University of Kent | Computing

CXXR: Refactoring the R Interpreter into C++

Andrew Runnalls

Computing Laboratory, University of Kent, UK

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);
- The `.C` and `.Fortran` interfaces, and the `R.h` and `S.h` APIs, are unaffected;
- Code compiled against `Rinternals.h` may need minor alterations.

Work started in May 2007, shadowing R-2.5.1; the current release (tested on Linux and Mac OS X) shadows R-2.7.1.

[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);
- The `.C` and `.Fortran` interfaces, and the `R.h` and `S.h` APIs, are unaffected;
- Code compiled against `Rinternals.h` may need minor alterations.

Work started in May 2007, shadowing R-2.5.1; the current release (tested on Linux and Mac OS X) shadows R-2.7.1.

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

Why Do This?

My medium-term objective is to introduce provenance-tracking facilities into CXXR: so that for any R data object, it is possible to determine exactly which original data files it was produced from, and exactly which sequence of operations was used to produce it. (Similar to the old S AUDIT facility, but usable directly within R.)

Also:

- By improving the internal documentation, and
- Tightening up the internal encapsulation boundaries within the interpreter,

we hope that CXXR will make it easier for other researchers to produce experimental versions of the interpreter, and to enhance its facilities.

Why Do This?

My medium-term objective is to introduce provenance-tracking facilities into CXXR: so that for any R data object, it is possible to determine exactly which original data files it was produced from, and exactly which sequence of operations was used to produce it. (Similar to the old S AUDIT facility, but usable directly within R.)

Also:

- By improving the internal documentation, and
- Tightening up the internal encapsulation boundaries within the interpreter,

we hope that CXXR will make it easier for other researchers to produce experimental versions of the interpreter, and to enhance its facilities.

Progress So Far

- Memory allocation and garbage collection have been decoupled from each other and from R-specific functionality, and encapsulated within C++ classes.
- The `SEXPREC` union has been replaced by an extensible C++ class hierarchy.

Data Layout in CR

In CR (i.e. standard R), R data objects (nodes) are laid out in memory in one of these patterns:

Vectors:

SEXPTYPE and other info
Pointer to attributes
Pointer to next node (used by GC)
Pointer to prev. node (used by GC)
Length
'True length'
Vector data

Other nodes:

SEXPTYPE and other info
Pointer to attributes
Pointer to next node (used by GC)
Pointer to prev. node (used by GC)
Pointer
Pointer
Pointer

All the above objects are handled *via* a single C type `SEXPREC`; the `SEXPTYPE` field identifies the particular kind of object it is, e.g. pairlist (`LISTSXP`), expression (`LANGSXP`), or vector of integers (`INTSXP`).

Data Layout in CR

Drawbacks

SEXPTYPE and other info
Pointer to attributes
Pointer to next node (used by GC)
Pointer to prev. node (used by GC)
Length
'True length'
Vector data

SEXPTYPE and other info
Pointer to attributes
Pointer to next node (used by GC)
Pointer to prev. node (used by GC)
Pointer
Pointer
Pointer

- Data allocation and garbage collection work directly in terms of these node patterns.
- Consequently, introducing an object type that doesn't conform to the pattern is a big deal. There is a tendency to shoehorn objects into the 'three pointers' pattern, and to use data fields for purposes different from what was originally intended.
- Checking that a node is of a type appropriate to its context is always done at run-time, never at compile-time.
- The CR code is filled with switches and tests on the `SEXPTYPE`.

Data Layout in CR

Drawbacks

SEXPTYPE and other info
Pointer to attributes
Pointer to next node (used by GC)
Pointer to prev. node (used by GC)
Length
'True length'
Vector data

SEXPTYPE and other info
Pointer to attributes
Pointer to next node (used by GC)
Pointer to prev. node (used by GC)
Pointer
Pointer
Pointer

- Data allocation and garbage collection work directly in terms of these node patterns.
- Consequently, introducing an object type that doesn't conform to the pattern is a big deal. There is a tendency to shoehorn objects into the 'three pointers' pattern, and to use data fields for purposes different from what was originally intended.
- Checking that a node is of a type appropriate to its context is always done at run-time, never at compile-time.
- The CR code is filled with switches and tests on the `SEXPTYPE`.

