R Internals

Version 2.4.0 (2006-11-25)

R Development Core Team

ISBN 3-900051-14-3

Table of Contents

1 R Internal Structures

This appendix is the beginnings of documentation about R internal structures. It is written for the R core team and others studying the code in the 'src/main' directory.

It is a work-in-progress, first begun for R 2.4.0, and should be checked against the current version of the source code.

1.1 SEXPs

What R users think of as *variables* or *objects* are symbols which are bound to a value. The value can be thought of as either a SEXP (a pointer), or the structure it points to, a SEXPREC (and there are alternative forms used for vectors, namely VECSXP pointing to VECTOR_SEXPREC structures). So the basic building blocks of R objects are often called *nodes*, meaning SEXPRECs or VECTOR_SEXPRECs.

Note that the internal structure of the SEXPREC is not made available to R Extensions: rather SEXP is an opaque pointer, and the internals can only be accessed by the functions provided.

Both types of node structure have as their first three fields a 32-bit sxpinfo header and then three pointers (to the attributes and the previous and next node in a doubly-linked list), and then some further fields. On a 32-bit platform a node[1] occupies 28 bytes: on a 64-bit platform typically 56 bytes (depending on alignment constraints).

The first five bits of the sxpinfo header specify one of up to 32 SEXPTYPEs.

1.1.1 SEXPTYPEs

Currently SEXPTYPEs 0:10 and 13:25 are in use. Values 11 and 12 were used for internal factors and ordered factors and have since been withdrawn. Note that the SEXPTYPEs are stored in saved objects and that the ordering of the types is used, so the gap cannot easily be reused.

no	SEXPTYPE	Description
0	NILSXP	NULL
1	SYMSXP	symbols
2	LISTSXP	pairlists
3	CLOSXP	closures
4	ENVSXP	environments
5	PROMSXP	promises
6	LANGSXP	language objects
7	SPECIALSXP	special functions
8	BUILTINSXP	builtin functions
9	CHARSXP	internal character strings
10	LGLSXP	logical vectors
13	INTSXP	integer vectors
14	REALSXP	numeric vectors
15	CPLXSXP	complex vectors

[1] strictly, a SEXPREC node; VECTOR_SEXPREC nodes are slightly smaller but followed by data in the node.

16	STRSXP	character vectors
17	DOTSXP	dot-dot-dot object
18	ANYSXP	make “any” args work
19	VECSXP	list (generic vector)
20	EXPRSXP	expression vector
21	BCODESXP	byte code
22	EXTPTRSXP	external pointer
23	WEAKREFSXP	weak reference
24	RAWSXP	raw vector
25	S4SXP	S4 classes not of simple type

Many of these will be familiar from R level: the atomic vector types are `LGLSXP`, `INTSXP`, `REALSXP`, `CPLXSP`, `STRSXP` and `RAWSXP`. Lists are `VECSXP` and names (also known as symbols) are `SYMSXP`. Pairlists (`LISTSXP`, the name going back to the origins of R as a Scheme-like language) are rarely seen at R level, but are for example used for argument lists. Character vectors are effectively lists all of whose elements are `CHARSXP`, a type that is rarely visible at R level.

Language objects (`LANGSXP`) are calls (including formulae and so on). Internally they are pairlists with first element a reference[2] to the function to be called with remaining elements the actual arguments for the call (and with the tags if present giving the specified argument names).

Expressions are of type `EXPRSXP`: they are a vector of language objects most often seen as the result of `parse()`.

The functions are of types `CLOSXP`, `SPECIALSXP` and `BUILTINSXP`: where `SEXPTYPE`s are stored in an integer these are sometimes lumped into a pseudo-type `FUNSXP` with code 99. Functions defined via `function` are of type `CLOSXP` and have formals, body and environment.

The `SEXPTYPE` `S4SXP` was introduced in R 2.4.0 for S4 classes which were previously represented as empty lists, that is objects which do not consist solely of a simple type such as an atomic vector or function.

1.1.2 Rest of header

The `sxpinfo` header is defined as a 32-bit C structure by

```
struct sxpinfo_struct {
    SEXPTYPE type      :  5;  /* discussed above */
    unsigned int obj   :  1;  /* is this an object with a class attribute? */
    unsigned int named :  2;  /* used to control copying */
    unsigned int gp    : 16;  /* general purpose, see below */
    unsigned int mark  :  1;  /* mark object as ‘in use’ in GC */
    unsigned int debug :  1;
    unsigned int trace :  1;
    unsigned int spare :  1;  /* unused */
    unsigned int gcgen :  1;  /* generation for GC */
    unsigned int gccls :  3;  /* class of node for GC */
};  /*              Tot: 32 */
```

[2] a pointer to a function or a symbol to look up the function by name, or a language object to be evaluated to give a function.

The `debug` bit is used for closures and environments. For closures it is set by `debug()` and unset by `undebug()`, and indicates that evaluations of the function should be run under the browser. For environments it indicates whether the browsing is in single-step mode.

The `trace` bit is used for functions for `trace()` and for other objects when tracing duplications (see `tracemem`).

The `named` field is set and accessed by the `SET_NAMED` and `NAMED` macros, and take values `0`, `1` and `2`. R has a 'call by value' illusion, so an assignment like

```
b <- a
```

appears to make a copy of `a` and refer to it as `b`. However, if neither `a` nor `b` are subsequently altered there is no need to copy. What really happens is that a new symbol `b` is bound to the same value as `a` and the `named` field on the value object is set (in this case to `2`). When an object is about to be altered, the `named` field is consulted. A value of `2` means that the object must be duplicated before being changed. (Note that this does not say that it is necessary to duplicate, only that it should be duplicated whether necessary or not.) A value of `0` means that it is known that no other `SEXP` shares data with this object, and so it may safely be altered. A value of `1` is used for situations like

```
dim(a) <- c(7, 2)
```

where in principle two copies of `a` exist for the duration of the computation as (in principle)

```
a <- `dim<-`(a, c(7, 2))
```

but for no longer, and so some internal functions can be optimized to avoid a copy in this case.

