
DOI

Link to record in KAR

http://kar.kent.ac.uk/50499/

Document Version

UNSPECIFIED
PROVENANCE-AWARE CXXR

A THESIS SUBMITTED TO

The University of Kent

IN THE SUBJECT OF COMPUTER SCIENCE

FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY.

By
Christopher Anthony Silles
January 2014
Abstract

A provenance-aware computer system is one that records information about the operations it performs on data to enable it to provide an account of the process that led to a particular item of data. These systems allow users to ask questions of data, such as “What was the sequence of steps involved in its creation?”, “What other items of data were used to create it?”, or “What items of data used it during their creation?.”

This work will present a study of how, and the extent to which the CXXR statistical programming software can be made aware of the provenance of the data on which it operates. CXXR is a variant of the R programming language and environment, which is an open source implementation of S. Interestingly S is notable for becoming an early pioneer of provenance-aware computing in 1988.

Examples of adapting software such as CXXR for provenance-awareness are few and far between, and the idiosyncrasies of an interpreter such as CXXR—moreover the R language itself—present interesting challenges to provenance-awareness: such as receiving input from a variety of sources and complex evaluation mechanisms. Herein presented are designs for capturing and querying provenance information in such an environment, along with serialisation facilities to preserve data together with its provenance so that they may be distributed and/or subsequently restored to a CXXR session. Also presented is a method for enabling this serialised provenance information to be interoperable with other provenance-aware software.

This work also looks at the movement towards making research reproducible, and considers that provenance-aware systems, and provenance-aware CXXR in particular, are well-positioned to further the goal of making computational research reproducible.
Acknowledgements

I owe a tremendous debt of gratitude to my supervisor, Andrew Runnalls, whose expertise, patience, and guidance has made this feasible for me; and whose generosity, kindness, and warmth has made it an absolute pleasure. I will forever be grateful.

I am grateful to my formal panel members, both past: Ursula Fuller and Tim Hopkins; and present: Andy King, Eerke Boiten, and Sally Fincher, for their searching questions and invaluable advice.

A special thank you to Prof. Roger Peng, of John Hopkins Bloomberg School of Public Health for his input.

Thanks, also, to my original officemates, Patrick and Radu, who initially made me feel so welcome and who subsequently became such good friends.

Thank you to all of my friends and family, for showing understanding during the more testing times and during my periods of self-imposed reclusion. In particular, I am incredibly grateful for the support my parents have always given me.

It would be absolutely uncharacteristic and entirely without precedent for me to thank Dr. Beulah “Snoofy” McKenzie without consummate levity; but I’ll do my best. To say that this—and so much more besides—would not have been possible for me without her friendship, support, and encouragement over the last decade would be an egregious understatement.

Finally, an enormous thank-you to my darling Jasmine\textsuperscript{1} for her love, support, spirit, and companionship (and not to mention patience and tolerance) throughout the last year. I feel so fortunate that our friendship has become something so special, and at such a wonderful time.

\textsuperscript{1}I hope you find my colour selection to be to your liking!
1.4.4 Benefits ....................................................... 23
1.4.5 Resistance ................................................... 24
1.4.6 Existing Approaches ....................................... 25
1.4.7 Approaches in R ............................................. 27
1.5 Motivation and Research Goals ............................. 28
1.5.1 Research Goals ............................................. 31
1.6 Overview of this Thesis ...................................... 33

2 CXXR ................................. 34
2.1 History ......................................................... 34
2.2 R ................................................................. 34
   2.2.1 Expressions ................................................ 35
   2.2.2 Objects ..................................................... 36
   2.2.3 Flow Control .............................................. 44
   2.2.4 Language .................................................... 46
   2.2.5 Packages ................................................... 47
   2.2.6 Bindings and Environments .............................. 47
2.3 CXXR ............................................................. 52
   2.3.1 Introduction ................................................ 52
   2.3.2 Progressive Development ................................ 52
   2.3.3 Layers ....................................................... 53
   2.3.4 Class Hierarchy .......................................... 53
   2.3.5 Memory Management ..................................... 55
   2.3.6 Other aspects of CXXR ................................. 60

3 Provenance in CXXR .................................... 61
3.1 Provenance Questions ........................................ 61
3.2 Design - Recording ......................................... 62
   3.2.1 Entity ....................................................... 62
   3.2.2 Activity ..................................................... 65
   3.2.3 Algorithm .................................................. 74
3.3 Design - Querying .......................................... 74
   3.3.1 In-interpreter Interface ................................ 74
3.4 Implementation .............................................. 75
   3.4.1 Monitors .................................................... 75
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.2</td>
<td>Containers</td>
<td>78</td>
</tr>
<tr>
<td>3.4.3</td>
<td>ProvenanceTracker</td>
<td>80</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Querying</td>
<td>84</td>
</tr>
<tr>
<td>3.5</td>
<td>Example</td>
<td>85</td>
</tr>
<tr>
<td>3.6</td>
<td>Evaluation</td>
<td>88</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Provenance Questions</td>
<td>88</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Performance</td>
<td>89</td>
</tr>
<tr>
<td>3.6.3</td>
<td>PROV Characterisation</td>
<td>90</td>
</tr>
<tr>
<td>3.6.4</td>
<td>Further Work</td>
<td>91</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Evaluation</td>
<td>115</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Illustrative Example</td>
<td>115</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Real-World Example</td>
<td>115</td>
</tr>
<tr>
<td>4.5</td>
<td>Provenance Interchange</td>
<td>118</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Design</td>
<td>118</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Algorithm</td>
<td>123</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Implementation</td>
<td>123</td>
</tr>
<tr>
<td>4.5.4</td>
<td>Evaluation</td>
<td>129</td>
</tr>
<tr>
<td>5.1</td>
<td>Expressions from Outside</td>
<td>132</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Introduction</td>
<td>133</td>
</tr>
</tbody>
</table>
7.2 Further Work ................................................................. 196
  7.2.1 Provenance-Aware CXXR ........................................... 196
  7.2.2 Reproducible Research .............................................. 198

Bibliography ............................................................................. 199

A Exploring R ............................................................................ 210
  A.1 Operators in R ................................................................. 210

B XML Serialization ................................................................... 212

C Air Quality Analysis .............................................................. 215
List of Figures

1.1 UML Use Case for exploratory data analysis in computational statistics package .................................................. 2
1.2 S AUDIT audit plot for example session, reproduced from [11]. ............... 5
1.3 Victoria Sponge Cake Provenance [76] ......................................................... 8
1.4 Key concepts of PROV illustrated by exemplifying John’s process of baking a cake, which was previously encountered as OPM exemplar. ............... 11
1.5 Reproducibility spectrum illustration, reproduced from [88]. ................. 20
1.6 UML activity diagram depicting an EDA of fine air particle pollution in the United States between 1999 and 2012 ............................. 30
1.7 UML Use Case depicting scenario following EDA in CXXR .................. 32

2.1 Bindings exist within environments and connect symbols to values. In this case, the symbol ‘three’ with a singleton integer vector ‘3’ ............. 48
2.2 Each environment is enclosed by another .............................................. 51
2.3 Layers within CXXR ....................................................................... 53
2.4 CXXR version 0.26 RObject class hierarchy ................................. 56

3.1 UML class diagram depicting attribute relationships of the binding class . . 63
3.2 UML class diagram depicting attributes of the Provenance class .......... 63
3.3 UML class diagram showing attribute relationships surrounding the Provenance class ......................................................... 64
3.4 Example of a Provenance hierarchy .................................................... 66
3.5 UML sequence digram illustrating the Read-Evaluate-Print-Loop mechanism 67
3.6 Activity diagram depicting occurrence of read operation on a binding .... 70
3.7 Activity diagram depicting occurrence of write operation on a binding ... 70
3.8 Activity diagram depicting when the read monitor is triggered ............ 71
3.9 Activity diagram depicting when the write monitor is triggered .......... 71
5.11 Sequence diagram of establishing the **makecounter** closure, and then evaluating it and binding the result (another closure) to **counter** .................. 171

5.12 Class diagram depicting interpreter state following creation of **makecounter** .......................... 172

5.13 Class diagram depicting interpreter state following creation of **makecounter**
and binding the result of its evaluation to **counter** ...................... 173

5.14 Sequence diagram of evaluating the **counter** closure ......................... 174

5.15 Sequence diagram showing frame monitoring .................................. 177

5.16 Class diagram showing new attributes and operations in class Frame .... 177

6.1 **example.pdf** generated using Sweave ........................................... 190

7.1 Example dependencies, depicting relationships between (a) bindings, and
(b) expressions ................................................................. 197
<table>
<thead>
<tr>
<th>Section</th>
<th>Algorithm Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Provenance-aware CXXR recording algorithm</td>
<td>74</td>
</tr>
<tr>
<td>3.2</td>
<td>Determine the ancestors of a (set of) binding state(s)</td>
<td>76</td>
</tr>
<tr>
<td>4.1</td>
<td>CXXR session serialisation algorithm</td>
<td>97</td>
</tr>
<tr>
<td>4.2</td>
<td>CXXR session deserialisation algorithm</td>
<td>98</td>
</tr>
<tr>
<td>4.3</td>
<td>Import bindings algorithm</td>
<td>98</td>
</tr>
<tr>
<td>4.4</td>
<td>Object Serialisation/Deserialisation algorithms</td>
<td>99</td>
</tr>
<tr>
<td>4.5</td>
<td>Serialise/Deserialise Environment</td>
<td>100</td>
</tr>
<tr>
<td>4.6</td>
<td>Serialise/Deserialise Symbol</td>
<td>101</td>
</tr>
<tr>
<td>4.7</td>
<td>Serialise/DeserialiseCachedString</td>
<td>102</td>
</tr>
<tr>
<td>4.8</td>
<td>The cxxr2prov algorithm</td>
<td>124</td>
</tr>
<tr>
<td>4.9</td>
<td>cxxr2prov: symbol_stop handler</td>
<td>125</td>
</tr>
<tr>
<td>4.10</td>
<td>cxxr2prov: provenance_stop handler</td>
<td>128</td>
</tr>
<tr>
<td>4.11</td>
<td>cxxr2prov: chronicle_stop handler</td>
<td>128</td>
</tr>
<tr>
<td>5.1</td>
<td>Refined granularity of provenance in source</td>
<td>135</td>
</tr>
<tr>
<td>5.2</td>
<td>An updated write monitor to allow overriding of inclusion in the seen set</td>
<td>152</td>
</tr>
<tr>
<td>5.3</td>
<td>The ProvenanceTracker flagXenogenesis operation</td>
<td>157</td>
</tr>
<tr>
<td>5.4</td>
<td>Functionality added to write monitor to, depending on ProvenanceTracker state, declare a Provenance xenogenous and preserve the present value of the binding</td>
<td>159</td>
</tr>
<tr>
<td>5.5</td>
<td>The Frame enableFrameMonitoring operation</td>
<td>177</td>
</tr>
<tr>
<td>5.6</td>
<td>The Frame disableFrameMonitoring operation</td>
<td>178</td>
</tr>
</tbody>
</table>
# List of Code Listings

1.1 Example S audit file [11] ......................................................... 4
1.2 Example PROV-N notation for baking a Victoria sponge cake ........ 11
1.3 List of data objects in CXXR workspace after execution of EDA use case . 31
2.1 Example R statements .............................................................. 35
2.2 Dynamic Typing example ......................................................... 36
2.3 Basic Vector Manipulation ......................................................... 37
2.4 Example of R’s list ................................................................. 39
2.5 Use of R function `search` to inspect the current search path and how this
   is affected by the attachment of a data frame ............................... 51
2.6 Trivial R Example ................................................................. 52
2.7 The `do_abs` function which implements R’s primitive function `abs` .... 57
2.8 Example usage of `GCStackRoot` in a function that returns a reversed copy
   of a `PairList` ........................................................................... 59
2.9 Traditional CR garbage collection mechanism example .................. 59
3.1 Trivial R Example (reprise) ...................................................... 62
3.2 Example loop in R ................................................................. 67
3.3 Expansion of loop given in Listing 3.2 ....................................... 68
3.4 The `findVar` function from `envir.cpp` ................................... 76
3.5 The `findBinding` function from `envir.cpp` ................................. 77
3.6 `Frame::Binding::value()` ......................................................... 77
3.7 The `Frame::Binding::assign` method ....................................... 78
3.8 class `CXXR::Provenance::CompTime` ...................................... 79
3.9 Methods for resetting in preparation for new REPL iteration .......... 81
3.10 `ProvenanceTracker::readMonitor` ........................................... 81
3.11 `ProvenanceTracker::writeMonitor` ........................................... 82
3.12 `Provenance::announceBirth()` ............................................. 83
3.13 `Provenance::announceDeath()` ............................................. 83
5.9 Provenance tracking pseudo-RNG 
5.10 Using edit to define a function sq. 
5.11 Extract from class Provenance header file showing relevant additions to 
   track provenance of xenogenous values 
5.12 Extract from source file of class Provenance, illustrating mutator method 
5.13 Extracts from header file for class ProvenanceTracker 
5.14 Extracts from source file for class ProvenanceTracker 
5.15 The write monitor ProvenanceTracker::CommandScope::writeMonitor 
   which identifies xenogenous bindings 
5.16 Extract of code from function do_pedigree which underlies the R function 
   pedigree 
5.17 Provenance interrogation using the pedigree() function. Illustrates the 
   way in which corresponding list elements describe a particular binding. 
5.18 Example of granularity issue when R code is defined within a code block. 
5.19 The ‘counter’ example 
5.20 The ‘counter’ example, augmented to assign results to variables 
5.21 Result of the ‘counter’ example illustrating the omission of provenance 
   tracking in local environments 
5.22 Additions to definition of class Frame to enable it to maintain a collection 
   of its instances 
5.23 The constructor of Frame modified to register this Frame instance with the 
   set of frames 
5.24 The destructor of Frame modified to deregister this Frame instance with the 
   set of frames 
5.25 The Frame::enableFrameMonitoring(bool) method 
5.26 Outline of Rf_ReplIteration function with control of frame monitoring 
5.27 Result of the ‘counter’ example illustrating the inclusion of provenance 
   tracking in local environments 
5.28 Illustration of side-effect of local environment provenance tracking 
6.1 Trivial R Code for cacher() example 
A.1 Extract a list of primitive functions from R-2.15.1 
A.2 Distinguish between builtin and special primitive functions 
Chapter 1

Introduction

1.1 Exploratory Data Analysis

CXXR is a variant of the R environment, which is an open source implementation of the S programming language for statistical analysis and visualisation. The development and usage of statistical computing packages, in particular S, was encouraged during the 1970s by a shift in approach to statistical analysis towards Exploratory Data Analysis (EDA), championed by John W. Tukey [114] whose belief it was that traditional use of statistics for only confirmatory data analysis (i.e. hypothesis testing) was insufficient:

“We often forget how science and engineering function. Ideas come from previous exploration more often than from lightning strokes. Important questions can demand the most careful planning for confirmatory analysis. Broad general inquiries are also important. Finding the question is often more important than finding the answer”

Pre-dating the widespread availability of computers, Tukey’s philosophy of EDA was originally proposed with the intention that its techniques employed physical means and would equip the analyst with ability to see in data not only what was being expressly searched for, but for whatever the data could reveal [115]:

“If we need a short suggestion of what exploratory data analysis is, I would suggest that 1. It is an attitude, AND 2. A flexibility, AND 3. Some graph paper (or transparencies, or both).

No catalog of techniques can convey a willingness to look for what can be seen, whether or not anticipated. Yet this is at the heart of exploratory data analysis. The graph paper—and transparencies—are there, not as a technique,
but rather as a recognition that the picture-examining eye is the best finder we have of the wholly unanticipated.”

Statistical software, especially S, brought with it an expansion to the data analyst’s “picture-examining eye”: the means for readily manipulating and visualising data in a manner far more efficient than with the graph paper originally suggested by Tukey.

Figure 1.1 depicts a UML use case of a typical exploratory data analysis conducted using software such as CXXR.

![Exploratory Data Analysis in CXXR](image)

Figure 1.1: UML Use Case for exploratory data analysis in computational statistics package

1.2 Provenance

The term provenance has, according to the Oxford English Dictionary, been used since the 18th century to mean “The fact of coming from some particular source or quarter; origin, derivation,” and since the late 19th century to refer to “The history of the ownership
of a work of art or an antique, used as a guide to authenticity or quality; a documented record of this.”

Today, provenance is a well-understood concept in many different areas including art, antiques and memorabilia. However, it is only relatively recently that the term has been used in the context of computing, in particular in its application to data, the provenance of which we consider to be “the process that led to that piece of data” [44]. This type of information is becoming not only of increased use but also of necessity as computer systems have taken on significant roles in many disciplines where it is critical to provide an evidential trail of how data has been managed throughout its lifetime. While these software computer systems are able to cope with producing, collating and manipulating vast quantities of data, for instance by means of complex modelling and simulation, facilities to provide provenance information to accompany this data have not always kept pace.

When a system is able to determine, record and interrogate the provenance of the data on which it operates, we consider it to be provenance-aware. There is currently significant interest in creating provenance-aware computer systems for use in areas as diverse as e-science; medical physics (CT, MRI, fMRI, PET etc.); proteomics; finance; and weather monitoring.

1.2.1 Definition and Characterisation

The definition of provenance given in the introduction may be generalised as “Provenance of a resource is a record that describes entities and processes involved in producing and delivering or otherwise influencing that resource” [41].

Provenance information naturally qualifies as metadata—some data that describes data; although this is not a symmetric relationship—not all metadata is provenance.

One of the initial motivators for research in the area of provenance-aware computing was Romeu, who contended that information about data (i.e. metadata) is critical for distinguishing good data from bad [95]. Further to this, Goble summarises the main uses of provenance information as follows [43]:

- **Quality.** Using evidence of the data’s derivation to support its integrity.

- **Auditing.** Prove that a data underwent a particular process.

- **Reproducibility.** By having access to information about derivation of data, processes can be repeated for purposes of establishing accuracy; updating result data
with respect to source data; and allowing validation by way of reproducing results.

- **Ownership.** Provenance can be used to attribute ownership, or establish liability.

