CXXR: An Ideas Hatchery for Future R Development

Andrew R. Runnalls and Chris A. Silles*

Abstract

The continued growth of CRAN is testament to the increasing number of developers engaged in R development. But far fewer researchers have experimented with the R interpreter itself. The code of the interpreter, written for the most part in C, is structured in a way that will be foreign to students brought up with object-oriented programming, and the available documentation, though giving a general understanding of how the interpreter works, does not really enable a newcomer to start modifying the code with any confidence.

The CXXR project is progressively refactoring the interpreter into C++, whilst all the time preserving existing functionality. By restructuring the code into tightly encapsulated and carefully documented classes, CXXR aims to open up the interpreter to more ready experimentation by statistical computing researchers.

This paper focusses on two example tasks: (*a*) providing, as a package, a new type of data vector, and (*b*) adding the capability to track the provenance of R objects. The paper shows how CXXR greatly facilitates these tasks by internal changes to the structure of the interpreter, and by offering a higher-level interface for packages to exploit.

1. CXXR

The object of the CXXR project is gradually to reengineer the fundamental parts of the R interpreter from C into C++, with the intention that:

- Full functionality of the standard distribution of R (including the recommended packages) is preserved;
- The behaviour of R code is unaffected (unless it probes into the interpreter internals);
- There is no change to the existing interfaces for calling out from R to other languages (`.C`, `.Fortran`, `.Call` and `.External`).
- Likewise there is no change to the main APIs (`R.h` and `S.h`) for calling into R. However, a broader API is made available to external C++ code.

Work on CXXR started in May 2007, at that time shadowing R-2.5.1; the current release (as of August 2011) shadows R-2.12.1, and an upgrade to 2.13.1 is in progress.

This paper will refer to the standard R interpreter as **CR**, in contradistinction to CXXR.

The original motivation for embarking on the CXXR project was to introduce **provenance-tracking** facilities into R, and progress on this front will be described later in the paper. But CXXR has a broader mission, which is to make the R interpreter more accessible to developers and researchers, by various means. For example, the C++ classes and other code structures within CXXR are carefully documented. Secondly, CXXR tightens up the encapsulation boundaries within the interpreter, so that a developer can be more confident that modifying code in one place will not unexpectedly break things elsewhere. This is part-and-parcel of moving towards an object-oriented software structure, something that graduate students are increasing familiar with, and which the R *language* itself has moved towards. Finally CXXR aims to express the internal algorithms of the interpreter at a higher level of abstraction, and make them available to external code through CXXR's own API, and we shall see some examples of this shortly.

*School of Computing, University of Kent, Canterbury CT2 7NF, UK

1.1 Layers

CXXR code falls into three categories, which can be considered roughly to form three concentric layers. At the centre is the **CXXR core**, which is written as far as possible in idiomatic C++, making free use of the C++ standard library, and some use of the Boost libraries of peer-reviewed, portable C++ [Boost (2011)]. Everything in the core layer is placed in the C++ namespace `CXXR`. The following is a summary of the functionality currently within the core:

- Memory allocation and garbage collection.
- `SEXPREC` union replaced by an extensible class hierarchy rooted at class `RObject` (described below).
- Environments (i.e. variable to object mappings), with hooks to support provenance tracking.
- Expression evaluation. (S3 method dispatch partially refactored; S4 dispatch not yet refactored.)
- Contexts and indirect flows of control (with some loose ends).
- Unary and binary function generics. `[`-subscripting.
- Object duplication is now handled by C++ copy constructors. (In an experimental development branch, object duplication is managed automatically, removing the need for `NAMED()` and `SET_NAMED()`.)

On the outside is the **packages and modules layer**, and the aim here is that existing R packages should work with CXXR with minimal alteration—usually none at all. A paper [Runnalls (2010)] at the useR!2010 conference explored the extent to which this had been achieved by testing CXXR with 50 key packages (other than the base and 'recommended' packages, which are routinely maintained as part of CXXR development) from the central R archive CRAN, namely the packages on which most other packages in the archive depend. The upshot was that, apart from fixing some latent bugs in the packages (and bugs in CXXR itself), only three lines of package code needed to be modified for all the tests included in these 50 packages to pass. All these changes were in the C code which some packages include; in no case did a package's R code need to be changed.

In between the core and the packages layer is the **transition layer**, which consists of C files from the CR interpreter adapted to work with the CXXR core. In most cases these C source files have been redesignated as C++, but the programming idioms are largely those of CR (which in addition to C idioms frequently exhibits those of Fortran and especially LISP).