The `gp` bits are by definition 'general purpose'. As of version 2.4.0 of R, bit 4 (i.e., the fifth bit) is turned on to mark S4 objects. Bits 0-3 and bits 14-15 have been used previously as described below (from detective work on the sources).

The bits can be accessed and set by the `LEVELS` and `SETLEVELS` macros, which names appear to date back to the internal factor and ordered types and are now used in only a few places in the code. The `gp` field is serialized/unserialized for the `SEXPTYPE`s other than `NILSXP`, `SYMSXP` and `ENVSXP`.

If we label the bits from 0, bits 14 and 15 of `gp` are used for 'fancy bindings'. Bit 14 is used to lock a binding or an environment, and bit 15 is used to indicate an active binding. (For the definition of an 'active binding' see the header comments in file '`src/main/envir.c`'.) Bit 15 is used for an environment to indicate if it participates in the global cache.

Almost all other uses seem to be only of bits 0 and 1, although one reserves the first four bits.

The macros `ARGUSED` and `SET_ARGUSED` are used when matching actual and formal function arguments, and take the values 0, 1 and 2.

The macros `MISSING` and `SET_MISSING` are used for pairlists of arguments. Four bits are reserved, but only two are used (and exactly what for is not explained). It seems that bit 0 is used by `matchArgs` to mark missingness on the returned argument list, and bit 1 is used to mark the use of a default value for an argument copied to the evaluation frame of a closure.

Bit 0 is used by macros `DDVAL` and `SET_DDVAL`. This indicates that a `SYMSXP` is one of the symbols `..n` which are implicitly created when `...` is processed, and so indicates that it may need to be looked up in a `DOTSXP`.

Bit 0 is used for PRSEEN, a flag to indicate if a promise has already been seen during the evaluation of the promise (and so to avoid recursive loops).

Bit 0 is used for HASHASH, on the PRINTNAME of the TAG of the frame of an environment.

Bits 0 and 1 are used for weak references (to indicate 'ready to finalize', 'finalize on exit').

Bit 0 is used by the condition handling system (on a VECSXP) to indicate a calling handler.

1.1.3 The 'data'

A SEXPREC is a C structure containing the 32-bit header as described above, three pointers (to the attributes, previous and next node) and the node data, a union

```
union {
    struct primsxp_struct primsxp;
    struct symsxp_struct symsxp;
    struct listsxp_struct listsxp;
    struct envsxp_struct envsxp;
    struct closxp_struct closxp;
    struct promsxp_struct promsxp;
} u;
```

All of these alternatives apart from the first (an int) are three pointers, so the union occupies three words.

The vector types are RAWSXP, CHARSXP, LGLSXP, INTSXP, REALSXP, CPLXSXP, STRSXP, VECSXP, EXPRSXP and WEAKREFSXP. Remember that such types are a VECTOR_SEXPREC, which again consists of the header and the same three pointers, but followed by two integers giving the length and 'true length'[3] of the vector, and then followed by the data (aligned as required: on most 32-bit systems with a 24-byte VECTOR_SEXPREC node the data can follow immediately after the node). The data is a block of memory of the appropriate length to store 'true length' elements (rounded up to a multiple of 8 bytes, with the 8-byte blocks being the 'Vcells' referred in the documentation for gc()).

The 'data' for the various types are given in the table below. A lot of this is interpretation, i.e. the types are not checked.

NILSXP
: There is only one object of type NILSXP, R_NilValue, with no data.

SYMSXP
: Pointers to three nodes, the name, value and internal, accessed by PRINTNAME (a CHARSXP), SYMVALUE and INTERNAL. (If the symbol's value is a .Internal function, the last is a pointer to the appropriate SEXPREC.) Many symbols have SYMVALUE R_UnboundValue.

LISTSXP
: Pointers to the CAR, CDR (usually a LISTSXP or NULL) and TAG (usually a SYMSXP).

CLOSXP
: Pointers to the formals (a pairlist), the body and the environment.

[3] This is almost unused. The only current use is for hash tables of environments (VECSXPs), where length is the size of the table and truelength is the number of primary slots in use, and for the reference hash tables in serialization(VECSXPs), where truelength is the number of slots in use.

ENVSXP
: Pointers to the frame, enclosing environment and hash table (NULL or a VECSXP). A frame is a tagged pairlist with tag the symbol and CAR the bound value.

PROMSXP
: Pointers to the value, expression and environment (in which to evaluate the expression). Once an promise has been evaluated, the environment is set to NULL.

LANGSXP
: A special type of LISTSXP used for function calls. (The CAR references the function (perhaps via a symbol or language object), and the CDR the argument list with tags for named arguments.) R-level documentation references to 'expressions' / 'language objects' are mainly LANGSXPs, but can be symbols (SYMSXPs) or expression vectors (EXPRSXPs).

SPECIALSXP
BUILTINSXP
: An integer giving the offset into the table of primitives/.Internals.

CHARSXP
: length, truelength followed by a block of bytes (allowing for the nul terminator).

LGLSXP
INTSXP
: length, truelength followed by a block of C ints (which are 32 bits on all R platforms).

