Noname manuscript No.
(will be inserted by the editor)

Aspects of CXXR Internals

Andrew R. Runnalls

Received: date / Accepted: date

Abstract The CXXR project aims gradually to refactor the fundamental parts of the R interpreter from C into C++ whilst retaining the full functionality of the standard distribution of R. It is hoped that this will enable researchers more easily to enhance the functionality of R by allowing them to extend the interpreter's internal C++ class hierarchy. The paper summarises progress to date and describes key aspects of the internal implementation of the CXXR where this differs from the standard interpreter. It also explains stratagems used to facilitate updating CXXR to reflect a new release of R, and examines the relative performance of CXXR and the standard interpreter.

Keywords R · CXXR · C++ · software refactoring

1 Introduction

The aim of the CXXR project (`http://www.cs.kent.ac.uk/projects/cxxr`) is gradually to refactor the fundamental parts of the R interpreter—the material within the `src/main` directory of the R source distribution—from C into C++, whilst all the time retaining the full functionality of the standard distribution of R. In particular, it is intended that the behaviour of R code be unaffected when run under the CXXR interpreter, unless the code probes into the interpreter internals, and that the R functions `.C`, `.Fortran` and `.Call` work as before to enable R code to call code written in C, C++ and Fortran. Moreover this foreign-language code will need no change provided it communicates with R only *via* the `R.h` or `S.h` APIs. However, external functions compiled against the `Rinternals.h` API may need small changes, as indicated in the CXXR documentation.

Work on CXXR started in May 2007, at that time shadowing R 2.5.1; the current release shadows R 2.9.2. This paper will refer to the standard R interpreter as **CR**, and will assume some familiarity with the internals and interfaces of the standard interpreter, as documented in 'R Internals' (R Core 2009a) and 'Writing R Extensions' (R Core 2009b).

Andrew Runnalls, School of Computing, University of Kent, Canterbury, Kent CT2 7NF, UK

An important motivation for the project was to provide a basis for introducing **provenance tracking** into R, so that for any R data object, an R user can determine exactly which primary data sources it is based on, and exactly which sequence of operations was used to produce it from the primary data, thus emulating and extending the AUDIT facility of S (Becker et al. 1988).[1] However, it is also hoped that CXXR will help other researchers to produce experimental versions of the R interpreter, this adaptability being achieved partly by improving the internal documentation, and partly by tightening up the encapsulation boundaries within the software. Although CXXR was initiated independently of the proposals in (Temple Lang 2009), it reflects very similar aspirations.

2 CXXR Structure

CXXR code falls into three categories, which can be considered roughly to form three concentric layers. At the centre is the **CXXR core**, which contains functionality that has been fully refactored into C++, following as far as possible the programming idioms of that language, and making free use of the C++ standard library, including some use of the TR1 library extensions (Becker 2006). Everything in the CXXR core is placed in the C++ namespace `CXXR`. The interfaces to the core are defined by header files in `src/include/CXXR` and have been carefully documented using the Doxygen documentation tool (van Heesch 1997).

The following functionality is now within the CXXR core:

- Memory allocation and garbage collection.
- An extensible class hierarchy replacing CR's `SEXPREC` union.
- Object duplication (now handled essentially by C++ copy constructors).
- Environments (i.e. variable to object mappings), with hooks to support provenance tracking.

These topics will be described in more detail in later sections.

The outermost of the three layers is the **packages and modules layer**, which as the name suggests comprises R packages and modules. As was the intention, very few changes have been necessary here, particularly in the packages: for example, only seven files under `src/library` have any (non-comment) changes, and in several cases these rectify what are at least arguably bugs in the CR code. It should be remarked, however, that only packages forming part of the standard R distribution have so far been tested.

In between the core and the packages and modules layer is the **transition layer**, which consists of the CR files in `src/main` adapted to work with the CXXR core. With a few exceptions, C source files have been redesignated as C++, but the programming idioms largely remain those of CR (which in addition to C idioms frequently exhibits those of Fortran and especially LISP). Indeed the general philosophy within the transition layer—not always respected in the early days of CXXR

[1] Provenance tracking facilities are being developed as a separate developmental thread by C. Silles, and are not (yet) an integral part of CXXR; consequently, they will be mentioned only incidentally in this paper.

development—is to avoid any changes to transition layer files that are not strictly required to deploy functionality from the CXXR core. This conservative approach facilitates upgrading CXXR when there is a new release of CR.

