CXXR: An Ideas Hatchery for Future R Development

Andrew Runnalls

School of Computing, University of Kent,

Canterbury, UK

Outline

The CXXR Project

The aim of the CXXR project[1] is progressively to reengineer the fundamental parts of the R interpreter from C into C++, with the intention that:

- Full functionality of the standard R distribution is preserved;
- The behaviour of R code is unaffected (unless it probes into the interpreter internals);
- No change to the existing interfaces for calling out from R to other languages (`.C`, `.Fortran`. `.Call` and `.External`).
- 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 started in May 2007, shadowing R-2.5.1; the current release shadows R-2.12.1, and an upgrade to 2.13.1 is in progress.

We'll refer to the standard R interpreter as **CR**.

[1] www.cs.kent.ac.uk/projects/cxxr

The CXXR Project

The aim of the CXXR project[1] is progressively to reengineer the fundamental parts of the R interpreter from C into C++, with the intention that:

- Full functionality of the standard R distribution is preserved;
- The behaviour of R code is unaffected (unless it probes into the interpreter internals);
- No change to the existing interfaces for calling out from R to other languages (`.C`, `.Fortran`. `.Call` and `.External`).
- 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 started in May 2007, shadowing R-2.5.1; the current release shadows R-2.12.1, and an upgrade to 2.13.1 is in progress.

We'll refer to the standard R interpreter as **CR**.

[1] www.cs.kent.ac.uk/projects/cxxr

Why Do This?

An initial motivation was to be able to introduce provenance-tracking facilities into CXXR: more on this later.

But CXXR has a broader mission: to make the R interpreter more accessible to developers and researchers.

This is being achieved by various means, including:

- Improving the internal documentation;
- Tightening up the internal encapsulation boundaries within the interpreter;
- Moving to an object-oriented structure, thus reflecting a programming approach with which students are increasingly familiar.
- Expressing internal algorithms at a higher level of abstraction, and making them available to external code through the CXXR API.

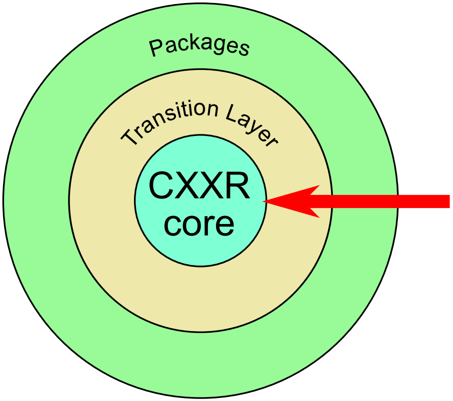
Why Do This?

An initial motivation was to be able to introduce provenance-tracking facilities into CXXR: more on this later.

But CXXR has a broader mission: to make the R interpreter more accessible to developers and researchers.

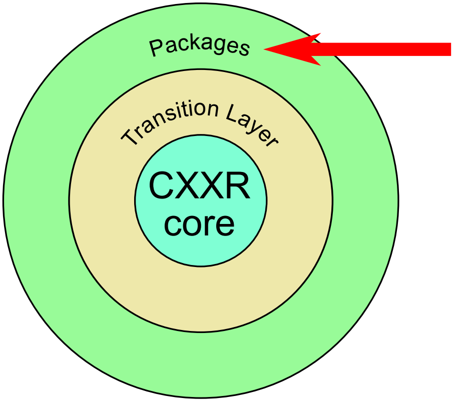
This is being achieved by various means, including:

- Improving the internal documentation;
- Tightening up the internal encapsulation boundaries within the interpreter;
- Moving to an object-oriented structure, thus reflecting a programming approach with which students are increasingly familiar.
- Expressing internal algorithms at a higher level of abstraction, and making them available to external code through the CXXR API.



- Written as far as possible in idiomatic C++, making free use of the C++ standard library, and some use of the peer-reviewed Boost libraries.
- Contained in the C++ namespace `CXXR`.
- Interfaces thoroughly documented using doxygen.
- As far as possible self-contained: avoids calls into the outer layers.

CXXR Layers
Packages



The aim is that existing R packages should work with CXXR with minimal alteration, usually none at all. This is achieved by leaving the primary interfaces unchanged.

CRAN Packages Tested

for useR! 2010 paper

Paper at useR! 2010 explored the compatibility of CXXR with 50 key packages from CRAN: those on which the largest number of other CRAN packages depend.

abind	gdata	MEMSS	RColorBrewer	scatterplot3d
akima	gee	mix	rgl	slam
ape	gtools	mlbench	rlecuyer	snow
biglm	kernlab	mlmRev	Rmpi	sp
bitops	leaps	multicore	robustbase	SparseM
car	lme4	mvtnorm	RODBC	timeDate
coda	logspline	numDeriv	rpvm	timeSeries
DBI	mapproj	nws	rsprng	tkrPlot
e1071	maps	quantreg	RSQLite	tripack
fBasics	mclust	randomForest	RUnit	xtable

Package versions were those current on 2010-05-05.