Data Layout in CR
Drawbacks

SEXPTYPE and other info
Pointer to attributes
Pointer to next node (used by GC)
Pointer to prev. node (used by GC)
Length
'True length'
Vector data

SEXPTYPE and other info
Pointer to attributes
Pointer to next node (used by GC)
Pointer to prev. node (used by GC)
Pointer
Pointer
Pointer

- Data allocation and garbage collection work directly in terms of these node patterns.
- Consequently, introducing an object type that doesn't conform to the pattern is a big deal. There is a tendency to shoehorn objects into the 'three pointers' pattern, and to use data fields for purposes different from what was originally intended.
- Checking that a node is of a type appropriate to its context is always done at run-time, never at compile-time.
- The CR code is filled with switches and tests on the `SEXPTYPE`.

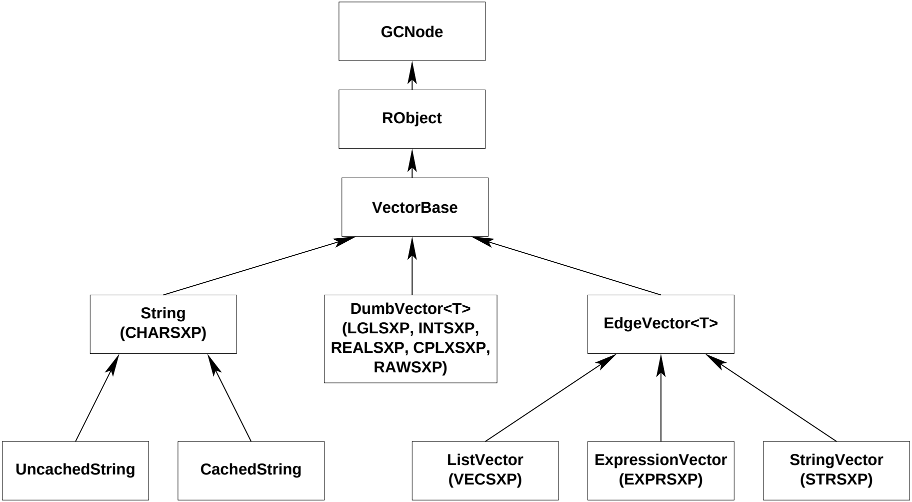
Data Layout in CR
Drawbacks

SEXPTYPE and other info
Pointer to attributes
Pointer to next node (used by GC)
Pointer to prev. node (used by GC)
Length
‘True length’
Vector data

SEXPTYPE and other info
Pointer to attributes
Pointer to next node (used by GC)
Pointer to prev. node (used by GC)
Pointer
Pointer
Pointer

