_hook(item)))    /* variable reuse :-) */
        return count;
    /* now count is zero */
    while (iterator != R_NilValue) {
        R_assert(TYPEOF(list) == LISTSXP);
        ++count;
        if (CAR(iterator) == item)
            return count;
        iterator = CDR(iterator);
    }
    return 0;
}

/* a generic cons function which also sets the tag value. */
static SEXP
cky_cons (SEXPTYPE type, SEXP tag, SEXP head, SEXP tail)
{
    SEXP atom;

    if (head == R_NilValue)
        return tail;
    atom = (type == LANGSXP) ? LCONS(head, tail) : CONS(head, tail);
    TAG(atom) = tag;
    return atom;
}
```

```
/*
 * One thing I've found out is that you have to build all the lists together
 * or you risk getting infinite loops. Of course, the method used here
 * somehow shoots functional programming in the head—sorry.
 */
static void
cky_make_lists (SEXP obj, SEXP *sym_list, SEXP *env_list)
{
    int count, length;

    if (cky_save_special_hook(obj))
        return;
    switch (TYPEOF(obj)) {
    case SYMSXP:
        if (cky_lookup(obj, *sym_list))
            return;
        *sym_list = CONS(obj, *sym_list);
        break;
    case ENVSXP:
        if (cky_lookup(obj, *env_list))
            return;
        *env_list = CONS(obj, *env_list);
        /* FALLTHROUGH */
    case LISTSXP:
    case LANGSXP:
    case CLOSXP:
    case PROMSXP:
        cky_make_lists(TAG(obj), sym_list, env_list);
        cky_make_lists(CAR(obj), sym_list, env_list);
        cky_make_lists(CDR(obj), sym_list, env_list);
        break;
    case VECSXP:
    case EXPRSXP:
        length = LENGTH(obj);
        for (count = 0; count < length; ++count)
            cky_make_lists(VECTOR(obj)[count], sym_list, env_list);
        break;
    }
    cky_make_lists(ATTRIB(obj), sym_list, env_list);
}

/* e.g., OutVec(fp, obj, INTEGER, OutInteger) */
#define OutVec(fp, obj, accessor, outfunc)                              \
    do {                                                                \
        int count;                                                      \
        for (count = 0; count < LENGTH(obj); ++count) {                 \
            OutSpace(fp);                                               \
            outfunc(fp, accessor(obj)[count]);                          \
        }                                                               \
    } while (0)
```

```
/*
 * simply outputs the string associated with a CHARSXP, one day this will
 * handle null characters in CHARSXPs and not just blindly call OutString.
 */
static void
OutCHARSXP (FILE *fp, SEXP s)
{
    R_assert(TYPEOF(s) == CHARSXP);
    OutString(fp, CHAR(s));
}

static void
cky_write_vec (SEXP s, SEXP sym_list, SEXP env_list, FILE *fp)
{
    int count;

    /*
     * I can assert here that s is one of the vector types, but it'll
     * turn out to be one big ugly statement... so I'll do it at the
     * bottom.
     */
    OutInteger(fp, LENGTH(s));
    switch (TYPEOF(s)) {
    case CHARSXP:
        OutSpace(fp);
        OutCHARSXP(fp, s);
        break;
    case LGLSXP:
    case INTSXP:
        OutVec(fp, s, INTEGER, OutInteger);
        break;
    case REALSXP:
        OutVec(fp, s, REAL, OutReal);
        break;
    case CPLXSXP:
        OutVec(fp, s, COMPLEX, OutComplex);
        break;
    case STRSXP:
        OutVec(fp, s, STRING, OutCHARSXP);
        break;
    case VECSXP:
    case EXPRSXP:
        for (count = 0; count < LENGTH(s); ++count) {
            OutSpace(fp);
            cky_write_item(VECTOR(s)[count], sym_list, env_list, fp);
        }
        break;
    default:
        error("cky_write_vec called with non-vector type");
    }
    OutNewline(fp);
}

