Provenance-Awareness in R

Chris A. Silles and Andrew R. Runnalls

School of Computing
University of Kent
Canterbury, UK
{C.A.Silles,A.R.Runnalls}@kent.ac.uk

Abstract. It is generally acknowledged that when, in 1988, John Chambers and Richard Becker incorporated the *S AUDIT* facility into their *S* statistical programming language and environment, they created one of the first provenance-aware applications. Since then, *S* has been spiritually succeeded by the open-source *R* project; however, *R* has no such facility for tracking provenance. This paper looks at how provenance-awareness is being introduced to *CXXR* (`http://www.cs.kent.ac.uk/projects/cxxr`), a variant of the *R* interpreter designed to allow creation of experimental *R* versions. We explore the issues surrounding recording, representing, and interrogating provenance information in a command-line driven interactive environment that utilises a lazy functional programming language. We also characterise provenance information in this domain and evaluate the impact of adding facilities for provenance tracking.

1 Introduction

The use of computer systems for recording information has proliferated in recent years; however, facilities for recording *how* this data has come to be in its present state have only recently started to catch up due to research in the field of provenance-aware computing. This discipline has developed quickly over the last decade and is now reaching maturity with the Open Provenance Model for the representation and exchange of provenance information [1].

In this paper we look at how facilities for recording and examining provenance have been introduced to the interactive statistical environment and programming language, CXXR, which is based on the popular R project [2]. Recording process documentation for the purpose of reproducible computing in R has previously been researched in *Sweave* [3], a system based on concepts of literate programming [4].

Making applications provenance-aware is not in itself a new concept [5]; however, CXXR presents some novel challenges, primarily to the way in which provenance is represented conceptually, but also to the way in which provenance needs to be presented to the user, and how particular features of the language require modelling in order to capture complete provenance.

The structure of this paper is as follows. Section 2 provides an introduction to the software and describes the approach to handling provenance therein. We then, in Section 3, detail the implementation steps we have taken. We conclude with a summary of findings and what we are looking forward to in Section 4.

D.L. McGuinness, J.R. Michaelis, and L. Moreau (Eds.): IPAW 2010, LNCS 6378, pp. 64–72, 2010.

2 CXXR

2.1 History

CXXR is a variant of *R*, which is an open-source implementation of *S*.

S. S is a language and interactive environment for statistical computing, graphics, and exploratory data analysis [6]. It was developed during the mid-1970s at Bell Laboratories by John Chambers and Richard Becker. S emerged from Bell Labs at around the same time as the C programming language, and this is reflected in both its syntax and name. Despite this, it uses the semantics of a functional programming language, including employment of lazy evaluation.

S AUDIT. 'New S' was released in 1988 sporting a new feature entitled *S AUDIT* [7]. While a user operated a session within S, a record was maintained of each top-level expression evaluated; as well as objects read from and written to during the course of evaluation.

The accompanying S AUDIT program was able to process this record and allow the user to interrogate it. S AUDIT was able to perform a number of queries on the audit record, such as displaying the full sequence of statements; those statements responsible for reading from, or writing to, a specific object; or simply providing a list of all objects in the session.

A more intriguing feature of S AUDIT was its ability to create an *audit plot*, which was a directed-acyclic graph with statements as nodes arranged on the circumference of a circle (anti-clockwise in order of creation), and edges each representing an object written by one statement, later being read by another. Audit plots enabled users to visualise how objects were being used over their lifetime.

New S, therefore, became one of the first *provenance-aware* software applications, and even featured visualisation of provenance: features that were at the time innovative, and still remain novel today.

R and CXXR. While S as an application continues life as a commercial product called S+ retailed by TIBCO [8], the language, library and environment have been reimplemented as part of the open-source R project [2]. The R distribution comprises an interpreter and a number of packages for common functionality, which have been written in a mixture of C, Fortran, and R itself. It is maintained by a nineteen-strong team of core developers, and enjoys a large and active userbase working in areas as diverse as retail strategy, genetics, education, pharmacology, proteomics, and data and text mining.

CXXR is a project to reengineer the fundamental components of the R interpreter from C into C++, while fully preserving functionality of the standard R distribution [9]. The primary objective of CXXR is to enable experimental versions of the R interpreter to be created, allowing new functionality to be easily introduced.