### 1.2.2 Early Provenance-Aware Computing

The role assumed by “New S” in early provenance-aware computing is one of particular pertinence in the context of this work because “New S” is a distant relation of CXXR (as Chapter 2 will describe).

“New S” was the sequel to the ‘S’ statistical language and environment, which was released in 1988 and sported a new feature entitled **S AUDIT** [11]. While an S session was in operation a record was maintained of the expressions that had been entered by the user and then evaluated by the interpreter; as well as objects read from and written to during the evaluation of an expression. The accompanying S AUDIT program was able to process this record and allow the user to ask questions about it, discovering details about the session which previously would not have been known. 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.

Listing 1.1 shows an example audit file generated by S. The lines beginning with # indicate an action; such as beginning a session, reading (‘get’) objects, and writing (‘put’) objects. The ‘get’ and ‘put’ lines show the path of the object’s data file, a timestamp of when the action took place, and its data mode which was used for maintaining accurate type information while data was serialised.

One advantage of this method was that lines beginning with # were treated as comments by the S interpreter and thus ignored, therefore the S audit file could be used directly as a source file to re-execute the statements it contained.

**Listing 1.1: Example S audit file [11]**

```
# New session: Time: 542034997; Version: "S Tue Mar 3 10:14:20 EST 1987"
> m<-matrix(read("brain.body"),byrow=T,ncol=2)
> # put "/usr/rab/.Data/m" 542035057 "structure"
> brain<-m[,1]
> # get "/usr/rab/.Data/m" 542035057 "any"
> # put "/usr/rab/.Data/brain" 542035066 "real"
> body<-m[,2]
```
A more intriguing feature of S AUDIT was its *audit plot* facility, which plotted a directed-acyclic graph with statements as nodes arranged in a circle (anti-clockwise in order of occurrence); and edges representing an object written by one statement, later being read by another. The audit plot for the example session given above is shown in Figure 1.2.

![Audit plot example](image)

Figure 1.2: S AUDIT audit plot for example session, reproduced from [11].

Therefore New S became the first *provenance-aware* software application, long before the phrase had been coined.

### 1.2.3 Modern Provenance-Aware Computing

Provenance-aware computer systems have been developed in different disciplines in slightly different ways: a sort of parallel evolution. Although this is primarily concerned with the type of data whose provenance is to be recorded, it also has implications for how data is stored and queried. At a very coarse classification, distinct approaches to creating provenance-aware systems can be seen in fields such as the semantic web; workflows;
service-oriented architectures; databases and data warehouses; automatic provenance collection at the file system.

The primary forum for discussion of provenance-aware computing has been the International Provenance and Annotation Workshop (IPAW), which began in 2006 \cite{77} and has since been held bi-annually, at which numerous papers are presented detailing different systems for collecting, querying, characterising and understanding provenance information.

One of the outcomes of the first IPAW was the need to gain greater understanding of the similarities and differences of these systems, and thus the First Provenance Challenge was established \cite{78}. Participating teams were given the same pre-defined workflow for aggregating fMRI images and were required to execute this workflow in their own systems. Following this, each team answered the same set of questions relating to the output: such as finding the process that led to the output of a particular image, and all invocations of a particular process.

One of the main conclusions drawn from the First Provenance Challenge was that comparing each system’s recorded provenance information was impossible. To ensure that the evolution of the various systems did not diverge, a second Provenance Challenge was established, with a focus on interoperability of provenance data. Participating teams performed the same workflow as in the first challenge, but then made the results (i.e. all recorded provenance information) publicly available. Each system then operated on provenance data collected by a different system as if it had been produced by itself. This allowed insight to be gained into how well data in one system can be translated for use in another, as well as how different systems can aggregate provenance over a number of individual processes. As a result of the experiment conducted in the Second Provenance Challenge, it became clear that a common model for representing provenance information was necessary, and there was agreement on how provenance should be represented. In particular it became clear that provenance was naturally representable by a directed acyclic graph (DAG).

1.2.4 Vocabularies, Ontologies and Representations

The need to represent provenance information in diverse application domains led to parallel creation of provenance models, some of which are general-purpose, others are less domain-agnostic. Some examples of these include: Dublin Core Terms \cite{117}, Provenir ontology \cite{100}, Provenance Vocabulary \cite{48}, Proof Markup Language \cite{30}, PREMIS (PRE-
servation Metadata: Implementation Strategies) [4], SWAN Provenance Ontology [26], WOT Schema [15], Semantic Web Publishing Vocabulary [19], and Changeset Vocabulary [116].

The discussion process that followed the first provenance challenge in 2006 made clear the need for a standardised model to represent provenance information, and work started towards reaching this objective. The result of this was the Open Provenance Model (OPM) which was published in 2008 [75] (later refined in 2009 [76]), which defined the following objectives:

- Exchange of provenance information between systems;
- Allow development of tools for provenance data;
- Precise definition of a technology-agnostic model;
- Define how provenance graphs may be interpreted.

### 1.2.5 Open Provenance Model

The OPM conceptually represents provenance information as a directed graph, whose nodes represent entities of the following types:

- **Artifact.** An immutable piece of state. Either a physical object, or item of data.
- **Process.** Actions performed on artifacts, resulting in new artifacts.
- **Agent.** A catalyst for initiating and controlling processes.

The graph’s edges represent the following dependencies or causal relationships:

- **used.** A process used an artifact;
- **wasGeneratedBy.** An artifact was generated by a process;
- **wasControlledBy.** A process was controlled by an agent;
- **wasTriggeredBy.** One process triggering another;
- **wasDerivedFrom.** One artifact being derived from another.

**Accounts** allow for an OPM graph to incorporate alternative explanations for a given execution, perhaps at different levels of detail such that one account is said to refine another. **Roles** in the OPM are annotations on edges to provide a context for dependencies between entities.
Figure 1.3 shows the provenance of the task of creating a Victoria Sponge Cake. Central to this is the Bake process, which was controlled by John, in the role of cook. The Bake process used four artifacts in various different roles: 100g Butter, 2 Eggs, 100g Sugar, and 100g Flour. The result of this process is the generation of the Cake artifact.

1.2.6 W3C Provenance Incubator and Working Group

In response to the increased interest of provenance in the field of the semantic web the W3C established the Provenance Incubator Group in 2008 with a charter to “provide a state-of-the-art understanding and develop a roadmap in the area of provenance and possible recommendations for standardization efforts.”

This effort concluded in December 2010 with the publication of its final report [41]. The products of this group are summarised as follows:

- Development of a shared working definition of provenance;
- Deduce a set of key dimensions for provenance—grouped into content, management and use;
- Collate use cases: three of which were honed into flagship scenarios;
- Developed provenance requirements of these scenarios. Initially in terms of user requirements, from which were derived technical requirements;
- Mappings for existing provenance vocabularies [99], using OPM as a reference model expressed in terms of SKOS (Simple Knowledge Organization System) [73];
• A state-of-the-art report, which identified the need for standards for publishing and accessing provenance information;

• Provenance in web architecture;

• Roadmap and recommendations, which included a proposed charter for a working group on provenance, and a series of proposed deliverables. Including conceptual model of provenance and a related formal model; how provenance should be accessed and queried; and XML serialisation.

The charter proposed by the Incubator Group was enacted by the W3C in April 2011 with the formation of the Provenance Working Group, whose deliverables were set out to satisfy the recommendations of the Provenance Incubator Group. Over the course of its lifetime, the Provenance Working Group defined a family of specifications, known collectively as PROV, which provide a definition for how provenance information can be represented and interchanged. Its approach is largely based on that of OPM but offers extensions in several directions. The group closed in June 2013, and its contribution marks a significant event in the field of provenance. The PROV family of specifications comprises the following W3C recommendations [42]:

• PROV-DM [79]. The PROV conceptual data model for provenance;

• PROV-CONSTRAINTS [24] Constraints that apply to define validity of a PROV instance: uniqueness constraints, event ordering constraints, impossibility constraints, and type constraints;

• PROV-O [64]. The PROV ontology. An OWL2 ontology representation of PROV-DM allowing the mapping of PROV to RDF;

• PROV-N [80]. A notation for representing PROV-DM provenance intended for human consumption.

As well as the above Recommendations, a number of Notes were published:

• Introductory documents PROV-OVERVIEW and PROV-PRIMER;

• An XML schema PROV-XML for representing PROV-DM instances, which will be discussed in Chapter 4;

• PROV-AQ for access and query of provenance;
The PROV model allows for the description of the provenance record of anything, known as an entity, which may be physical, digital, conceptual or otherwise. Examples of entities would be a printed document, a PDF document, a \LaTeX\ markup file, a data set, or a chart.

An entity is said to be generated by an activity, which may be any process, either virtual or of the real-world. In the course of performing its function, an activity may make use of entities. These interrelations between entities and activities are described in the past tense, such as in this instance ‘used’ and ‘wasGeneratedBy’.

An agent assumes the role of being responsible (to a degree) for the execution of some activity. An agent again may be physical in its nature such as a person or object, or virtual, such as a software program. An entity may be attributed to an agent who was responsible for its creation, and an activity can be associated with an agent.

If a user ran the \LaTeX\ typesetting program \texttt{pdflatex} to transform a \LaTeX\ file into a PDF document, this constituted an activity, with which the user and the program were associated as agents; the input \LaTeX\ file is an entity which is used by the process; and the resulting PDF document is an entity which was generated by that process.

One important feature of the PROV-O ontology is its extensibility. One such extension of PROV-O is the PAV (“Provenance, Authoring and Versioning”) ontology [27], which intends to support the authoring and versioning information of web resources.

The example of John baking a cake that was represented in Figure 1.3 to illustrate the use of OPM has been depicted in PROV terms as shown in Figure 1.4, which utilises the PROV diagram and colour specification.

The PROV-N specification describes a notation for representing PROV-DM instances in a human-readable form. It uses functional-style predicates such as \texttt{entity} and \texttt{activity} followed by a list of terms. The cake-baking example might be expressed as shown in Listing 1.2.
Figure 1.4: Key concepts of PROV illustrated by exemplifying John’s process of baking a cake, which was previously encountered as OPM exemplar.

Listing 1.2: Example PROV-N notation for baking a Victoria sponge cake

```plaintext
document

default <http://victoriaspongeexample.org/>

entity(100gbutter)
entity(100gflour)
entity(100gsugar)
entity(2eggs)
entity(cake)

agent(john, [ prov:type='prov:Person', name="John" ])
// The above illustrates the optional use of a list of attributes
// to further describe the agent with identifier 'john'

activity(bake)

used(bake, 100gbutter)
used(bake, 100gflour)
used(bake, 100gsugar)
used(bake, 2eggs)

wasGeneratedBy(cake, bake)
wasAssociatedWith(bake, john)

endDocument
```
1.3 Provenance-Aware Software

1.3.1 Classification

It is possible to consider that there are—broadly-speaking—two approaches to provenance-aware computer systems: that which requires the user manually to enter provenance information; and that which looks after this process on the user’s behalf without any explicit user interaction.

The exclusive use of the former approach is not favourable as it is obviously little more than ad-hoc annotation volunteered by the user and lacks the systemic involvement to offer the required degree of structure or guarantee of accuracy; however, it does have use in supplementing automatically recorded provenance information with user-defined annotations.

The latter category can be further divided two ways: those items of software that are capable (either naturally, or having been retrospectively adapted) of self-determining which provenance to record, recording it, and allowing the user some means of interrogating it; and those which enable the automatically collection of provenance from existing applications, by means of observation.

In the case of software that has not had the benefit of any human involvement to specify what provenance is to be recorded, one of the principal issues is that of granularity. Granularity refers to the extent to which a recorded process has been subdivided, and systems range from coarse-grained with fewer, larger processes; to fine-grained with more, smaller processes. The term granularity is similarly applied to the data items on which a process is performed; for example an XML file within a filesystem, an individual XML record, a database tuple, or an individual byte within memory.

Granularity is a particular issue for systems that automatically record provenance empirically (i.e. by process observation) as they are naturally oblivious to the context of the information about which they are recording provenance, and therefore have difficulty in determining an appropriate level of granularity. These systems typically operate at a very low-level (e.g. often that of system calls) and so naturally provide a fine-grained record. In practicality terms for these, granularity can be considered as the “mismatch between the operating system’s observation of a sequence of system calls and the scientific user’s desire to record provenance”.

A potentially useful intermediate position between software that has been adapted to collect provenance, and automated observation, is software that intrinsically records its provenance by instrumenting these facilities by way of automatically adapting its source
code.

1.3.2 System-Level Provenance

One of the more successful attempts at recording provenance of existing systems has been the Provenance Aware Storage System (PASS) [81]. PASS is implemented using a modified Linux kernel and Berkeley DB back-end storage, and because of this it is able to operate at a very low level, recording provenance information by intercepting system calls relating to input and output to files and pipes. The result of this is incredibly fine-grained provenance information. For instance, during the execution of a program launched from a command line, PASS records the path of the executable; path(s) of any input files; all environment variables; kernel version and loaded modules.

We can illustrate how PASS would record the execution of the example S session given in Listing 1.1, in which case the property of S storing all of its data objects in individual files becomes particularly useful. It would be possible to store the code given in Listing 1.1 in a file, and provide it as input to S running on a PASS volume. PASS would record the invocation of the S executable; the opening for reading of the input source file; and the opening of S object files whenever an object was read from or written to; as well as a potential plethora of environment variables and other information. From the records generated by PASS of files opened and closed for reading and writing, it would be straightforward to identify which objects were created and read during a session; however, there would be no indication of how those objects were used, and whether for instance one object were used in the creation of another object. In order to identify this, the source file (and its interpretation) would be required. It therefore follows that the source file is required to annotate the process in order to provide a satisfactory level of granularity; were the source file not present to accompany the record created by PASS (or similar system), for instance if the statements were instead entered manually by the user, there would be no record of precisely what operations were performed on the data objects.

1.3.3 Versioning File-Systems

Provenance-aware systems have been based at the level of file systems, either keeping a historic record of files by preserving each version—a ‘versioning file system’—or as a useful point to intercept file operations performed by certain ‘monitored’ processes, and thus being able to attribute those files opened for reading as being used for ‘input’, and those opened for writing as ‘output’ [103].
A versioning file-system retains copies of previous versions of files. ElephantFS for instance employed a copy-on-write approach, which instead of overwriting a file with a new version, created a copy of it [101]. This concept was improved upon by VersionFS, which was able to work on top of any underlying file system, as well as providing more control over storage and retention policies [82]. Such a system operates at a necessarily coarse granularity—that of a file—so while it may able to account implicitly for the evolution of data by examination of the history of a file, it maintains no record of the specific processes involved.

1.3.4 Adapting Software

One predominant work on adapting existing software to become provenance-aware is Para-view, an open source application for data analysis and visualisation, to which has been integrated VisTrails.

VisTrails allows the recording of provenance information pertaining to data exploration and workflows by maintaining a record of the data provenance to track workflow (or in VisTrails parlance, a ‘dataflow’) evolution and recording provenance information in a structured way—as defined by an XML schema—allowing it to be queried and mined and dataflows compared [17].

More specifically, this is accomplished by separating the notions of a dataflow specification from its instances, so while a dataflow instance comprises a record of the sequence of steps performed in the generation of a particular visualisation, which would be sufficient to regenerate the visualisation, this may be ‘abstracted’ to a more general dataflow specification that may be used as a template for visualisations with different parameters.

VisTrails provides user interfaces for building and interacting with dataflows, allowing the user to explore and return to previous versions.

It is VisTrails’ approach of change-based provenance, whereby only changes to state are recorded, and not the state itself, that enables its integration to applications, and to its particular case-study Paraview [18].

The foundation of this approach to adapting an application for provenance-awareness is in an application’s use of the model-view-controller paradigm of graphical user interface, which stipulates the decoupling of the user interface (view) from the application’s logic and processing (model), by use of an event handling process known as a controller. Because all user actions within the application pass through the controller, it can be modified to intercept and record these actions so they may be replayed later. In practice, these
actions are those typically represented by the application’s undo-redo action stack, and a particularly useful side-effect of ‘hijacking’ the undo-redo stack in this fashion is that it utilises this as a form of pre-defined, application-specific granularity—there is no need to define what constitutes a useful action from an outside perspective: it has already been determined by the developer of the application.

When an action is captured, it is passed to a Provenance Explorer process that runs alongside the target application in its own thread, via a defined Communications API, and is recorded along with both automatic and manual metadata. Automatic metadata includes the time and date of the action, the user who created it, and an assigned unique identifier for the action, as well as a reference to its preceding action. Manual metadata in the form of annotations or tags may be defined in the Provenance Explorer.

The Provenance Explorer graphically displays the different application states as a tree, and allows the user to return to any of these states by replaying the sequence of recorded actions. It does this by controlling the target application via the Communications API—clearing the application state and executing the series of actions.

The principal advantage of this method is that it utilises the domain-specific granularity as defined by the application’s undo-redo feature; however, it is thereby limited to applications that feature undo-redo capabilities, and allow for application actions to be captured and indeed controlled in this way.

1.3.5 Automatically Adapting Source Code

SourceSource investigates the approach of automatically adapting source code to enable it to record its own process documentation [74].

The process documentation created by SourceSource is defined in terms of the OPM. Statements are represented by OPM processes and variables are represented by OPM artifacts. Because a variable may take on any number of values during the execution of the program, each new assignment to a variable will result in the creation of a new artifact, and a mapping is maintained by the recording library to reflect the most recent artifact of each variable. When a variable is used in a statement, a causal relation (OPM used) is created between the process (statement) and the most recent artifact associated with the variable.

SourceSource recognises the requirement for control over granularity, and enables the user a degree of configuration to determine whether or not a source component—such as a Java class file—is adapted to record fine-grained provenance, or remains (relatively)
opaque, in which case a record of it being invoked is made, but not the processes it performs, and so records coarse-grained provenance. Calls to third-party libraries, or any other components for which the source is not available for adaptation, are treated as Opaque components, for which adapters can be created to document the execution more extensively. Varying levels of granularity can be accommodated by OPM’s account identifier, and SourceSource always records the coarse-grained account, and if the relevant source component has been adapted, the fine-grained account will also be recorded. These two accounts can be linked by an OPM refinement relationship, in which the fine-grained account refines the coarse-grained.

