Article type: Advanced Review

CXXR: A Workbench for Future R Development

Andrew Runnalls

School of Computing, University of Kent

Keywords

R, CXXR, provenance, serialization

Abstract

This paper describes CXXR, a project to refactor the R interpreter from C into C++, with a view to making the internals of the interpreter more readily accessible to researchers and developers.

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.

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 2012) shadows R-2.14.1, and an upgrade to 2.15.1 is in progress.[1]

This paper will refer to the standard R interpreter as **CR**, in contradistinction to CXXR. Bare **R** will be used to refer to the language and user interface that both these interpreters implement.

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.

[1]The CXXR initiative is not part of the official R project; however, the author wishes to acknowledge the help and encouragement of many members of the R Core Team.

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 [2012]). 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.

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. Runnalls [2010] 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 almost all 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).

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 fact this union (named `SEXPREC`) comprises no fewer than 23 distinct types, corresponding to the different types of R object. All of these types have in common an integer-valued field known as the `SEXPTYPE` which is used at runtime to determine which type of object it is; the various values of the `SEXPTYPE` are given mnemonic names such as `INTSXP` to identify an integer vector, or `BUILTINSXP` to identify an R function implemented directly within the interpreter.

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, by examining the `SEXPTYPE` field: the 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 wanted to add a new type of R object, that would mean delving into the heart of the interpreter and making fundamental modifications, something which package developers have rightly been reluctant to do. The adverse effect of this can be gauged by studying the code of the powerful `Matrix` package (Bates and Maechler [2012]): the code is constantly having to convert matrices from a format representable using CR's limited range of datatypes into the data structures required by third-party libraries such as LAPACK, and then—having applied a function from a third-party library—to convert the result back into a CR-compatible format. As we shall now see, CXXR allows this to be short-circuited, by allowing any desired C/C++ data structure to be incorporated directly into an R object.

In CXXR, 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.[2] Readers familiar with R will have no difficulty mapping most[3] of 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, vectors of complex numbers, vectors of strings (R's `character` type), 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. Secondly, CXXR uses C++ templates to express key algorithms at a higher level of abstraction.

[2]Although the C++ language contains a number of built-in mechanisms for determining which type an object belongs to within a class inheritance hierarchy, CXXR nevertheless also retains the `SEXPTYPE` field of CR. This is for backwards compatibility with existing code.

[3]With the exception of the `BailOut` classes, which are not visible to R users. These are used in CXXR to implement indirect flows of control in R (e.g. `return`, `break`) without incurring the overhead of throwing C++ exceptions.

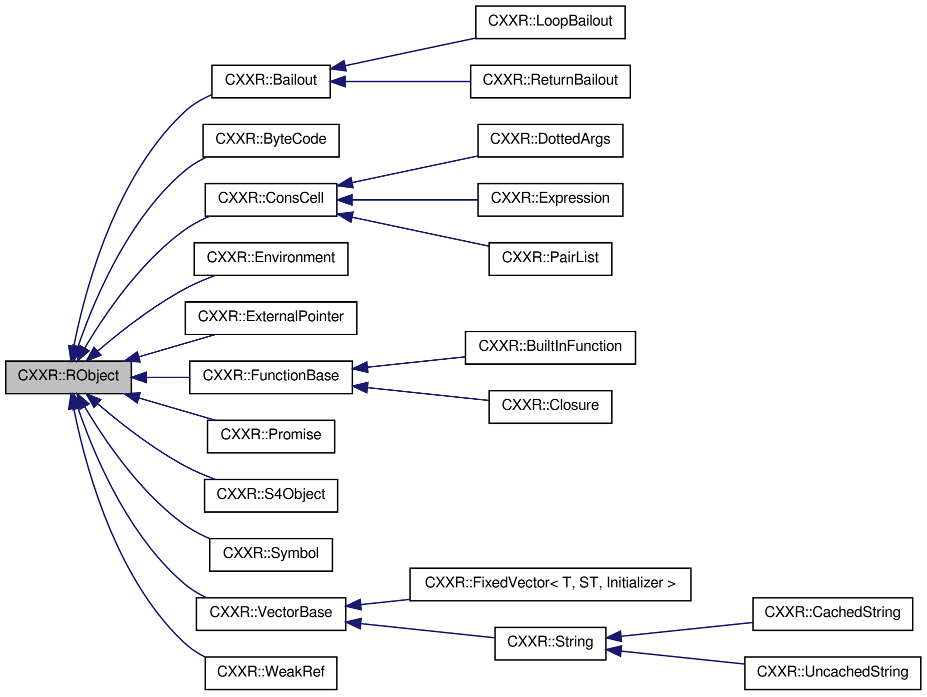

Figure 1: The `RObject` class hierarchy. (Adapted from a diagram produced by doxygen.)