Listing 1. Example R commands

```
> 1+2
[1] 3
> three <- 1+2
> square <- function(x) { x*x }
> nine <- square(three)
> nine
[1] 9
```

2.2 How R Works

Data Types. R has many data types, the most important of which is the *vector*. Vectors are homogeneous arrays of data, and may be composed of elements of types including integers, booleans, strings, and real and complex numbers. Vectors are ubiquitous in R. Even a single value (e.g. 3.14) is treated as a vector having only one element.

Example. Listing 1 shows the evaluation of some commands in R. The `>` character is the prompt, at which the user enters commands. The first statement performs a simple addition, and R prints the result. The square brackets indicate that the result is a vector, and the number signifies the index of the element at the beginning of the line. The second and third statements show assignment of a vector and a function to objects respectively.

Objects. When the user performs an assignment, a *binding* is created between a *symbol* object and a *value* object. The space in which bindings are stored is known as an *environment*. Environments are used, among other things, to define scope. The two environments with which we are concerned here are the *global* and *base* environments. The workspace the user operates in is the *global* environment, and the standard library functions reside in the *base* environment. In the second statement of Listing 1, a binding is created in the global environment between the symbol `three`, and an integer vector containing the single element '3', as illustrated by Figure 1.

Garbage Collection. R — and CXXR likewise — is *garbage collected*, so objects that can no longer be accessed by the user because they have either been manually deleted or bindings to them have been reassigned to reference other objects, will at some point be destroyed by the garbage collector, which then releases unused memory.

2.3 Making CXXR Provenance-Aware

The principal objective of this work is to enable CXXR to identify the following information of any given object: -

1. The *process* that led to it – the sequence of commands executed;
2. Its *ancestors* – which other objects it depends on;
3. Its *descendants* – which other objects depend on it.

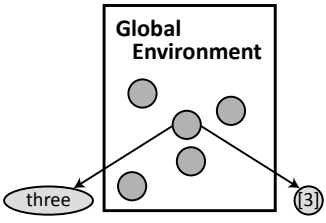

Fig. 1. Bindings exist within environments and connect symbols to values. In this case, the symbol 'three' with a singleton integer vector '3'.

2.4 What Provenance?

The use of the word 'object' in R is unfortunately ambiguous. As mentioned above, what is commonly referred to as an 'object' in R is really a binding in an environment between a symbol and an object representing a value. The R language is *dynamically typed*, which means a variable (i.e. 'object') has no intrinsic type, and simply takes on the type of the object assigned to it. When referring to 'object x', what is often intended is *the value of a binding referred to by symbol* `x` *in a particular environment*. So what exact provenance are we interested in?

A binding allows an object to exist in an environment and be utilised in expressions, but it also gives an object meaning.

Consider the following R code:

```
> x <- 1:5
> y <- x
```

The first expression creates an integer vector composed of the values 1 to 5, and establishes a binding between the symbol `x` and the newly created vector. The second expression assigns `x` to `y`; or speaking more strictly, it binds `y` to a copy of `x`'s vector. It's a trivial example, but understanding what happens in a case like this is critical to understanding how provenance is defined in this context.

The object referred to by `x` — in the strict sense, meaning the integer vector 1,2,3,4,5 — has not changed. All that has happened to it is that a clone of it has been created. To understand where `x` and `y` have come from, we need to know what has been bound to them in a particular environment.

Therefore, in order for provenance information to have meaning, it needs to be associated not with an *object*, but with a *binding*.

3 Implementation

The fundamental addition to CXXR required for recording provenance is the introduction of read and write *monitors*, which are attached to environments and get triggered when a binding in that environment is either read from or is created or overwritten.

3.1 Storing

Three C++ containers have been introduced to store various aspects of provenance information.

The `Provenance` class is central to storing provenance for a binding. It is composed of the *timestamp* of when the binding was created; the top-level *expression* that was being evaluated; the *symbol* that is bound; and references to the *parentage* and *children* of the binding.

Binding B1 is a **parent** of binding B2 (and conversely B2 is a **child** of B1) if binding B1 was read in the course of evaluating the top-level expression that gave rise to binding B2. Parentage is represented by the `Parentage` class, which inherits from the C++ Standard Template Library (STL) `std::vector` class, and stores pointers to `Provenance` objects.