In order for the resultant provenance information to be queried and understood by the user, SourceSource attributes identifiers to statement executions, variables and program executions. Each individual statement execution is identified by its scope identifiers (e.g. package/class/method names), a unique statement identifier and the iteration of its execution. Each use of a variable is identified by the statement execution using it, its scope and its name. Program executions are identified by a generated execution identifier. These identifiers are used to annotate the processes and artifacts in the OPM graph so that it may be queried by these attributes.

The process of adapting source code for use with SourceSource comprises three stages: explicate, identify and augment, which can be achieved automatically using a tool for rewriting source code based on rules. The first stage, explicate is simply preparing the source code for processing, in particular making explicit any implicit code blocks by the addition of braces. The second stage, identify, is to introduce identifiers to each statement with a unique name, comprising its scope identifiers as well as its position within the method. In the third stage, augment, each occurrence of a process or artifact (i.e. statement or variable) is augmented by a recording statement.

The approach outlined in SourceSource attempts automatically to adapt source code for provenance-awareness, thus eliminating (or perhaps at least reducing) the need for manually adapting software to become provenance-aware. This has only been tested as a proof of concept in a limited capacity, albeit one which was not entirely trivial, as it was taken from the Third Provenance Challenge, in submission to which several approaches to provenance-tracking were made. However, it remains to be seen how scalable this approach is and whether or not it would be applicable to an application such as an interpreter, whose function is to handle ad-hoc processes and arbitrary data.
1.4 Reproducible Research

1.4.1 Introduction

The journal publication process has long been considered to be the fundamental method of dissemination of scientific findings and scholarly discussion. It has recently been contended that “whilst statistical practice has evolved to encompass more computation and larger and more complex datasets and models, the primary vehicle for delivery has remained the static, printed page” [40]. The fundamental scientific method remains the same: a scientific claim must be reproducible. It is this “culture of replication” that weeds out spurious claims and ensures integrity. As advances in computational processing power increase researchers’ ability to collect and process increasingly large sets of data with increasingly complex models and simulations, so too does the challenge of ensuring reproducibility.

The significance of this requirement can be seen against a modern landscape which provides demand for scientific claims to be ‘startling’; not just by headline writers for newspapers creating digests of scientific findings for public consumption—even journals exhibit a publication bias towards papers that discover something new [121], so it is more critical than ever that scientific analyses are accessible and their repeatability is validated.

Results from observational studies have in particular been drawn into question [121, 57] as an excessive reliance on statistical significance by publishers can lead researchers to manipulate findings to exploit this bias. A study in 2005 of 49 highly-cited studies that made observational claims showed that 14 either failed entirely to replicate or the magnitude of the claim could not be repeated [56]. A study by the same author into 18 articles in the journal Nature showed only two could be fully reproduced; six could be partially reproduced or reproduced with some discrepancy; and 10 could not be reproduced at all [58].

It is not only observational studies that are susceptible to lack of reproducibility; randomised, controlled trials (RCTs) may be considered the “king of study-designs”, but even these may require repetition to provide definitive conclusions. Even independent research teams tasked with answering the same research question and armed with the same data will not necessarily arrive at the same answer, due to the subjective approach to the research question; the differences in evidence selection and analytic methods; and editing of the report [63].

Errors in science have not escaped public attention and have been the subject of media scrutiny, even recently making appearances in mainstream media publications. The New York Times reported that the American Society for Microbiology’s Infection and
Immunity journal was forced to retract [113] six papers by one author who—it transpired—had manipulated results [84]. The case dubbed the “Duke University Scandal” [54]—the narrative of which provides a suitably illustrative exemplar of the benefits of reproducible research, or perhaps more precisely: the cost of irreproducible research—was reported in both The Economist [109] and the New York Times [85]. With an even greater focus on the need for reproducibility in science, The Economist in October 2013 published an article under the descriptive heading “Unreliable Research: Trouble at the Lab” [110], which serves as an excellent primer to reproducible research and what efforts are being made to further reproducibility.

A study in the field of personalised medicine was conducted at Duke University in 2006. The aim was to establish whether a patient’s genetic make-up could be used to predict their responsiveness to various chemotherapy treatments; the research correlated drug sensitivity data with results from bio-markers identified by micro-arrays, and made predictions against patient samples. According to the researchers’ publication in Nature [92], these predictions were successful, and so provided a proof of concept that signalled a break-through in the avoidance of chemotherapeutic regime failure. The findings were so significant that the M.D.Anderson Cancer Center appointed its bio-statisticians Keith Baggerly and Kevin Coombes to investigate. When supplied with the original data and code used, Baggerly and Coombes encountered issues and identified ‘sensitive’ cell lines attributed as ‘resistant’ and vice-versa; unintelligible data; mislabelled data and descriptions of irreproducible analytical steps. When the original researchers refused to acknowledge these findings, Baggerly and Coombes wrote to the (by now numerous) journals who had published the Duke results. A response from the Journal of Clinical Oncology in 2008 was given as:

“A focus on these errors as presented by Baggerly et al is misleading since it suggests they are a contributing factor in the supposed lack of reproducibility, which is not the case. Most importantly, the claim that they cannot reproduce the results of the study, when in fact they did not even try to do so, is an egregious flaw in their commentary. To reproduce means to repeat, using the same methods of analysis as reported. It does not mean to attempt to achieve the same goal of the study but with different methods.”

The original research was used as the foundation of a clinical trial, which eventually involved 109 patients. Despite an internal inquiry by Duke University into the practices of its researchers, in response to criticism by Baggerly and Coombes, the University passed
the work as being valid. The situation was only resolved when other institutes were unable to reproduce the original results; a copy of the report of the internal Duke investigation was acquired; and it was discovered that the principal author and key researcher had falsified elements of his curriculum vitae. It transpired they had provided to the inquiry data that had been modified in an attempt to cover their tracks. Ultimately four journal papers were retracted. Baggerly wrote a brief summary of the incident in Nature and took the opportunity to make a call to the scientific community to “Disclose all data in publications”:

To counter this problem, journals should demand that authors submit sufficient detail for the independent assessment of their paper’s conclusions. The quality of scientific output will benefit from setting these standards. As a community, we owe it to patients and to the public to do what we can to ensure the validity of the research we publish.

Whilst most journals encourage attribution of supplementary materials to articles, providing data and code in this fashion tends not to be completely satisfactory because there is no consistent way to package these and convey the exact set of steps involved.

With particular respect to reproducing the results of computing code; it has been stated that natural language descriptions of code lead to ambiguity, and errors even exist in ‘perfect’ descriptions. Even journals that employ policies that insist on code release via formal descriptions or pseudocode cannot guarantee that this achieves the objective of imparting the specific functionality of the code without ambiguity.

This is the ultimate objective of the reproducible research movement.

1.4.2 Terminology

The term reproducible research was coined in 1992 by geophysicist Jon Claerbout, who went on to describe the initial motivation of recreating one’s own analyses: “In the mid-1980s, we realized that our laboratory’s researchers often had difficulty reproducing their own computations without considerable agony.” This perspective was distilled by Buckheit and Donohu to what has been referred to as Claerbout’s Principle:

An article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software development environment and the complete set of instructions which generated the figures.
A further definition has been given as “a piece of reproducible research is an article that provides readers with all the materials that are needed to produce the same results as described in the publication” [51]. A view corroborated by Gentleman and Temple Lang¹: “By reproducible research, we mean research papers with accompanying software tools that allow the reader to directly reproduce the results and employ the computational methods that are presented in the research paper.” [40].

Efforts have been made to define a distinction between reproducibility and replication, although there is no consensus about this. One such distinction has been given by Peng [90], who describes reproducibility as the availability of original data and code, which may be subjected to independent verification as well as alternative or extended analyses; while replication as the process of independent investigators using independent data, analytical methods, laboratories and instruments. Replication is described as the higher standard to which all scientific evidence should be held, but it is proposed that reproducibility should be a ‘minimum standard’ that should be met.

Drummond describes this relationship in opposite terms—i.e. with ‘replicability’ as the simple repetition of original analyses, and reproducibility as the independent conducting of analyses in attempt to repeat the outcome, without repeating the exact method [34].

Both Davison [31] and Peng [88] identify a reproducibility spectrum, as shown in Figure 1.5. At the ‘not reproducible’ end is just the publication; and at the other ‘gold standard’ end is ‘full replication’. In between these two poles are—in respective order—the publication with code; publication with code and data; and publication with linked and executable code and data.

![Reproducibility Spectrum](image)

Figure 1.5: Reproducibility spectrum illustration, reproduced from [88].

The intrinsic meanings of the terms replication and reproduction are not sufficiently different to allow them to denote distinct concepts. They will herein be treated synonymously.

¹Gentleman and Temple Lang are, incidentally, contributors to the R project. In fact, Robert Gentleman is a founder of the project, along with Ross Ihaka—in part, hence the name ‘R’, which also is a play on ‘S’. 
Mesirov introduced some formal—if not widely adopted—terms for referring to components of a *Reproducible Research System* (RRS) [71]. The first is a *Reproducible Research Environment*, which provides the necessary tools to conduct analysis and automatically track the provenance of data, analyses and results, and package them (or persistent pointers to them) for redistribution. Secondly, a *Reproducible Research Publisher* (RRP) is typesetting system such as Word or \LaTeX{} that provides facilities for incorporating artifacts from the RRE.

### 1.4.3 Journal Interest, Policy and Practice

IEEE’s *Computing in Science and Engineering* (CiSE) has to date featured two special issues on reproducible research [36, 107] featuring guest editorials and articles detailing approaches taken towards furthering reproducible research in individual disciplines.

Elsevier’s *Journal of Computational Science* in 2010 launched a competition inviting researchers to propose methods for ‘executable papers’, to “improve how data intensive research is represented in a scholarly journal” [35]. Its winner, *Collage* [83], was announced at a workshop [39] held at *International Conference on Computer Science 2011*. The Association for Computing Machinery’s Special Interest Group on Management of Data (SIGMOD) has since 2008 offered the opportunity to assess the papers accepted to its annual conference in terms of repeatability, as well as going further to “workability”: how well parameters in computations may be changed [68].

*Annals of Internal Medicine* stated in 2007 its desire for authors to include in their articles submitted for consideration a *reproducible research statement* [62] and in 2013 conducted a study along with Yale Open Data Access, whereby two research teams were asked to answer the same research question, using the same dataset and produce papers detailing their findings. The resulting differences serve to “illustrate the value of evidence synthesis, data sharing, peer review, editing and reproducible research in helping us get closer to the truth” [63].

Following a letter to the editor of the *Biometrical Journal* identifying numerical problems of a Markov Chain Monte Carlo analysis in a paper the journal published, *Biometrical Journal* conducted a study into the extent to which the articles published in one of its volumes were reproducible [51]. Of the articles that included either simulations and/or illustrative examples, 32% provided access to data; 15% provided access to code; and 11% access to both code and data. The *Biometrical Journal* described this finding as “not too bad” and acknowledged the room for improvement, and went on to state its aims to “in-
crease the quality, usefulness and scientific impact of Biometrical Journal articles through reproducibility”. To achieve this, the journal has appointed an ‘Associate Editor for Reproducible Research’ who is responsible for checking reproducibility of an article using its supporting information and providing assistance to authors to make their publications reproducible.

The British Medical Journal [45] in 2009 stated its intention to encourage authors to make available raw research data, and include in their submissions a ‘data sharing statement’ to explain which additional data is available and how it is accessed. However, the focus here is on the sharing of data (“raw numbers, analyses, facts, ideas and images that do not make it into published articles”), and there is little mention of the processing of the data to derive ‘facts, ideas and images’.

Stodden et al. conducted a study in 2013 to evaluate the data sharing policies for a referent set of 170 journals for 2011 and 2012 [108]. Of the journals studied in 2012, 38% had a policy governing data sharing; 22% had a code policy; and 66% a policy on supplementary materials. In comparison to the previous year, this represents increases of 16%, 30% and 7% in respective areas. Usefully this study analyses a five-point spectrum of requirements of the individual policies—from ‘no mention’ up to a ‘requirement as condition of publication’. At the maximum end of this scale, only 11% of journals require sharing of data as a prerequisite of publication; only 3.5% require code sharing; and 3.5% require sharing of supplementary materials (actually down from 4.7% the previous year).

The journal Biostatistics implemented a policy of encouraging authors of accepted papers to make their work reproducible by others [87]. The journal also offers a “reproducibility review” to its authors, in which the associate editor for reproducibility runs the submitted code and data to verify the results in the manuscript. A series of ‘kite-marks’ are offered to appear on the first page of an article that have included data (“D”), code (“C”), and pass the reproducibility review (“R”). After two years of this policy, 21 of 125 published articles have been attributed kite-marks; including five “R”s [88].

Sonnenburg et al. have stated the machine learning discipline’s requirement for a more stringent reproducibility policy in the Journal of Machine Learning Research [105]; although this has been—quite specifically—countered by Drummond [34].

It is worth noting that policy adoption does not always translate to an adoption of practice—and that not only scientific disciplines require reproducible research. The Journal of Money, Credit and Banking since 1996 has required its authors to submit their code and data to an archive; however, a 2006 study revealed that of 150 papers published
over the past decade, only 15 could be independently reproduced using the materials provided [69].

1.4.4 Benefits

Although making processes reproducible is typically for the benefit of allowing third parties to use existing work on their own, it may also benefit the original author [102]. Details of how artifacts were derived may be lost or forgotten over time, and adequate reproducibility provisions may be implemented to prevent this.

Transparency and sharing of code can be seen as beneficial for reasons other than direct reproducibility. The UK Engineering and Physical Sciences Research Council Software Sustainability Institute promotes unity between scientists and software developers in an effort to encourage the cross-discipline sharing of common code, to prevent “researchers wasting time reinventing the wheel for each new application” [70].

The British Medical Journal suggests the “potential benefits of sharing data include quicker scientific discovery and learning, better understanding of research methods and results, more transparency about the quality of research, and greater ability to confirm or refute research through replication” [45].

The 2007 case of NASA’s Surface Temperature Analysis software [7, 70] illustrates how irreproducibility can engender public scepticism, and also how transparency alone is not enough to change this perception. In this instance, NASA’s Goddard Institute for Space Studies (GISS) made claims regarding climate change that were based on its GISS Surface Temperature Analysis (GISTEMP) software, which analysed data of global surface temperature since 1880. These claims were widely reported, and almost as widely criticised, owing in no small part to the lack of accompanying software, despite this being quite usual practice. In response to criticism for publishing findings without the software used to derive them, GISS released the software source code. At this point, the focus of the criticism shifted to the nature of the software itself—it was largely written in FORTRAN, a language with which these days few are familiar; it was poorly organised and had no build system or test framework; it was written in several languages and required very specific versions of compilers and it was not portable between big- and little-endian architectures. These difficulties surrounding the usability of the released code served only to further fuel theories of conspiracy regarding the global warming claims made by the software and published by NASA. Therefore it is not a trivial assertion that simple transparency is of itself valuable; the value lies in—and what was required in this instance
was—reproducibility. However, the benefit of this transparency was in having the code in question available in the public domain so that it may be examined. It is this transparency that has enabled the code’s reimplementation in Python and the foundation of the Clear Climate Code project, which has not only been able to verify the original code’s results, but also consequently to allay fears of a conspiracy [6].

One other noted benefit of this exercise has been the identification of defects in the GISTEMP software—truncation of floating point digits during parsing of input data; weak loop termination conditions; inflexibility of internal array representations to changing parameters.

### 1.4.5 Resistance

The principal barriers to adoption of reproducible research methods are suggested by Peng to be “code no longer [being] available”; “the lack of a deeply engrained culture that simply requires reproducibility for all scientific claims”, and “the lack of an integrated infrastructure for distributing reproducible research to others” [88]. The first point here can be addressed by enabling software to be provenance-aware, although it is noted that this is not feasible for closed-source, or proprietary software.

Donoho presents numerous potential barriers as well as responses [33]. These are divided into two categories; the first being “Knee-Jerk Objections”, which include: “Reproducibility takes time and effort”; “No one else does it, so I won’t get any credit for it”; “Strangers will use your code to compete with you”; “My computing environment is too complicated to publish like this”. The second category is “Thoughtful Objections” and include “Reproducibility undermines the creation of intellectual capital”; “Reproducibility destroys time-honored motivations for collaboration”; “Reproducibility floods journals with marginal science”; and “True reproducibility means reproducibility from first principles”.

The British Medical Journal, in its statement of intent to encourage its authors to share data, describes the sharing of clinical research data as “a new and challenging concept” for most medical journals [45]. One general observation made is that nature of clinical data in respect of data protection, and the (strict) requirement of patient data anonymisation.

Drummond describes the lower standard of ‘replicability’—i.e. repetition of analyses without change—as the “poor cousin” of the higher standard of independent reproducibility, and that it is “not worth having”, “would cause a great deal of wasted effort” and “at best, would serve as little more than a policing tool, preventing outright fraud” [34]. This
CHAPTER 1. INTRODUCTION

is very much in the minority of expressed opinion and it could be contended that this perspective is naïve in its assumption that research does not require policing, or acknowledge that by facilitating ‘replication’ of findings to as great an extent as is possible, credibility and trust in those findings may be engendered. Retractions from journals increased 10-fold in the decade 2000-2010 [119], with 44% of those attributable to ‘misconduct’ (comprising fabrication or falsification 11%; self-plagiarism 17%; plagiarism 16%) and 11% because of irreproducible results.