CRAN Packages Tested

for useR! 2010 paper

Paper at useR! 2010 explored the compatibility of CXXR with 50 key packages from CRAN: those on which the largest number of other CRAN packages depend.

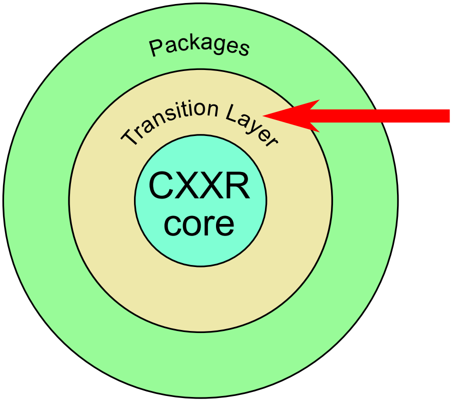
abind	gdata	MEMSS	RColorBrewer	scatterplot3d
akima	gee	mix	rgl	slam
ape	gtools	mlbench	rlecuyer	snow
biglm	kernlab	mlmRev	Rmpi	sp
bitops	leaps	multicore	robustbase	SparseM
car	lme4	mvtnorm	RODBC	timeDate
coda	logspline	numDeriv	rpvm	timeSeries
DBI	mapproj	nws	rsprng	tkrPlot
e1071	maps	quantreg	RSQLite	tripack
fBasics	mclust	randomForest	RUnit	xtable

Apart from fixing latent bugs, only three lines of package code needed to be modified for all the tests included in the packages to pass.

All these changes were in package C code, never R code.

Package versions were those current on 2010-05-05.

CXXR Layers
Transition layer



C files from CR adapted as necessary to work with the core.

With some exceptions, these C files have been redesignated as C++.

Outline

How R Objects are Implemented in CR

In CR, all R objects (as listed by the R command `objects()`) are implemented using a C ‘union’. This is a way of telling the C compiler that a particular memory address may hold any one of several distinct datatypes: in this case 23 types, corresponding to the different types of R object.

This has several disadvantages:

- The compiler doesn’t know which of the 23 types is occupying a particular union block. Consequently all type checking must be done at run-time; the possibilities of compile-time type checking are not exploited.
- Debugging at the C level is difficult.
- Introducing a new type of R object means modifying a data definition at the very heart of the interpreter.

How R Objects are Implemented in CR

In CR, all R objects (as listed by the R command `objects()`) are implemented using a C 'union'. This is a way of telling the C compiler that a particular memory address may hold any one of several distinct datatypes: in this case 23 types, corresponding to the different types of R object.

This has several disadvantages:

- The compiler doesn't know which of the 23 types is occupying a particular union block. Consequently all type checking must be done at run-time; the possibilities of compile-time type checking are not exploited.
- Debugging at the C level is difficult.
- Introducing a new type of R object means modifying a data definition at the very heart of the interpreter.

How R Objects are Implemented in CR

In CR, all R objects (as listed by the R command `objects()`) are implemented using a C ‘union’. This is a way of telling the C compiler that a particular memory address may hold any one of several distinct datatypes: in this case 23 types, corresponding to the different types of R object.

This has several disadvantages:

- The compiler doesn’t know which of the 23 types is occupying a particular union block. Consequently all type checking must be done at run-time; the possibilities of compile-time type checking are not exploited.
- Debugging at the C level is difficult.
- Introducing a new type of R object means modifying a data definition at the very heart of the interpreter.