REALSXP
: length, truelength followed by a block of C doubles

CPLXSXP
: length, truelength followed by a block of C99 double complexs, or equivalent structures.

STRSXP
: length, truelength followed by a block of pointers (SEXPs pointing to CHARSXPs).

DOTSXP
: A special type of LISTSXP for the value bound to a ... symbol: a pairlist of promises.

ANYSXP
: This is used as a place holder for any type: there are no actual objects of this type.

VECSXP
EXPRSXP
: length, truelength followed by a block of pointers. These are internally identical (and identical to STRSXP) but differ in the interpretations placed on the elements.

BCODESXP
: For the future byte-code compiler.

EXTPTRSXP
: Has three pointers, to the pointer, the protection value (an R object which if alive protects this object) and a tag (a SYMSXP?).

WEAKREFSXP
: A WEAKREFSXP is a special VECSXP of length 4, with elements 'key', 'value', 'finalizer' and 'next'. The 'key' is NULL, an environment or an external pointer, and the 'finalizer' is a function or NULL.

RAWSXP
: length, truelength followed by a block of bytes.

S4SXP
: two unused pointers and a tag.

1.1.4 Allocation classes

As we have seen, the field `gccls` in the header is three bits to label up to 8 classes of nodes. Non-vector nodes are of class 0, and 'small' vector nodes are of classes 1 to 6, with 'large' vector nodes being of class 7. The 'small' vector nodes are able to store vector data of up to 8, 16, 32, 48, 64 and 128 bytes: larger vectors are `malloc`-ed individually whereas the 'small' nodes are allocated from pages of about 2000 bytes.

1.2 Environments and variable lookup

What users think of as 'variables' are symbols which are bound to objects in 'environments'. The word 'environment' is used ambiguously in R to mean *either* the frame of an `ENVSXP` (a pairlist of symbol-value pairs) *or* an `ENVSXP`, a frame plus an enclosure.

There are additional places that 'variables' can be looked up, called 'user databases' in comments in the code. These seem undocumented in the R sources, but apparently refer to the **RObjectTable** package at http://www.omegahat.org/RObjectTables/.

The base environment is special. There is an `ENVSXP` environment with enclosure the empty environment `R_EmptyEnv`, but the frame of that environment is not used. Rather its bindings are part of the global symbol table, being those symbols in the global symbol table whose values are not `R_UnboundValue`. When R is started the internal functions are installed (by C code) in the symbol table, with primitive functions having values and `.Internal` functions having what would be their values in the field accessed by the `INTERNAL` macro. Then `.Platform` and `.Machine` are computed and the base package is loaded into the base environment followed by the system profile.

The frames of environments (and the symbol table) are normally hashed for faster access (including insertion and deletion).

By default R maintains a (hashed) global cache of 'variables' (that is symbols and their bindings) which have been found, and this refers only to environments which have been marked to participate, which consists of the global environment (aka the user workspace), the base environment plus environments[4] which have been `attach`ed. When an environment is either `attach`ed or `detach`ed, the names of its symbols are flushed from the cache. The cache is used whenever searching for variables from the global environment (possibly as part of a recursive search).

1.2.1 Search paths

S has the notion of a 'search path': the lookup for a 'variable' leads (possibly through a series of frames) to the 'session frame' the 'working directory' and then along the search path. The search path is a series of databases (as returned by `search()`) which contain the system functions (but not necessarily at the end of the path, as by default the equivalent of packages are added at the end).

R has a variant on the S model. There is a search path (also returned by `search()`) which consists of the global environment (aka user workspace) followed by environments which have been attached and finally the base environment. Note that unlike S it is not possible to attach environments before the workspace nor after the base environment.

[4] Remember that attaching a list or a saved image actually creates and populates an environment and attaches that.

However, the notion of variable lookup is more general in R, hence the plural in the title of this subsection. Since environments have enclosures, from any environment there is a search path found by looking in the frame, then the frame of its enclosure and so on. Since loops are not allowed, this process will eventually terminate: until R 2.2.0 it always terminated at the base environment, but nowadays it can terminate at either the base environment or the empty environment. (It can be conceptually simpler to think of the search always terminating at the empty environment, but with an optimization to stop at the base environment.) So the 'search path' describes the chain of environments which is taken once the search reaches the global environment.

1.2.2 Name spaces

Name spaces are environments associated with packages (and once again the base package is special and will be considered separately). A package *pkg* with a name space defines two environments `namespace:`*pkg* and `package:`*pkg*: it is `package:`*pkg* that can be `attach`ed and form part of the search path.

The objects defined by the R code in the package are symbols with bindings in the `namespace:`*pkg* environment. The `package:`*pkg* environment is populated by selected symbols fron the `namespace:`*pkg* environment (the exports). The enclosure of this environment is an environment populated with the explicit imports from other name spaces, and the enclosure of *that* environment is the base name space. (So the illusion of the imports being in the name space environment is created via the environment tree.) The enclosure of the base name space is the global environment, so the search from a package name space goes via the (explicit and implicit) imports to the standard 'search path'.

The base name space environment `R_BaseNamespace` is another `ENVSXP` that is special-cased. It is effectively the same thing as the base environment `R_BaseEnv` *except* that its enclosure is the global environment rather than the empty environment: the internal code diverts lookups in its frame to the global symbol table.

1.3 Attributes

As we have seen, every `SEXPREC` has a pointer to the attributes of the node (default `R_NilValue`). The attributes can be accessed/set by the macros/functions `ATTRIB` and `SET_ATTRIB`, but such direct access is normally[5] only used to check if the attributes are `NULL` or to reset them. Otherwise access goes through the functions `getAttrib` and `setAttrib` which impose restrictions on the attributes. One thing to watch is that if you copy attributes from one object to another you may (un)set the `"class"` attribute and so need to copy the object bit as well.