1.4.6 Existing Approaches

The ReDoc system [102] was developed by Schwab, Karrenbach and Claerbout at Stanford University in the 1990s, and has been used in their department to provide reproducible material accompaniment to journal papers, PhD theses as well as books. ReDoc is described as “a simple software filing system for authors that lets readers easily reproduce computational results using standardized rules and commands”. The system is based on GNU Make [106], a standard Unix utility, typically used for creating software build systems. A file that is under the control of make is known as a ‘target’ file. For each target file there is defined a list of source files (or dependencies) and a sequence of commands (or rules) that should be executed to create that target file. When make is executed, it will determine whether a target file is out-of-date with respect to its dependencies, and if it is, the rules will be executed. This process is also applied to a target file’s dependencies, and so ensures everything is up-to-date according to the defined dependencies. ReDoc’s reader interface leverages this functionality, and augments it with a framework of make rules. As well as the article or paper and source code, a ReDoc is composed of project-specific makefiles, a set of universal make rules, and naming conventions for files. ReDoc classifies files as follows: fundamental files are those which are not generated by a process, such as source files, data sets and makefiles; result files are the targets ultimately to be reproduced; and intermediate files that are generated during the process of producing result files from fundamental files. ReDoc provides the following commands: burn for removing easily reproducible result files; build for reproducing result files; view for launching an appropriate viewing program for a result file; and clean for removing intermediate files. The pre-defined naming conventions allow universal rules to handle files appropriately, for instance, files with the suffix .0 or base name junk are removed by the clean command. This system represents a trivial step for users of make, as only the naming conventions need to be adhered to.
Authors of the previously-mentioned VisTrails system (page 14) identify its \textit{provenance-awareness} as a “step toward simplifying the creation and review of reproducible results” [37], and thus it is rare amongst the reproducible research literature in its direct relation of the terms \textit{reproducible research} and \textit{provenance}.

One of the ways in which this goes beyond simple provenance-awareness and towards enabling reproducible research is Vistrail’s support for creating “provenance-rich” papers, which allow the direct embedding of VisTrails workflows into \LaTeX and Microsoft Word and Powerpoint, as described in [8], and also for support of ‘executable papers’ with whose data and visualisations a user may interact [61].

Mesirov describes an expressly user-friendly reproducible research system, comprising the reproducible research environment of the \textit{GenePattern} computational genomics environment, and Microsoft Word as the related publisher [71]. \textit{GenePattern}, like \textit{Paraview}, handles workflows and automatically tracks the versions of ‘pipelines’—connections between modules in the workflow; tracks the users’ analytic session; regenerates corresponding pipelines; and packages them for redistribution. An add-in to Microsoft Word enables connection to a \textit{GenePattern} server—often running on the local machine—and offers the author the ability to embed text, tables and figures derived from previously executed pipelines, which may then be persistently stored within the document. The reader may then interrogate an embedded artifact, such as a figure, directly within the document, and see the pipeline responsible for its creation. The reader is also able to connect to a \textit{GenePattern} server, to execute the embedded pipeline—with or without modified parameters—to update the embedded elements, which can all then be saved along with updated provenance.

The BURRITO Linux-based system aims to automate the ‘tedious’ aspects of everyday research activities: managing file versions, logging parameters and experimental outputs, writing notes, and organising notes [47]. It achieves this by combining a versioning file system (NILFS [60]) to preserve historic versions of files; tracers for capturing operating system-level execution provenance (similar to PASS) and GUI interactions; plugins for collecting provenance in specific applications; and a variety of utilities for interrogating and disseminating the recorded events, such as an \textit{Activity Feed} similar to that of Twitter.

ReproZip [25] and CDE [46] are similar in their approach to providing a means for both recording provenance of an execution, and packaging it for reproduction in a different environment. The former utilises the SystemTap [3] library for intercepting system calls, while the latter uses the \texttt{ptrace} system call. ReproZip cites Burrito as an influence and
claims some advantages over CDE including greater control over collected provenance and
customisation of the reproducible package; better caching of provenance in a database;
and greater focus on usability for authors and reviewers. Both are, however, limited to
operating on Linux operating systems. It is possible to overcome this limitation by using
a Virtual Machine to emulate a Linux system on which the package can be executed;
although this necessitates additional configuration and will incur a performance cost.

Sumatra [31] is a Python library “on which may be built interfaces adapted to individu-
al scientists’ workflows”; or, a method of constructing—what Mesirow might call—
Reproducible Research Environments. This is essentially a system for the automatic col-
lection of provenance, even though it’s not explicitly stated as such—to which end, two
approaches are described: “taking a digital copy of the entire environment using a hard-
ware virtualisation approach”; and “capturing and storing metadata about the code and
environment that lets it be recreated later”. In the latter approach, the execution en-
vironment (such as hardware platform, architecture, operating system, software versions,
command line parameters etc.) is recorded; the version of code used—according to a ver-
sion number attributed by a version control system such as Git or Mercurial; the input
data—again identified by VCS version number; and output data.

Reproducible research has been facilitated by means of the concept of Literate Pro-
gramming described by Knuth [59] in which programming code is interwoven with an
explanation of its logic expressed in natural language. Two preprocessing routines exist
to distil out components for consumption by computer and human—tangle extracts from
the input file a compilable source file; and weave which creates formatted documentation
for (human) viewing.

One system that utilises literate programming for the purpose of reproducible research
is Lepton [67].

It has been proposed that virtual machines could be used to facilitate packaging of
reproducible research [13]. At the point at which a researcher has completed an analysis,
the state of the (virtual) machine is saved. This is then packaged in an archive with
auxiliary files (such as the paper, its text and figures) which may then be distributed.

1.4.7 Approaches in R

There exists in R a package entitled Sweave, which facilitates the literate programming
of R code [65]. This was extended in 2011 [66] to the $R^2$ platform, which proposes using
R packages (see Section 2.2.5) as containers for reproducible research papers, and R’s
built-in **CMD check** facility to evaluate the commands necessary to reproduce and validate the paper and its results. There is also a proposed $R^2$ server for storage and subsequent retrieval of packaged papers.

Building on the *Sweave* approach, Gentleman and Temple Lang propose *Compendium* as a “container for one or more dynamic documents and the different elements needed when processing them, such as code and data” [40]. This approach describes a dynamic document as an ordered composition of *code chunks*—sequences of commands, e.g. R, the execution of which are needed to produce output—and *text chunks*—the (natural language) descriptions of the problems, code, results and interpretations. As in literate programming, one is intended for consumption by computer and the other by the reader. Auxiliary software may also be included in a compendium; for example user defined functions that will not appear (directly) in the document, but notwithstanding are required for reproduction.

Peng describes the R package *cacher* as an alternative approach to that of literate programming [86]—one more akin to that described by Claerbout [102]. The objective of this package is to “provide a means by which an author can assemble the code and data used in a statistical analysis into a single package that can be distributed easily to others.” It does this without intertwining the code with a human-readable document as would be the case in literate programming and so does not require the user to be familiar with the necessary mark-up languages employed for this purpose, such as \LaTeX. When an analysis is evaluated under the supervision of *cacher*, it establishes a directory within the file system and saves to it cached versions of all resultant objects. This directory can be made accessible via any protocol (e.g. HTTP/FTP) with which R is compatible, so that any user who wishes to interact with the code and data from that analysis may invoke the function *clonecache* with the relevant URI as an argument to obtain the repository of data and code used. A further advantage of this method is that when an analysis is repeated only those expressions that have not previously been evaluated (and therefore whose resultant objects may be different) will be re-evaluated, while those that have remained the same can load the values for objects directly from the cache.

### 1.5 Motivation and Research Goals

The principal motivation for this work is to overcome the lack of transparency of data provenance that faces the analyst conducting, or perhaps returning to a long-ago conducted, exploratory data analysis in the CXXR statistical environment. As this chapter has
described, the value of data is increased by accompanying it with its provenance; furthermore, in this particular context of statistical analysis, establishing the provenance of an analysis has clear value when it comes to being able to reproduce that analysis.

The use of a computational statistical platform for conducting exploratory data analysis presents inherent challenges for the analyst, whose typical workflow will employ techniques for presenting data in various ways—e.g. tables, summaries, and plots (stem-and-leaf, boxplots, scatterplots etc.)—in attempts to allow the data to reveal its underlying structure.

After conducting an analysis—either immediately or after loading a previously saved session—the user is faced with significant degree of opacity in the data—there is no metadata. The result of this is that the user is only able to see what data objects were created in the session and display (either textually or graphically as appropriate) their values. There is no provision for asking questions about how any of the data objects came into existence, so it is therefore not feasible to piece together a coherent picture of the analysis.

To exemplify this effect we will look at an example analysis into changes in fine air particle pollution (PM) in the United States between 1999 and 2012 using data available on the website of the US government’s Environmental Protection Agency. This analysis was written in R by Roger Peng [89] and will be encountered throughout this work (the full analysis is presented in Appendix C.) The workflow of this analysis is presented in Figure 1.6 as a UML activity diagram to describe its various stages.
CHAPTER 1. INTRODUCTION

Figure 1.6: UML activity diagram depicting an EDA of fine air particle pollution in the United States between 1999 and 2012

In an environment such as CXXR, the use of such techniques can quickly cause a cluttered and confusing workspace: each instance of representing a dataset is likely to require the creation of new data objects, or altered versions of existing data objects. The analysis shown in Figure 1.6 was conducted in a CXXR session, and Listing 1.3 uses the R function `ls()` to list the data objects—the names enclosed in quotation marks—that were created during the execution of this analysis. The particulars of this example will
be described more thoroughly in due course, but for now serves to illustrate the sort of problem faced by the data analyst in this scenario. This analysis consists of 50 expressions, and results in the creation of 23 data objects, many of which have arbitrary, cryptic names such as \texttt{x1} and \texttt{tab}, from which very little information about the analysis can be gleaned.

Listing 1.3: List of data objects in CXXR workspace after execution of EDA use case

```
> ls()
[1] "both" "both.county" "both.id" "cnames"
[5] "cnt0" "cnt1" "dates" "dates0"
[9] "dates1" "missing.months" "negative" "pm0"
[13] "pm0sub" "pm1" "pm1sub" "rng"
[17] "site0" "site1" "tab" "x0"
[21] "x0sub" "x1" "x1sub"
```

More generally, Figure 1.7 depicts a use case of what might follow that described in Figure 1.1, in which the original user Alice wishes to reflect on her analysis, or a new user Bob loads Alice’s session and wishes to pick up where Alice left off. In this case, the users wish to elicit details about the original analysis by asking questions of the data objects to gain an understanding of how the analysis evolved and in what ways the data was used.

At present it is not possible for these such questions to be answered in CXXR. As a consequence the user is prevented from easily (or perhaps at all) being able to re-use the original analysis—its value is, at best, diminished; at worst, extinguished—and there is no certainty in being able to replicate the analysis.

1.5.1 Research Goals

Motivated by the problems encountered in such scenarios, the principal objective of this thesis is:

\textit{To understand how and the extent to which CXXR can be adapted to record and interrogate the provenance of the data on which it operates.}

To accomplish this the following goals will be undertaken:

1. Understand the role of provenance in the context of typical use cases of statistical computing for exploratory data analysis (already discussed in this chapter)

2. Define, from 1, \textbf{provenance questions} that a provenance-aware CXXR would be necessarily capable of answering (Chapter 3)
3. Design and implement facilities for enabling provenance questions to be answered by CXXR (Chapter 3 in the general case; Chapter 5 for special cases)

4. Design and implement facilities for provenance questions to be answered in a cross-session capacity (Chapter 4)

5. Understand how provenance recorded in CXXR can be used in other systems (Chapter 4)

6. Evaluate the efficacy of provenance-awareness in CXXR with respect to the provenance questions defined (Chapters 3, 4, 5)

A secondary objective of this thesis is as follows:

What can a Provenance-Aware CXXR contribute to reproducible research in R?
1.6 Overview of this Thesis

The remainder of this thesis is structured as follows:

**Chapter 2** Introduces the software that is the subject of this study, **CXXR**, and its development as a variant of **R**, which may be considered a spiritual descendent of **S**. R is a language and environment for statistical analysis and graphics. This chapter also primes the reader in the language of R, so that the examples encountered throughout may be understood.

**Chapter 3** Details the **provenance questions** to which provenance-aware CXXR should offer answers. This chapter describes the software design to achieve this and its implementation to CXXR, and illustrates its effect with examples, and discusses how we can understand this view of provenance in terms of standard provenance models.

**Chapter 4** Describes how cross-session provenance tracking is achieved by reimplementing the interpreter’s serialisation process to include provenance information.

**Chapter 5** Details a series of ‘special cases’ of provenance tracking in CXXR, for which the facilities introduced in Chapter 3 do not adequately cater, and special design decisions are required in order for accurate representation of provenance.

**Chapter 6** The first of two conclusion chapters which specifically evaluates this work with respect to the field of reproducible research.

**Chapter 7** Draws more general conclusions from the study, critically evaluates the understanding outcomes, and also considers how further work may extend the knowledge.
Chapter 2

CXXR

CXXR is a variant of the R environment, which is an open source implementation of S. This chapter gives a brief examination of how CXXR has evolved from R and in turn S, and an introduction to the R language and environment. Individual aspects of the R language and CXXR version of the environment will be discussed in detail in later chapters as they become more relevant.

2.1 History

S is a language and interactive environment for statistical computing, graphics, and exploratory data analysis [10]. 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 its name.

The follow-up to S, ‘New S’ was released in 1988 (accompanied by the “Blue Book” [9]) and it sported a new feature entitled S AUDIT [11], and in so doing it became one of the first provenance-aware applications.

While S continued life as a commercial product called S-PLUS, retailed by TIBCO Software Inc. [111] until at least 2010, the language, library and environment have been reimplemented as part of the open-source R project [93] that was started as a research project by Ross Ihaka and Robert Gentleman in the 1990s [53].

2.2 R

The R distribution comprises an interpreter, written for the most part in C with a splash of Fortran, and packages for common functionality, which are written in a combination of C, Fortran and R itself. R is maintained by a team of core developers that is currently
20-strong and includes original S developer John Chambers, and enjoys a large and active userbase working in areas as diverse as retail strategy, genetics, education, pharmacology, proteomics, and data and text mining.

This section serves as an introduction to the R language and environment.

### 2.2.1 Expressions

Listing 2.1: Example R statements

```r
1 > 1 + 2
2 [1] 3
3 > three <- 1 + 2
```

The R environment is **interactive**; that is to say that during a session it is operated by a user, who provides **expressions** that are **evaluated** and the result of that evaluation is then displayed. An R session begins at the **prompt**, which by default is indicated by the string "> " , and it is here that expressions are entered by the user for evaluation. Expressions in R are ubiquitous, so to identify those that are entered here at the prompt by the user, they are referred to as **top-level** expressions, which correctly implies that there are other levels of expression. Although graphical user interfaces to R exist on many common platforms (Windows, Linux, Mac OS X etc.), these perform little function when it comes to performing any data analysis or programming, but serve a useful purpose of facilitating easy control over the session by its user, such as customising its look and feel, presenting visualisations in windows, managing packages and data sources, saving or loading the workspace, and providing access to help documentation.

Listing 2.1 shows an extract of a session in which two top-level expressions are evaluated. The first expression simply outputs the result of the integer addition, while the second attributes the result to the name **three** whose value will remain until such a time when **three** is either deleted, or given another value. The latter expression is an example of an **assignment** operation (and can be referred to as either an assignment expression or an assignment operation equally validly), as denoted by the operator "<-"\(^1\). Also appearing in the second top-level expression as the right hand side operand of the assignment is addition, as traditionally denoted by the operator "+", and like the assignment operator and indeed all other binary operators in R, it is essentially only ‘syntactic sugar’ for an

\(^1\)The more traditional assignment operator = is also available at (only) the top-level in R; however, the ‘arrow’ is generally preferred in any case.
underlying binary function, whose arguments become the left and right hand side oper-

ands. In the second top-level expression, the addition (1+2) forms an expression of its

own, within the top-level expression.

However, this is a slight simplification. What goes on “under the hood” in the above

example (Listing 2.1) is a little more complex. When 1 + 2 is evaluated in each of the two

statements it gives rise to an object representing the result of the calculation—3—which

in the first statement is discarded\(^2\) after its value is printed, but the second expression is

an assignment: we can think of this as “assigning 3 to three”.

The high-level view that most R users tend to take (as do most R introductory texts)

would suggest here that three is an ‘object’ or ‘variable’ and conceptually this is perfectly

sufficient in most contexts, but we need to treat this issue with finer accuracy. What

actually happens is more like so: The symbol three becomes bound to the value rep-

resenting 3 that resulted from evaluating the right hand side operand (consisting of the

expression 1 + 2) of the assignment operator. It would therefore be more accurate to say

that three refers to an object whose value is 3, or more generally: a binding connects

a symbol to an object.

### 2.2.2 Objects

Creating and using objects is one of the concepts central to R—it is practically a mantra of

S and R that “everything is an object” [21]. An object belongs to a class, which describes

its underlying data type and therefore how it may be used by functions. Common classes

within R include the following vector classes: logical, numeric, integer, character, list, complex, raw, expression, and the non-vector class function. Less common classes

are call, name and environment.

**Dynamic Typing**

A binding in an R session is not declared in such a way that it has a fixed (or static) data

type. Instead, its type is determined dynamically, and it therefore assumes the class of

the object to which it is bound. The function class(x) returns a string representation of

binding x’s class. Examples of dynamic typing in R are shown in Listing 2.2.

---

\(^2\)Strictly-speaking, it is bound to the symbol .Last.value but this is outside our scope for the moment.
CHAPTER 2. CXXR

Vectors

A vector in R is an homogeneous indexable array of data, which may be composed of elements of any vector class. Vectors in R are pervasive to such an extent that even single values of objects of a vector class (e.g. TRUE) are represented by a vector of just one element—a singleton vector. Therefore in the initial example (Listing 2.1), 1 + 2 is an expression which adds two singleton integer vectors, to produce another singleton integer vector—3. Scalar objects do not exist in R.

Listing 2.3: Basic Vector Manipulation

```
> x <- "a string"
> class(x)
[1] "character"
> x <- 1
> class(x)
[1] "numeric"
> x <- TRUE
> class(x)
[1] "logical"

> x <- seq(from=1, length.out=10, by=2)
> x
[1]  1  3  5  7  9 11 13 15 17 19
> x[c(1,5,9)]
[1]  1  9 17
> x[c(TRUE,FALSE,TRUE,FALSE,TRUE,FALSE,TRUE,FALSE,TRUE,FALSE)]
[1]  1  5  9 13 17
> x * 2
[1]  2  6 10 14 18 22 26 30 34 38
> x[x > 10]
[1] 11 13 15 17 19
```

Listing 2.3 shows some basic vector operations. The first statement creates a vector x using the seq() function to generate a vector of 10 elements, starting from 1, with increment 2, which results in the odd numbers between 0 and 20. The second statement simply causes this vector to be printed.
In general, a vector \( v \)'s elements can be addressed using the syntax \( v[i] \), where \( i \) is a numeric (i.e. integer) vector or a logical vector. When \( i \) is a numeric vector, elements of \( v \) corresponding to the elements of \( i \) are addressed, as shown in the third statement. When \( i \) is a logical vector, elements of \( v \) that correspond by position to those elements in \( i \) that are \textbf{TRUE} are addressed (as shown in the fourth statement). The fifth and sixth statements illustrate how vector operations are performed \textit{element-wise}, for example, in the case of multiplying two vectors \( a \) and \( b \), the resulting vector is composed of \( a[1] \times b[1] \), \( a[2] \times b[2] \) and so on. If \( a \) and \( b \) are of unequal length, then the shorter is repeated, as illustrated in the fifth statement, where the \textbf{vector 2} is repeated for each element of \( x \) (i.e. \texttt{length(x)} times), and the sixth statement where the logical vector \( c(\text{TRUE,FALSE}) \) extracts just the odd-numbered elements of \( x \), making the cumbersome fourth expression more succinct.