2. The `RObject` Class Hierarchy

One of the major changes in CXXR is the way that objects visible in R are implemented. In CR, these objects are implemented using a C **union**, which can be thought of as a way of telling the C compiler that a particular memory address may hold any one of several specified datatypes: in this case no fewer than 23 distinct types, corresponding to the different types of R object.

This approach has several disadvantages. First of all, the compiler does not know which of the 23 types is occupying a particular address: this is determined only at runtime. Consequently much of the type checking in the interpreter must itself be done at runtime: the

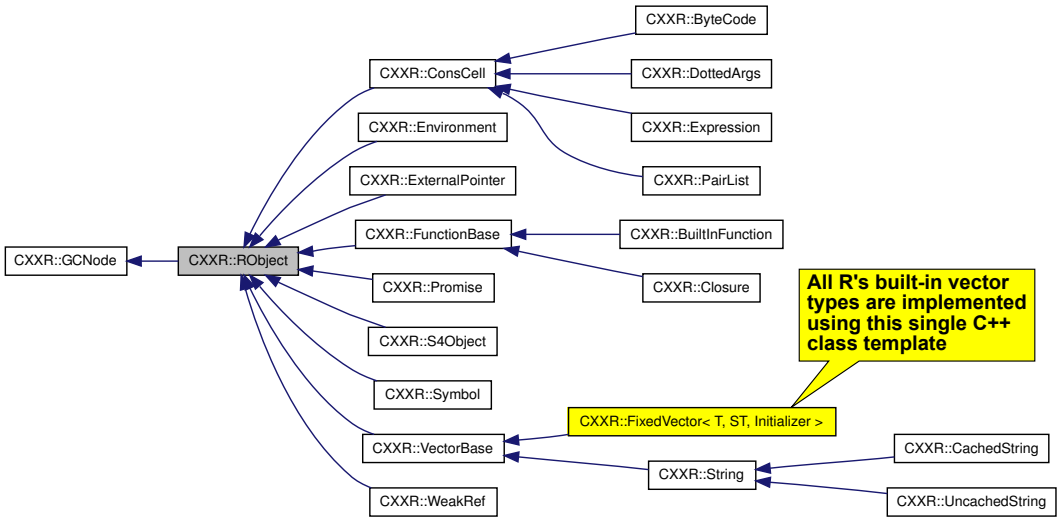

Figure 1: The `RObject` class hierarchy.

implementation simply doesn't leverage the compiler's own considerable capabilities for type checking. This type ambiguity can also make debugging at the C level difficult.

Another snag is that this approach in effect turns the set of R data types into a *closed list*: if you want to add a new type of R object, that means delving into the heart of the interpreter and making fundamental modifications.

CXXR uses quite a different approach: the different types of R object are implemented as a C++ class inheritance hierarchy, rooted at an abstract class unimaginatively named **RObject**, and shown in Fig. 1. Readers familiar with R will have no difficulty mapping the C++ class names in this figure onto the corresponding type of object in the R language. A particular point to note is that all of R's built-in vector types are implemented using a single C++ class template called `FixedVector`: this includes vectors of integers, vectors of reals and vectors of complex numbers as well as R lists.

Along with this change to a C++ class hierarchy, CXXR is carrying out a wholesale restructuring of the interpreter code. First of all, the project is endeavouring gradually to move all code relating to a particular data type into one place, and then to use C++'s public/protected/private mechanisms to conceal implementational details and to defend class invariants.

Most important of all, CXXR aims to make it easy for developers to extend the class hierarchy, and the paper will now give an example of this: of how a developer can add a new type of object to CXXR at the interpreter level.

2.1 `MyGMP`

R of course has the ability to handle vectors of integers, but the range of values that can be represented by these built-in integers is limited by the word-length of the computer you're using; in fact currently R's built-in integers are limited to 32 bits, even on a 64-bit architecture.

Suppose a developer wanted to write a package, which we shall call **MyGMP**, adding to R the capability of handling arbitrarily large integers. The developer is aware that there is a free GNU library, the **GNU MP library** [GNU MP (2011)] that provides '**bigint**'s, as they are called, for C and C++, and would like to build on that.

(Some readers will be aware that there already is such an R package, the **gmp** package [Lucas *et al.* (2010)], which offers bigints and much else besides. The rudimentary package

described in this paper does not in any way match the capabilities of gmp: its purpose is simply to illustrate how relatively easy it is to get such a package off the ground using CXXR.)