The code assumes that the attributes of a node are either `R_NilValue` or a pairlist of non-zero length (and this is checked by `SET_ATTRIB`). The attributes are named (via tags on the pairlist). The replacement function `attributes<-` ensures that `"dim"` precedes `"dimnames"` in the pairlist. Attribute `"dim"` is one of several that is treated specially: the values are checked, and any `"names"` and `"dimnames"` attributes are removed. Similarly, you cannot set `"dimnames"` without having set `"dim"`, and the value assigned must be a list of the correct length and with elements of the correct lengths (and all zero-length elements are replaced by `NULL`).

[5] An exception is the internal code for `terms.formula` which directly manipulates the attributes.

The other attributes which are given special treatment are `"names"`, `"class"`, `"tsp"`, `"comment"` and `"row.names"`. For pairlist-like objects the names are not stored as an attribute but (as symbols) as the tags: however the R interface makes them look like conventional attributes (one reason why the `"names"` attribute has to be treated specially). The C code ensures that the `"tsp"` attribute is an `REALSXP`, the frequency is positive and the implied length agrees with the number of rows of the object being assigned to. Classes and comments are restricted to character vectors, and assigning a zero-length comment or class removes the attribute. Setting or removing a `"class"` attribute sets the object bit appropriately. Integer row names are converted to and from the internal compact representation.

Care needs to be taken when adding attributes to objects of the types with non-standard copying semantics. There is only one object of type `NILSXP`, `R_NilValue`, and that should never have attributes (and this is enforced in `installAttrib`). For environments, external pointers and weak references, the attributes should be relevant to all uses of the object: it is for example reasonable to have a name for an environment, and also a `"path"` attribute for those environments populated from R code in a package.

When should attributes be preserved under operations on an object? Becker, Chambers & Wilks (1988, pp. 144–6) give some guidance. Scalar functions (those which operate element-by-element on a vector and whose output is similar to the input) should preserve attributes (except perhaps class, and if they do preserve class they need to preserve the `OBJECT` bit). Binary operations normally call `copyMostAttributes` to copy most attributes from the longer argument (and if they are of the same length from both, preferring the values on the first). Here 'most' means all except the `names`, `dim` and `dimnames` which are set appropriately by the code for the operator.

Subsetting (other than by an empty index) generally drops all attributes except `names`, `dim` and `dimnames` which are reset as appropriate. On the other hand, subassignment generally preserves such attributes even if the length is changed. Coercion drops all attributes. For example:

```
> x <- structure(1:8, names=letters[1:8], comm="a comment")
> x[]
a b c d e f g h
1 2 3 4 5 6 7 8
attr(,"comm")
[1] "a comment"
> x[1:3]
a b c
1 2 3
> x[3] <- 3
> x
a b c d e f g h
1 2 3 4 5 6 7 8
attr(,"comm")
[1] "a comment"
> x[9] <- 9
> x
a b c d e f g h
```

```
1 2 3 4 5 6 7 8 9
attr(,"comm")
[1] "a comment"
```

1.4 Contexts

Contexts are the internal mechanism used to keep track of where a computation has got to (and from where), so that control-flow constructs can work and reasonable information can be produced on error conditions, (such as traceback) and otherwise (the `sys.xxx` functions).

Execution contexts are a stack of C `structs`:

```
typedef struct RCNTXT {
    struct RCNTXT *nextcontext; /* The next context up the chain */
    int callflag;               /* The context 'type' */
    JMP_BUF cjmpbuf;            /* C stack and register information */
    int cstacktop;              /* Top of the pointer protection stack */
    int evaldepth;              /* Evaluation depth at inception */
    SEXP promargs;              /* Promises supplied to closure */
    SEXP callfun;               /* The closure called */
    SEXP sysparent;             /* Environment the closure was called from */
    SEXP call;                  /* The call that effected this context */
    SEXP cloenv;                /* The environment */
    SEXP conexit;               /* Interpreted on.exit code */
    void (*cend)(void *);       /* C on.exit thunk */
    void *cenddata;             /* Data for C on.exit thunk */
    char *vmax;                 /* Top of the R_alloc stack */
    int intsusp;                /* Interrupts are suspended */
    SEXP handlerstack;          /* Condition handler stack */
    SEXP restartstack;          /* Stack of available restarts */
} RCNTXT, *context;
```

plus additional fields for the future byte-code compiler. The 'types' are from

```
enum {
    CTXT_TOPLEVEL = 0,  /* toplevel context */
    CTXT_NEXT     = 1,  /* target for next */
    CTXT_BREAK    = 2,  /* target for break */
    CTXT_LOOP     = 3,  /* break or next target */
    CTXT_FUNCTION = 4,  /* function closure */
    CTXT_CCODE    = 8,  /* other functions that need error cleanup */
    CTXT_RETURN   = 12, /* return() from a closure */
    CTXT_BROWSER  = 16, /* return target on exit from browser */
    CTXT_GENERIC  = 20, /* rather, running an S3 method */
    CTXT_RESTART  = 32, /* a call to restart was made from a closure */
    CTXT_BUILTIN  = 64  /* builtin internal function */
};
```

where the `CTXT_FUNCTION` bit is on wherever function closures are involved.

Contexts are created by a call to `begincontext` and ended by a call to `endcontext`: code can search up the stack for a particular type of context via `findcontext` (and jump there) or jump to a specific context via `R_JumpToContext`. `R_ToplevelContext` is the 'idle' state (normally the command prompt), and `R_GlobalContext` is the top of the stack.