How R Objects are Implemented in CR

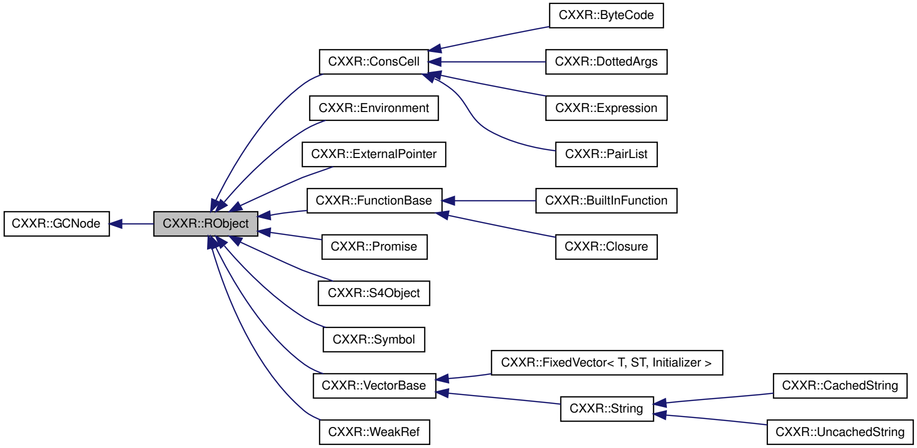
In CR, all R objects (as listed by the R command `objects()`) are implemented using a C ‘union’. This is a way of telling the C compiler that a particular memory address may hold any one of several distinct datatypes: in this case 23 types, corresponding to the different types of R object.

This has several disadvantages:

- The compiler doesn’t know which of the 23 types is occupying a particular union block. Consequently all type checking must be done at run-time; the possibilities of compile-time type checking are not exploited.
- Debugging at the C level is difficult.
- Introducing a new type of R object means modifying a data definition at the very heart of the interpreter.

The `RObject` Class Hierarchy

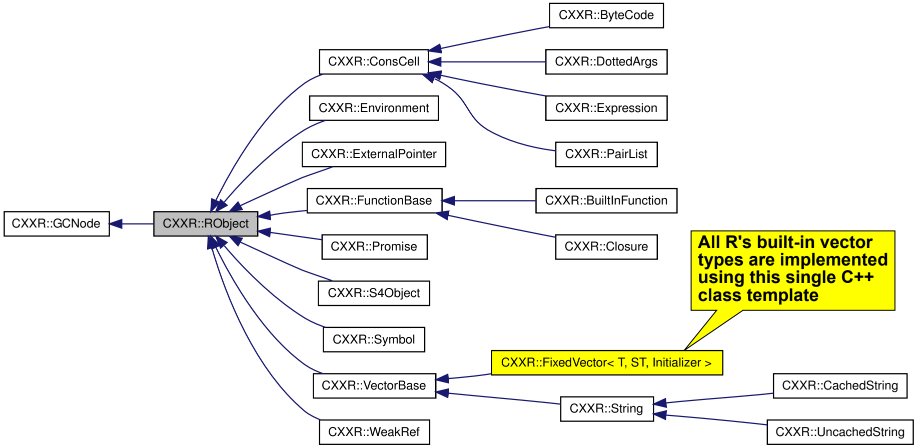
In CXXR, the various sorts of R objects are implemented using a C++ class inheritance hierarchy rooted at `RObject`:



(Based on a diagram produced by Doxygen.)

The RObject Class Hierarchy

In CXXR, the various sorts of R objects are implemented using a C++ class inheritance hierarchy rooted at `RObject`:



(Based on a diagram produced by Doxygen.)

The `RObject` Class Hierarchy
Objectives

- As far as possible, move all program code relating to a particular datatype into one place.
- Use C++'s public/protected/private mechanism to conceal implementational details.
- Allow developers readily to extend the class hierarchy.

The RObject Class Hierarchy
Objectives

- As far as possible, move all program code relating to a particular datatype into one place.
- Use C++'s public/protected/private mechanism to conceal implementational details.
- Allow developers readily to extend the class hierarchy.

The `RObject` Class Hierarchy
Objectives

- As far as possible, move all program code relating to a particular datatype into one place.
- Use C++'s public/protected/private mechanism to conceal implementational details.
- Allow developers readily to extend the class hierarchy.

Extending the Class Hierarchy: Example

Introducing arbitrarily large integers

Suppose we wanted to write a package adding to R the capability of handling arbitrarily large integers, drawing on the GNU Multiple Precision Library at gmplib.org.

In fact there already is such a package: the **GMP** package by Antoine Lucas *et al.* which does this and much more ...

... however the purpose of this example is to show how CXXR makes this task relatively straightforward. We'll call our nascent package **MyGMP**.

Extending the Class Hierarchy: Example

Introducing arbitrarily large integers

Suppose we wanted to write a package adding to R the capability of handling arbitrarily large integers, drawing on the GNU Multiple Precision Library at gmplib.org.

In fact there already is such a package: the **GMP** package by Antoine Lucas *et al.* which does this and much more . . .

. . . however the purpose of this example is to show how CXXR makes this task relatively straightforward. We'll call our nascent package **MyGMP**.

Extending the Class Hierarchy: Example

Adding NA to what the GNU library provides

The GNU MP library defines a C++ class `mpz_class` to represent an arbitrarily large integer.