A `ProvSet` of provenance objects is used to store references to `Provenance` objects. This collection is an `std::set`, and its members are ordered by time of creation. It is used primarily for storing references to children.

The class collaboration diagram for the relationship between new classes and existing CXXR classes is shown in Figure 2.

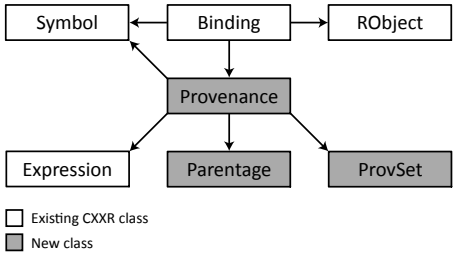

Fig. 2. Class collaboration diagram

3.2 Recording

The mechanism responsible for reading commands from the standard input, evaluating them, and printing the result is known as the Read-Evaluate-Print-Loop (REPL). Provenance for each REPL iteration is recorded according to the following algorithm: -

- Begin with the following empty collections:
  - *Seen* set: Provenance of bindings either read from or written to;
  - *Parentage* list: Provenance of bindings read from (in sequence).
- On read of binding to symbol `x`:
  - If `x` is not in the *Seen* set, add it to *Parentage* and *Seen*.
- On write of binding to symbol `y`:
  - Create a new *Provenance* object comprising:
    - A reference to the current top-level expression;
    - A reference to symbol `y`;

    - A reference to the current *Parentage*;
    - The current timestamp;
    - An empty set of children;
- Register the new Provenance object as a child of each of its parents, as recorded by the current Parentage list;
- Associate this *Provenance* object with the *Binding* of `y`;
- Add `y` to *Seen*.

3.3 Retrieval

In order for the user to be able to interrogate provenance information a couple of new R commands have been introduced. The `provenance(x)` function returns a list detailing the provenance of the current binding of `x`: the date and time of its creation, the expression immediately responsible for its current state, its symbol, and a list of both its parent and child Provenances.

The `pedigree(x)` function describes the full sequence of commands executed that led to the current binding of `x`. A full ancestry is collated by recursively looking at each Provenance's parentage starting from `x`; ordering all ancestors by time of binding creation; and printing their respective expressions, which are by definition relevant and their order chronological.

Listing 2 shows the result of these functions applied to one of the bindings resulting from evaluating the expressions shown in Listing 1. Firstly, the call to the `provenance` function shows information about `nine`, most interestingly that it has two parents: `square`, a function; and `three`, an integer vector of a single element. Secondly, the sequence of commands resulting in the current state of binding `nine` is detailed by the `pedigree()` function.

3.4 Issues

Loops. Although their use is not generally encouraged, loops are present in R. Consider the following loop to compute the sum of integers 1 to 5 and store this in object `x`:

```
> x <- 0              # Initialise x to zero
> for (n in 1:5)      # n = {1..5}
+   x <- x + n        # Increment x by n
```

There are two top-level expressions being evaluated here: The first initialises `x`, and the second (split across two lines, as indicated by the continuation prompt beginning with `+`) is a loop in which `n` iteratively takes the values from 1 to 5 and gets added to `x`. During each iteration of the loop, bindings `x` and `n` are both read and written.

Our initial implementation did not model this behaviour correctly because for each iteration of the loop, Provenances of bindings to `x` and `n` were added multiple times to the current parentage.

A more natural representation of this is, when a binding is read, to only add the associated Provenance to the current Parentage if it has not previously been

Listing 2. Example of provenance inspection functions

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

$symbol
nine

$timestamp
[1] "03/15/2010 03:34:27 PM.241776"

$parents
[1] "square" "three"