Note that whilst all calls to closures set a context, those to special internal functions never do, and those to builtin internal functions have done so only recently (and prior to that only when profiling).

Dispatching from a S3 generic (via `UseMethod` or its internal equivalent) or calling `NextMethod` sets the context type to `CTXT_GENERIC`. This is used to set the `sysparent` of the method call to that of the `generic`, so the method appears to have been called in place of the generic rather than from the generic.

The R `sys.frame` and `sys.call` work by counting calls to closures (type `CTXT_FUNCTION`) from either end of the context stack.

Note that the `sysparent` element of the structure is not the same thing as `sys.parent()`. Element `sysparent` is primarily used in managing changes of the function being evaluated, i.e. by `Recall` and method dispatch.

`CTXT_CCODE` contexts are currently used in `cat()`, `load()`, `scan()` and `write.table()` (to close the connection on error), by `PROTECT`, serialization (to recover from errors, e.g. free buffers) and within the error handling code (to raise the C stack limit and reset some variables).

1.5 Argument evaluation

As we have seen, functions in R come in three types, closures (`SEXPTYPE CLOSXP`), specials (`SPECIALSXP`) and builtins (`BUILTINSXP`). In this section we consider when (and if) the actual arguments of function calls are evaluated. The rules are different for the internal (special/builtin) and R-level functions (closures).

For a call to a closure, the actual and formal arguments are matched and a matched call (another `LANGSXP`) is constructed. This process first replaces the actual argument list by a list of promises to the values supplied. It then constructs a new environment which contains the names of the formal parameters matched to actual or default values: all the matched values are promises, the defaults as promises to be evaluated in the environment just created. That environment is then used for the evaluation of the body of the function, and promises will be forced (and hence actual or default arguments evaluated) when they are encountered.

If the closure is an S3 generic (that is, contains a call to `UseMethod`) the evaluation process is the same until the `UseMethod` call is encountered. At that point the argument on which to do dispatch (normally the first) will be evaluated if it has not been already. If a method has been found which is a closure, a new evaluation environment is created for it containing the matched arguments of the method plus any new variables defined so far during the evaluation of the body of the generic. (Note that this means changes to the values of the formal arguments in the body of the generic are discarded when calling the method, but *actual* argument promises which have been forced retain the values found when they were forced. On the other hand, missing arguments have values which are promises to use the default supplied by the method and not the generic.) If the method found is a special or builtin it is called with the matched argument list of promises (possibly already forced) used for the generic.

The essential difference[6] between special and builtin functions is that the arguments of specials are not evaluated before the C code is called, and those of builtins are. In each case positional matching of arguments is used. Note that being a special/builtin is separate from being primitive or `.Internal`: `function` is a special primitive, `+` is a builtin primiitve, `switch` is a special `.Internal` and `grep` is a builtin `.Internal`.

Many of the internal functions are internal generics, which for specials means that they do not evaluate their arguments on call, but the C code starts with a call to `DispatchOrEval`. The latter evaluates the first argument, and looks for a method based on its class. If it finds a method, it dispatches to that method with a call based on promises to evaluate the remaining arguments. If no method is found, the remaining arguments are evaluated before return to the internal generic.

1.5.1 Missingness

Actual arguments to (non-internal) R functions can be fewer than are required to match the formal arguments of the function. Having unmatched formal arguments will not matter if the argument is never used (by lazy evaluation), but when the argument is evaluated, either its default value is evaluated (within the evaluation environment of the function) or an error is thrown with a message along the lines of

```
argument "foobar" is missing, with no default
```

Internally missingness is handled by two mechanisms. The object `R_MissingArg` is used to indicate that a formal argument has no (default) value. When matching the actual arguments to the formal arguments, a new argument list is constructed from the formals all of whose values are `R_MissingArg` with the first `MISSING` bit set. Then whenever a formal argument is matched to an actual argument, the corresponding member of the new argument list has its value set to that of the matched actual argument, and if that is not `R_MissingArg` the missing bit is unset.

This new argument list is used to form the evaluation frame for the function, and if named arguments are subsequently given a new value (before they are evaluated) the missing bit is cleared.

Missingness of arguments can be interrogated via the `missing()` function. An argument is clearly missing if its missing bit is set or if the value is `R_MissingArg`. However, missingness can be passed on from function to function, for using a formal argument as an actual argument in a function call does not count as evaluation. So `missing()` has to examine the value (a promise) of a non-yet-evaluated formal argument to see if it might be missing, which might involve investigating a promise and so on . . . .

1.5.2 Dot-dot-dot arguments

Dot-dot-dot arguments are convenient when writing functions, but complicate the internal code for argument evaluation.

The formals of a function with a `...` argument represent that as a single argument like any other argument, with tag the symbol `R_DotsSymbol`. When the actual arguments are matched to the formals, the value of the `...` argument is of `SEXPTYPE DOTSXP`, a pairlist of promises (as used for matched arguments) but distinguished by the `SEXPTYPE`.

[6] There is currently one other difference: when profiling builtin functions are counted as function calls but specials are not.

Recall that the evaluation frame for a function initially contains the *name=value* pairs from the matched call, and hence this will be true for `...` as well. The value of `...` is a (special) pairlist whose elements are referred to by the special symbols `..1`, `..2`, `...` which have the `DDVAL` bit set: when one of these is encountered it is looked up (via `ddfndVar`) in the value of the `...` symbol in the evaluation frame.

Values of arguments matched to a `...` argument can be missing.