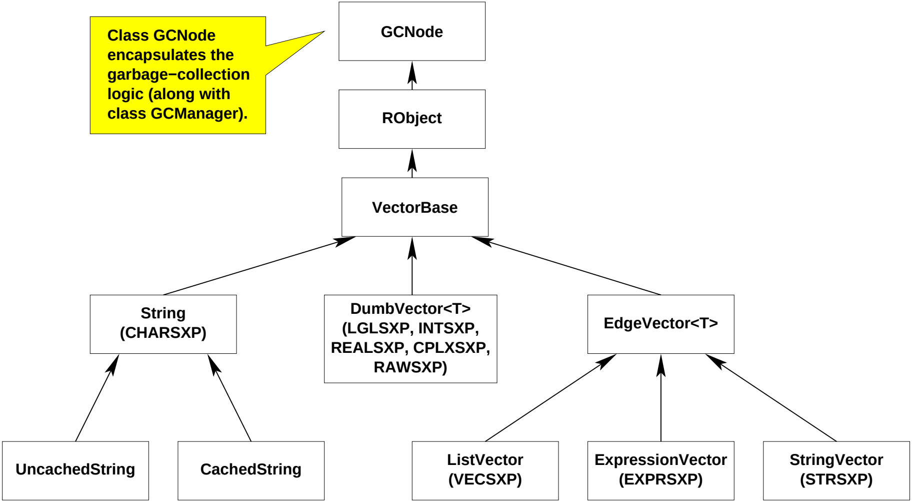
- Data allocation and garbage collection work directly in terms of these node patterns.
- Consequently, introducing an object type that doesn’t conform to the pattern is a big deal. There is a tendency to shoehorn objects into the ‘three pointers’ pattern, and to use data fields for purposes different from what was originally intended.
- Checking that a node is of a type appropriate to its context is always done at run-time, never at compile-time.
- The CR code is filled with switches and tests on the `SEXPTYPE`.

Vector Classes in CXXR



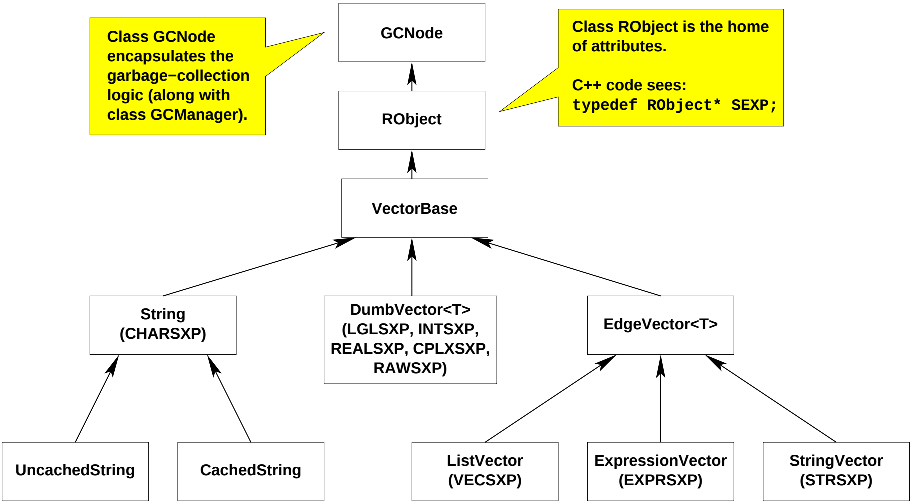
This class inheritance hierarchy is readily extensible.

Vector Classes in CXXR



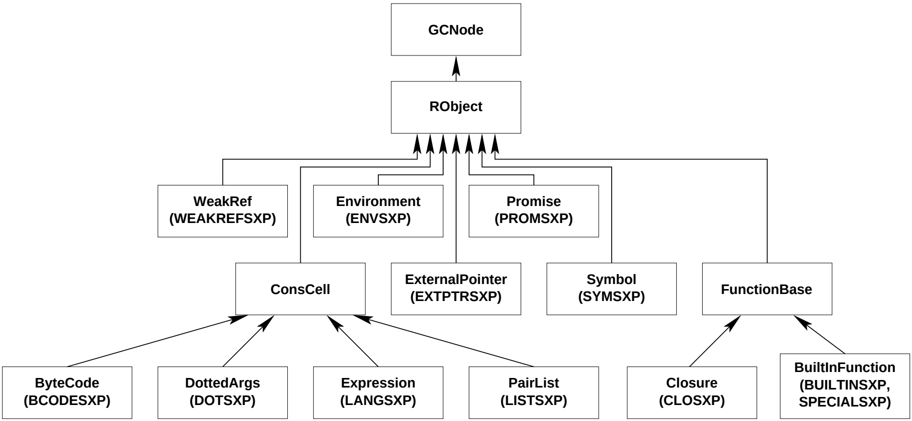
This class inheritance hierarchy is readily extensible.