The GNU MP library defines a C++ class **mpz_class** to represent an arbitrarily large integer. But one of the attractive features of R for statistical analysis is that it can flag individual data points as being '**not available**', represented **NA** in R. As it stands `mpz_class` doesn't have this capability: it can represent positive integers, negative integers, zero, but not NA.

Fortunately, in CXXR we can put this right essentially in one line of C++ code, using the class template **NAAugment**, which does what the name suggests:

```
namespace MyGMP {
    typedef CXXR::NAAugment<mpz_class> BigInt;
}
```

This type definition gives us a new C++ class **BigInt** which can represent an arbitrarily large integer *or NA*, and this is all set up in such a way that the generic algorithms in CXXR can handle NAs with little or no attention from the package writer.

Each object of the new class `BigInt` represents a single bigint. But of course R works primarily with *data vectors*—or matrices or higher dimensional arrays. No problem: we can introduce vectors of `BigInt`s into CXXR essentially with one further line of code, which reinstantiates the same C++ class template `FixedVector` that is used for the built-in vector types:

```
namespace MyGMP {
    typedef CXXR::FixedVector<BigInt,
                              CXXSXP,
                              ApplyBigIntClass> BigIntVector;
}
```

With this one step, `BigIntVector`s have now joined the `RObject` class hierarchy. We can now assign `BigIntVector`s to R variables, and facilities such as garbage collection, and the dimensioning of matrices and arrays—all this is automatically in place.

2.2 Binary functions

Obviously the `MyGMP` package needs to have the capability to carry out arithmetic on `BigIntVector`s: multiplication for example. Let us first review how binary operations such as multiplication work for R vectors. Basically each element of the result is determined by multiplying together the corresponding elements of the two operands, so for example the first element of the result is the product of the first elements of the two operands.

But there are some complications. For example, if either of the operand elements is NA, then the corresponding result element must be set to NA. If the operands are of unequal length, the elements of the shorter operand are reused in rotation. But R gives a warning if the longer operand is not an exact multiple of the length of the shorter operand. It is also necessary to consider attributes: for example in R you can give names to individual elements of vectors, or to the rows and columns of matrices. Somehow the attributes of the result of a binary operation must be inferred from the attributes of the operands. There are some further complications if the operands are matrices or arrays.

CXXR defines a **generic algorithm** for implementing R binary functions, and makes it available to package C++ code *via* the CXXR API. To use the algorithm the programmer needs to specify four things:

1. The elementwise operation that is to be performed. In the case of multiplication this operation is given to us directly by the GNU MP library, which overloads the C++ multiplication operator "$*$" for operands of `mpz_class`.

2. The two operands (in the present case `BigIntVectors`).

3. The type of vector (or other vector-like container) to be produced as the result: in the present case again a `BigIntVector`;

4. Finally, it is necessary to specify how the attributes of the result are inferred from those of the operands. Fortunately, this is better standardised in R for binary operations than it is for unary operations, so usually a default value can be used for this parameter.

Using this algorithm, introducing multiplication of `BigIntVectors` to the `MyGMP` package requires just the following C++ code:

```
extern "C" {
    BigIntVector* MyGMP_multiply(const BigIntVector* vl,
                                 const BigIntVector* vr)
    {
        using namespace CXXR::VectorOps;
        return
            BinaryFunction<GeneralBinaryAttributeCopier,
                           std::multiplies<mpz_class> >()
            .apply<BigIntVector>(vl, vr);
    }
}
```

which is invoked via a simple R function in the package:

```
`*.BigInt` <- function(vl, vr) {
  .Call("MyGMP_multiply", as.bigint(vl), as.bigint(vr))
}
```

All in all, the developer does not need to do much programming at all in the `MyGMP` package before it becomes possible, for example, to compute large factorials in R. Here is an example session:

```
> f <- as.bigint(c(1:20, NA))
> for (i in 3:21) f[i] <- f[i]*f[i-1]
> f
 [1] "1"                   "2"                   "6"
 [4] "24"                  "120"                 "720"
 [7] "5040"                "40320"               "362880"
[10] "3628800"             "39916800"            "479001600"
[13] "6227020800"          "87178291200"         "1307674368000"
[16] "20922789888000"      "355687428096000"     "6402373705728000"
[19] "121645100408832000"  "2432902008176640000" NA
```

This example illustrates the fact that NAs propagate as expected without needing special consideration within the package code.

2.3 Subscripting

The factorial example above included the use of square brackets to perform subscripting on a `BigIntVector`, and this facility is something that needs to be provided explicitly by the `MyGMP` package.