But an attractive characteristic of R is its ability to flag individual data points as ‘not available’: `NA`. As it stands `mpz_class` does not have this capability.

Fortunately, in CXXR we can put this right essentially in one line of C++ code:

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

This type definition gives us a new C++ class which can represent an arbitrarily large integer or ‘`NA`’. This is set up in such a way that CXXR’s generic algorithms can detect and handle `NA`s with little or no attention from the package writer.

Extending the Class Hierarchy: Example
Adding NA to what the GNU library provides

The GNU MP library defines a C++ class `mpz_class` to represent an arbitrarily large integer.

But an attractive characteristic of R is its ability to flag individual data points as ‘not available’: `NA`. As it stands `mpz_class` does not have this capability.

Fortunately, in CXXR we can put this right essentially in one line of C++ code:

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

This type definition gives us a new C++ class which can represent an arbitrarily large integer or ‘`NA`’. This is set up in such a way that CXXR’s generic algorithms can detect and handle `NA`s with little or no attention from the package writer.

Extending the Class Hierarchy: Example

Vectors of BigInts

So far we can represent an individual BigInt. But of course R works primarily with **vectors** (or matrices or higher dimensional arrays). We can introduce vectors/matrices/array of BigInts into CXXR essentially with one further line of C++ code:

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

BigIntVectors have now joined the RObject class hierarchy alongside the built-in data vector types. We can now assign BigIntVectors to R variables, and facilities such as garbage collection, copy management, dimensioning and so on are automatically in place.

Extending the Class Hierarchy: Example

Vectors of `BigInt`s

So far we can represent an individual `BigInt`. But of course R works primarily with **vectors** (or matrices or higher dimensional arrays). We can introduce vectors/matrices/array of `BigInt`s into CXXR essentially with one further line of C++ code:

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

`BigIntVector`s have now joined the `RObject` class hierarchy alongside the built-in data vector types. We can now assign `BigIntVector`s to R variables, and facilities such as garbage collection, copy management, dimensioning and so on are automatically in place.

Binary Operations in R

Consider a binary operation on R vectors:

```
vr <- v1*v2
```

Basically this involves determining each element of the result by applying the binary operation to the corresponding elements of the two operands, so for example vr[1] is set to v1[1]*v2[1].

But there are complications. For example:

- If either operand element is NA, 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 give a warning if its length is not a submultiple of that of the longer operand.
- Attributes (e.g. element names) of the result must be inferred somehow from the corresponding attributes of the operands.

There are some further complications if the operands are matrices or higher dimensional arrays.

Binary Operations in R

Consider a binary operation on R vectors:

```
vr <- v1*v2
```

Basically this involves determining each element of the result by applying the binary operation to the corresponding elements of the two operands, so for example vr[1] is set to v1[1]*v2[1].

But there are complications. For example:

- If either operand element is NA, 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 give a warning if its length is not a submultiple of that of the longer operand.
- Attributes (e.g. element names) of the result must be inferred somehow from the corresponding attributes of the operands.

There are some further complications if the operands are matrices or higher dimensional arrays.

Binary Operations in R

Consider a binary operation on R vectors:

```
vr <- v1*v2
```

Basically this involves determining each element of the result by applying the binary operation to the corresponding elements of the two operands, so for example vr[1] is set to v1[1]*v2[1].

But there are complications. For example:

- If either operand element is `NA`, 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 give a warning if its length is not a submultiple of that of the longer operand.
- Attributes (e.g. element names) of the result must be inferred somehow from the corresponding attributes of the operands.

There are some further complications if the operands are matrices or higher dimensional arrays.

Generic Algorithm for R Binary Functions

CXXR defines a **generic algorithm** (based on the C++ class template `CXXR::VectorOps::BinaryFunction`) for implementing R binary functions, and makes it available to package C++ code via the CXXR API.

To use this algorithm the package writer need only specify:

- The elementwise operation to be performed, e.g. the multiplication operation defined for `mpz_class` by the GNU MP library.
- The two operands.
- The type of vector (or other vector-like container) to be produced as the result.
- The way in which attributes of the result (e.g. row and column names) are to be inferred from the operands (and usually a default value suffices for this).

Generic Algorithm for R Binary Functions

CXXR defines a **generic algorithm** (based on the C++ class template `CXXR::VectorOps::BinaryFunction`) for implementing R binary functions, and makes it available to package C++ code via the CXXR API.

To use this algorithm the package writer need only specify:

- The elementwise operation to be performed, e.g. the multiplication operation defined for `mpz_class` by the GNU MP library.
- The two operands.
- The type of vector (or other vector-like container) to be produced as the result.
- The way in which attributes of the result (e.g. row and column names) are to be inferred from the operands (and usually a default value suffices for this).