Most important of all, CXXR aims to make it easy for developers to extend the class hierarchy.

Example: `MyGMP`

We will now give an extended example illustrating the ease with which CXXR enables new sorts of objects to be introduced into R at the C++ code level, by extending the `RObject` class hierarchy.

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. This developer is aware that there is a free GNU library, the **GNU MP library** (GMP [2012]) 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. [2004]), 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 `BigInts` 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;
}
```

The `FixedVector` class template takes three template parameters. The first is the type of element the vector is to contain, here `BigInt`. The second is the `SEXPTYPE` to be ascribed to objects of this type. `CXXSXP` is a special `SEXPTYPE` value used in CXXR to flag up any type of `RObject` that does not correspond to a CR object type: in the event that code inherited from CR encounters this `SEXPTYPE` value, it will normally raise an error. The third template parameter must be the name of an 'initializer' class, that is to say a class which defines a static `initialize()` function which is to be used to initialize newly-created objects. In the present case this is the class `ApplyBigIntClass`, defined as follows:

```
namespace MyGMP {
     struct ApplyBigIntClass
     {
          static void initialize(CXXR::RObject* obj)
          {
            obj->setAttribute(CXXR::ClassSymbol,
                              CXXR::asStringVector("BigInt"));
        }
    };
}
```

The effect of this is to ensure than newly created `BigIntVector` objects are given the R class attribute `BigInt`.

With the few lines of code listed above, `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.

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 normally 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`. This appears in the code below as the template parameter to the class template `BinaryFunction`.
2. The two operands (in the present case `BigIntVector`s).
3. The type of vector (or other vector-like container) to be produced as the result: in the present case again a `BigIntVector`. This appears in the code below as the template parameter to the templated function `BinaryFunction::apply()`.

4. Finally, it is necessary to specify how the attributes of the result are inferred from those of the operands, and this is determined by an optional second template parameter to the `BinaryFunction` class template. Fortunately, this is better standardised in R for binary operations than it is for unary operations, so usually—as here—this parameter can be left to take its default value.

Using this generic 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<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))
}
```

Unary functions are handled in a similar way.

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.

Subscripting

The factorial example above included the use of square brackets to perform subscripting on a `BigIntVector`, and this capability 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 CR's built-in data types. Consequently, as it stands, we can't directly exploit this code to bring subscripting facilities to `BigIntVector`s.

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 (Adler et al. [2012]). In either case the developer can use CXXR's subscripting algorithms to index into these new vector types.

As far as `BigIntVector`s 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:

```
using namespace CXXR;
using namespace MyGMP;

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), ...)
  }
```

(In the C++ code above, the templated function `SEXP_downcast()` is used to cast a pointer to some type in the `RObject` class hierarchy to a pointer to a type further down the hierarchy (i.e. further to the right in Fig. 1). According to the configuration options with which CXXR is built, this cast may be either checked (using a C++ `dynamic_cast`) or unchecked (`static_cast`).)

Serialization

For our class `MyGMP` to be of full value, it is necessary that objects of this class can be saved at the end of an R session alongside other types of R object, and restored at the start of the next session. In other words this class must participate in session serialization and deserialization.

The current CR code for serialization is not readily extensible. It is built around the limited number of datatypes comprised by `SEXPREC` union, and the serialization code is concentrated into a few huge functions. Moreover, the serialization format is inherently binary in nature, which makes debugging very difficult: it is difficult even to determine whether a bug is happening during serialization or during deserialization.

CXXR now has the capability to save and restore user sessions using an XML serialization format. This capability builds upon the facilities of the ingenious Boost C++ Serialization Library (Ramey [2009]). This library handles well the task of serializing a directed graph in which the nodes are heap-borne C++ objects of various types, and the edges are pointers. Each node in the graph will be serialized only once, even if it is pointed to by several pointers, and on deserialization all pointers will be set up correctly, even though the graph nodes will now be at different memory addresses. The library also takes care of the possibility that a pointer of type `T*`, say, may actually point to an object of some type inheriting from `T`.