1.6 The write barrier and the garbage collector

R has since version 1.2.0 had a generational garbage collector, and bit `gcgen` in the `sxpinfo` header is used in the implementation of this. This is used in conjunction with the `mark` bit to identify two previous generations.

There are three levels of collections. Level 0 collects only the youngest generation, level 1 collects the two youngest generations and level 2 collects all generations. After 20 level-0 collections the next collection is at level 1, and after 5 level-1 collections at level 2. Further, if a level-*n* collection fails to provide 20% free space (for each of nodes and the vector heap), the next collection will be at level *n+1*. (The R-level function `gc()` performs a level-2 collection.)

A generational collector needs to efficiently 'age' the objects, especially list-like objects (including `STRSXP`s). This is done by ensuring that the elements of a list are regarded as at least as old as the list *when they are assigned*. This is handled by the functions `SET_VECTOR_ELT` and `SET_STRING_ELT`, which is why they are functions and not macros. Ensuring the integrity of such operations is termed the *write barrier* and is done by making the `SEXP` opaque and only providing access via functions (which cannot be used as lvalues in assignments in C).

All code in R extensions is by default behind the write barrier. The only way to obtain direct access to the internals of the `SEXPREC`s is to define '`USE_RINTERNALS`' before including '`Rinternals.h`', which is normally defined in '`Defn.h`'. To enable a check on the way that the access is used, R can be compiled with flag '`--enable-strict-barrier`' which ensures that '`Defn.h`' does not define '`USE_RINTERNALS`' and hence that `SEXP` is opaque in most of R itself. (There are some necessary exceptions: foremost '`memory.c`' where the accessor functions are defined and also '`size.c`' which needs access to the sizes of the internal structures.)

For background papers see http://www.stat.uiowa.edu/~luke/R/barrier.html and http://www.stat.uiowa.edu/~luke/R/gengcnotes.html.

1.7 Serialization Formats

Serialized versions of R objects are used by `load`/`save` and also at a lower level by `.saveRDS`/`.readRDS` and `serialize`/`unserialize`. These differ in what they serialize to (a file, a connection, a raw vector) and whether they are intended to serialize a single object or a collection of objects (typically a workspace). `save` writes a header indicating the format at the beginning of the file (a single LF-terninated line) which the lower-level versions do not.

R has used the same serialization format since R 1.4.0 in December 2001. Reading of earlier formats is still supported via `load`, but they are not described here. (Files of most of these formats can still be found in 'data' directories of packages.) The current serialization format is called 'version 2', and has been expanded in back-compatible ways since R 1.4.0, for example to support additional SEXPTYPEs.

save() works by first creating a tagged pairlist of objects to be saved, and then saving that single object preceded by a single-line header (typically RDX2\n for a binary save). load() reads the header line, unserializes a single object (a pairlist or a vector list) and assigns the elements of the list in the appropriate environment.

Serialization in R needs to take into account that objects may contain references to environments, which then have enclosing environments and so on. (Environments recognized as package or name space environments are saved by name.) Further, there are 'reference objects' which are not duplicated on copy and should should remain shared on unserialization. These are weak references, external pointers and environments other than those associated with packages, name spaces and the global environment. These are handled via a hash table, and references after the first are written out as a reference marker indexed by the table entry.

Serialization first writes a header indicating the format (normally 'X\n' for an XDR format binary save) and the version number of the format and of two R versions (as integers). (Unserialization interprets the two versions as the version of R which wrote the file followed by the minimal version of R needed to read the format.) Serialization then writes out the object recursively using function WriteItem in file 'src/main/serialize.c'.

Some objects are written as if they were SEXPTYPEs: such pseudo-SEXPTYPEs cover R_NilValue, R_EmptyEnv, R_BaseEnv, R_GlobalEnv, R_UnboundValue, R_MissingArg and R_BaseNamespace.

For all SEXPTYPEs except NILSXP, SYMSXP and ENVSXP serialization starts with a integer with the SEXPTYPE in bits 0:7[7] followed by the object bit, two bits indicating if there are any attributes and if there is a tag (for the pairlist types), an unused bit and then the gp field[8] in bits 12:27. Pairlist-like objects write their attributes (if any), tag (if any), CAR and then CDR (using tail recursion): other objects write their attributes after themselves. Atomic vector objects write their length followed by the data: generic vector-list objects write the length followed by a call to WriteItem for each element. The code for CHARSXPs special-cases NA_STRING and writes it as length -1 with no data.

Environments are treated in several ways: as we have seen, some are written as specific pseudo-SEXPTYPEs. Package and name space environments are written with pseudo-SEXPTYPEs followed by the name. 'Normal' environments are written out as ENVSXPs with an integer indicating if the environment is locked followed by the enclosure, frame, tag and attributes.

1.8 Internal use of global and base environments

This section notes known use by the system of these environments: the intention is to minimize or eliminate them.

[7] only 0:4 will currently be used for SEXPTYPEs but values 241:255 are used for pseudo-SEXPTYPEs.

[8] Currently the only relevant bits are 0:1, 4, 14:15.

1.8.1 Base environment

The graphics devices system maintains two variables `.Device` and `.Devices` in the base environment: both are always set. The variable `.Devices` gives a list of character vectors of the names of open devices, and `.Device` is the element corresponding to the currently active device. The null device will always be open.

There appears to be a variable `.Options`, a pairlist giving the current options settings. But in fact this is just a symbol with a value assigned, and so shows up as a base variable.

Similarly, the evaluator creates a symbol `.Last.value` which appears as a variable in the base environment.

Errors can give rise to objects `.Traceback` and `last.warning` in the base environment.