These two techniques may be combined to select elements of a vector based on a boolean condition, such as that of the seventh statement, which indexes \( x \) by a logical vector that is created by applying the logical operation \( > 10 \) to each element of \( x \).

\textbf{Functions}

Function definitions take the following form:

\begin{verbatim}
function (formal arguments) body
\end{verbatim}

Typically a function body is enclosed by braces and split over multiple lines for readability, and functions are typically bound to symbols.

Formal arguments are defined as a comma-separated list of names, each of which may be given a \textit{default expression} that gets evaluated in the absence of an actual argument. For instance, the formal arguments for the function \texttt{seq} are defined as follows:

\begin{verbatim}
seq(from = 1, to = 1, by = ((to - from)/(length.out - 1)),
length.out = NULL, along.with = NULL, ...)
\end{verbatim}

During evaluation of a function, the \textit{actual} arguments, which are also comma-separated and enclosed in parentheses after the name of the function, are matched to the formal parameters by \textit{position} or \textit{name}, or a combination of the two. For example, a call to \texttt{seq} to generate an integer vector of the numbers 1 to 10 could be expressed equally validly as \texttt{seq(from=1,to=10)}, \texttt{seq(to=10,from=1)}, or \texttt{seq(1,to=1)}, or \texttt{seq(1,10)}.

Some built-in functions are evaluated not for their result, but for their \texttt{side-effect(s)} such as the function \texttt{plot()}, which does not return a result, but instead uses a graphics device to show a plot.
Operators and Primitive Functions

R uses C-style mathematical, relational and logical operators: + - * / == != < > <= >= & !, along with operators for exponentials (^), integer division (\%/%) and modulo division (%%). R has other language-specific operators such as assignment <-, non-local assignment <<-, collection subscripting [], collection named-component access $, package-qualification ::, and a number of others that will generally not be encountered here.

The use of binary operators with infix syntax, such as a + b is only ‘syntactic sugar’ for a call to a function in prefix style, in which the operator is placed in backticks, and followed by its operands given as arguments: `+`(a,b).

R’s operators are one example of its set of functions known as primitives, which do not correspond to the previously-encountered function type (that has formal arguments and a body). Their implementation is internal to R which holds a mapping between each primitive ‘name’ and a C routine to be executed. There are two internal types of primitives: builtin, whose arguments are first evaluated before being passed to the internal routine, and special, whose arguments are passed to the internal routine as unevaluated as expressions.

The majority of primitive functions in R are of the builtin variety as these operations, which include all mathematical operations such as addition `+`, multiplication `*`, and logical negation `!`, will necessarily require their operands to be evaluated in order to perform their function; while a special primitive such as `if` allows flexibility in determining which of its operands (if any) are evaluated and when.

Lists

The R list datatype is a heterogeneous collection of (optionally named) objects. Lists are typically returned as the result of calls to functions, but may be constructed manually using the list() function, an example of which is shown in Listing 2.4: a list newlist is constructed with a function element named oneToN; the list is then printed; the function newlist$oneToN is invoked with argument 7; another list mylist is then constructed, consisting of two elements: the result of newlist$oneToN(10) named intvector, and a string; the entire list is printed; and then one of the named elements is extracted.

---

3Primitive functions reside in the base package. An investigation into primitives can be found in Appendix A.1.
Classes and Object Orientation

Object-Orientated Programming (OOP) allows for an object’s attributes and behaviour to be encapsulated and defined by a class, possibly in relation to other classes to provide inheritance. When called, a method determines its behaviour based on the class of the object on which it operates. R natively has three systems to support object-oriented programming, two of which it has inherited from versions of its predecessor language, S: S3 informal ‘classes’ present a simple and ad-hoc system to provide rudimentary object-orientation based on single method dispatch; however, their use for new code is now considered deprecated, although a significant legacy S3 codebase is still in use (including in R itself); and S4 classes, which work similarly to S3 classes, but have formal class definitions (and are hence known as formal classes) and allow multiple dispatch. Finally, Reference Classes have come along far more recently and unlike S3 and S4 classes, these operate on a different principle of message-passing object-orientation to better capture the semantics of other object-orientated languages such as Java or C++.

S3 classes were introduced to the version of S developed around 1990, and first described in the accompanying “White Book” [22]. Under this mechanism, objects are given an attribute class that is used to describe its class and from what other classes it inherits.

---

4 There are a number of add-on packages to support OOP available for R, including R.oo
The \texttt{class} attribute contains one or more strings representing names of classes, the first of which is class of the object, and the remainder are classes from which the object inherits. The class of an object can be set using the \texttt{class} function.

\begin{verbatim}
> x <- "The Sport of Rugby"
> class(x) <- "rugby"
\end{verbatim}

A \textbf{generic method} is one which examines the class of its argument(s) and determines to which method a call should be dispatched. Under \texttt{S3}, this is determined purely on the basis of the class name(s) involved—if \texttt{m} is a generic method and \texttt{obj} is an object with class \texttt{c}, then in a call to \texttt{m(obj)} will result in attempt to dispatch a call to \texttt{m.c(obj)}. It should be noted that full stop (\texttt{.}) is a valid character in identifiers and it does not carry any member-extraction connotations, unlike C, C++ or Java. If no such method exists, then methods with names corresponding to the other classes from which \texttt{obj} inherits are sought, defaulting to \texttt{m.default(obj)} if no specialised method pertaining to one of \texttt{obj}’s classes exists. This is called single-dispatch because an S3 generic method may only accept one parameter.

\begin{verbatim}
> ball <- function(x) UseMethod("ball", x)
> ball.football <- function(x) "sphere"
> ball.rugby <- function(x) "prolate spheroid"
> ball.default <- function(x) "generic ball shape"
> ball(x)
[1] "prolate spheroid"
\end{verbatim}

\texttt{S3} classes offer an easy way to provide limited \textbf{polymorphism}, but this is limited by the single dispatch restriction and lack of robust structure and type-safety. In particular, there is no definition for what types the elements of an \texttt{S3} class must have.

Nowadays the creation of new \texttt{S3} classes is generally discouraged \cite[p. 362]{21} in favour of \texttt{S4} classes, although they are still widely-used and there is still appetite from the user-base.

\texttt{S4} classes arrived in the 1998 version of S, which is described in the “Green Book” \cite{20}. This mechanism works loosely on the same principle of method dispatch as \texttt{S3}, but it uses a formal definition of a class known as its \textbf{representation}, which specifies what attributes the class has and from what (if any) other classes it inherits.

A class is created with the \texttt{setClass} function, which establishes its \textbf{representation}, which is a collection of \texttt{named} and \texttt{typed} variables known as \texttt{slots}. 

\textit{CHAPTER 2. CXXR}
The `setMethod` function is used to establish which generic methods may be called on an object of a given class. In contrast with S3, S4 generic methods support multiple-dispatch, so they may choose methods based on the class of any number of arguments instead of just one.

```
> setMethod("show", "BallSport", function(object) {
  + cat("The sport of", object@officialname, "is played with a",
  +       paste0(object@ball, "-shaped"), "ball\n")
  + })
[1] "show"
> rugby
The sport of Rugby Union is played with a prolate spheroid-shaped ball
```

A new generic method (such as one to be used as an accessor function for a class slot) is first created with the `setGeneric` function, before being established for a particular class using `setMethod`.

```
> setGeneric("getBall", function(object) standardGeneric("getBall"))
[1] "getBall"
> setMethod("getBall", "BallSport", function(object) object@ball)
[1] "getBall"
> sapply(c(rugby, football), getBall)
[1] "prolate spheroid" "sphere"
```

S4 classes offer a more formalised approach to object-orientation in R; however, like S3 classes, their methods are not encapsulated as part of their definition, they are simply functions. They are also subject to R’s copy-on-change semantics: when a slot’s value is altered, the change does not occur ‘in-place’, instead it forces the construction of a whole new object. While this is a valid characteristic for R’s heritage as a functional language and makes it easy to reason about programs with S4 classes, it can be inefficient where large objects and numerous modifications are concerned.

R version 2.12 released in October 2010 contained for the first time the reference class (RC) facility for defining classes whose objects behave in a fashion similar to those of OOP languages like Java and C++. Reference classes are based on S4 classes, and the
RC vernacular and interface is broadly similar to that of S4, but there are a couple of significant differences which set RC apart from its ancestor. Firstly, they present tighter encapsulation of methods: unlike S4 classes whose methods are functions declared suitable for joining in with the method dispatch arrangement, RC methods actually belong to objects. Secondly, RC objects are mutable: they contain state that can be modified without the object being subject to R’s copy-on-change policy.

A reference class is created using the \texttt{setRefClass} function to specify its name as a string, and lists comprising fields and methods. The fields list’s elements each represent a field of the class: the name of the field and its value is a string representing the field’s type. Similarly for the methods list, whose element names represent the names of the methods, and whose values are the functions.

\begin{verbatim}
> BallSport <- setRefClass("BallSport",
  +   fields = list(officialname = "character", ball = "character"),
  +   methods = list(getBall = function() ball)
  + )
\end{verbatim}

It is conventional to use the object returned by \texttt{setRefClass} in the creation of new objects of a reference class. The operator \$ is used to denote access to a member (either field or method) of an RC object, for instance \texttt{object$method(args)} denotes invoking a method method with arguments args on an RC object object, while \texttt{object$field} accesses a field.

\begin{verbatim}
> rugby <- BallSport$new(officialname = "Rugby Union", ball = "prolate spheroid")
> rugby

Reference class object of class "BallSport"
Field "officialname":
[1] "Rugby Union"
Field "ball":
[1] "prolate spheroid"
> rugby$getBall()
[1] "prolate spheroid"
\end{verbatim}

As their name implies, RC objects display reference semantics, meaning they are not copied-on-modification, and they are passed by reference.

\begin{verbatim}
> notRugby <- rugby
> notRugby$ball <- "A silly shape"
> rugby$ball
[1] "A silly shape"
\end{verbatim}
In the general R world with its copy-on-modify semantics, it is safe to assume that a function $f$ will not modify object $a$ in the call $f(a)$—if it were to modify its argument $a$, then a copy of it would be created within the scope of $f$, without modifying $a$ in the calling code; however, if $a$ is a reference class object, then this assumption is no longer safe.

**Data Frames**

Data frames are one example of the legacy of S3 classes in R, and are described as “the fundamental data structure by most of R’s modelling software”\(^5\).

The concept of data frames in R is commonly considered to be naturally representative of experimental data obtained by observation (i.e. one or more variables being measured against another variable, such as time). In general, a data frame is a collection of linked variables which, when it is visualised as a table, has unique row names and named columns. An example of how data frames may be used is given in Section 2.2.6.

### 2.2.3 Flow Control

R’s ethos as a functional language encourages that, whenever possible, new functionality is implemented by function accretion, where new functions are built on top of existing functions, and operations are applied to vectors (or vectorised). However, R is not a purely functional language and so does not enforce a requirement for code to be composed exclusively of function application; in practice, R code typically looks imperative in nature.

R includes constructs for flow control: conditional code execution and loop constructs for repeated code execution.

R provides two forms of conditional code execution:

- `if (condition) expression_1`
- `if (condition) expression_1 else expression_2`

Similarly to other programming languages, if the logical expression `condition` evaluates to `TRUE` then `expression_1` is then evaluated, otherwise if the optional `else` clause is defined, `expression_2` is then evaluated.

As in the semantics of a functional language, the `if` construct returns a value. The value resulting from evaluation of an `if` statement is the value that results from evaluation of the final line in the branch, or NULL in the case where no `else` branch is defined and the condition evaluated to `FALSE`). For example, the following function `f` returns the result

---

\(^5\) According to the online help for `data.frame`
of an if, which is either the result of evaluating \( 42 > 42 \), i.e. \texttt{FALSE}; or the result of evaluating a \texttt{cat} call, i.e. \texttt{NULL}:

```r
> f <- function(x) {
+   if (x > 0) {
+     cat("if branch\n")
+     42 > 42
+   } else {
+     cat("else branch\n")
+   }
+ }
> rc <- f(rnorm(1))
if branch
> rc
[1] FALSE
> rc <- f(rnorm(1))
else branch
> rc
NULL
```

R also supports three types of loops for repeating operations:

- The \texttt{for each} loop: \texttt{for (var in range) body}.

  During each iteration of the loop, \texttt{var}’s value is iteratively set to each element of \texttt{range}, which is typically a vector, and the expression \texttt{body} is evaluated. Unless otherwise explicitly terminated, the loop terminates when \texttt{range} has been exhausted. This is akin to a for-each loop in other languages (e.g. Perl, Bash, Java).

- The \texttt{while} loop: \texttt{while (condition) body}.

  This is a traditional \texttt{while}, which evaluates the expression \texttt{condition} and if \texttt{TRUE}, evaluates the expression \texttt{body} and repeats until the test \texttt{condition} evaluates to \texttt{FALSE}.

- The \texttt{repeat} loop: \texttt{repeat body}.

  The expression \texttt{body} should include its own test to determine when to terminate the loop.

  Two statements which may appear in the \texttt{body} expressions of loops are \texttt{next} (akin to \texttt{continue} in other languages) which stops evaluation of the current loop iteration and proceeds to the next; and \texttt{break} which terminates the loop evaluation.
It is not only as a matter of style that loops are generally not preferred in R and S: a discussion has long surrounded the use of vector operations for potentially achieving greater computational efficiency over using looping constructs. The term vectorisation refers to the use of a single expression operating on a vector in place of a loop that iterates over the vector. While the potential to be gained from thorough vectorisation is often exaggerated, there are instances where this can result in greater efficiency, such as reducing the number of calls dispatched to C functions from R. The general idea is to move from a situation involving a large number of function calls that each operate on a small amount of data, to a smaller number of function calls that each operate on a larger amount of data.

2.2.4 Language

The S language, of which R is one implementation, evolved slowly into its current state of being a multi-paradigm language that draws influence primarily from functional languages, but also imperative and—with the introduction of S4 classes and in particular Reference Classes—object-oriented languages.

A pure function in computer science is one whose result depends exclusively (although not necessarily) on its arguments and therefore will always evaluate the same result given the same arguments; nor may it have any semantically observable side-effects, such as modifying external state or perform I/O interactions.

R does not enforce this practice and cannot guarantee this behaviour of functions written in R, so it may not be considered as a purely functional language; however, it does allow for functions to be written according to the spirit of functional programming, and gain (to a greater or lesser extent) the inherent advantages of pure functions.

Some common R functions have side-effects, which in some cases may be considered relatively legitimate (although still contrary to the principles of functional programming), such as plot() (a “generic function for plotting of R objects”), whose only purpose is the side-effect of drawing a plot, and those that accept user input; more illegitimate sources of side-effects exist in the form of accessing and mutating state.

While R is not a strictly-functional language, R allows for software to be written that makes use of functions and functional programming concepts, and even encourages well-written software to do just that [23].
2.2.5 Packages

R’s system of packages allows R code to be grouped, saved, distributed and loaded as required. The R distribution includes a number of packages for common functionality: the base packages, which are loaded on start-up; and the recommended packages, which are included as part of the standard distribution but not automatically loaded. Tables 2.1 and 2.2 each give a list and description of the base and recommended packages respectively at the time of writing [50].

Packages are written to provide access to additional functionality and or sets of data. It is common practice for a statistics textbook to have an R package accompaniment. The R Project also maintains CRAN, the Comprehensive R Archive Network, which is a package repository currently featuring over 5000 packages covering a vast array of functionality [2].

Functions and data sets in packages that are loaded are often subject to lazy loading, so that although they appear in scope, their contents are not actually loaded until required. This reduces the time taken for R to start, as well as keeping large data sets out of memory until they are needed. Lazy loading is discussed further in Chapter 5.

2.2.6 Bindings and Environments

When the term ‘object’ is used to describe an entity in R it is often used ambiguously and its true meaning is not quite as obvious as it may first appear. What is commonly referred to as an ‘object’ in R is actually a binding in an environment between a symbol and a value (Figure 2.1).

Consider the following R expression:

\[
> \text{three} \leftarrow 3
\]

This expression creates a singleton integer vector composed only of the integer 3, and establishes a binding between that and the symbol \text{three}. As this expression was evaluated at the top level (indicated by the command prompt >), it is effective in the global environment.