/*
 *  If we change cky_write_auxiliary, cky_read_auxiliary has to be changed
 *  too.
 */
static void
cky_write_auxiliary (SEXP obj, SEXP sym_list, SEXP env_list, FILE *fp)
{
    OutSpace(fp); cky_write_item(ATTRIB(obj), sym_list, env_list, fp);
    OutSpace(fp); OutInteger(fp, LEVELS(obj));
    OutSpace(fp); OutInteger(fp, OBJECT(obj));
}

static void
cky_write_item (SEXP s, SEXP sym_list, SEXP env_list, FILE *fp)
{
    int i;
    SEXP iterator;

    if ((i = cky_save_special_hook(s))) {
        OutInteger(fp, i);
    } else {
        OutInteger(fp, TYPEOF(s)); OutSpace(fp);
        switch (TYPEOF(s)) {
        /*
         *  we shouldn't encounter NILSXP here, as it should've been
         *  handled above (in cky_save_special_hook).  Unless there
         *  are more NILSXP's than R_NilValue.
         */
        case SYMSXP:
            i = cky_lookup(s, sym_list);
            R_assert(i);
            OutInteger(fp, i); OutNewline(fp);
            break;
        case LISTSXP:
        case LANGSXP:
            iterator = s;
            while (iterator != R_NilValue) {
                cky_write_item(TAG(iterator), sym_list, env_list, fp);
                cky_write_item(CAR(iterator), sym_list, env_list, fp);
                iterator = CDR(iterator);
            }
            /*  This is our sentinel  */
            cky_write_item(R_NilValue, sym_list, env_list, fp);
            cky_write_item(R_NilValue, sym_list, env_list, fp);
            break;
        case ENVSXP:
            i = cky_lookup(s, env_list);
            R_assert(i);
            OutInteger(fp, i); OutNewline(fp);
            break;
        case CLOSXP:
        case PROMSXP:
            cky_write_item(TAG(s), sym_list, env_list, fp);
            cky_write_item(CAR(s), sym_list, env_list, fp);
            cky_write_item(CDR(s), sym_list, env_list, fp);
            break;
        case SPECIALSXP:
        case BUILTINSXP:
            OutString(fp, PRIMNAME(s));
            break;
        case CHARSXP:
        case LGLSXP:
        case INTSXP:
        case REALSXP:
        case CPLXSXP:
        case STRSXP:
        case VECSXP:
        case EXPRSXP:
            /* The blasted vector thingies. . . . */
            cky_write_vec(s, sym_list, env_list, fp);
            break;
        default:
            /* don't know—maybe just write TAG CAR & CDR */
            warning("cky_write_item: unknown type %i", TYPEOF(s));
            cky_write_item(TAG(s), sym_list, env_list, fp);
            cky_write_item(CAR(s), sym_list, env_list, fp);
            cky_write_item(CDR(s), sym_list, env_list, fp);
        }
        cky_write_auxiliary(s, sym_list, env_list, fp);
    }
    OutNewline(fp);
}

/*
 * General format: the total number of symbols, then the total number of
 * environments. Then all the symbol names get written out, followed by
 * the environments, then the items to be saved. If symbols or environments
 * are encountered, references to them are made instead of writing them
 * out totally.
 */

void
cky_DataSave (SEXP s, FILE *fp)
{
    SEXP the_sym_list = R_NilValue, the_env_list = R_NilValue, iterator;
    int sym_count, env_count;

    cky_make_lists(s, &the_sym_list, &the_env_list);
    OutInit(fp);
    OutInteger(fp, sym_count = length(the_sym_list)); OutSpace(fp);
    OutInteger(fp, env_count = length(the_env_list)); OutNewline(fp);
    for (iterator = the_sym_list; sym_count--; iterator = CDR(iterator)) {
        R_assert(TYPEOF(CAR(iterator)) == SYMSXP);
        OutString(fp, CHAR(PRINTNAME(CAR(iterator))));
        OutNewline(fp);
    }
    for (iterator = the_env_list; env_count--; iterator = CDR(iterator)) {
        R_assert(TYPEOF(CAR(iterator)) == ENVSXP);
        cky_write_item(ENCLOS(CAR(iterator)), the_sym_list, the_env_list, fp);
        cky_write_item(FRAME(CAR(iterator)), the_sym_list, the_env_list, fp);
        cky_write_item(TAG(CAR(iterator)), the_sym_list, the_env_list, fp);
    }
    /*
     * If you stare at the incoming data structure hard enough, you can
     * actually see it as an environment frame.
     */
    /* this is for the fictitious 'enclos' */
    cky_write_item(R_NilValue, the_sym_list, the_env_list, fp);
    cky_write_item(s, the_sym_list, the_env_list, fp);
    /* this is for the fictitious 'tag' */
    cky_write_item(R_NilValue, the_sym_list, the_env_list, fp);
    OutTerm(fp);
}

#define InVec(fp, obj, accessor, infunc, length)                        \
    do {                                                                \
        int count;                                                      \
        for (count = 0; count < length; ++count)                        \
            accessor(obj)[count] = infunc(fp);                          \
    } while (0)

static SEXP
InCHARSXP (FILE *fp)
{
    SEXP s;
    char *tmp;