Vector Classes in CXXR



This class inheritance hierarchy is readily extensible.

Other Node Classes in CXXR



This is a fairly simple-minded first cut, and is subject to change.

```
void insertAfter(ConsCell* location, RObject* car,
                 RObject* tag = 0)
{
    GCRoot<PairList> tail(location->tail());
    PairList* node = new PairList(car, tail, tag);
    location->setTail(node);
}
```

(This is only an illustrative example, not part of the CXXR code base.)

Some Features of CXXR Internal Code

```
void insertAfter(ConsCell* location, RObject* car,
                 RObject* tag = 0)
{
    GCRoot<PairList> tail(loca…ion->tail());
    PairList* nod…                (car, tail, tag);
    location->set…
}
```

The default is for the newly inserted node to have no tag: in CXXR, R_NilValue is simply a null pointer.

(This is only an illustrative example, not part of the CXXR code base.)

Some Features of CXXR Internal Code

```
void insertAfter(ConsCell* location, RObject* car,
                 RObject* tag = 0)
{
    GCRoot<PairList> tail(location->tail());
    PairList* node = new PairList(car, tail, tag);
    location->setTail(node);
}
```

GCRoot is a (templated) 'smart pointer' type. It can be used like a pointer (PairList* in this case), but protects whatever it points to from the garbage collector.

(This is only an illustrative example, not part of the CXXR code base.)

Some Features of CXXR Internal Code

```
void insertAfter(ConsCell* location, RObject* car,
                 RObject* tag = 0)
{
    GCRoot<PairList> tail(location->tail());
    PairList* node = new PairList(car, tail, tag);
    location->setTail(node);
}
```

The invocation of 'new' may result in a garbage collection.

(This is only an illustrative example, not part of the CXXR code base.)

Some Features of CXXR Internal Code

```
void insertAfter(ConsCell* location, RObject* car,
                 RObject* tag = 0)
{
    GCRoot<PairList> tail(location->tail());
    PairList* node = new PairList(car, tail, tag);
    location->setTail(node);
}
```

The GCRoot goes out of scope here, so the GC−protection it offers to tail ends automatically: no need to balance PROTECT()/ UNPROTECT() 'by hand'.

(This is only an illustrative example, not part of the CXXR code base.)

Some Features of CXXR Internal Code

```
void insertAfter(ConsCell* location, RObject* car,
                 RObject* tag = 0)
{
    location->setTail(new PairList(car,
                                   location->tail(),
                                   tag));
}
```

(This is only an illustrative example, not part of the CXXR code base.)

Benchmarks

The following tests were carried out on a 2.8 GHz Pentium 4 with 1 MB L2 cache, comparing R-2.7.1 with CXXR 0.14-2.7.1, in each case using `gcc -O2` and no `USE_TYPE_CHECKING`.

Benchmark	CR (secs)	CXXR (secs)	Ratio
`bench.R` (Jan de Leeuw)	108.0 ± 0.3	108.0 ± 0.2	≈ 1
`mass-Ex.R` (Simon Urbanek)	29.68 ± 0.03	42.38 ± 0.06	1.43
`stats-Ex.R`	23.04 ± 0.01	34.50 ± 0.01	1.50

The reasons for the time penalty in CXXR are not yet fully understood: the target is to get it down to 30% or better.

Tentative Roadmap

1. Further adjustments to the class hierarchy.
2. Reimplement `duplicate()` using C++ copy constructors and an `RObject::clone()` virtual function.
3. Reimplement `eval()` as a C++ virtual function.
4. New serialisation format, probably XML-based. This is to make it easier to introduce new node classes, and to support provenance-tracking information.
5. Reengineer the `Environment` class, which will lie at the centre of provenance tracking.