R is rightly renowned for the power of its subscripting operations: subscript expressions can be used to access or replace elements of vectors, rows and/or columns of matrices, and slices of higher dimensional arrays. To give a few examples: a row of a matrix (for example) can be specified either by its position, or by its name (if it has one). Negated row numbers signify 'include all rows except these'. Odd numbered rows of a matrix `m` can be selected by specifying a logical expression as the index: `m[c(TRUE, FALSE), ]`, the elements of the 'logical' vector `c(TRUE, FALSE)` being recycled as necessary up to the total number of rows in `m`.

In fact the *R Language Definition* document devotes over four of its 51 pages to describing R's subscripting facilities, and even that glosses over some edge cases. In the CR interpreter there are some 2000 C language statements implementing these facilities. But in a sense this C code is 'locked up': it isn't made available to external code *via* a documented API, and it is written exclusively to handle R's built-in data types. Consequently, as it stands, we can't directly exploit this code to bring subscripting facilities to `BigIntVectors`.

CXXR takes a more open approach. It makes subscripting facilities available through its API using a class with the obvious name: **Subscripting**. This class works using C++ generic algorithms, which abstract away from the type of the elements of the vector (or matrix or higher-dimensional array). Consequently the algorithms are not restricted to R's built-in data types: `BigInt` elements work just fine.

But not only do the algorithms abstract away from the type of elements of the vector, they also abstract away from the particular data structure used to implement the vector. So this opens the way to using the algorithms with packed data: perhaps a developer implements a new vector object type in which 32 DNA bases are packed into a 64-bit word. Or a developer may introduce into the `RObject` class hierarchy a new vector implementation for large datasets in which most of the data are held on disk, after the fashion of the **ff** package. In either case the developer can use CXXR's subscripting algorithms to index into these new vector types.

As far as `BigIntVectors` are concerned, using CXXR's `Subscripting` class means that we can carry across the full power of R subscripting with just a few lines of package code. Here for example is the package C++ code needed to implement **subassignment**, i.e. the replacement operation that occurs when square brackets appear on the left-hand side of an assignment:

```
extern "C" {
    BigIntVector* MyGMP_bigintsubassign(const PairList* args)
    {
        args = args->tail();
        BigIntVector* lhs
            = SEXP_downcast<BigIntVector*>(args->car());
        args = args->tail();
        const BigIntVector* rhs
            = SEXP_downcast<const BigIntVector*>(args->car());
        args = args->tail();
        return Subscripting::subassign(lhs, args, rhs);
    }
}
```

which is invoked via the following R function in the package:

```
`[<-.BigInt` <- function(v, ..., value) {
  .External("MyGMP_bigintsubassign", v, as.bigint(value), ...)
}
```

3. Provenance Tracking

This paper will now briefly describe another line of development building on CXXR, namely work to provide provenance tracking facilities.

Have you ever returned to a data analysis after a period of months, and asked yourself questions such as the following?

- How exactly was a particular data object or model derived from the original data?
- What data points were discarded, and how were they identified as being suspect?
- One of the several datasets that went into the original analysis is now known to have been corrupt. Which results does this invalidate?

In other words you are interested in **interrogating the provenance** of data objects and models. Provenance-awareness is a topic of increasing importance in information science, and is the subject of biennial conferences: the International Provenance and Annotation Workshops (IPAW).

Interestingly one of the first provenance-aware applications was S, and [Becker and Chambers (1988)] is often cited as a pioneer paper. An S session would maintain an audit file, recording all the top-level commands, and identifying the data objects read and modified by the commands. This audit file could then be analysed using a separate tool, `S AUDIT`, described in [Becker, Chambers and Wilks (1988)].

We have been exploring the possibility of introducing this capability into CXXR, but with the difference that provenance information can be interrogated directly from within an R session. A preliminary report on this was given in [Silles and Runnalls (2010)]. To illustrate this, consider the following simple R command session:

```
> one <- 1
> two <- c("deux", "zwei")
> two <- one + one
> three <- 3
> square <- function(x) x*x
> four <- square(two)
> five <- four + 1
> nine <- square(three)
> rm(two, five)
```

Work to date enables us to ask about the provenance of a particular object (or, to speak more strictly, the provenance of a **binding** of an R variable to an object). For example:

```
> provenance(nine)
$command
nine <- square(three)

$symbol
nine

$timestamp
[1] "01/07/11 15:50:43.497459"

$parents
[1] "square" "three"

$children
NULL
```

from which we can see that the current binding of the symbol **nine** was created at 15:50 on July 1st by the top-level command `nine <- square(three)`.

