Provenance Tracking in R

Andrew Runnalls and Chris Silles

University of Kent | Computing

CXXR

The CXXR project aims gradually to reengineer the fundamental parts of the R interpreter **from C into C++** in such a way that:

- the 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 interpreter internals);
- there is no change to the existing interfaces for calling out from R to other languages such as C or Fortran, nor to the main APIs for calling into R.

CXXR achieves a high degree of compatibility with R packages from the **CRAN** repository: see [1].

The **AUDIT** facilities [2] that once formed part of S and S-plus were an invaluable feature, and one motivation behind CXXR was to introduce similar but better facilities into the R interpreter. Early work on a provenance-enabled variant of CXXR was presented in [3].

Work on CXXR started in May 2007, at that time shadowing R-2.5.1. Since then CXXR has been **regularly upgraded** to keep pace with the major releases of R (usually synching on the .1 minor release), so for example over the last year CXXR has shadowed the increasing deployment of the **bytecode compiler** within standard R. The current release of CXXR shadows R-2.14.1.

R Objects in CXXR

Standard R provides for only a fixed range of object types (implemented as a C union) to be assigned to R variables, and to participate in the interpreter's garbage collection scheme. In contrast, data objects in CXXR are implemented as a C++ class hierarchy, which can be extended at will. The provenance-tracking variant of CXXR leverages this feature extensively.

OPM Relationship

The provenance-enabled variant of CXXR maps the concepts of the OPM 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.

Agent: Not currently used in CXXR.

Why bindings? You may be surprised that it is R bindings, rather than R objects themselves, that are taken as artifacts. But consider the R object 0 (i.e. the 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.

Provenance-enabled CXXR instruments the reading and writing of bindings within the 'global environment' (R's main workspace), 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: this marks a difference from the S AUDIT facility, which required a separate tool to query provenance data.

Serialization

A recent development in CXXR (not yet incorporated into the development trunk) is to reengineer the way that data are serialized and deserialized between one session and the next, by drawing on the serialization facilities of the well-regarded open-source Boost C++ libraries (`www.boost.org`). This means that not only can developers extend the range of data types usable within the interpreter, they can also—within the new C++ class definitions themselves—specify how objects of that class are saved to and restored from the session archive (which now uses an XML format). This applies not least to the classes implementing the provenance audit trail, so that this is carried forward from one CXXR session to the next.

Example

Imagine you've just come back from vacation, and are on the point of resuming an R data analysis. Probably you'd start with…

```
> ls()
 [1] "air.boot"       "air.fun"        "air.rg"         "cleanEx"
 [5] "diff.means"     "grav1"          "grav.fun"       "gravity"
 [9] "grav.L"         "grav.mom"       "grav.p"         "grav.q"
[13] "grav.tilt"      "grav.tilt.boot" "grav.w"         "grav.z0"
[17] "mean.diff"      "new.data"       "new.fit"        "nuke"
[21] "nuke.boot"      "nuke.data"      "nuke.diag"      "nuke.fun"
[25] "nuke.lm"        "nuke.res"       "pkgname"        "ratio"
```

…and suddenly wish you'd kept better notes of what you'd done before!

Fortunately, with provenance-enabled CXXR, you can easily query the 'pedigree' of R variables, like this:

```
> pg <- pedigree("grav.tilt.boot")
> pg$commands
[[1]]
load("gravity.rda")

[[2]]
grav1 <- gravity[as.numeric(gravity[, 2]) >= 7, ]

[[3]]
grav.fun <- function(dat, w, orig) {
    strata <- tapply(dat[, 2], as.numeric(dat[, 2]))
    d <- dat[, 1]
    ns <- tabulate(strata)
    w <- w/tapply(w, strata, sum)[strata]
    mns <- tapply(d * w, strata, sum)
    mn2 <- tapply(d * d * w, strata, sum)
    s2hat <- sum((mn2 - mns^2)/ns)
    as.vector(c(mns[2] - mns[1], s2hat, (mns[2] - mns[1] - orig)/sqrt(s2hat)))
}

[[4]]
grav.z0 <- grav.fun(grav1, rep(1, 26), 0)

[[5]]
grav.L <- empinf(data = grav1, statistic = grav.fun, stype = "w",
    strata = grav1[, 2], index = 3, orig = grav.z0[1])

[[6]]
grav.tilt <- exp.tilt(grav.L, grav.z0[3], strata = grav1[, 2])

[[7]]
grav.tilt.boot <- boot(grav1, grav.fun, R = 199, stype = "w",
    strata = grav1[, 2], weights = grav.tilt$p, orig = grav.z0[1])
```