Ambiguity of this terminology will herein be avoided as far as is possible by using explicit phrasing; however, in the interest of readability, the reader is advised that a shorthand description of ‘objects’ may be employed: for instance, “x is a data frame” should be read as “a binding whose value is of type data frame is associated to the symbol x”. The environment in which the binding occurs will most likely be qualified, but in many scenarios this will be the global environment.
Table 2.1 R’s *base* packages

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>Base R functions (and datasets before R 2.0.0).</td>
</tr>
<tr>
<td>compiler</td>
<td>R byte code compiler (added in R 2.13.0).</td>
</tr>
<tr>
<td>datasets</td>
<td>Base R datasets (added in R 2.0.0).</td>
</tr>
<tr>
<td>grDevices</td>
<td>Graphics devices for base and grid graphics (added in R 2.0.0).</td>
</tr>
<tr>
<td>graphics</td>
<td>R functions for base graphics.</td>
</tr>
<tr>
<td>grid</td>
<td>A rewrite of the graphics layout capabilities, plus some support for interaction.</td>
</tr>
<tr>
<td>methods</td>
<td>Formally defined methods and classes for R objects, plus other programming tools, as described in the Green Book.</td>
</tr>
<tr>
<td>parallel</td>
<td>Support for parallel computation, including by forking and by sockets, and random-number generation (added in R 2.14.0).</td>
</tr>
<tr>
<td>splines</td>
<td>Regression spline functions and classes.</td>
</tr>
<tr>
<td>stats</td>
<td>R statistical functions.</td>
</tr>
<tr>
<td>stats4</td>
<td>Statistical functions using S4 classes.</td>
</tr>
<tr>
<td>tcltk</td>
<td>Interface and language bindings to Tcl/Tk GUI elements.</td>
</tr>
<tr>
<td>tools</td>
<td>Tools for package development and administration.</td>
</tr>
<tr>
<td>utils</td>
<td>R utility functions.</td>
</tr>
</tbody>
</table>

Figure 2.1: Bindings exist within environments and connect symbols to values. In this case, the symbol ‘three’ with a singleton integer vector ‘3’
### Table 2.2 R’s *recommended* packages

<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASS</td>
<td>Functions and datasets from the main package of Venables and Ripley, “Modern Applied Statistics with S”. (Contained in the VR bundle for R versions prior to 2.10.0.)</td>
</tr>
<tr>
<td>Matrix</td>
<td>A Matrix package. (Recommended for R 2.9.0 or later.)</td>
</tr>
<tr>
<td>class</td>
<td>Functions for classification (k-nearest neighbor and LVQ). (Contained in the VR bundle for R versions prior to 2.10.0.)</td>
</tr>
<tr>
<td>cluster</td>
<td>Functions for cluster analysis.</td>
</tr>
<tr>
<td>codetools</td>
<td>Code analysis tools. (Recommended for R 2.5.0 or later.)</td>
</tr>
<tr>
<td>foreign</td>
<td>Functions for reading and writing data stored by statistical software like Minitab, S, SAS, SPSS, Stata, Systat, etc.</td>
</tr>
<tr>
<td>lattice</td>
<td>Lattice graphics, an implementation of Trellis Graphics functions.</td>
</tr>
<tr>
<td>mgcv</td>
<td>Routines for GAMs and other generalized ridge regression problems with multiple smoothing parameter selection by GCV or UBRE.</td>
</tr>
<tr>
<td>nlme</td>
<td>Fit and compare Gaussian linear and nonlinear mixed-effects models.</td>
</tr>
<tr>
<td>nnet</td>
<td>Software for single hidden layer perceptrons (“feed-forward neural networks”), and for multinomial log-linear models. (Contained in the VR bundle for R versions prior to 2.10.0.)</td>
</tr>
<tr>
<td>rpart</td>
<td>Recursive PARTitioning and regression trees.</td>
</tr>
<tr>
<td>spatial</td>
<td>Functions for kriging and point pattern analysis from “Modern Applied Statistics with S” by W. Venables and B. Ripley. (Contained in the VR bundle for R versions prior to 2.10.0.)</td>
</tr>
<tr>
<td>survival</td>
<td>Functions for survival analysis, including penalized likelihood.</td>
</tr>
</tbody>
</table>
Each environment comprises a **frame**, which stores the mapping between symbols and bindings, and an **enclosure**, which is a pointer to an enclosing environment. This mechanism of having one environment enclosing another is used to define the **search path** that is traversed when performing variable look-up. When variables are used in expressions they are referenced by their symbol, therefore when expressions are being evaluated the symbols they contain need to be dereferenced to enable their current value to be used in place of the symbol. To know where to begin searching for bindings, expression evaluation takes place in a given environment, and if a binding cannot be located within the frame of that environment then its enclosing environments are iteratively searched until either a binding with the desired symbol has been located, or the chain of environments has been exhausted (the sequence is terminated by a special environment known as the **empty environment**).

When an environment is at some point added for inclusion in this chain, it is said to be **attached**. For instance when a package that exposes constants or functions is loaded, the environment in which its bindings reside is attached, meaning it is incorporated into the search path so that when referenced in an expression, its bindings can be located by their symbols and their values used. It is also possible to attach data frames and lists, each of which have named components—these are the columns of a data frame and individual elements within a list—which may then be accessed by name without explicit reference to the data frame or list. When one of these is attached a new environment is created and then populated by bindings whose symbols correspond to the names of the elements within the data frame or list, and whose values are copies of the elements. This environment is then placed in the search path which enables its elements to be accessed directly by name; for example if a data frame `dfr` has a column `col`, ordinarily this column would be referred to by `dfr$col`, but after `attach(dfr)`, then its column can be referred to simply as `col`.

While an environment is permitted only one direct enclosure, it is possible for an environment to be the enclosure of zero or more environments, which means environments form a tree structure, whose root is the **empty environment**; however in practice this is often more of a linked list. This is illustrated in Figure 2.2, which depicts the *mise-en-scène* of environments in a vanilla R session. This was obtained using the R function `search()` as shown in Listing 2.5, which returns a character vector representation of environments on the current search path beginning with the global environment (represented by `.GlobalEnv`) and ultimately (and necessarily) ending with the base package. Environments of packages and R ‘objects’ (such as data frames) are represented by their name.
Empty Environment

package:base

Autoloads

package:methods

package:datasets

package:utils

package:grDevices

package:graphics

package:stats

Global Environment

Figure 2.2: Each environment is enclosed by another.

attributes. This listing also shows how an attached data frame, in this instance women from package:datasets, is incorporated into the search path and how its named components may now be addressed.

Listing 2.5: Use of R function search to inspect the current search path and how this is affected by the attachment of a data frame

```r
> search()
[1] ".GlobalEnv" "package:stats" "package:graphics"

> attach(women)

> search()
[1] ".GlobalEnv" "women" "package:stats"

> height
[1] 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
```
Listing 2.6: Trivial R Example

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

Listing 2.6 shows some simple top-level R expressions. The first creates an integer vector composed of the values 1 to 5, and by assigning to the symbol `x` establishes a binding between the two. The second expression assigns `x` to `y`; or speaking more strictly, it binds symbol `y` to a copy of the value that is bound to symbol `x`.

## 2.3 CXXR

### 2.3.1 Introduction

The CXXR project [97] founded at the University of Kent is a project to progressively reengineer the fundamental components of the R interpreter from C into C++, while fully preserving functionality of the standard R distribution, which will be referred to as CR where necessary in contradistinction to CXXR. The primary objective of CXXR is to enable experimental versions of the R interpreter to be created, allowing developers to introduce new functionality, which would otherwise be highly impractical to incorporate into the standard R interpreter.

Additional consequences of conducting this refactoring include improving the internal documentation, which in the case of CR could quite feasibly be a barrier to a developer looking to alter core functionality of the interpreter; as well as tightening internal encapsulation boundaries within the interpreter, which again assists developers by instilling confidence that modifications to the code are localised and will not impact other areas.

### 2.3.2 Progressive Development

Work started on CXXR in 2007, shadowing R version 2.5.1. CXXR shadows developments to the CR distribution, as well as progressing with its own development. Over the course of this work, CXXR has been kept up-to-date with respect to numerous versions of CR, and the provenance-aware variant has in turn spanned several versions of CXXR.

The initial Provenance-Aware variant of CXXR described in Chapter 3 and Chapter 4 and Sections 5.1 and 5.2 of this work was based on CXXR 0.26-2.10.1, and it is this version that is introduced here in this chapter. This variant was later brought up to date with

---

6The two-part version number refers both to internal CXXR version [0.26] and – after the hyphen – the CR version on which it is based [2.10.1]
respect to CXXR version 0.40-2.15.1, upon which was based work that is described in Sections 5.3 and 5.4.

Where it is crucial to the understanding of the work described here, any relevant changes to the provenance-aware variant’s CXXR underpinning will be detailed as required.

### 2.3.3 Layers

The refactored CXXR interpreter can be considered to comprise three distinct layers, as shown in Figure 2.3.

![Figure 2.3: Layers within CXXR](image)

The CXXR core contains CR code that has been refactored as far as possible into idiomatic C++ and is defined within the C++ namespace CXXR. The packages layer resides on the outside of CXXR and, similarly to CR, is composed of the base and recommended packages that form the standard R distribution, as well as the optional multitude of packages available from CRAN. It is CXXR’s objective that packages should work with little or no alteration, and it has been shown that this has been achieved [96]. The transition layer consists of C code from CR redesignated as C++, which has been adapted where necessary to work with the CXXR core but not yet comprehensively refactored to employ CXXR or C++ idioms [98].

### 2.3.4 Class Hierarchy

In an R session, the user creates and interacts with object values that have been bound to symbols. In this arrangement the objects (and symbols alike) are represented internally to CR by one of two nodes: either a `struct SEXP` or a `struct VECTOR_SEXP`. Each node has an opaque pointer type, respectively defined as:
typedef struct SEXPREC *SEXP;
typedef struct VECTOR_SEXPREC *VEXSEXP;

Access to the contents of a struct SEXPREC or struct VECTOR_SEXPREC is permitted only through the provided functions, whose argument types are the opaque pointer type SEXP. For instance, elemental access of an integer vector x is achieved with the function INTEGER(x) and retrieving the head of a list y is achieved with CAR(y).

Both nodes comprise a struct sxpinfo header; three pointers—one to the object’s attributes, and one each to the next and previous nodes in a doubly-linked list (used for memory management); and finally the data.

The most important aspect of the header is the field SEXPTYPE type, where SEXPTYPE is, for the time being at least\(^7\), defined as typedef unsigned int SEXTYPE, with a #define for each one of the 27 different types of object:

```c
typedef unsigned int SEXTYPE;

#define NILSXP 0 /* nil = NULL */
#define SYMSXP 1 /* symbols */ [...]
#define LGLSXP 10 /* logical vectors */
#define INTSXP 13 /* integer vectors */
#define REALSXP 14 /* real variables */
#define CPLXSXP 15 /* complex variables */ [...]
#define FUNSXP 99 /* Closure or BuiltIn or Special */
```

The data portion of the two nodes vary. The SEXP node is used to represent a number of types: primitives, symbols, lists, environments, closures, and promises, so its data component is a C union of these individual structures.

```c
union {
    struct primsxp_struct primsxp;
    struct symsxp_struct symsxp;
    struct listsxp_struct listsxp;
    struct envsxp_struct envsxp;
    struct closxp_struct closxp;
    struct promsxp_struct promsxp;
} u;
```

While the VECSEXP node only represents vector types (which include logical, integer, real, etc.) and has as its data portion a struct vecsxp_struct vecsxp; used to store

---

\(^7\) There is a currently a movement to replace the current definition of SEXPTYPE values with an enum
the length of the vector\textsuperscript{8}. This node is then followed by a block of memory large enough
to store the required number of vector elements, whose individual size depends upon the
type of vector. For instance a C \texttt{int} type is used to represent elements of integer and
logical vectors and C99 \texttt{double complex} is used for complex vectors.

Since both nodes share a common preamble section (of header and three pointers), it is
common practice for a \texttt{VECSEXP} to be treated as though it were a \texttt{SEXP}: even the function
responsible for allocating vectors has the signature:

\begin{verbatim}
SEXP allocVector(SEXPTYPE type, R_len_t length)
\end{verbatim}

This makes working with objects in CR a very type-unsafe prospect. As with all C
unions, exactly which one of its possible constituent types a \texttt{SEXPREC} is representing is not
known at compile-time and determined only at runtime. There are two main implications
of this: the compiler’s typechecking abilities are not utilised; and at runtime, debugging
the interpreter is significantly more difficult—two characteristics that are incompatible
with CXXR’s motivation to enable creation of experimental versions of interpreter, and
indeed incompatible with any experimenter with a sense of self-preservation.

CXXR replaces this union with an extensible class hierarchy rooted in class \texttt{RObject},
which is shown in Figure 2.4.

\subsection*{2.3.5 Memory Management}

CXXR’s core instruments facilities for the allocation and management of memory using
automatic garbage collection.

The class \texttt{MemoryBank} is responsible for memory allocation and can allocate memory in
one of two ways depending upon the size of the allocations required. Memory allocations
of less than 128 bytes can be drawn from CXXR’s \texttt{CellPool}, which comprises preallocated
cells of fixed size (e.g. 8 bytes, 16 bytes). This enables more efficient allocation
and deallocation of memory for smaller objects \textsuperscript{72}. Larger allocations of memory are
accomplished using C++’s \texttt{::operator new}.

The two principal clients of \texttt{MemoryBank} are \texttt{GCNode::operator new} and
\texttt{CXXR::Allocator<T>}. The former will be discussed in the following section; the lat-
ter is an STL-compatible Allocator to enable STL collections to use memory allocated via
\texttt{MemoryBank}.

\textsuperscript{8}As well as a field called \texttt{truelength}, which is not used in the vast majority of cases.
CXXR employs a garbage collection mechanism for automatically deallocating objects when they are no longer required. This is primarily based on reference counting, which dictates that when an object is no longer referenced by any other object then it is no longer possible to access it and so it may be deleted. Mark-sweep garbage collection is employed as a backstop to collect up cases where objects may refer to each other but are not otherwise accessible from any other objects.

Objects subject to garbage collection inherit—either directly or indirectly—from the class GCNode (so called as it represents a node in the graph of objects), which incorporates a reference count. When a GCNode is instantiated, its construction will involve requesting memory for itself via MemoryBank.

GCNodes refer to each other by way of a GCEdge<T>, which is a templated smart pointer that automatically adjusts the reference count of the GCNode to which it points.

A garbage collection may be triggered at any point when memory is requested from the memory bank. Typically these garbage collections are based only on reference counting, as it is trivial to ascertain which objects are no longer required; this is known as a lightweight
garbage collection. A mark sweep garbage collection is less frequent, and a threshold for the point above which these are conducted is maintained in class \texttt{GCManager}.

From the point that \texttt{GCNode} is created and exposed to the garbage collector, it needs to be protected from unwanted destruction by the garbage collector. One way in which this is achieved is by use of a \texttt{GCEdge}, but this is only appropriate in situations where the reference to the \texttt{GCNode} is from another \texttt{GCNode}, in other situations it is necessary to specifically instruct the garbage collector to \texttt{protect} a vulnerable \texttt{GCNode}.

One typical scenario where the need for these measures occurs is within the C implementation of the R primitive functions, such as \texttt{abs(x)}—which is analogous to (and implemented using) C’s \texttt{abs} function—which constructs an object in which to store the result of the operations to be returned to the calling code. Such an object needs to be protected from being prematurely destroyed before it is returned.

In CR, protecting an object from the garbage collector is handled entirely manually using the macros \texttt{PROTECT()}, \texttt{UNPROTECT()}, and (less commonly) \texttt{REPROTECT()}. When an object is to be protected from the garbage collector, then a call to \texttt{PROTECT()} is made with it given as an argument. Pointers to protected objects are stored on a stack, and as such are removed in a last-in-first-out arrangement. When objects no longer require protection, \texttt{UNPROTECT} is used to \texttt{pop} a given number of objects from the protection stack. This behaviour is illustrated in Listing 2.7 which shows the \texttt{do_abs} C function that implements R’s primitive function \texttt{abs}.

When a vector object is created—either on line 16 or 22—it is immediately protected by the call to \texttt{PROTECT}. Before the function returns it, it is necessary to \texttt{UNPROTECT} it (line 30).

Listing 2.7: The \texttt{do_abs} function which implements R’s primitive function \texttt{abs}

```c
SEXP attribute_hidden do_abs(SEXP call, SEXP op, SEXP args, SEXP env)
{
    SEXP x, s = R_NilValue /* -Wall */;

    checkArity(op, args);
    check1arg(args, call, "x");
    x = CAR(args);

    if (DispatchGroup("Math", call, op, args, env, &s))
        return s;
```
if (isInteger(x) || isLogical(x)) {
    /* integer or logical ==> return integer, 
    factor was covered by Math.factor. */
    int i, n = length(x);
    PROTECT(s = allocVector(INTSXP, n));
    /* Note: relying on INTEGER(.) === LOGICAL(.) : */
    for(i = 0 ; i < n ; i++)
        INTEGER(s)[i] = abs(INTEGER(x)[i]);
} else if (TYPEOF(x) == REALSXP) {
    int i, n = length(x);
    PROTECT(s = allocVector(REALSXP, n));
    for(i = 0 ; i < n ; i++)
        REAL(s)[i] = fabs(REAL(x)[i]);
} else if (isComplex(x)) {
    return do_cmathfuns(call, op, args, env);
} else
    errorcall(call, R_MSG_NONNUM_MATH);
DUPLICATE_ATTRIB(s, x);
UNPROTECT(1);
return s;

As can be seen in the illustrative example, this mechanism requires parity between
the number of objects protected and subsequently unprotected, and places responsibility
for ensuring this with the programmer. Therefore, this mechanism can be (and often is)
prone to errors, particularly in conditional branches from which the method may return,
as the number of objects that require unprotecting can be variable. Because only a single
protection stack exists for the entire call stack, if an object is not unprotected at the
appropriate time and exposed to the garbage collector, then it will persist and result in
a memory leak; conversely, the consequence of too many objects being unprotected is
the premature garbage collection of an object possibly outside the scope of the current
method.

The CXXR analogue of this mechanism is the C++ smart pointer class GCStackRoot,
which offers protection against garbage collection for the object to which it refers. As
with any C++ variable, its lifetime is determined by its scope, which makes it particularly
suited to short-term protection against garbage collection. When the GCStackRoot object
goes out of scope (either when a method returns or the code block in which it was defined
closes), the object to which it referred is then exposed to the garbage collector. As the name
implies, GCStackRoot is implemented as a stack, and so objects of this type are required to be destroyed in the reverse order of their creation, which is also a suitable characteristic for short-term garbage collection protection. Long-term protection against garbage collection is better achieved using GCRoot, which works in a similar way to GCStackRoot but is not subject to the latter’s restriction in destruction order and as a consequence construction and destruction is slightly more time-consuming.

An example of how a GCStackRoot can be employed to protect an object is shown in Listing 2.8. This can be contrasted with Listing 2.9 which gives one way in which the same scenario is handled with only traditional CR facilities. There is no particularly favourable way of handling this situation as the protection stack works against efficiency.