Despite this general philosophy, there have been a number of systematic changes to files in the transition layer. For example, in choosing identifier names CR coders often choose C++ reserved words such as `this`, `new` and `class`, and these have to be changed to something else. Also, C++ often requires explicit type conversions in places where C allows implicit conversions. Thirdly, within transition-layer files that are now in C++, C-style casts have everywhere been replaced by C++ casts (such as `static_cast`, `const_cast`, `reinterpret_cast`), which are more conspicuous and give a clearer indication of the programmer's intent. As CXXR development has proceeded, it has proved to be particularly useful to make these systematic changes in such a way that they can easily be reversed by a simple Perl script `uncxxr.pl`.

When it is necessary to upgrade CXXR to a new release of CR, changes to files within the transition layer are scrutinised using a graphical three-way differencing tool to compare (*a*) a file as it appeared in the previous release of CR, (*b*) the file as it appears in the new release of CR, and (*c*) the corresponding CXXR file back-transformed towards CR using `uncxxr.pl`. This backtransformation removes a lot of 'noise' from the comparison, and makes it easier to see where substantive changes have taken place in CR that need to be incorporated into CXXR: these changes can then be accomplished by editing the CXXR code in a separate editor window.

Files within the CXXR transition layer occasionally exhibit unorthodox use of whitespace, and use C preprocessor macros defined within the header file `uncxxr.h` (contrary to general C++ practice which is to shun the use of macros). All this is to enable `uncxxr.pl` better to do its work.

3 The `RObject` Class Hierarchy

A ubiquitous data type within the CR code is `SEXP`, whose underlying definition is 'pointer to `SEXPREC`' (`SEXPREC*`). `SEXPREC` itself is a C union of no fewer than 23 distinct datatypes, distinguished from each other—after the fashion of a Pascal variant record—by a '`SEXPTYPE`' field. However, most CR code does not see the definition of `SEXP` as 'pointer to `SEXPREC`'; instead it sees `SEXP` defined as an opaque pointer, i.e. a pointer to an undefined C `struct`. Consequently, the CR code can access and manipulate `SEXPREC`s only *via* a number of predefined functions such as `REAL()` or `CAR()`.

The resulting encapsulation has been of enormous help to the CXXR refactorisation effort. However, the fact that `SEXPREC` is a C union has disadvantages. It means that the majority of type checking within the CR interpreter must be done at runtime. It has also allowed some questionable programming practices to find their way into the CR codebase. In particular, access functions which are intended to be applied to one type within the union are sometimes applied to an object of another such type, and this inevitably makes it difficult subsequently to change the underlying type definitions.

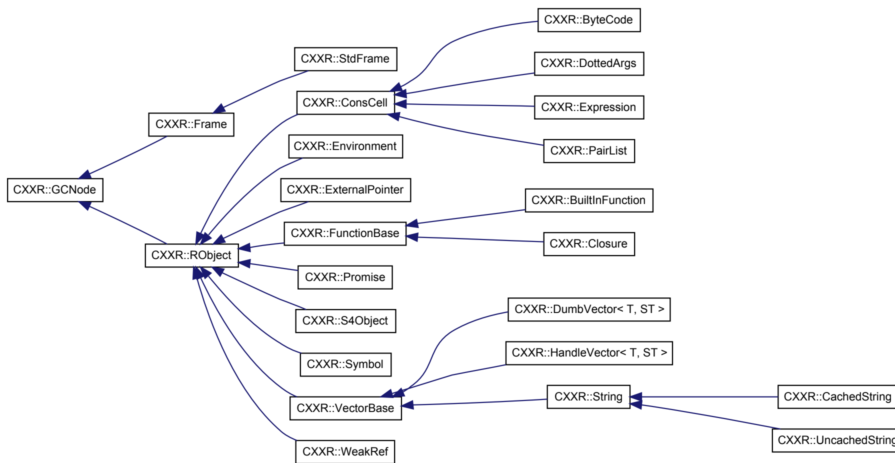

Fig. 1 The main class hierarchy within CXXR (generated using Doxygen and slightly simplified).