1.8.2 Global environment

The seed for the random number generator is stored in object `.Random.seed` in the global environment.

Some error handlers may give rise to objects in the global environment: for example `dump.frames` by default produces `last.dump`.

The `windows()` device makes use of a variable `.SavedPlots` to store display lists of saved plots for later display. This is regarded as a variable created by the user.

1.9 Modules

R makes use of a number of shared objects/DLLs stored in the '`modules`' directory. These are parts of the code which have been chosen to be loaded 'on demand' rather than linked as dynamic libraries or incorporated into the main executable/dynamic library.

For a few of these (e.g. `vfonts`) the issue is size: the database for the Hershey fonts is included in the C code of the module and was at one time an appreciable part of the codebase for a rarely used feature. However, for most of the modules the motivation has been the amount of (often optional) code they will bring in via libraries to which they are linked.

`internet` The internal HTTP and FTP clients and socket support, which link to system-specific support libraries.

`lapack` The code which makes use of the LAPACK library, and is linked to '`libRlapack`' or an external LAPACK library.

`vfonts` The Hershey font databases and the code to draw from them.

`X11` (Unix-alikes only.) The `X11()`, `jpeg()` and `png()` devices. These are optional, and link to the `X11`, `jpeg` and `libpng` libraries.

'`Rbitmap.dll`'
(Windows only.) The code for the BMP, JPEG and PNG devices and for saving on-screen graphs to those formats. This is technically optional, and needs source code not in the tarball.

'`Rchtml.dll`'
(Windows only.) A link to an ActiveX control that displays Compiled HTML help. This is optional, and only compiled if CHTML is specified.

‘iconv.dll’
: (Windows only.) A DLL compiled via Visual C++ which contains the routines to convert between character sets.

‘internet2.dll’
: (Windows only.) An alternative version of the internet access routines, compiled against Internet Explorer internals (and so loads ‘wininet.dll’ and ‘wsock32.dll’).

2 .Internal vs .Primitive

C code compiled into R at build time can be called "directly" or via the `.Internal` interface, which is very similar to the `.External` interface except in syntax. More precisely, R maintains a table of R function names and corresponding C functions to call, which by convention all start with '`do_`' and return a SEXP. Via this table (`R_FunTab` in file '`src/main/names.c`') one can also specify how many arguments to a function are required or allowed, whether the arguments are to be evaluated before calling or not, and whether the function is "internal" in the sense that it must be accessed via the `.Internal` interface, or directly accessible in which case it is printed in R as `.Primitive`.

R's functionality can also be extended by providing corresponding C code and adding to this function table.

In general, all such functions use `.Internal()` as this is safer and in particular allows for transparent handling of named and default arguments. For example, `axis` is defined as

```
axis <- function(side, at = NULL, labels = NULL, ...)
    .Internal(axis(side, at, labels, ...))
```

However, for reasons of convenience and also efficiency (as there is some overhead in using the `.Internal` interface wrapped in a function closure), there are exceptions which can be accessed directly. Note that these functions make no use of R code, and hence are very different from the usual interpreted functions. In particular, `args`, `formals` and `body` return `NULL` for such objects, and argument matching is purely positional (with empty positions being dropped).

The list of these "primitive" functions is subject to change: currently, it includes the following.

1. "Special functions" which really are *language* elements, however exist as "primitive" functions in R:

   ```
   {        (          if      for       while   repeat   break   next
   return   function   quote   on.exit
   ```

2. Basic *operator*s (i.e., functions usually *not* called as `foo(a, b, ...)`) for subsetting, assignment, arithmetic and logic. These are the following 1-, 2-, and N-argument functions:

   ```
                     [    [[    $     <-    <<-   =     [<-   [[<-   $<-

   +    -    *    /    ^    %%    %*%   %/%
   <    <=   ==   !=   >=   >
   |    ||   &    &&   !
   ```

3. "Low level" 0- and 1-argument functions which belong to one of the following groups of functions:

   a. Basic mathematical functions with a single argument, i.e.,

      ```
      sign    abs
      floor   ceiling  trunc
      ```

```
sqrt    exp
cos     sin     tan
acos    asin    atan
cosh    sinh    tanh
acosh   asinh   atanh

cumsum  cumprod
cummax  cummin

Im      Re
Arg     Conj    Mod
```

Note however that the R function `log` has an optional named argument `base`, and therefore is defined as

```
log <- function(x, base = exp(1)) {
  if(missing(base))
    .Internal(log(x))
  else
    .Internal(log(x, base))
}
```

in order to ensure that `log(x = pi, base = 2)` is identical to `log(base = 2, x = pi)`.

b. Functions rarely used outside of "programming" (i.e., mostly used inside other functions), such as

```
nargs          missing
interactive    is.xxx
.Primitive     .Internal
symbol.C       symbol.For
globalenv      baseenv      emptyenv      pos.to.env
unclass
seq_along      seq_len
```

(where *xxx* stands for almost 30 different notions, such as `function`, `vector`, `numeric`, and so forth, but not `is.loaded`).

c. The programming and session management utilities

```
debug     undebug    trace  untrace
browser   proc.time  gc.time
```

4. The following basic assignment and extractor functions

```
length      length<-
class       class<-
oldClass    oldCLass<-
attr        attr<-
attributes  attributes<-
dim         dim<-
dimnames    dimnames<-
            environment<-
```

5. The following few *N*-argument functions are "primitive" for efficiency reasons:

```
:          ~          c            list
call       as.call    as.character
expression substitute as.environment
UseMethod  invisible  standardGeneric
.C         .Fortran   .Call        .External
.Call.graphics        .External.graphics
.subset    .subset2   .primTrace   .primUntrace
rep        seq.int
```

`rep` and `seq.int` manage their own argument matching and so do work in the standard way.