Generic Algorithm for R Binary Functions

CXXR defines a **generic algorithm** (based on the C++ class template `CXXR::VectorOps::BinaryFunction`) for implementing R binary functions, and makes it available to package C++ code via the CXXR API.

To use this algorithm the package writer need only specify:

- The elementwise operation to be performed, e.g. the multiplication operation defined for `mpz_class` by the GNU MP library.
- The two operands.
- The type of vector (or other vector-like container) to be produced as the result.
- The way in which attributes of the result (e.g. row and column names) are to be inferred from the operands (and usually a default value suffices for this).

Generic Algorithm for R Binary Functions

CXXR defines a **generic algorithm** (based on the C++ class template `CXXR::VectorOps::BinaryFunction`) for implementing R binary functions, and makes it available to package C++ code via the CXXR API.

To use this algorithm the package writer need only specify:

- The elementwise operation to be performed, e.g. the multiplication operation defined for `mpz_class` by the GNU MP library.
- The two operands.
- The type of vector (or other vector-like container) to be produced as the result.
- The way in which attributes of the result (e.g. row and column names) are to be inferred from the operands (and usually a default value suffices for this).

Generic Algorithm for R Binary Functions

CXXR defines a **generic algorithm** (based on the C++ class template `CXXR::VectorOps::BinaryFunction`) for implementing R binary functions, and makes it available to package C++ code via the CXXR API.

To use this algorithm the package writer need only specify:

- The elementwise operation to be performed, e.g. the multiplication operation defined for `mpz_class` by the GNU MP library.
- The two operands.
- The type of vector (or other vector-like container) to be produced as the result.
- The way in which attributes of the result (e.g. row and column names) are to be inferred from the operands (and usually a default value suffices for this).

Extending the Class Hierarchy: Example

Computing factorials

With very little programming at the package level, we are already in a position to calculate some largish factorials:

```
> 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
```

Subscripting in R

R is renowned for the power of its subscripting operations.

```
> mx

Country     Latest  Qtr  2011  2012
  Austria      3.9  3.6   2.7   1.9
  Belgium      3.0  4.3   2.3   1.8
  France       2.2  3.8   2.1   1.7
  Germany      5.4  6.1   3.4   2.2
  Greece      −5.5  0.7  −4.1  −0.1
```

(GDP growth, *The Economist*, 2011-07-09)

Subscripting in R

R is renowned for the power of its subscripting operations.

```
> mx

Country     Latest  Qtr  2011  2012
  Austria      3.9  3.6   2.7   1.9
  Belgium      3.0  4.3   2.3   1.8
  France       2.2  3.8   2.1   1.7
  Germany      5.4  6.1   3.4   2.2
  Greece      −5.5  0.7  −4.1  −0.1
```

(GDP growth, *The Economist*, 2011-07-09)

On the right are some examples of subscripting the R matrix above.

```
> mx[c("France", "Germany"), −2]

Country     Latest  2011  2012
  France       2.2   2.1   1.7
  Germany      5.4   3.4   2.2
```

```
> mx[,3]
Austria  Belgium   France
    2.7      2.3      2.1
Germany   Greece
    3.4     −4.1
```

```
> mx[,3, drop=FALSE]

Country    2011
  Austria   2.7
  Belgium   2.3
  France    2.1
  Germany   3.4
  Greece   −4.1
```

Subscripting in R

R is renowned for the power of its subscripting operations.

```
> mx

Country    Latest  Qtr  2011  2012
  Austria     3.9  3.6   2.7   1.9
  Belgium     3.0  4.3   2.3   1.8
  France      2.2  3.8   2.1   1.7
  Germany     5.4  6.1   3.4   2.2
  Greece     −5.5  0.7  −4.1  −0.1
```

(GDP growth, *The Economist*, 2011-07-09)

On the right are some examples of subscripting the R matrix above.

Subscripting expressions may also appear on the left-hand side of an assignment.

```
> mx[,"2012"] <− mx[,"2012"] + 0.5
> mx

Country    Latest  Qtr  2011  2012
  Austria     3.9  3.6   2.7   2.4
  Belgium     3.0  4.3   2.3   2.3
  France      2.2  3.8   2.1   2.2
  Germany     5.4  6.1   3.4   2.7
  Greece     −5.5  0.7  −4.1   0.4
```

Subscripting in R
(continued)

The *R Language Definition* document devotes over four of its 51 pages to describing subscripting facilities. . . and even that doesn't tell the whole story.

The CR interpreter includes about 2000 C-language statements to implement these facilities.