`SEXPREC` does not exist in CXXR: it has been replaced by a class hierarchy rooted at the class `RObject`. This is part of a continuing programme to try, as far as possible, to move all program code relating to a particular datatype into one place, and then to use C++ public/protected/private access controls to conceal details of implementation and to defend class invariants. To give an example of a class invariant within CXXR, the code of the base class `RObject` ensures that every attribute of an `RObject` has a distinct `Symbol` object as its tag.

CXXR aims to make it easy for researchers to extend the class hierarchy, and so introduce new sorts of object into R at the interpreter level.

Fig. 1 shows the resulting class hierarchy. `RObject` is the base class of objects intended to be visible in the R language. C++ code sees `SEXP` defined as 'pointer to `RObject`'; C code continues to see `SEXP` as an opaque pointer. `RObject` is also the home of R attributes. For the most part the leaf nodes of the subtree rooted at `RObject` correspond in an obvious way to the various `SEXPTYPE`s described in (R Core 2009a), and merit no particular comment. An exception is `CHARSXP`, which corresponds to `CXXR::String` in the diagram; this is an abstract class with two implementations, `CachedString` and `UncachedString`, the latter of which is obsolescent.

All the vector classes inherit from an abstract class `VectorBase`, which is provided mainly to allow a simple implementation of CR's function `LENGTH()`, which gives the number of elements in the vector. Data vectors (`LGLSXP`, `RAWSXP`, `REALSXP` etc.) are implemented as instantiations of the C++ template `DumbVector<T, ST>`, where the template parameter `T` indicates the C++ type of the elements of the vector, and `ST` indicates the `SEXPTYPE` of the vector as a whole. 'Dumb' in the class name reflects the fact that this template is for use with element types with trivial constructors and destructors and bitwise copying logic: 'plain old data' as it is sometimes called.

R vector types such as `VECSXP` and `EXPRSXP` which comprise pointers to other `RObject`s are implemented using the C++ class template `HandleVector<T, ST>`; the reason for 'handle' in the name will become apparent in Sec. 5.

Classes CachedString and Symbol are unusual in that their constructors are private. Instances of these classes have to be produced using the relevant class's static member function obtain(), and these functions enforce the requirements that at any time there can be at most one CachedString object with a given text and character encoding, and at most one Symbol object corresponding to a given CachedString. Applying these constraints means that various hash tables within CXXR can be keyed on the *address* of a CachedString object, or the *address* of a Symbol, knowing that this corresponds uniquely to the underlying text and encoding. Incidentally, hash tables within the CXXR core are always based on the relevant templates from the TR1 extensions to the C++ standard library, which offer tighter typing than the hashing code within CR.

CR includes a special 'null object', of type NILSXP, pointed to by the global variable R_NilValue. This null object is used a stub, to indicate the absence of a more substantive object. For example, the CDR (successor) field of the last element in a list points to the null object. Because the various SEXPTYPEs of CR are mapped onto distinct C++ types within CXXR, having a real null object would be unworkable, because this object would need to belong simultaneously to all the C++ types that the null object stands in for. Instead of a null *object*, CXXR simply uses a null *pointer*: it defines R_NilValue as a macro expanding to NULL. This entails some extra complexity in the implementation of accessor functions such as CAR(), which in CR are frequently applied to R_NilValue; CXXR needs to handle this case specially.

A deficiency in the RObject class hierarchy as currently defined is the fact that in several places, for example in the inheritance from ConsCell, inheritance represents commonality of implementation rather than the 'is a' relationship favoured by canons of object-oriented programming. Future restructuring of the hierarchy is consequently likely.

4 Memory Allocation and Garbage Collection

Memory allocation and garbage collection (GC) are entirely decoupled from each other in CXXR. Memory allocation is handled by the class MemoryBank, all of whose members are static. Principal among its member functions are allocate() and deallocate, which present an interface similar to malloc() and free() in C. MemoryBank::allocate() routes requests for 'large' memory blocks (currently those of at least 128 bytes) directly to C++'s global operator new. Requests for smaller blocks are serviced from pools of preallocated cells carved out of 'superblocks' of about 4 kB (cf. Meyers 1998, Item 10). These pools are implemented by class CellPool and are provided for various cell sizes from 8 bytes upwards. MemoryBank keeps track of the total number of bytes currently allocated.

There are two principal clients of MemoryBank. First, there is a class called CXXR::Allocator, which is a memory allocator of the sort used by the C++ standard library containers. Using this it is straightforward to declare standard vectors, sets, maps and so on which draw their memory from MemoryBank.