    AllocBuffer(MAXELTSIZE - 1);
    /*
     * FIXME: rather than use strlen, use actual length of string when
     * sized strings get implemented in R's save/load code.
     */
    tmp = InString(fp);
    s = allocVector(CHARSXP, strlen(tmp));
    memcpy(CHAR(s), tmp, strlen(tmp));
    return s;
}

static SEXP
cky_read_vec (SEXPTYPE type, SEXP sym_table, SEXP env_table, FILE *fp)
```

```
{
    int length, count;
    SEXP my_vec;

    length = InInteger(fp);
    my_vec = allocVector(type, length);
    switch (type) {
    case CHARSXP:
        my_vec = InCHARSXP(fp);
        break;
    case LGLSXP:
    case INTSXP:
        InVec(fp, my_vec, INTEGER, InInteger, length);
        break;
    case REALSXP:
        InVec(fp, my_vec, REAL, InReal, length);
        break;
    case CPLXSXP:
        InVec(fp, my_vec, COMPLEX, InComplex, length);
        break;
    case STRSXP:
        InVec(fp, my_vec, STRING, InCHARSXP, length);
        break;
    case VECSXP:
    case EXPRSXP:
        for (count = 0; count < length; ++count)
            VECTOR(my_vec)[count] = cky_read_item(sym_table, env_table, fp);
        break;
    default:
        error("cky_read_vec called with non-vector type");
    }
    return my_vec;
}

static void
cky_read_auxiliary (SEXP obj, SEXP sym_table, SEXP env_table, FILE *fp)
{
    ATTRIB(obj) = cky_read_item(sym_table, env_table, fp);
    LEVELS(obj) = InInteger(fp);
    OBJECT(obj) = InInteger(fp);
}

static SEXP
cky_read_item (SEXP sym_table, SEXP env_table, FILE *fp)
{
    SEXPTYPE type;
    SEXP s, tmp1, tmp2, tmp3;
    int pos;

    R_assert(TYPEOF(sym_table) == VECSXP && TYPEOF(env_table) == VECSXP);
    type = InInteger(fp);
```

```
if ((s = cky_load_special_hook(type)))
    return s;
switch (type) {
case SYMSXP:
    pos = InInteger(fp);
    s = pos ? VECTOR(sym_table)[pos - 1] : R_NilValue;
    break;
case LISTSXP:
case LANGSXP:
    /*
     * HACK ALERT: lists have to be done this way, instead of
     * just cons'ing tmp2 on top of the list, or else the list
     * will come out reversed.
     */
    tmp1 = cky_read_item(sym_table, env_table, fp);
    tmp2 = cky_read_item(sym_table, env_table, fp);
    tmp3 = s = cky_cons(type, tmp1, tmp2, R_NilValue);
    while (tmp1 != R_NilValue || tmp2 != R_NilValue) {
        tmp1 = cky_read_item(sym_table, env_table, fp);
        tmp2 = cky_read_item(sym_table, env_table, fp);
        tmp3 = CDR(tmp3) = cky_cons(type, tmp1, tmp2, R_NilValue);
    }
    break;
case ENVSXP:
    pos = InInteger(fp);
    s = pos ? VECTOR(env_table)[pos - 1] : R_NilValue;
    break;
case CLOSXP:
case PROMSXP:
    PROTECT(s = allocSExp(type));
    TAG(s) = cky_read_item(sym_table, env_table, fp);
    CAR(s) = cky_read_item(sym_table, env_table, fp);
    CDR(s) = cky_read_item(sym_table, env_table, fp);
    UNPROTECT(1);
    break;
case SPECIALSXP:
case BUILTINSXP:
    AllocBuffer(MAXELTSIZE - 1);
    s = mkPRIMSXP(StrToInternal(InString(fp)), type == BUILTINSXP);
    break;
case CHARSXP:
case LGLSXP:
case INTSXP:
case REALSXP:
case CPLXSXP:
case STRSXP:
case VECSXP:
case EXPRSXP:
    s = cky_read_vec(type, sym_table, env_table, fp);
    break;
default:
        warning("cky_read_item: unknown type %i", type);
        PROTECT(s = allocSExp(type));
        TAG(s) = cky_read_item(sym_table, env_table, fp);
        CAR(s) = cky_read_item(sym_table, env_table, fp);
        CDR(s) = cky_read_item(sym_table, env_table, fp);
        UNPROTECT(1);
    }
    cky_read_auxiliary(s, sym_table, env_table, fp);
    return s;
}

SEXP
cky_DataLoad (FILE *fp)
{
    int sym_count, env_count, count;
    SEXP sym_table, env_table;