But this C code is effectively 'locked up' for two related reasons:

- it isn't made available via a documented API,
- it is hard-wired around CR's built-in data types.

Consequently this code is not, as it stands, suitable for providing subscripting facilities for our `BigIntVector`s.

Subscripting in R
(continued)

The *R Language Definition* document devotes over four of its 51 pages to describing subscripting facilities... and even that doesn't tell the whole story.

The CR interpreter includes about 2000 C-language statements to implement these facilities.

But this C code is effectively 'locked up' for two related reasons:

- it isn't made available via a documented API,
- it is hard-wired around CR's built-in data types.

Consequently this code is not, as it stands, suitable for providing subscripting facilities for our `BigIntVector`s.

Subscripting in R
(continued)

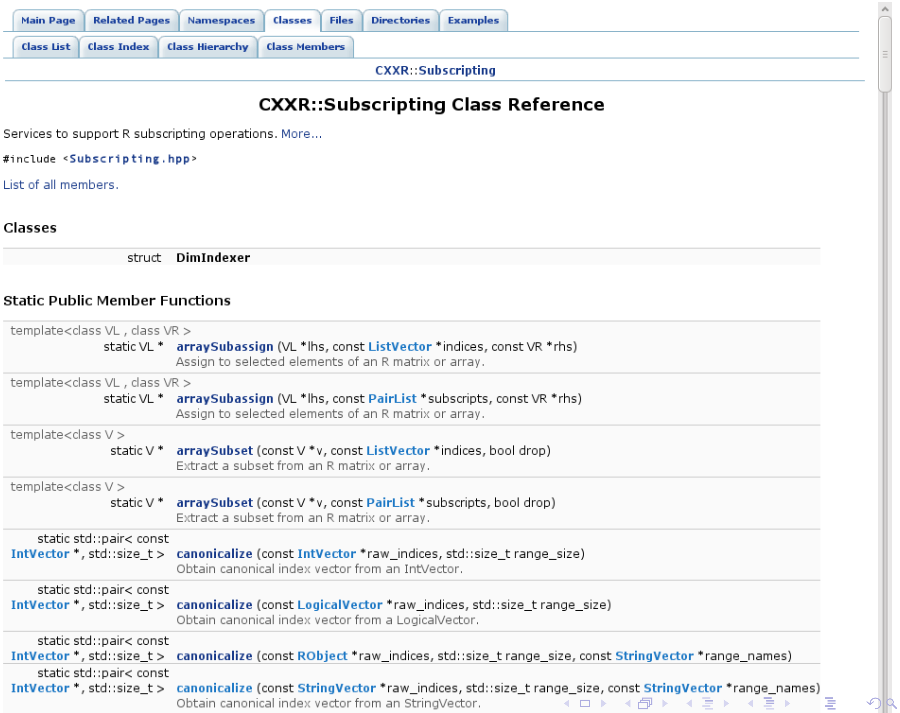
The *R Language Definition* document devotes over four of its 51 pages to describing subscripting facilities. . . and even that doesn't tell the whole story.

The CR interpreter includes about 2000 C-language statements to implement these facilities.

But this C code is effectively 'locked up' for two related reasons:

- it isn't made available via a documented API,
- it is hard-wired around CR's built-in data types.

Consequently this code is not, as it stands, suitable for providing subscripting facilities for our `BigIntVector`s.



CXXR::Subscripting Class Reference

Services to support R subscripting operations. More...

`#include <Subscripting.hpp>`

List of all members.

Classes

struct	**DimIndexer**

Static Public Member Functions

template<class VL , class VR > static VL *	**arraySubassign** (VL *lhs, const ListVector *indices, const VR *rhs) Assign to selected elements of an R matrix or array.
template<class VL , class VR > static VL *	**arraySubassign** (VL *lhs, const PairList *subscripts, const VR *rhs) Assign to selected elements of an R matrix or array.
template<class V > static V *	**arraySubset** (const V *v, const ListVector *indices, bool drop) Extract a subset from an R matrix or array.
template<class V > static V *	**arraySubset** (const V *v, const PairList *subscripts, bool drop) Extract a subset from an R matrix or array.
static std::pair< const IntVector *, std::size_t >	**canonicalize** (const IntVector *raw_indices, std::size_t range_size) Obtain canonical index vector from an IntVector.
static std::pair< const IntVector *, std::size_t >	**canonicalize** (const LogicalVector *raw_indices, std::size_t range_size) Obtain canonical index vector from a LogicalVector.
static std::pair< const IntVector *, std::size_t >	**canonicalize** (const RObject *raw_indices, std::size_t range_size, const StringVector *range_names)
static std::pair< const IntVector *, std::size_t >	**canonicalize** (const StringVector *raw_indices, std::size_t range_size, const StringVector *range_names) Obtain canonical index vector from an StringVector.