Secondly, there is class `GCNode`, which is the base class for all objects subject to garbage collection. `GCNode` defines its own `operator new` method, which obtains memory *via* `MemoryBank`.

During 2009 CXXR development has experimented with different garbage collection approaches. At the time of the DSC2009 conference, the svn repository was split three ways:

- On the development `trunk`, the approach used was generational mark-sweep, much as in CR itself.
- The branch `refcount+gen` used reference counting, backed up by generational mark-sweep to handle cyclic references. This was an intermediate refactorisation step.
- The branch `gclab` used reference counting backed up by simple, i.e. non-generational, mark-sweep.

The third of these approaches, which is comparable to that in (DeTreville 1990), has moved onto the development trunk as from release 0.21-2.8.1 of CXXR. Reference counting has the advantage that once a memory block becomes garbage, it is quickly reclaimed and reallocated, thus increasing the probability that a newly allocated memory block is already mapped by a processor cache. Reverting from generational mark-sweep to simple mark-sweep means that it is no longer necessary to enforce a 'write barrier', and this greatly simplifies coding, and in particular makes it possible to deploy the facilities of the C++ standard library more extensively.

The remainder of this paper will assume this third approach unless otherwise indicated. In this approach, each `GCNode` has a one-byte saturating reference count: if the reference count ever reaches 255, it stays at that value, and the node can then only be garbage-collected by the mark-sweep mechanism.[2]

When one `GCNode` needs to refer to another, it does so not with an ordinary pointer but with a 'smart pointer' (cf. Meyers 1996, Item 28) from the `GCEdge<T>` class template; such a pointer automatically adjusts the reference count of the node referred to.

Class `GCManager` manages the mark-sweep garbage collection process, including ensuring the correct handling of weak references and finalisers. (Each `WeakRef` object refers to itself *via* a `GCEdge`, to prevent weak references being culled by the reference counting mechanism.) `GCManager` also controls the threshold at which mark-sweep garbage collection takes place; this threshold is based exclusively on the number of bytes allocated from `MemoryBank`, unlike CR which also considers the number of nodes allocated.

A frequent requirement within the interpreter is temporarily to protect a `GCNode` against garbage collection, either because it has not yet been incorporated into the main directed graph of object references, or because it has been detached from this graph. In CR, this is achieved using the `PROTECT()`, `UNPROTECT()` and kindred functions. These functions continue to be available in CXXR (for use in the transition layer), but a better approach is to use a smart pointer from the `GCStackRoot<T>`

[2] Note that this reference count is not visible outside the `GCNode` class. In particular, it does not replace the `NAMED` mechanism used within CR for controlling object duplication, which for the time being continues unchanged within CXXR.

template. The template parameter T indicates the type of GCNode pointed to, by default RObject itself. For most purposes, a GCStackRoot can be handled exactly like an ordinary pointer of type T*: it can be dereferenced, and a pointer destination can be copied from an ordinary T* pointer to a GCStackRoot<T> pointer using ordinary assignment syntax, and *vice versa*. But the destination object of GCStackRoot pointer will be protected from garbage collection for as long as the GCStackRoot points to it.

The GC protection offered by a GCStackRoot object automatically ceases when the GCStackRoot itself ceases to exist, typically when it goes out of scope: this is accomplished by the class destructor. Consequently, using GCStackRoots sidesteps the need manually to balance PROTECT() and UNPROTECT() operations that is necessary in CR. Another advantage is that if a GCStackRoot is declared in the outermost block of a function, its GC protection extends up to *and including* the return statement of the function, whereas CR's UNPROTECT() must necessarily precede the return.

Conceptually, GCStackRoot objects form a stack, and they are consequently subject to the constraint that they must destroyed in the reverse order of their creation, as with pop and push operations on a stack. (There is just one stack, irrespective of the template parameter T.) It is recommended GCStackRoot pointers are always automatic C++ variables, i.e. variables stored on the process's execution stack, in which case the C++ compiler will itself ensure that they are destroyed in the reverse order of creation.

Because of this constraint, GCStackRoots are best suited to short-term protection against GC. Long-term protection is achieved using the GCRoot<T> smart pointer template, which behaves in exactly the same way as GCStackRoot but is not subject to any destruction-order constraints. The disadvantage with GCRoot is that construction and destruction take longer than for GCStackRoot. Increasingly, R objects which need to be in existence throughout an R session are implemented within CXXR using GCRoot static members of the relevant class: an example is R_NaString (the 'not available' string).