3 R coding standards

R is meant to run on a wide variety of platforms, including Linux and most variants of Unix as well as 32-bit Windows versions and on MacOS X. Therefore, when extending R by either adding to the R base distribution or by providing an add-on package, one should not rely on features specific to only a few supported platforms, if this can be avoided. In particular, although most R developers use GNU tools, they should not employ the GNU extensions to standard tools. Whereas some other software packages explicitly rely on e.g. GNU make or the GNU C++ compiler, R does not. Nevertheless, R is a GNU project, and the spirit of the *GNU Coding Standards* should be followed if possible.

The following tools can "safely be assumed" for R extensions.

- As from R 2.3.0 an ISO C99 C compiler. Note that extensions such as POSIX 1003.1 must be tested for, typically using Autoconf unless you are sure they are supported on all mainstream R platforms (including Windows and MacOS X). Packages will be more portable if written assuming only C89, but this should not be done where using C99 features will make for cleaner or more robust code.
- A FORTRAN 77 compiler (but not Fortran 9x).
- A simple `make`, considering the features of `make` in 4.2 BSD systems as a baseline.

  GNU or other extensions, including pattern rules using '`%`', the automatic variable '`$^`', the '`+=`' syntax to append to the value of a variable, the ("safe") inclusion of makefiles with no error, conditional execution, and many more, must not be used (see Chapter "Features" in the *GNU Make Manual* for more information). On the other hand, building R in a separate directory (not containing the sources) should work provided that `make` supports the `VPATH` mechanism.

  Windows-specific makefiles can assume GNU `make` 3.75 or later, as no other `make` is viable on that platform.
- A Bourne shell and the "traditional" Unix programming tools, including `grep`, `sed`, and `awk`.

  There are POSIX standards for these tools, but these may not fully be supported. Baseline features could be determined from a book such as *The UNIX Programming Environment* by Brian W. Kernighan & Rob Pike. Note in particular that '`|`' in a regexp is an extended regexp, and is not supported by all versions of `grep` or `sed`. The Open Group Base Specifications, Issue 6, which is technicall identical to ISO/IEC 9945 and IEEE Std 1003.1 (POSIX), 2004, are available at http://www.opengroup.org/onlinepubs/009695399/mindex.html.

Under Windows, most users will not have these tools installed, and you should not require their presence for the operation of your package. However, users who install your package from source will have them, as they can be assumed to have followed the instructions in "the Windows toolset" appendix of the "R Installation and Administration" manual to obtain them. Redirection cannot be assumed to be available via `system` as this does not use a standard shell (let alone a Bourne shell).

In addition, the following tools are needed for certain tasks.

- Perl version 5 is needed for converting documentation written in Rd format to plain text, HTML, LaTeX, and to extract the examples. In addition, several other tools, in particular `check` and `build` require Perl.

The R Core Team has decided that Perl (version 5) can safely be assumed for building R from source, building and checking add-on packages, and for installing add-on packages from source. On the other hand, Perl cannot be assumed at all for installing *binary* (pre-built) versions of add-on packages, or at run time.

- Makeinfo version 4.7 is needed to build the Info files for the R manuals written in the GNU Texinfo system. (Future distributions of R may contain the Info files.)

It is also important that code is written in a way that allows others to understand it. This is particularly helpful for fixing problems, and includes using self-descriptive variable names, commenting the code, and also formatting it properly. The R Core Team recommends to use a basic indentation of 4 for R and C (and most likely also Perl) code, and 2 for documentation in Rd format. Emacs users can implement this indentation style by putting the following in one of their startup files. (For GNU Emacs 20: for GNU Emacs 21 or later use customization to set the `c-default-style` to `"bsd"` and `c-basic-offset` to 4.)

```
;;; C
(add-hook 'c-mode-hook
          (lambda () (c-set-style "bsd")))
;;; ESS
(add-hook 'ess-mode-hook
          (lambda ()
            (ess-set-style 'C++)
            ;; Because
            ;;                                 DEF GNU BSD K&R C++
            ;; ess-indent-level                  2   2   8   5   4
            ;; ess-continued-statement-offset    2   2   8   5   4
            ;; ess-brace-offset                  0   0  -8  -5  -4
            ;; ess-arg-function-offset           2   4   0   0   0
            ;; ess-expression-offset             4   2   8   5   4
            ;; ess-else-offset                   0   0   0   0   0
            ;; ess-close-brace-offset            0   0   0   0   0
            (add-hook 'local-write-file-hooks
                      (lambda ()
                        (ess-nuke-trailing-whitespace)))))
(setq ess-nuke-trailing-whitespace-p 'ask)
;; or even
;; (setq ess-nuke-trailing-whitespace-p t)
;;; Perl
(add-hook 'perl-mode-hook
          (lambda () (setq perl-indent-level 4)))
```

(The 'GNU' styles for Emacs' C and R modes use a basic indentation of 2, which has been determined not to display the structure clearly enough when using narrow fonts.)

4 Testing R code

When you (as R developer) add new functions to the R base (all the packages distributed with R), be careful to check if *`make test-Specific`* or particularly, *`cd tests; make no-segfault.Rout`* still works (without interactive user intervention, and on a standalone computer). If the new function, for example, accesses the Internet, or requires GUI interaction, please add its name to the "stop list" in '`tests/no-segfault.Rin`'.

[To be revised: use `make check-devel`, check the write barrier if you change internal structures.]

Function and variable index

Concept index