CXXR's `Subscripting` Class

CXXR's `Subscripting` class aims to encapsulate R's subscripting facilities within a number of generic algorithms.

These algorithms abstract away from:

- The type of the elements of the R vector/matrix/array. (`BigInt`s work just fine!)
- The data structure used to implement the vector/matrix/array itself. This opens the door to using the algorithms with packed data (e.g. A/T/G/C DNA bases), or with vector structures for large datasets which hold data on disk (in the style of the `ff` package).

Using the `Subscripting` class, we can carry across the full power of R subscripting to `BigIntVector`s with just a few lines of package code.

CXXR's `Subscripting` Class

CXXR's `Subscripting` class aims to encapsulate R's subscripting facilities within a number of generic algorithms.

These algorithms abstract away from:

- The type of the elements of the R vector/matrix/array. (`BigInt`s work just fine!)
- The data structure used to implement the vector/matrix/array itself. This opens the door to using the algorithms with packed data (e.g. A/T/G/C DNA bases), or with vector structures for large datasets which hold data on disk (in the style of the `ff` package).

Using the `Subscripting` class, we can carry across the full power of R subscripting to `BigIntVector`s with just a few lines of package code.

CXXR's `Subscripting` Class

CXXR's `Subscripting` class aims to encapsulate R's subscripting facilities within a number of generic algorithms.

These algorithms abstract away from:

- The type of the elements of the R vector/matrix/array. (`BigInt`s work just fine!)
- The data structure used to implement the vector/matrix/array itself. This opens the door to using the algorithms with packed data (e.g. A/T/G/C DNA bases), or with vector structures for large datasets which hold data on disk (in the style of the `ff` package).

Using the `Subscripting` class, we can carry across the full power of R subscripting to `BigIntVector`s with just a few lines of package code.

Summary So Far

CXXR aims to open up the R interpreter to developers. In particular:

- Objects visible to R are implemented using a C++ class hierarchy which developers can easily extend.
- Key algorithms embodying R functionality are being rewritten at a higher level of abstraction and published via the CXXR API.

Summary So Far

CXXR aims to open up the R interpreter to developers. In particular:

- Objects visible to R are implemented using a C++ class hierarchy which developers can easily extend.
- Key algorithms embodying R functionality are being rewritten at a higher level of abstraction and published via the CXXR API.

Outline

Tracking Provenance During Data Analysis

Have you ever returned to a data analysis after a gap of months (or maybe years) and asked yourself such questions as the following?

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

In other words, you are interesting in interrogating the *provenance* of data objects, models, etc. This is a topic of increasing importance in information science.

Tracking Provenance During Data Analysis

Have you ever returned to a data analysis after a gap of months (or maybe years) and asked yourself such questions as the following?

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

In other words, you are interesting in interrogating the *provenance* of data objects, models, etc. This is a topic of increasing importance in information science.

Tracking Provenance During Data Analysis

Have you ever returned to a data analysis after a gap of months (or maybe years) and asked yourself such questions as the following?

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

In other words, you are interesting in interrogating the *provenance* of data objects, models, etc. This is a topic of increasing importance in information science.

Tracking Provenance During Data Analysis

Have you ever returned to a data analysis after a gap of months (or maybe years) and asked yourself such questions as the following?

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

In other words, you are interesting in interrogating the *provenance* of data objects, models, etc. This is a topic of increasing importance in information science.

Provenance Tracking in S and R

One of the pioneer provenance-aware applications was S, with its AUDIT facility: the classic paper *Auditing of Data Analyses*[2] by Becker and Chambers is widely cited in the provenance-awareness literature.

An S session would maintain an **audit file**, recording all the top-level commands issued in this and previous sessions within the workspace, and identifying the data objects read and modified by the commands. This audit file could then be analysed using a separate tool, `S AUDIT`.

Chris Silles has been exploring the possibility of introducing such a facility into R, building on CXXR . . . but with the difference that provenance information can be interrogated directly from within an R session.

[2] SIAM J. Sci. Stat. Comput. 9 [1988] pp. 747–60

Provenance Tracking in S and R

One of the pioneer provenance-aware applications was S, with its AUDIT facility: the classic paper *Auditing of Data Analyses*[2] by Becker and Chambers is widely cited in the provenance-awareness literature.

An S session would maintain an **audit file**, recording all the top-level commands issued in this and previous sessions within the workspace, and identifying the data objects read and modified by the commands. This audit file could then be analysed using a separate tool, `S AUDIT`.