4.1 Class GCNode's internal lists

Internally, class GCNode arranges all GCNode objects on a number of lists, implemented as static members of the GCNode class. Principal among these are the **live list** s_live and the **moribund list** s_moribund. Newly created nodes are placed on the live list. If a GCNode's reference count falls to zero, it is transferred to the moribund list.

Such moribund nodes cannot immediately be deleted, because it is quite common for the reference count to go up again after falling to zero: in fact this happens in about one case in four. This arises because CR code relies widely on the fact that no garbage collection will take place unless memory allocation is attempted. For example, it is very common for functions within the interpreter to return an unprotected SEXP: a pointer to an RObject that *was* protected within the body of the function now returning, and *will* again be protected within the calling code, but which for the moment—in CXXR—has a reference count of zero.

The destructor of class `GCNode` is private, and likewise the destructors of all classes inheriting from `GCNode` are either private or protected. This means that the only possible way of creating such an object is dynamic allocation using the `new` operation (cf. Meyers 1996, Item 27). However, `GCNode` defines its own `operator new`, so that when an object of a class inheriting from `GCNode` is created, the compiler automatically routes the request for memory to `GCNode`'s own `operator new` function, rather than to the global `operator new`.

What `GCNode::operator new` does first is to scan through the moribund list. Any node whose reference count is still zero gets deleted, and other nodes get returned to the live list. Then, if the number of bytes allocated *via* `MemoryBank` exceeds a threshold set by class `GCManager`, `GCNode::operator new` calls upon class `GCManager` to initiate a mark-sweep garbage collection. Finally, `operator new` allocates the requested number of bytes by calling `MemoryBank::allocate()`.

4.2 Infant Immunity

In one aspect, the way in which C++ works is rather at odds with automatic garbage collection. Suppose that `Foo` is a class that inherits from `GCNode`. From what has already been said, one might suppose that a statement such as the following would be sufficient to create a `Foo` object pointed to by `f`:

```
GCStackRoot<Foo> f = new Foo(···);
```

where `Foo(···)` is a constructor expression.

The compiler will convert this into the following sequence of operations:

1. `GCNode::operator new` is called to allocate enough memory for a `Foo` object;
2. The relevant constructor of `Foo` is called to initialise the object. However, before this constructor can start its own work, the C++ language requires it first (implicitly or explicitly) to call a constructor of `GCNode` to initialise the `GCNode` part of the object.
3. After the `GCNode` constructor returns, the `Foo` constructor initialises the `Foo` part of the object.
4. The code then returns a pointer to the newly-created `Foo`.
5. This pointer is assigned to the smart pointer `f`, and this assignment increments the reference count of the newly-created `Foo` and takes other steps to protect it from garbage collection.

But consider what happens if `Foo`'s constructor needs to create a subobject which is also a `GCNode`. (As we shall see in Sec. 5, this happens for example in the copy constructor of class `PairList`, which automatically copies subsequent elements of the list.) Suppose further that allocating memory for the subobject gives rise to a mark-sweep garbage collection. What happens is that the object under construction—the `Foo` object—gets garbage-collected. The subobject is allocated at Step 3, in `Foo`'s constructor. Step 2 is already complete, which means the garbage collector is aware of the new node: it is already on the live list. But not until Step 4 do we get a pointer to the node under construction, so it cannot be protected from garbage collection until Step 5.

The solution that CXXR adopts is for each GCNode to be immune from garbage collection while it is construction. This means that code must specifically signal to class GCNode that the construction of a node is complete, and this is done using the expose() method, which acts like an identity function, but with the side effect of terminating infant immunity. The object construction idiom thus becomes:

```
GCStackRoot<Foo> f = GCNode::expose(new Foo(···));
```

If CXXR is built with the preprocessor variable CHECK_EXPOSURE defined, checks are inserted to verify that each node has been exposed to garbage collection before any substantive use is made of it.

During CXXR development, various approaches have been explored for implementing infant immunity on a node by node basis, but all these approaches proved to have substantial drawbacks. The approach now used is much simpler: GCNode::operator new will not initiate a mark-sweep collection if *any* GCNode is still under construction. The 'under construction' status is signalled by a flag within each GCNode object, which is set by the constructor and cleared by the expose() function. Class GCNode also keeps a tally of the number of nodes currently under construction.