All of this is achieved using C++ templates, and in a way which delegates to individual classes the details of how objects of that class will be serialized. So alongside the code defining the functionality of class `MyGMP`, some of which we reviewed above, we can place code defining how `BigIntVector` objects are serialized. In fact the following code suffices:

```
template <class Archive>
void load(Archive& ar, mpz_class& bi,
          const unsigned int version)
{
    std::string d;
    ar >> BOOST_SERIALIZATION_NVP(d);
    bi = d;
}

template <class Archive>
void save(Archive& ar, const mpz_class& bi,
          const unsigned int version)
{
    std::string d = bi.get_str();
    ar << BOOST_SERIALIZATION_NVP(d);
}

template <class Archive>
void serialize(Archive& ar, mpz_class& bi,
               const unsigned int version)
{
    boost::serialization::split_free(ar, bi, version);
}
```

To understand this code, recall that `BigIntVector` is `typedef`ed to an instantiation of the class template `FixedVector`. This class template already defines a (templated) member function `serialize()` which defines how `FixedVector` objects should be serialized using the facilities of `boost::serialization`. This function delegates the task of serializing the individual elements of the `FixedVector` according to the type of these elements. In the present case (`BigIntVector`) the type of the elements is of course `BigInt`, and as we saw earlier `BigInt` is `typedef`ed to an instantiation of the class template `NAAugment<T>`. This class template in its turn defines a member function `serialize()` defining how `NAAugment` objects should be serialized, and this in its turn delegates the task of serializing its payload, according to the type `T` of that payload.

In the present case the payload type is `mpz_class`, so all that remains is to specify how objects of `mpz_class` should be serialized. This is a class from a third-party library which does not define a `serialize()` member function. That is not a problem, however, since the `boost::serialization` library will also accept a free-standing `serialize()` function as defined in the code above.

This `serialize()` function calls the (inlined) library function `split_free()`, which simply forwards the call either to `save()`, if the template parameter `Archive` is an output archive type (i.e. for serialization), or to `load()` if `Archive` is an input archive type (deserialization). The `save()` function is passed a reference to the `mpz_class` object to be serialized, and represents this object as a decimal string within the output archive; `load()` performs the converse operation.

The code excerpt above calls the macro BOOST_SERIALIZATION_NVP, which is provided as part of the serialization library and handles 'name-value pairs'; this call causes the string d to be serialized within an XML element with name "d". The following snippet shows how a single element of a BigIntVector is represented within an archive:

```
<item>
  <m_value>
    <d>2432902008176640000</d>
  </m_value>
  <m_na>0</m_na>
</item>
```

The reader may be puzzled by the unused parameter 'version' of the functions above. This reflects another feature of the Boost serialization library that is very valuable within CXXR, namely that the serialization format is versioned *on a class by class* basis. That is to say, whenever an archive includes the serialized form of one or more objects of a particular class Cls, the archive records which version of serialization for this class was used: by default this is Version 0. If in the course of time, developers decide that a new serialization format is necessary for this class, perhaps because a new data member has been added to Cls, then they can change the output serialization code accordingly, and designate the resulting format as Version 1. They will also need to amend the code for input archives so that it can handle both archives in the new Version 1 and in the superseded Version 0 format. But note that these changes affect only the code associated with class Cls: there is no need for any changes to the higher-level serialization code, nor in the 'magic numbers' at the start of an archive file.

Once development of the new serialization approach is complete, it is intended that CXXR will switch to a serialization format based on compressed XML as its default serialization format; however CXXR will of course retain the ability to load sessions serialized in CR format, and to save CR-compatible data in CR format. Among the issues that remain to be addressed is the fact that, for example, the MyGMP package—and in particular its dynamically-loaded C++ library—must already have been loaded before any attempt is made to deserialize an archive containing the serialized forms of MyGMP objects. (Ideally it would also be possible to load the non-MyGMP-dependent content from an archive without loading the package; whilst this is feasible in principle, providing this capability is not seen as a priority.)

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 two series of conferences: IPAW (International Provenance and Annotation Workshop), and USENIX TaPP (Theory and Practice of Provenance).

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 et al. [1988]. A development branch of the CXXR code base has been exploring the possibility of introducing this capability for R. A preliminary report on this was given in Silles and Runnalls [2010].