$children
NULL

> pedigree(nine)
three <- 1 + 2
square <- function(x) x*x
nine <- square(three)
```

written to or read from during the current top-level expression evaluation. This is the purpose of the *seen* set. In the case of the above loop, this strategy records only one parent for each `x` and `n`: the initial binding of `x` created by the first expression. This is illustrated by Listing 3.

Promises. The R language is capable of *lazy evaluation* of expressions, meaning they are not evaluated unless and until their value is required. The mechanism at the heart of lazy evaluation in R is a *promise*, which comprises an expression to be evaluated, and an environment in which the expression is to be evaluated. As in other programming languages, lazy evaluation prevents expressions from being evaluated unnecessarily in function bodies. R also installs the standard library functions into the base environment as promises that only load the full function definition when it is required. This practice is referred to as *lazy loading*.

When a promise is *forced*, that is to say its expression gets evaluated, its original binding may be succeeded by a new one. According to the algorithm outlined above, this would then get placed in the *seen* set and thus be excluded from appearing in the current parentage. This meant that during the first invocation of a lazily-loaded function, it could not appear as a parent to any object written. Subsequent invocations worked as desired because no additional binding creation precluded attribution of parentage. We handle this by not including in the *seen* set any binding created as a result of forcing a promise.

Source. R's `source(input)` function reads expressions from file `input` and evaluates each line in turn. This needs to be handled as a special case as these evaluations fall outside of the main Read-Evaluate-Print-Loop (REPL) mechanism.

Listing 3. Example illustrating how our refined implemention handles loops

```
> x<-0
> for (n in 1:5) x<-x+n
> provenance(x)
$command
for (n in 1:5) x <- x + n

$symbol
x

$timestamp
[1] "11/06/2009 11:39:11.230680"

$parents
[1] "x"
```

For the purposes of provenance collection, we view `source` as a *white box*, so that objects written are directly attributed to the precise statement within the file that resulted in their creation. This is opposed to a *black box* approach, which would simply describe a resulting object as having been created by a call to *source* with a particular input.

This more precisely describes the sequence of commands responsible for the current state of data, but not how that sequence came to be evaluated, since no record of the input file usage is made.

4 Conclusion

This work demonstrates how it is possible to introduce facilities for provenance awareness into an interactive, command-line driven statistical environment. CXXR has provided a number of challenges, the most novel of which are the necessity of attaching provenance to bindings rather than objects; facilities for lazy loading; and evaluating expressions from a file as opposed to the command line.

4.1 Further Work

Looking forward, one of our priorities is to enable *cross-session provenance tracking*. That is to say, when the user terminates a session, the objects are serialised along with relevant provenance information so the user is then able to restore a session with not only the object data, but also the pedigree of that data. This will require modifying the serialisation formats of CXXR, and draws into question how best the provenance information collected can be mapped to the Open Provenance Model [1].

CXXR is currently only aware of provenance in the global and base environments. Other environments, such as local environments in user defined functions,

and those associated with attached data frames, will eventually have their provenances tracked. This will present new challenges, in particular the user interface will need to provide an effective method of allowing the user to inspect provenance in different environments, and displaying the information in an intuitive way.

References

1. Moreau, L., Clifford, B., Freire, J., Gil, Y., Groth, P., Futrelle, J., Kwasnikowska, N., Miles, S., Missier, P., Myers, J., Simmhan, Y., Stephan, E., den Bussche, J.V.: The open provenance model — core specification (v1.1). Future Generation Computer Systems (December 2009)
2. The R Foundation: The R Project for Statistical Computing, `http://www.r-project.org`
3. Gentleman, R.: Reproducible research: A bioinformatics case study. Statistical Applications in Genetics and Molecular Biology 4(1), Article 2 (2005)
4. Knuth, D.E.: Literate programming. Comput. J. 27(2), 97–111 (1984)
5. Callahan, S.P., Freire, J., Scheidegger, C.E., Silva, C.T., Vo, H.T.: Towards provenance-enabling paraview, pp. 120–127 (2008)
6. Becker, R.A.: A brief history of S. Computational Statistics – Papers Collected on the Occasion of the 25th Conference on Statistical Computing at Schlosz Reisensburg, pp. 81–110 (1994)
7. Becker, R.A., Chambers, J.M.: Auditing of Data Analyses. SIAM Journal on Scientific and Statistical Computing 8, 747–760 (1988)
8. TIBCO Software Inc: Spotfire S+, `http://spotfire.tibco.com`
9. Runnalls, A.R.: CXXR project, `http://www.cs.kent.ac.uk/projects/cxxr`