5 Object Duplication

Each class inheriting from RObject defines whether and how objects of that class can be duplicated. Class RObject defines a virtual method clone(), but by default that returns a null pointer, signifying that the object is not clonable. Many classes in the RObject hierarchy define a C++ copy constructor, which encapsulates the behaviour of CR's duplicate() function with regard to that class, and most such classes also reimplement clone() so that foo->clone() returns a pointer to a copy of foo made with the copy constructor.

CXXR here exploits yet another smart pointer template called RObject::Handle<T>. This inherits from GCEdge<T>, and like GCEdge<T> each handle encapsulates a pointer—possibly null—to a particular type T of RObject. When a (non-null) handle is copied, it tries to copy the object it points to by invoking clone(). If that succeeds, i.e. returns a non-null pointer, then the copied handle points to the copied object, but if the clone fails, the copied handle points to the original object. This greatly simplifies the implementation of copy constructors, and in some cases the default copy constructor supplied by the compiler does exactly what is needed.

To illustrate this, Fig. 2 shows how a PairList (LISTSXP) object is laid out in CXXR.[3] Notice that the attributes field m_attrib is declared as a handle, so that when a PairList object is cloned, the copy constructor will automatically try to duplicate the attributes too. Similarly for the 'CAR'. But the tag is simply declared as a GCEdge, so the cloned PairList will point to the original tag: there is no attempt to duplicate it.

[3] The last few fields in the figure, from m_missing onwards, have been 'pushed down' the inheritance hierarchy from the gp field of SEXPREC; some or all of them may be removed in future refactorisation.

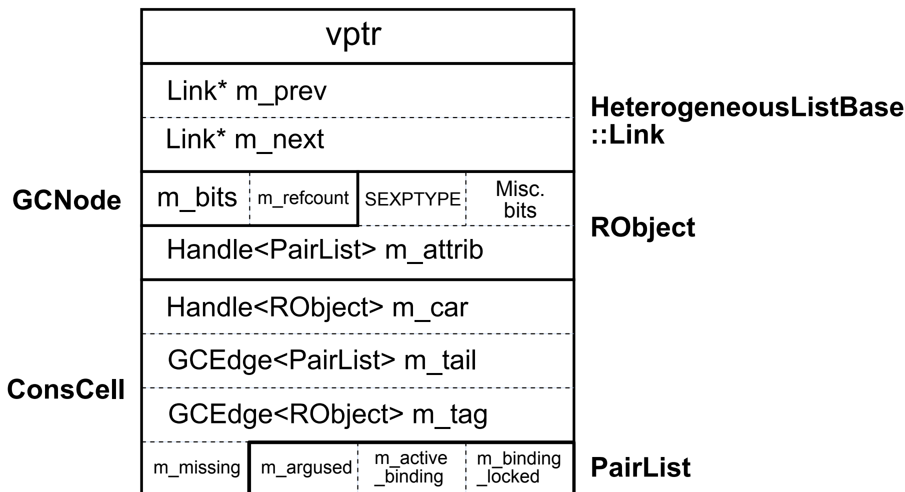

Fig. 2 Layout of a `PairList` object (schematic, for 32-bit architecture). `vptr` is a compiler-generated pointer to class `PairList`'s table of virtual functions.

The tail (CDR) is a special case: the tail *is* duplicated on copy, but not using the `Handle` mechanism, which would entail recursing down the list. Instead there is special code inside the `PairList` copy constructor that duplicates the tail using iteration.

The rationale behind the name of `HandleVector<T, ST>` will probably now be apparent: the class is implemented using a C++ standard vector with elements of type `RObject::Handle<T>`. Consequently, cloning a `HandleVector` will attempt to clone the objects its elements point to.

6 Environments

In CXXR, all environments work in the same way, and this includes the base environment and base namespace, which CR handles in a completely different way from other environments. Each environment contains a pointer to its enclosing environment, and a pointer to an object of class `Frame`. The only unusual feature of the base environment and the base namespace is that they both point to the same `Frame`, though they have different enclosing environments.

`Frame` itself is an abstract class—it defines an interface without defining an implementation—which defines a mapping from `Symbol` objects to `Frame::Binding` objects. These `Frame::Bindings` in turn refer to arbitrary `RObjects`. This structure means that it is entirely straightforward to encapsulate lazy loading strategies into classes inheriting from `Frame`. Similarly, functionality similar to R's `RObjectTables` package (Temple Lang 2001) is readily obtainable: for example it is possible to de-

fine a type of Frame that looks symbols up in an external database. (However, CXXR does not support the RObjectTables package directly.)