Two key concepts of the Open Provenance Model (OPM, see Moreau et al. [2010]) are an **artifact**, which it defines as "Immutable piece of state, which may have a physical embodiment in a physical object, or a digital representation in a computer system", and **process**, which it defines as "Action or series of actions performed on or caused by artifacts, and resulting in new artifacts". (A third key concept of the OPM is **agent**, but this is not currently used in CXXR.) Artifacts and processes are together considered to define a directed hypergraph, with the artifacts as nodes, and the processes as hyperedges.[4] CXXR maps these concepts onto R concepts as follows:

Artifact: A **binding** of an R symbol (variable) to an R object.

Process: An R **top-level command**, i.e. an expression entered directly at the interpreter prompt.

The reader may be surprised that it is R bindings, rather than R objects themselves, that are taken as artifacts. But consider the R object 0 (represented in R as an integer vector of length 1 containing just zero). The provenance of 0 may be of interest to philosophers, but what is of more practical interest is the fact that a particular R variable (e.g. `num.outliers`) *has the value* 0.

Note also that R **top-level commands** are treated as black boxes from the point of view of provenance tracking: there is no attempt to track the evaluation of subexpressions or the function calls that are involved in evaluating a top-level expression.

Provenance-enabled CXXR instruments the reading and writing of bindings within the 'global environment' (R's main workspace), plus certain other environments, and maintains an audit trail defining the OPM hypergraph leading up to all extant bindings. This provenance information can then be interrogated within the interpreter itself.

As an illustration, consider the following simple R command session:

[4]These are hyperedges rather than plain edges because a process may take more than one artifact as an input, and/or produce more than one artifact as an output.

```
> 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)
```

With provenance-enabled CXXR, we can ask about the 'pedigree' of a binding, i.e. the entire sequence of top-level commands that are relevant to a current symbol binding:

```
> pedigree("four")$commands
[[1]]
one <- 1

[[2]]
two <- one + one

[[3]]
square <- function(x) x * x

[[4]]
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 provenance record also includes the times when these top-level commands were issued:

```
> pedigree("four")$timestamps
[1] "2012-07-27 08:03:09 GMT" "2012-07-27 08:03:42 GMT"
[3] "2012-07-27 08:04:03 GMT" "2012-07-27 08:04:09 GMT"
```

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())$commands
[[1]]
one <- 1

[[2]]
two <- one + one

[[3]]
three <- 3
```

```
[[4]]
square <- function(x) x * x

[[5]]
four <- square(two)

[[6]]
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**.

For a discussion of some outstanding issues with provenance-enabled CXXR, see Runnalls and Silles [2012].

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. 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.

References

Daniel Adler, Jens Oehlschlägel, et al. ff: memory-efficient storage of large data on disk and fast access functions. Available as package `ff` from CRAN, `http://cran.r-project.org`, 2012.

Douglas Bates and Martin Maechler. Matrix: Sparse and dense matrix classes and methods. Available as package `Matrix` from CRAN, `http://cran.r-project.org`, 2012. This package is included in the standard distributions of CR and CXXR.

Richard A. Becker and John M. Chambers. Auditing of data analyses. *SIAM J. Sci. and Stat. Comput.*, 9(4):747–60, July 1988.

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

Boost. Boost C++ libraries. Available at `http://www.boost.org`, 2012.

GMP. The GNU multiple precision arithmetic library. Available at `http://gmplib.org`, 2012.

Antoine Lucas et al. RBigIntegers. Available as package `gmp` from CRAN, `http://cran.r-project.org`, and from `http://mulcyber.toulouse.inra.fr/projects/gmp`, 2004.

Luc Moreau et al. The Open Provenance Model, core specification (v1.1). Available from `http://eprints.soton.ac.uk/271449/`, 2010.

Robert Ramey. Boost serialization. Available at `http://www.boost.org`, 2009.

Andrew R. Runnalls. CXXR and add-on packages. From useR! 2010 conference, available at `http://user2010.org/Slides/Runnalls.pdf`, 2010.

Andrew R. Runnalls and Chris A. Silles. Provenance tracking in R. From IPAW 2012, available in poster form at `http://www.cs.kent.ac.uk/projects/cxxr/pubs/IPAW2012_poster.pdf`, 2012.

Chris A. Silles and Andrew R. Runnalls. Provenance-awareness in R. In D. McGuinness, J. Michaelis, and L. Moreau, editors, *Provenance and Annotation of Data and Processes*, Lecture Notes in Computer Science. Springer, Berlin/Heidelberg, 2010. ISBN 978-3-642-17818-4.