Somewhat more interestingly, we can ask about the entire pedigree of a binding, i.e. the entire sequence of top-level commands that are relevant to a current symbol binding:

```
> pedigree(four)
one <- 1
two <- one + one
square <- function(x) x * x
four <- square(two)
```

Notice how the provenance of the user-defined function **square** is included here, as well as the provenance of the data objects **one**, **two** and **four** itself. This reflects the fact that in R, functions are fully-fledged objects.

The pedigree command can also be applied to a set of object names, so for example to find the joint pedigree of all the objects still existing in the session, we can use the command:

```
> pedigree(ls())
one <- 1
two <- one + one
three <- 3
square <- function(x) x * x
four <- square(two)
nine <- square(three)
```

Notice how the symbol **five** makes no appearance in this pedigree, because the object named **five** no longer exists, and its former value had no bearing on the value of any current bindings. However, the symbol **two** does make an appearance: even though no object named **two** still exists, one of the former values of the symbol **two** played a part in computing the value of the current object named **four**.

4. Cross-session Provenance Tracking and Serialization

In the examples above we saw how provenance data was maintained during a single R session. That is useful enough, but the real value comes when we resume a data analysis after a lapse of time. This means that it is essential that provenance information is saved at the end of a session, alongside the data objects it relates to.

This has led us to review the approach used for **serialization** and **deserialization**: the process by which a set of R objects is rendered into a form suitable for saving in a file, and subsequently restored. The primary objective is to delegate to each C++ class in CXXR responsibility for serializing and deserializing objects of that class. (This is in contrast to CR, which concentrates all serialization functionality into a few huge functions.) This includes the C++ classes used to implement provenance-tracking, so provenance information will automatically be serialized alongside the data it relates to.

Reorganising serialization in this way will have a further benefit. We saw earlier how CXXR enables developers to introduce new sorts of object such as `BigIntVector` into the `RObject` class hierarchy. One serious omission at the moment is that there is no provision for `BigIntVector` objects to be carried across from one R session to the next. The new serialization framework under development will rectify this: using this frame-

work, the **BigIntVector** class can itself define how **BigIntVector** objects are serialized and deserialized.[1]

5. Conclusion

CXXR aims to open up the R interpreter to developers, providing a workbench on which they can develop their own ideas regarding extensions, modifications and adaptations to R. In other words, CXXR is an **ideas hatchery**. As this paper has illustrated, one important way in which CXXR achieves this is by implementing objects visible in R using a C++ class inheritance hierarchy which developers can extend. This is complemented by a drive to rewrite key algorithms within the R interpreter at a higher level of abstraction, and to make them available *via* the CXXR API.

As one possible extension of R, the paper has briefly described work on introducing provenance-tracking facilities to CXXR.

Much work remains to be done in the development of CXXR itself. A conspicuous gap at present is a Windows port of CXXR: until now it has been built and tested only on Linux and (to a lesser extent) on MacOS X. Volunteers to assist in this and other aspects of CXXR development are welcome.

References

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

[Becker, Chambers and Wilks (1988)] Richard A. Becker, John M. Chambers and Allan R. Wilks *The New S Language*, Wadsworth and Brooks, Pacific Grove, CA, ISBN 0-534-09192-X.

[Boost (2011)] "Boost C++ Libraries", available at `http://www.boost.org`, accessed on 30 August 2011.

[GNU MP (2011)] "The GNU Multiple Precision Arithmetic Library", available at `http://gmplib.org`, accessed on 30 August 2011.

[Lucas *et al.* (2010)] Antoine Lucas, Immanuel Scholz, Rainer Boehme and Sylvain Jesson "RBigIntegers", available from `http://mulcyber.toulouse.inra.fr/projects/gmp` (accessed on 30 August 2011) and as package `gmp` from CRAN, `http://cran.r-project.org`.

[Runnalls (2010)] Andrew R. Runnalls "CXXR and Add-on Packages", available at `http://user2010.org/Slides/Runnalls.pdf`, accessed on 30 August 2011.

[Silles and Runnalls (2010)] Chris A. Silles and Andrew R. Runnalls "Provenance-Awareness in R" in *Provenance and Annotation of Data and Processes* eds. D. McGuinness, J. Michaelis and L. Moreau, Lecture Notes in Computer Science, Springer, Berlin/Heidelberg, ISBN 978-3-642-17818-4.

[1] However, a drawback with the approach currently under development (which builds on the Boost serialization library) is that it would require the `MyGMP` package to be loaded before the previous session could be deserialized.