Chris Silles has been exploring the possibility of introducing such a facility into R, building on CXXR . . . but with the difference that provenance information can be interrogated directly from within an R session.

[2]SIAM J. Sci. Stat. Comput. 9 [1988] pp. 747–60

Provenance Tracking Example

Consider this example 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)
```

Provenance Tracking Example

Consider this example 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)
```

Querying the provenance of an R object (strictly, an R binding):

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

$symbol
nine

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

$parents
[1] "square" "three"

$children
NULL
```

Provenance Tracking Example

Consider this example 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)
```

Querying the full pedigree of an R object:

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

Provenance Tracking and Serialization

Although having provenance information available during a single R session is useful, its real value arises when a data analysis (perhaps carried out by someone else) is resumed after a lapse of time.

This means that it is essential that provenance information is saved at the end of a session along with the data objects to which it relates.

This has led us to look more generally at serialization and deserialization within CXXR: the process by which a set of R objects is rendered into a form suitable for saving in a file, and subsequently restored.

Provenance Tracking and Serialization

Although having provenance information available during a single R session is useful, its real value arises when a data analysis (perhaps carried out by someone else) is resumed after a lapse of time.

This means that it is essential that provenance information is saved at the end of a session along with the data objects to which it relates.

This has led us to look more generally at serialization and deserialization within CXXR: the process by which a set of R objects is rendered into a form suitable for saving in a file, and subsequently restored.

Serialization/Deserialization Objectives

- Delegate to each C++ class responsibility for serializing/deserializing objects of that class (instead of having a monolithic `serialize()` function as in CR).
- Provide for provenance information to be serialized/deserialized automatically alongside the data to which it relates.
- Provide an easy-to-use framework for package writers to have objects of package-supplied C++ classes (such as `BigIntVector`) serialized/deserialized along with other session data.

Current work is building on the facilities of the Boost serialization library.

Serialization/Deserialization Objectives

- Delegate to each C++ class responsibility for serializing/deserializing objects of that class (instead of having a monolithic `serialize()` function as in CR).
- Provide for provenance information to be serialized/deserialized automatically alongside the data to which it relates.
- Provide an easy-to-use framework for package writers to have objects of package-supplied C++ classes (such as `BigIntVector`) serialized/deserialized along with other session data.

Current work is building on the facilities of the Boost serialization library.

Serialization/Deserialization Objectives

- Delegate to each C++ class responsibility for serializing/deserializing objects of that class (instead of having a monolithic `serialize()` function as in CR).
- Provide for provenance information to be serialized/deserialized automatically alongside the data to which it relates.
- Provide an easy-to-use framework for package writers to have objects of package-supplied C++ classes (such as `BigIntVector`) serialized/deserialized along with other session data.

Current work is building on the facilities of the Boost serialization library.

Serialization/Deserialization Objectives

- Delegate to each C++ class responsibility for serializing/deserializing objects of that class (instead of having a monolithic `serialize()` function as in CR).
- Provide for provenance information to be serialized/deserialized automatically alongside the data to which it relates.
- Provide an easy-to-use framework for package writers to have objects of package-supplied C++ classes (such as `BigIntVector`) serialized/deserialized along with other session data.

Current work is building on the facilities of the Boost serialization library.

Outline

CXXR: An Ideas Hatchery

CXXR is offered as a workbench for you to try your own ideas out.

But volunteers to assist in the development of CXXR itself are also welcome. A conspicuous gap is a Windows port of CXXR: until now it has been tested only on Linux and (to a lesser extent) on MacOS X.

Functionality Now in CXXR Core

- Memory allocation and garbage collection.
- `SEXPREC` union replaced by an extensible class hierarchy rooted at class `RObject`.
- Environments (i.e. variable→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()`.)

Conway's 'Game of Life'

CPU time for 100 iterations over a square matrix with wraparound (toroidal topology):

Grid size	**CR (secs)**	**CXXR (secs)**
32×32	0·047	0·053
64×64	0·168	0·191
128×128	0·686	0·743
256×256	3·084	3·004
512×512	33·402	14·239
1024×1024	144·386	60·128

The tests were carried out on a 2.8 GHz Pentium 4 with 1 MB L2 cache, comparing R-2.12.1 with CXXR 0.35-2.12.1.

Extending the Class Hierarchy: Example

Multiplying vectors/arrays of arbitrarily large integers

Package R code:

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

Package 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);
    }
}
```

Extending the Class Hierarchy: Example
Subassignment: `[<-`

Package R code:

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

Package C++ code:

```
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);
    }
}
```