Class Frame inherits from GCNode, but not from RObject (cf. Fig. 1). The class includes the facility to set up callback functions to monitor when a binding within the frame is read or modified. Such monitor functions are used in object provenance tracking.

There is currently no analogue to CR's 'global cache', which provides a short cut for locating symbol definitions on the search path. This means that when CXXR looks up a symbol that is defined in the base environment, it has every time to work its way down the search list before it finds it. Although the use of package namespaces sidesteps the problem for many lookups, which will terminate at the base namespace rather than the base environment, the absence of a global cache nevertheless appears to have a serious impact on performance, and it is likely that some such mechanism will be reintroduced into CXXR in due course.

7 Performance

Benchmark	CR (secs)	CXXR GMS (secs)	CXXR RC/MS (secs)
bench.R (Jan de Leeuw)	111	113	112
kaltime10.R	95	144	113
stats-Ex.R	30	61	69

The performance of CXXR is currently disappointing. The table above shows timing tests on various benchmarks carried out on a 2.8 GHz Pentium 4 with 1 MB L2 cache, comparing R-2.8.1 with CXXR revision 599, using comparable optimisation options. The column headed 'CXXR GMS' is the implementation using generational mark-sweep for garbage collection, and corresponds closely to release 0.20-2.8.1 of CXXR. The right-hand column, headed 'CXXR RC/MS' uses reference counting backed up by simple mark-sweep, and corresponds fairly closely to release 0.21-2.8.1.

If we consider Jan de Leeuw's benchmark bench.R (Urbanek 2008), then CR and CXXR are about the same speed. But since most of the work in bench.R is done in C functions that are essentially unchanged in CXXR, that is unsurprising.

At the other extreme is stats-Ex.R, which takes over twice as long in CXXR as in CR, and is even slower in the reference-counted version of CXXR. The reasons for this are not yet fully understood.

Closer study has been made of the performance of CXXR with a third program, kaltime10.R, as follows:

```
kaltime <- function(d, n){
    phi <- matrix(0.9/d, nrow=d, ncol=d)
    p <- matrix(0, nrow=d, ncol=d)
    q <- diag(d)
    for (i in 1:n) p <- phi %*% p %*% phi + q
    p
}

kaltime(10, 5000000)
```

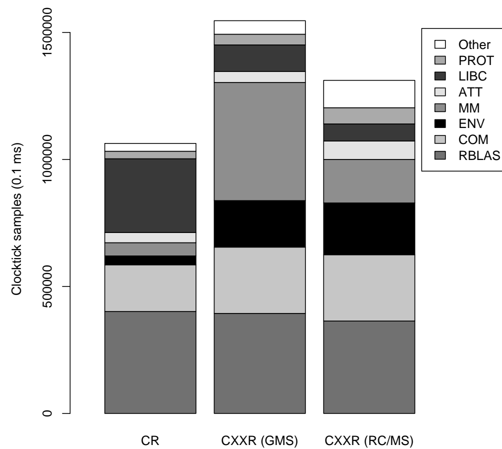

Fig. 3 Analysis of time consumption running `kaltime10.R`. Clocktick samples are allocated to the following categories. RBLAS: `libRblas.so` basic linear algebra system. COM: common, i.e. functions little changed between CR and CXXR. ENV: environments, inc. symbol look-up. MM: memory allocation, garbage collection, C++ destructors. ATT: R attributes. LIBC: `libc-2.9.so`. PROT: initiating and ending protection from garbage collection. Other: anything not clearly attributable to the preceding categories.

Basically, the code carries out 5 000 000 iterations of a Kalman filter time update (naïvely implemented, and with a matrix transposition omitted), operating on 10×10 matrices. With `kaltime10.R`, the reference-counted version of CXXR is about 20% slower than CR, but the generational mark-sweep version is about 50% slower.