    InInit(fp);
    sym_count = InInteger(fp);
    /*
     * the +1 is for the extra environment tacked on the end—the
     * list the user asked to save.
     */
    env_count = InInteger(fp) + 1;
    PROTECT(sym_table = allocVector(VECSXP, sym_count));
    PROTECT(env_table = allocVector(VECSXP, env_count));
    for (count = 0; count < sym_count; ++count) {
        AllocBuffer(MAXELTSIZE - 1);
        VECTOR(sym_table)[count] = install(InString(fp));
    }
    /* allocate the structures first */
    for (count = 0; count < env_count; ++count)
        PROTECT(VECTOR(env_table)[count] = allocSExp(ENVSXP));
    /* now stick stuff in them... don't want to use mkEnv here.... */
    for (count = 0; count < env_count; ++count) {
        SEXP tmp;

        tmp = VECTOR(env_table)[count];
        ENCLOS(tmp) = cky_read_item(sym_table, env_table, fp);
        FRAME(tmp) = cky_read_item(sym_table, env_table, fp);
        TAG(tmp) = cky_read_item(sym_table, env_table, fp);
    }
    /* the +2 is for the first two PROTECT()s (of {sym,env}_table) */
    UNPROTECT(env_count + 2);
    InTerm(fp);
    return FRAME(VECTOR(env_table)[env_count - 1]);
}
```

Appendix D Glossary

closure	A closure is a function with additional variable bindings. In R, these additional bindings come in the form of an attached environment.
composite object	An object built up of other objects (compare *simple object*). Linked lists and language objects are perhaps the most used composite object types in R.
cons cell	In R, basic building units of a linked list; one cons cell makes one list element. In Lisp, cons cells have more general uses that do not apply to R.
environment	Roughly, a set of variables, represented as a list of symbols with their values.
kernel code	In the context of this paper, program code that make up the R interpreter (compare *user code*); much of this are written in C, the rest in Fortran. This paper covers only the C side of the code.
kernel-level	Accessible only to kernel code; said typically of functions and data structures.
language object	A specialised linked list type, that is used to hold user-level code in R.
library function	A user-level function that all user code can access. Library functions usually provide more elegant interfaces to primitive functions; for example, `plot` is a library function that ultimately calls the primitive function `plot.xy`, but is more “user friendly” than the latter.
primitive function	A kernel-level function that is callable by user code. Primitive functions typically provide features that user code will be useless without. Arithmetic operators and assignment operators fall into this category; in one version of R there are 442 primitive functions.
simple object	A single standalone object (compare *composite object*). In R, the simple object types are logical, integer, real and complex vectors, as well as primitive function references.
symbol	An object that contains a unique name; these names might be names of variables, functions, whatever. Symbols are stored in a hashed list; for each different name, there is exactly one associated symbol (no two distinct symbols will have the same names).
tagged list	A list where the elements are named. The `ctor` function in Section 2.1.2 [State sharing], page 9 returns a tagged list.
user code	In the context of this paper, program code that are not part of the R interpreter (compare *kernel code*); such code are written in the R programming language, and are interpreted as they execute.
user-level	Accessible to user code (and by extension also kernel code); said typically of functions and data structures.

Colophon

This paper was typeset using *Texinfo*, a macro package that runs on top of TeX. I used a hacked version that changes the default fonts to Times, Courier, Helvetica, and (for the mathematical symbols) Computer Modern. The rendering of 'R' emulates the one seen in the LaTeX version of the R documentation; this was accomplished via the undocumented `@sf` command, which uses the sans-serif font family (much like `\textsf` in LaTeX).

Figures were done with *Xfig* (`http://www.xfig.org/`), a "Facility for Interactive Generation of figures". It's a very cool point-and-click program that, not very surprisingly, draws figures. These were then exported as encapsulated PostScript files and included using the '`epsf`' package (via the `@image` command).

Program code were marked up by hand. The new save/load code, as listed in Appendix C [Complete code listing], page 23, has been incorporated in R from version 0.90 onwards in '`src/main/saveload.c`'; you can use it by defining `USE_NEW_SAVE_FORMAT`. Be aware however that the format *has* changed since I wrote the code.

The program code was based on version 0.64.2 of R, and was originally developed on a 486 machine running Linux. From July 1999, it was moved to a Pentium II machine running OpenBSD, which was also used in making this document.

The source code to this document, its associated figures and macros, should be available at `http://cloud9.hedgee.com/doc/` by the time you read this.

I gratefully acknowledge the assistance of Dr Ross Ihaka, my project supervisor. Many of the ideas used in this project have been his, and the amount of patience he has shown is amazing, especially during his sabbatical. Thank you, Ross.

You can contact me about this paper via the email address `cky-doc@m.org.nz`.