This shows, in time order, **all the top-level R commands previously issued that are relevant to the current value of the R variable** `grav.tilt.boot`. Moreover you can find out when these commands were issued:

```
> pg$timestamps
[1] "2012-06-08 15:02:57 GMT" "2012-06-11 10:57:59 GMT"
[3] "2012-06-11 10:58:17 GMT" "2012-06-11 10:58:27 GMT"
[5] "2012-06-11 10:59:26 GMT" "2012-06-11 10:59:36 GMT"
[7] "2012-06-11 11:00:43 GMT"
```

Xenogenesis

Consider the following R session:

```
> x <- 1:10
> mysteryf <- edit(function(x){})
> y <- mysteryf(x)
> rm(mysteryf)
> y
 [1]  3  8  6 12  0  2  6 12  8  0
```

To understand fully the provenance of the current binding of `y`, it isn't sufficient to know that it was generated by the above sequence of commands. We also want to know what the function `mysteryf` was. But it has already been deleted from the R session!

The problem here is the function `edit`, which calls an external editor to edit an R object. Rather unusually among R functions, its return value doesn't depend only on its arguments, nor even on its arguments and on other bindings in the R session: *it depends on something external to the R session entirely*. (`load` in the main example is another such function.) We call functions like this **xenogenetic**, and the bindings they give rise to **xenogenous**: "due to an outside cause".

To work around this problem, the approach currently being explored is for the audit trail to identify whether a binding is xenogenous, and if so *to record the value* of that binding. So in the example above, the value bound to `mysteryf` by the top-level command `mysteryf <- edit(function(x){})` will be recorded in the audit trail *for as long as any artifact dependent on that binding exists*.

In the example, the mystery function can be retrieved like this:

```
> pedigree("y")$values[[2]]
function(x){
  (2 + x%%3)*(x%%5)
}
```

Environments

In R (and CXXR), an **environment** is—roughly speaking—a container for bindings. At present CXXR tracks the provenance of bindings within R's global environment `.GlobalEnv`. This tracking can be extended to other standard environments set up at the start of an R session, though this can result in a deluge of provenance data that would rarely be of value.

Each invocation of a function written in R results in the creation of a **local environment**. Normally these are evanescent, and can be ignored by the provenance tracker. But there are exceptions, and it is for example possible to define an R function—such as `counter` in the example below—which in effect has internal state, stored in a local environment and carried forward from one invocation to the next.

```
> makecounter <- function(){
+   count <- 0
+   function(){
+     count <<- count + 1
+     count
+   }
+ }
> counter <- makecounter()
> counter()  ###
[1] 1
> x <- counter()
> x
[1] 2
```

At present the CXXR provenance tracker does not realise that the command marked `###` is relevant to the provenance of `x`, but work is in the pipeline to rectify this.

One remaining concern is that there is currently no method of referring to local environments in a way that is meaningful between R sessions: this can hamper reproducibility, especially in the presence of xenogenesis.

Acknowledgments

CXXR would of course have been impossible without R and the tireless efforts of the R core team and other R contributors.

The example above is based on material from the R `boot` package, which in turn is based on an example from Davison and Hinkley [1997].

References

[1] Runnalls, A.: CXXR and add-on packages. In: useR! 2010. (2010) Available at `http://user2010.org/slides/Runnalls.pdf`.

[2] Becker, R.A., Chambers, J.M.: Auditing of data analyses. SIAM J. Sci. Stat. Comput. **9** (1988) 747–60

[3] Silles, C., Runnalls, A.: Provenance-awareness in R. In McGuinness, D., Michaelis, J., Moreau, L., eds.: Provenance and Annotation of Data and Processes. Volume 6378 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2010) 64–72