Fig. 3 presents an analysis of which parts of the interpreter consume most time when running `kaltime10.R`, by allocating clocktick samples to a number of categories. (The sample counts are based on flat profile data obtained using Intel's VTune tool; the use of this tool means that total timings are not quite in the same proportions as in the table earlier.) For the most part this is self-explanatory, but three comments are in order:

– Although the functions in category COM are *textually* little changed between CR and CXXR, they include many invocations of functions which are inlined in CXXR but not in CR. This appears largely to account for the timing differences in the category.
– In the case of CR much of the time in category LIBC is accounted for by calls to `isnan()`. In C++, `isnan()` is inlined, and the time will show up instead in the COM category.
– As regards the MM category, the difference between the two versions of CXXR seems almost entirely to be due to better cache utilisation in the reference-counted version. Why the reference-counted version is so much slower than CR is not fully understood.

One aspect of performance where the reference-counted version of CXXR shows particular promise is L2 cache misses, as the following table shows. The table was obtained using `cachegrind` (Valgrind Developers 2009) running `kaltime10.R` but with just 100 000 iterations.

	CR	CXXR GMS	CXXR RC/MS
D1 misses	12 864 395 (0.4%)	36 063 852 (1.1%)	21 130 898 (0.6%)
L2d misses	6 655 104 (0.2%)	14 193 172 (0.4%)	373 466 (0.0%)
I1 misses	25 843 389 (0.4%)	31 854 095 (0.5%)	73 659 529 (1.2%)
L2i misses	63 323 (0.0%)	97 394 (0.0%)	55 400 (0.0%)
Brch mispred.	37 427 870 (5.4%)	41 119 592 (5.8%)	47 800 867 (6.3%)

8 Future Work

Near-term priorities for future work on CXXR are as follows:

- Introduce a virtual function `RObject::evaluate()`, and use it to move evaluation code out from CR's `eval()` function into the appropriate place in the `RObject` class hierarchy. In the process reduce internal use by the interpreter of `PairList` objects in favour of lighter-weight data structures.
- Similarly factor out serialisation/deserialisation code into the class hierarchy. Also possibly switch to a serialisation format (XML perhaps) in which the end of the serialized form of any object can be determined on syntactical grounds alone. This will enable a deserialiser to make some sense of a saved session even if it does not understand every type of object in the session (e.g. because of missing packages). Extension to the serialisation facilities is also necessary to support cross-session provenance tracking.

With evaluation and serialisation thus moved out into the class hierarchy, it is to be hoped that more implementation details can be concealed within these classes, so improving encapsulation.

In the rather longer term, it is expected that the global cache will be reintroduced in some form, and that R contexts (`RCNTXT`) and error handling will be refactored in a manner that exploits C++ exception handling idioms. It would also be desirable for attributes and slots to be implemented by distinct mechanisms at the C++ level, and for the C++ classes implementing R types to enforce appropriate constraints on the setting of attributes and slots.

References

Richard A. Becker and John M. Chambers (1988) Auditing of Data Analyses, SIAM J. Sci. and Stat. Comput. 9:747–60.

Peter Becker (2006) The C++ Standard Library Extensions: a Tutorial and Reference, Addison-Wesley, Reading, MA.

John DeTreville (1990) Experience with Concurrent Garbage Collectors for Modula-2+, Tech. Rep. 64, DEC Systems Research Center. Available from `http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-64.pdf`, accessed on 25 September 2009.

Dimitri van Heesch (1997) Doxygen. `http://www.doxygen.org`. Accessed on 2 September 2009.

Scott Meyers (1996), More Effective C++, Addison-Wesley, Reading, MA.

Scott Meyers (1998), Effective C++, Second Edition, Addison-Wesley, Reading, MA.

R Development Core Team (2009a) R Internals. ISBN 3-900051-14-3. Available at `http://www.r-project.org`.

R Development Core Team (2009b) Writing R Extensions. ISBN 3-900051-11-9. Available at `http://www.r-project.org`.

Duncan Temple Lang (2001) User-Defined Tables in the R Search Path, `http://www.omegahat.org/RObjectTables/RObjectTables.pdf`. Accessed on 7 September 2009.

Duncan Temple Lang (2009) A Modest Proposal: an Approach to Making the Internal R System Extensible. Comput. Stat. 24:271–81.

Simon Urbanek (2008) R Benchmarks, `http://r.research.att.com/benchmarks`, Accessed on 7 September 2009.

Valgrind Developers (2009) Cachegrind: a Cache and Branch-Prediction Profiler, `http://valgrind.org/docs/manual/cg-manual.html`, Accessed on 7 September 2009.