Listing 2.8: Example usage of GCStackRoot in a function that returns a reversed copy of a PairList

```cpp
PairList* reverse(const PairList* pl)
{
    GCStackRoot<PairList> ans;
    while (pl) {
        ans = PairList::cons(pl->car()->clone(),
                             ans, pl->tag());
        pl = pl->tail();
    }
    return ans;
}
```

Listing 2.9: Traditional CR garbage collection mechanism example

```cpp
SEXP reverse(SEXP pl)
{
    SEXP ans = R_NilValue;
    int nprotect = 0;
    for (; pl != R_NilValue; pl = CDR(pl))
    {
        SEXP el = Rf_duplicate(CAR(pl));
        PROTECT(ans = CONS(el, ans)); ++nprotect;
        SET_TAG(ans, TAG(el));
    }
```

2.3.6 Other aspects of CXXR

This section will give a brief overview of how CXXR has addressed refactoring other aspects of the CR interpreter that, although these may not be of direct relevance to the remainder of this work, should nevertheless be useful in getting the flavour of what CXXR is about.

Handling of R errors and indirect flows of control such as return and break from within R code are handled by CR with the use of the C standard library functions setjmp/longjmp, which are incompatible with C++’s requirement for in-order stack unwinding. The natural candidate for re-engineering this functionality is C++ exceptions, which are used in CXXR to handle and report errors. However, the propagation of C++ exceptions when handling indirect flows of control incurred a significant overhead, and this led to a comprehensive refactoring of the CR notion of a context, and the introduction of classes Bailout and BailoutContext which offer a lightweight means to return control flow to its intended destination.

The means for object duplicating now utilises C++ copy constructors. The CR function Rf_duplicate to accomplish this previously utilised a gargantuan switch statement for each SEXPTYPE. Under CXXR, this function simply calls a virtual method clone on the relevant RObject, which in turn utilises the copy constructor of its class.

Unary and binary operations that are type-specific in CR have been replaced in CXXR with generic algorithms using C++ templates. CXXR’s extensible hierarchy enables the easy introduction of new data types, and the abstraction of mathematical and subscripting (and subassignment) operations that are exposed via the CXXR API, enables new data types to be up and running with R operations with a minimal amount of coding.
Chapter 3

Provenance in CXXR

This chapter will address the following research goals as set out in Section 1.5.1, in the ways respectively described:

- Goal 2 will be addressed by defining a set of provenance questions that a provenance-aware CXXR should be able to answer;

- Goal 3 involves firstly the development of a view of provenance in the context of CXXR; secondly, and in accordance with the aforementioned view of provenance in CXXR, the design of facilities to record the information necessary to allow provenance questions to be answered; and finally, the implementation of the design;

- Goal 6 is to be addressed by evaluation of the implemented facilities, in particular with respect to a real-world example of an exploratory data analysis.

3.1 Provenance Questions

Section 1.5 introduced the concept of, and the motivation for, the CXXR user wishing to ask provenance questions of the data in his or her session, with the primary aim to elicit further information about the data and thereby gain greater understanding of how an analysis was conducted.

The motivating questions to which a provenance-aware CXXR should be able to provide answers are given as follows:

1. What was the command that gave rise to an object?

2. When was an object created?

3. Which other objects were used in the creation of an object?
4. Which other objects during their creation used an object?

5. What is the full sequence of commands that was used to derive an object (or collection of objects)?

6. Who was the user that created, with which version of the software, an object?

3.2 Design - Recording

3.2.1 Entity

As implied by the provenance questions given above, the entity whose provenance we wish to record and subsequently query is the object. To answer the provenance questions, it is necessary to record operations performed on an object throughout its lifetime—from its initial creation to its deletion, and in the meantime, every occasion on which it is used to calculate the value of another object.

As introduced in Section 2.2.6, the colloquial ‘object’ is more strictly known as a binding.

Listing 3.1: Trivial R Example (reprise)

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

In the code shown in Listing 3.1 each of the top level expressions evaluated is an assignment and as such is responsible for the creation of a new binding; however, the second expression does not just create a binding, it also reads the value of an existing one—x, which it does to determine the value of the RHS operand of the assignment operation and therefore what value to give the newly-created binding to symbol y.

There is an important but subtle point to be made here regarding provenance. To understand how x and y in Listing 3.1 have been derived, we need to know what has been bound to these symbols. The value bound to x is an integer vector. During the assignment y <- x R’s copy-on-write policy dictates that a copy of the vector is made, to which symbol y is bound.

There is no way to determine from the perspective of the vectors here, how x and y have been derived.

It is clear, therefore, that provenance information in this context of variable bindings must be associated not with an object—in the loose sense—but with a binding.
The relationships between classes Symbol, Binding, and RObject are depicted in UML by Figure 3.1.

![Figure 3.1: UML class diagram depicting attribute relationships of the binding class](image)

**Provenance class: A binding state**

When a binding is established (i.e. an “object is written”, loosely speaking), such as `x` during the evaluation of the above expression `x <- 1:5`, this will be known as the creation of a binding state (of `x`).

The term “binding state” is used to denote the concept that a binding has a mutable nature; therefore, when an assignment expression is evaluated and causes a binding `B` to receive a value, `B`’s `value` attribute is altered, but the same `B` binding object persists.

A binding state is represented in this model by the class Provenance (Figure 3.2), which records the following details pertaining to a binding state:

- The symbol that was bound;
- The expression that gave rise to the binding;
- The timestamp of the binding’s establishment;
- The `parents` of the binding (if any);
- The `children` of the binding (if any).

When a binding state is created, a Provenance is created to represent this fact and it is then associated with the Binding by attribution.

![Figure 3.2: UML class diagram depicting attributes of the Provenance class](image)
Figure 3.3: UML class diagram showing attribute relationships surrounding the Provenance class

Figure 3.3 shows how the Provenance class is attributed to a binding.

Provenance P1 is a parent of Provenance P2 (and conversely P2 is a child of P1) if the binding state corresponding to P1 was read in the course of evaluating the expression that gave rise to binding state corresponding to P2. It is not necessary for a Provenance to have any parents at all, and there is no conceptual limit to the number of parents it may have.

Provenance Hierarchy

The Provenance class's self-referential attributes parents and children model the real-world relationships after which they are named to permit the formation of a Provenance family tree. Like any other family tree, one composed of Provenances is not limited to representing the relationships between only two generations, but an arbitrary number of generations. This chaining together of Provenances can provide an entire ancestry.

In the following example, three binding states of \( x \) are created; the latter two are each dependant upon the previous:

\[
\begin{align*}
> & \ x \leftarrow 1 \\
> & \ x \leftarrow x + 1 \\
> & \ x \leftarrow x + 2
\end{align*}
\]

The sequence of evaluating the above expressions is represented by object diagrams in Figure 3.4, which illustrates how a Provenance captures and preserves the particular state of a binding, as well as how Provenances relate to each other through their parent and
child attributes to form a hierarchy.

### 3.2.2 Activity

This model considers the provenance activities that occur to be the **evaluation of expressions**.

The CXXR environment, like that of CR, employs by default a **Read-Evaluate-Print-Loop (REPL)**. The steps involved are: *read* user input from the standard input; *parse* and *evaluate* that input; *print* the result; and *loop* (i.e. return to *read*). This type of behaviour is common to interactive toplevels, such as the general purpose UNIX shell, and environments for interpreted programming languages such as Python.

Figure 3.5 depicts by means of a sequence diagram the Read-Evaluate-Print-Loop mechanism that is employed by CXXR.

### Granularity

The REPL strategy naturally advances a provenance granularity of the **top-level expression**. A top-level expression is so-called in contradistinction to the sub-expressions of which it may be composed. It is considered that this granularity is compatible with provenance question 1 and it will therefore be utilised in this design.

One of the principal implications of this design choice is that the evaluation of a single top-level expression, E, may read a particular binding state B1 multiple times before writing binding state B2. There is no need, however, for this fact to be recorded: B1 is a parent of B2 by virtue of it being read at least once; therefore, B1 should appear only once as a parent of B2.

Similarly, evaluation of E may give rise to multiple bindings. If binding state B1 is written during the course of evaluating top-level expression E, then the definition of B1’s parents is those bindings read in the course of evaluating E before B1 is written.

By extension, it is conceivable that an individual binding B2 may be written to multiple times thereby going through numerous intermediate binding states in the course of evaluating E; however, to record this level of granularity would be too fine for the mandate of this design.

In order to maintain these constraints and ensure that granularity is restricted to the top-level expression, it is proposed that a set of Provenance objects known as the **seen set** is maintained to prevent binding states being erroneously recorded in a parentage.

One such scenario that requires a restriction in granularity by these means, is the
CHAPTER 3. PROVENANCE IN CXXR

(a) After first expression evaluation

(b) After second expression evaluation

(c) After third expression evaluation

Figure 3.4: Example of a Provenance hierarchy
Handling of Loops

R allows for the use of looping constructs, although their use is not favoured for operations on vectors. The R code shown in Listing 3.2 uses a for-each loop to compute the sum of integers 1 to 5 (inclusive) and stores the result in x.

Listing 3.2: Example loop in R

```r
1 > x <- 0  # Initialise x to zero
2 > for (n in 1:5) # n = {1 .. 5}
3 + 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 +) is a loop in which n iteratively takes the value of each element in the vector [1, 2, 3, 4, 5] (created by 1:5 which uses the infix binary operator : to generate a regular sequence), and gets added to x. During each iteration of the loop: a new binding state of n is established; the addition operation reads and sums the values of the bindings to n and x; the assignment operation
binds the result of the addition to `x`. In short, during each loop iteration the values of both bindings `x` and `n` are read, and new binding states of both `x` and `n` are established. Although `n` is used as a loop control variable and declared in the loop header, in R—unlike many languages—it’s lifetime is not limited to the body of the loop: expressions relating to it are evaluated in the global environment, which is therefore where its binding is created and will reside after the loop has terminated. At the end of the loop, there will remain a binding of `x` and a binding of `n`, but there were binding states of each that do not survive the evaluation of the `for` loop—in this instance they only survive the iteration in which they were established—since there may be at most only one binding to a symbol in any frame.

Listing 3.3 expands the above loop and shows each intermediate binding state denoted by a suffix.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>x0 = 0</code> # Initialise <code>x</code> to zero</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td><code>n0 = 1</code> # First Iteration</td>
</tr>
<tr>
<td>4</td>
<td><code>x1 = x0 + n0</code></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td><code>n1 = 2</code> # Second Iteration</td>
</tr>
<tr>
<td>7</td>
<td><code>x2 = x1 + n1</code></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td><code>n2 = 3</code> # Third Iteration</td>
</tr>
<tr>
<td>10</td>
<td><code>x3 = x2 + n2</code></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td><code>n3 = 4</code> # Fourth Iteration</td>
</tr>
<tr>
<td>13</td>
<td><code>x4 = x3 + n3</code></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td><code>n4 = 5</code> # Fifth Iteration</td>
</tr>
<tr>
<td>16</td>
<td><code>x5 = x4 + n4</code></td>
</tr>
</tbody>
</table>

The binding state of `x` that persists beyond the end of this expression evaluation is `x5`, which has as a parent `x4`, which has as a parent `x3`, and so on back up to `x0`. Similarly, the intermediate states of binding `n` are recorded as parents of their respective children.

According to the chosen strategy of attributing top-level expressions, the granularity of this information is **too fine**; it does not accurately represent the top-level expression that was issued, nor do we have adequate notation to differentiate between the various
intermediate binding states to elicit any meaning therefrom.

The seen set

The purpose of the seen set in this design is to prevent intermediate bindings from being recorded in the parentage, thereby maintaining the granularity of the top-level expression, as well as to prevent multiple occurrences of a single binding state in a parentage. For instance, in the case of \( b \leftarrow a + a \), although the value of \( a \) is used twice, it should appear only once as a parent to \( b \).

Whenever a binding is written to or read from during the course of evaluating a top-level expression, the Provenance associated with it is added to the seen set to indicate that reads of it should not be subsequently reflected to the parentage.

In the instance of the ‘for each’ loop given in Listing 3.2, the effect of the seen set is as follows. During the first loop iteration, \( n \) is bound to 1 causing the Provenance associated with this binding to be added to the seen set. When the value of \( n \) is required during the evaluation of the expression \( x \leftarrow x + n \), because the Provenance of \( n \) exists in the seen set, the reading of this value will not be reflected in (i.e. added to) the parentage. This expression evaluation will also invoke a read on the binding of \( x \) that was created immediately prior to the loop, therefore its Provenance will be added to the parentage as well as to the seen set; when the assignment operation established a new binding to \( x \), this too will be added to the seen set so that on the second iteration of the loop, it will be excluded from the parentage. This repeats for each iteration of the loop: the binding \( n \) established by the loop construct and the binding \( x \) established in the previous iteration are both excluded from the parents of the newly established \( x \). The ultimate effect of this is that the end of the evaluation of the for loop, the parentage only contains the Provenance of the binding of \( x \) established before the loop, not its subsequent bindings. As such the bindings \( x \) and \( n \) that survive the loop will each be attributed to the expression \( \text{for } (n \text{ in } 1:5) \ x \leftarrow x + n \) and have the prior binding of \( x \) as their sole parent.

Interactions

This model captures the interactions between activities (evaluation of top-level expressions) and entities (bindings), which are the read and write operations that are performed on a binding during the course of evaluating a top-level expression. For example, in the expression \( y \leftarrow 1 + x \), the value of the binding \( x \) is required, causing a read operation to be performed on it; and subsequently a write operation will occur to the binding \( y \) due
to the assignment operation. The occurrences of read and write types of interaction are depicted in activity diagrams shown in Figures 3.6 and 3.7 respectively.

Figure 3.6: Activity diagram depicting occurrence of read operation on a binding

Figure 3.7: Activity diagram depicting occurrence of write operation on a binding

To capture the interactions between an activity and entities, this design employs the notion of a monitor to be called when a binding is read from or written to.

Figures 3.8 and 3.9 illustrate by way of activity diagrams when the monitors on read and write operations are respectively triggered.
CHAPTER 3. PROVENANCE IN CXXR

Class ProvenanceTracker

This method of recording provenance requires the maintenance of some state for the lifetime of a REPL iteration as dictated by the chosen level of granularity: the current parentage, an ordered collection of Provenance to record those bindings read; and the seen set, an (unordered) collection of Provenance to record which bindings have previously been encountered and should not be subsequently recorded in the parentage.
This state, along with the operations to reset it for each REPL iteration, and the
monitor operations are defined by the ProvenanceTracker class (Figure 3.10).

```
ProvenanceTracker
- current_parentage : Provenance [*]
- seen : Provenance [*]
+ resetParentage (void) : void
+ resetSeen (void) : void
+ writeMonitor (Binding) : void
+ readMonitor (Binding) : void
```

Figure 3.10: UML class diagram depicting attributes and operations of the ProvenanceTracker class

The way in which ProvenanceTracker has been designed to be used as part of the REPL strategy is shown in Figure 3.11.

Monitors

It is the responsibility of the read and write monitors to intercept read and write operations performed on a binding during the course of top-level expression evaluation and perform the necessary housekeeping to ensure the provenance record is maintained.

The read monitor must, if a binding state has not been included in the seen set, include
this in both the parentage and seen set. The write monitor is responsible for encapsulating the binding state in a Provenance object (Figure 3.2); associating this Provenance with the binding; adding it to the seen set; and finally, registering it as a child of each parent.

The designed behaviour of the read and write monitors is depicted respectively in Figures 3.12 and 3.13.

![Figure 3.12: Activity diagram depicting the behaviour of the read monitor](image)

![Figure 3.13: Activity diagram depicting the behaviour of the write monitor](image)

**Binding Deletion**

When a binding state is established, it is registered as a child of each of its parents.

Conversely, when a binding B is deleted (e.g. when the user uses the `rm()` function to
delete an object) it is necessary for B to be deregistered as a child of each of its parents.

3.2.3 Algorithm

The approach taken in the above design to model and record provenance in CXXR is described in pseudocode in Algorithm 3.1.

**Algorithm 3.1 Provenance-aware CXXR recording algorithm**

1: procedure RECORDPROVENANCE
2:   GlobalEnv.WriteMonitor ← ProvenanceTracker.writeMonitor
3:   GlobalEnv.ReadMonitor ← ProvenanceTracker.readMonitor
4: for each REPL iteration do
5:   p_seen ← []  \(\triangleright\) Reset containers
6:   p_current ← []
7: procedure PROVENANCETRACKER.READMONITOR(bdg : Binding)
8:   if bdg.m_provenance ∉ p_seen then
9:     p_current.add(bdg.m_provenance)
10:    p_seen.add(bdg.m_provenance)
11: procedure PROVENANCETRACKER.WRITEMONITOR(bdg : Binding)
12:    P ← NEW PROVENANCE  \(\triangleright\) Create new binding state
13:    P.m_expression ← Current top-level expression
14:    P.m_symbol ← bdg.symbol
15:    P.m_parents ← p_current
16:    P.m_timestamp ← Current Time
17:    P.m_children ← []
18: for all parent ∈ P.m_parents  \(\triangleright\) Register as child of each parent
19:    parent.children.add(P)
20:    bdg.m_provenance ← P
21:    p_seen.add(P)

3.3 Design - Querying

3.3.1 In-interpreter Interface

The nature of the motivating provenance questions, for example Question 5, “What is the full sequence of commands that was used to derive an object?”, in the typical use case
(Figure 1.7), dictates that answers should be provided in (i) an immediate fashion; (ii) in an environment with which the user is already familiar; and (iii) in a format that can be fed back into the user’s workflow.

In consideration of these factors, the prime candidate interface for querying recorded provenance information in CXXR is in-interpreter, by employing R-level functions that the user evaluates at the command prompt.

Some of the advantages of this approach are:

• The use of R functions should already be a concept familiar to the user, for whom provenance information should therefore be readily accessible;

• The provenance information remains in-memory without the need to be continually serialised. This results in a simpler implementation;

• The output of an R function can be used as the input to another function, thereby offering substantial flexibility of which objects are queried and how the result is used.

Answers to provenance questions can be given by two functions:

• `provenance(symbol)` This function should return information pertaining to a current binding state given by reference to its symbol. This function answers provenance questions 1-4;

• `pedigree(symbol(s))` This function should return the sequence of commands used to derive the current state(s) of given binding(s) by traversing the hierarchy of Provenances in chronological order. This function answers provenance question 5. Algorithm 3.2 describes a routine for determining the ancestors of a set of binding states.

3.4 Implementation

3.4.1 Monitors

CXXR introduces the concept of monitors that are applied to a frame and triggered whenever a binding within that frame is accessed, either to have its value read or when it is established. A monitor is a pointer to a function which takes as argument a reference to the binding being accessed. The monitor data type is defined in the Frame class as follows:

```c
typedef void (*monitor)(const Binding&);
```
Algorithm 3.2 Determine the ancestors of a (set of) binding state(s)

1: procedure ANCESTORS(open : set of Provenance)

2:   closed ← ∅

3: while open¬∅ do

4:   P ∈ open

5:   for all parent ∈ P.parents do

6:     if parent /∈ closed then

7:       OPEN.ADD(parent)

8:       OPEN.REMOVE(P)

9:     closed.add(P)

10: return closed

The Frame class defines two member fields of this data type: m_read_monitor and m_write_monitor, each of which has a mutator method (e.g. setReadMonitor) to alter its value, and an accessor method (e.g. monitorRead), which will invoke the relevant monitor function if one has been set—since there is no requirement to implement monitors, a NULL monitor signifies such an absence.

Monitor Triggers

During the course of evaluating an expression the interpreter necessarily performs look-ups for any symbols used in the expression to denote references to variables. As previously mentioned (Section 2.2.6, this search begins in a given environment, which for symbols occurring in a top-level command will be the global environment. The process of looking up a symbol beginning in a given Environment is handled by the function findVar(SEXP symbol, SEXP rho) (shown in Listing 3.4) which was inherited by CXXR from CR. Most of R’s internal C functions pass arguments as SEXP opaque pointers. In CXXR SEXP is a pointer to RObject, which then requires downcasting to its actual type, either in a checked capacity, such as with the symbol argument, or in an unchecked capacity, such as with rho. rho is the name customarily designated to an environment argument.

Listing 3.4: The findVar function from envir.cpp

1 | SEXP findVar(SEXP symbol, SEXP rho) |
2 | { |
3 | if (TYPEOF(rho) == NILSXP) |
It uses the `findBinding` function, shown in Listing 3.5, to search an environment `env` for a binding with a given symbol `sym`. If one cannot be found in the environment given, then it repeats this search in each enclosing environment in turn.

Listing 3.5: The `findBinding` function from `envir.cpp`

```cpp
pair<Environment*, Frame::Binding*> findBinding(const Symbol* symbol, Environment* env) {
    while (env) {
        Frame::Binding* bdg = env->frame()->binding(symbol);
        if (bdg) {
            return make_pair(env, bdg);
        }
        env = env->enclosingEnvironment();
    }
    return pair<Environment*, Frame::Binding*>(0, 0);
}
```

If a binding associated with the specified symbol has been located in the environment’s frame, then `findVar` will return the value of it by calling the `value` method on it. It is at this point that the read monitor is called, as is shown in Listing 3.6.

Listing 3.6: `Frame::Binding::value()`

```cpp
RObject* Frame::Binding::value() const {
    RObject* ans = (isActive() ? getActiveValue(m_value) : m_value);
    m_frame->monitorRead(*this);
    return ans;
}
```
The assignment to a variable involves modification of a binding’s value, and is handled similarly. Initially the setVar method uses findBinding to locate a binding, which if successfully found, has its assign method called with the value being assigned to it as argument. The assign method will dispatch a call to the write monitor, as shown in Listing 3.7.

Listing 3.7: The Frame::Binding::assign method

```cpp
void Frame::Binding::assign(RObject* new_value, Origin origin)
{
    if (isLocked())
        Rf_error(_("cannot change value of locked binding for \"%s\""),
                  symbol()->name()->c_str());
    m_origin = origin;
    if (isActive()) {
        setActiveValue(m_value, new_value);
        m_frame->monitorRead(*this);
    } else {
        m_value = new_value;
        m_frame->monitorWrite(*this);
    }
}
```

### 3.4.2 Containers

The following containers have been introduced to CXXR to store the various aspects of provenance information. The class collaboration diagram for these new classes is shown in Figure 3.14

**class Provenance**

The Provenance class is central to storing provenance for a binding. Because it is desirable that Provenance objects which are no longer accessible are destroyed, this class is subject to the garbage collection outlined in Section 2.3.5 and as such inherits the CXXR class GCNode, and may utilise GCEdge smart pointers to refer to other GCNode objects.

- Its implementation comprises the following:
  - The `struct` timeval timestamp of the binding’s creation;
  - A pointer to a Set of children;
Figure 3.14: Class collaboration diagram of new/old CXXR classes.

- A pointer to a `Parentage` and an integer denoting the position in the parentage at which the binding was created;
- A GCEdge to a clone of the top-level `expression` that was being evaluated when the binding was created;
- A GCEdge to the `symbol` to which the binding is bound.

It also defines a nested class `CompTime` that is used by STL containers of `Provenance` objects to order their members chronologically according to their timestamp, as shown in Listing 3.8.

Listing 3.8: class `CXXR::Provenance::CompTime`

```cpp
class CompTime {
public:
    bool operator()(Provenance* lhs, Provenance* rhs) {
        return (lhs->m_timestamp.tv_sec==rhs->m_timestamp.tv_sec) ? 
            (lhs->m_timestamp.tv_usec<rhs->m_timestamp.tv_usec) : 
            (lhs->m_timestamp.tv_usec>rhs->m_timestamp.tv_usec);
    }
};
```

class `Parentage`

The `Parentage` class stores references to `Provenance` objects pertaining to those `Bindings` that have been read during the course of evaluating a top-level expression. It is necessary to
reference Provenances here as their association with a binding may only be transient, and may not outlast the evaluation of the top-level expression, such as the loop control variable \( n \) in the loop \( \text{for } (n \text{ in } 1:5) \) which will go through numerous bindings throughout the evaluation (as Section 3.2.2 will discuss).

This class inherits from the C++ Standard Template Library (STL) \texttt{std::vector} class; it does not inherit from GCNode. However, despite this, it does encapsulate its references to Provenance objects in GCEdges, but instead of relying on the garbage collection facilities from GCNode it performs its own manual reference counting together with class Provenance. This is in response to a garbage collection issue which resulted in the premature destruction of Parentage objects.

\textbf{class ProvenanceSet}

This class is a set of references to Provenance objects via GCEdges. It was introduced for use in circumstances where these objects would otherwise be in the firing line of the garbage collector and therefore require protection from it. This collection inherits from GCNode and therefore forms part of the garbage collection graph. It principally functions as the seen set during collection of provenance information described later in Section 3.2.2 and is defined as follows:

\begin{verbatim}
class ProvenanceSet : public GCNode,
    public std::set<GCEdge<Provenance>, Provenance::CompTime>
\end{verbatim}

\textbf{Set Provenance::Set}

This is a non-GC analogue of ProvenanceSet defined as:

\begin{verbatim}
typedef std::set<Provenance*, Provenance::CompTime> Set;
\end{verbatim}

This class is used for purposes where the Provenance objects it comprises are referenced elsewhere via GCEdges, typically as part of a Parentage. This class is used to represent the set of children attributed to a Provenance and creating, manipulating and traversing provenance graphs.

\textbf{3.4.3 ProvenanceTracker}

The class ProvenanceTracker is principally responsible for the execution of the above algorithm. It houses instances of the containers for the seen set and current parentage list; the read and write monitors that are instrumented to a frame to capture reads and writes of bindings; as well as methods for resetting the collections at the start of a REPL cycle.
All the members of this class—both its fields and methods—are static; this class is not instantiated, it is used purely in a static context.

REPL Reset

The ProvenanceTracker class provides two methods for resetting the member fields of class ProvenanceTracker: resetParentage() and resetExpression(), both of which are shown in Listing 3.9.

Listing 3.9: Methods for resetting in preparation for new REPL iteration

```cpp
void ProvenanceTracker::resetParentage() {
    (*p_seen)=GCNode::expose(new ProvenanceSet());
    (*p_current)->set(new Parentage());
    return;
}

void ProvenanceTracker::resetExpression() {
    setExpression(NULL);
}
```

Each of these methods is called at the beginning of each REPL iteration (handled by the function Rf_ReplIteration defined in main.cpp).

Monitor Hooks

The monitor functions that are triggered on read and writes of bindings are defined in this class, as is a method for attaching these monitors to the hooks of a particular environment, initEnv(Environment*), which gets called during initialisation of the global environment.

Read Monitor

The read monitor defined in ProvenanceTracker is simply as is stated in the algorithm above, and shown in Listing 3.10.

Listing 3.10: ProvenanceTracker::readMonitor

```cpp
void ProvenanceTracker::readMonitor(const Frame::Binding& bdg) {
    Frame::Binding& b=const_cast<Frame::Binding&>(bdg);
    Provenance* p=const_cast<Provenance*>(b.getProvenance());
}```
```cpp
if (!p) return;
GCEdge<Provenance> needle(p);
if (seen()->find(needle)==seen()->end())
    parentage()->pushProvenance(p);
seen()->insert(needle);
}
```

**Write Monitor**

Similarly, the write monitor defined by `ProvenanceTracker` is largely as appears in the above algorithm and is shown in Listing 3.11. The expression that is passed in the first argument to the constructor of `Provenance` is determined by the function `expression()`, which will be discussed in greater detail in Section 5.1.3.

**Listing 3.11: ProvenanceTracker::writeMonitor**

```cpp
void ProvenanceTracker::writeMonitor(const Frame::Binding &bind, {
    CXXR::Frame::Binding& bdg=const_cast<CXXR::Frame::Binding&>(bind);
    RObject* e=expression();
    Expression* expr=(e) ? static_cast<Expression*>(e->clone()) : NULL;
    Symbol* sym=const_cast<Symbol*>(bind.symbol());

    bdg.setProvenance(GCNode::expose(
        new Provenance(expr,sym,parentage())
    ));
    Provenance* prov=const_cast<Provenance*>(bdg.getProvenance());

    GCEdge<Provenance> tmp(prov);
    seen()->insert(tmp);
}
```

**Registering with Parents**

During the course of evaluating a top-level expression, only a single parentage is maintained and referenced by the provenances of all bindings created during the evaluation of that expression. This strategy was implemented to avoid needlessly duplicating `Provenance` references and creating `Parentage` objects of different sizes. Only those members of a `Parentage` that were read before a `Provenance` was created are considered to be its parents. This is realised by storing a position marker in `Provenance` to represent the number of
relevant members in the parentage.

When a Provenance is created, it is responsible for announcing itself as a child of each of its parents. This happens in Provenance::announceBirth(), shown in Listing 3.12, where registerChild(Provenance* p) simply adds p to its set of children. The at method called on a Parentage is inherited from std::vector.

Deregistering with Parents

Similarly it is also necessary for a Provenance object to deregister itself from its parents when it is no longer required. This occurs when the binding to which a Provenance is attached is no longer accessible, either directly from an environment (e.g. after being explicitly removed, or ‘replaced’ when a symbol is rebound to a different value) or as part of the ancestry of an accessible binding. This is determined automatically by the garbage collector, so that when a Provenance is no longer accessible it is destroyed, at which point it informs its parents in the announceDeath() method shown in Listing 3.13. This method also contains functionality for manually handling reference-counted garbage collection of Parentage objects.

Listing 3.12: Provenance::announceBirth()

```cpp
void Provenance::announceBirth() {
    if (!m_parentage) return;
    for (unsigned int i=0;i<m_parentpos;++i)
        m_parentage->at(i)->registerChild(this);
}
```

Listing 3.13: Provenance::announceDeath()

```cpp
void Provenance::announceDeath() {
    if (!m_parentage) return;
    /* Firstly, tell all of our parents we’re dying */
    for (unsigned int i=0;i<m_parentpos;i++)
        m_parentage->at(i)->deregisterChild(this);
    /* Manual garbage collection.
    If this is the last Provenance refering to this Parentage
    then we must destroy it. */
    if (!m_parentage->decRefCount()) {
        for (Parentage::iterator it=m_parentage->begin();
```
CHAPTER 3. PROVENANCE IN CXXR

3.4.4 Querying

This section describes the implementation of two R-level functions for querying recorded provenance information: `provenance(object)` and `pedigree(object,...)`.

The `provenance` function

The `provenance(x)` function expects parameter `x` to be of type `Symbol` and returns an R list detailing the provenance of the current binding of `x` (the search for which will begin in the global environment, assuming the function is invoked at the top-level), comprising the following named elements:

- `$command` – The `Expression` whose evaluation gave rise to the binding;
- `$symbol` – The `Symbol` to which the binding pertained;
- `$timestamp` – A string representation of the date and time at which the binding was written;
- `$parents` – A vector of string representations of `x`’s parents’ symbols;
- `$children` – A vector of string representations of `x`’s children’s symbols.

The `pedigree` function

The `pedigree(x)` function expects either a `Symbol` or an `Expression` that when evaluated, yields a string vector representation of `Symbols`, and prints the complete chronological sequence of top-level expressions that resulted in the current binding(s) of those symbols.

Firstly, it creates a `Set S1` of `Provenance` objects attached to `Bindings` resolved from looking up the given `Symbol(s)`, beginning in the global environment. This `Set` is then passed to the `Provenance::ancestors` method (defined below), which returns a `Set S2` containing `Provenance` objects of all ancestors of those given in `S1`. Because `Set` is ordered according to timestamp, `S2` is then traversed from beginning to end, iteratively printing each expression.
Ancestry

Like all representations of provenance data, the parentage records attributed to bindings form a *directed, acyclic graph* between *Provenances*, which can be easily traversed. The method `Provenance::ancestors(Set* open)` is an implementation of Algorithm 3.2 that collates all ancestors of *Provenances* contained in the *Set open*, and is shown in Listing 3.14.

```
Listing 3.14: The Provenance::ancestors(Set*) method
1 Provenance::Set* Provenance::ancestors(Set* open) {
2     Set *closed;
3     closed = new Set();
4
5     while (!open->empty()) {
6         Provenance* n = *(open->begin());
7         Parentage* p = n->getParentage();
8         if (p) {
9             for (unsigned int i = 0; i < n->m_parentpos; i++) {
10                Provenance* s = p->at(i);
11                // If s isn't in closed set, put it in open
12                if (closed->find(s) == closed->end())
13                    open->insert(s);
14            }
15        }
16        open->erase(n);
17        closed->insert(n);
18    }
19    return closed;
20 }
```

3.5 Example

Trivial

The trivial examples in this section will relate to the code given in Listing 3.15.

```
Listing 3.15: Example R code for demonstrating provenance recording and query
1 | > one <- 1
2 | > two <- one + one
```
> three <- 3
> sq <- function(x) x * x
> four <- sq(two)
> nine <- sq(three)

After evaluation of the expressions shown in Listing 3.15, the `provenance` function returns the following for three of the bindings:

Listing 3.16: Output of `provenance(three)`

```r
> provenance(three)
$command
three <- 3

$symbol
three

$timestamp
[1] "14/10/11 18:59:28.297549"

$parents
character(0)

$children
[1] "nine"
```

Listing 3.17: Output of `provenance(sq)`

```r
> provenance(sq)
$command
sq <- function(x) x * x

$symbol
sq

$timestamp
[1] "14/10/11 18:59:44.857478"

$parents
character(0)
```
CHAPTER 3. PROVENANCE IN CXXR

The following examples of the usage of `pedigree` are assumed to have taken place after evaluation of the expressions shown in Listing 3.15.

As Listing 3.19 shows, only those top-level expressions involved in the derivation of the binding `nine` are included in the output of `pedigree(nine)`. The other expressions which were evaluated in between these expressions, whose inclusion would not be ‘harmful’ in the derivation of `nine`, are not included unnecessarily as they are not required as part of its derivation. This is similarly depicted for `pedigree(four)` in Listing 3.20.
pedigree is also able to accept as its argument any expression that results in a character vector, whose elements are (string representations of) symbols. This enables the output of \texttt{ls()} to be used as input to show the pedigree of all current bindings, which (since no bindings have been removed) will give an exhaustive account of the session as shown in Listing 3.21.

\begin{verbatim}
one <- 1
two <- one + one
sq <- function(x) x * x
four <- sq(two)
\end{verbatim}

\begin{verbatim}
> ls()
[1] "four" "nine" "one" "sq" "three" "two"
> pedigree(ls())
one <- 1
two <- one + one
three <- 3
sq <- function(x) x * x
four <- sq(two)
nine <- sq(three)
\end{verbatim}

3.6 Evaluation

Roger Peng’s Air Quality Audit analysis (see Figure 1.6 for workflow; Appendix C for R code) was selected as a sufficiently complex example of a real-world R program. It comprises a moderately large number of statements (50) and operates on a large amount of data (approximately 2.5 million rows x 29 columns).

3.6.1 Provenance Questions

To demonstrate that the provenance questions defined in Section 3.1 can be satisfactorily answered they will hereby be asked of a data object—\texttt{pm1}—that is created in the course of executing the air quality audit analysis.

Answers to questions 1 to 4 are given in the output of the \texttt{provenance()} function; while the \texttt{pedigree()} function offers an answer to question 5 as shown in Listing 3.22.
3.6.2 Performance

The Peng AQA analysis was executed in CXXR with both provenance-tracking enabled and disabled, and for each execution its duration and the memory usage were recorded. Five runs were performed in each Provenance-Aware and Provenance-Unaware CXXR and of the measurements collected the mean and standard deviation were calculated. The results of this performance analysis are given in Table 3.1.

The overall performance impact of the implementation of provenance-awareness to CXXR appears, at least in the case of this example, to be minimal and is less than expected, particularly with respect to execution time. The increase in average memory consumption was expected due to the retention in memory of objects that in ordinary CXXR would either have not existed at all (e.g. Provenance) or have been garbage collected (e.g. Expression) earlier and have therefore existed for shorter periods of time. This result is therefore considered to be very positive.
### Table 3.1 Performance analysis of PA-CXXR vs. CXXR

<table>
<thead>
<tr>
<th></th>
<th>Duration (s)</th>
<th>Memory Usage (MiB)</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>S.D.</td>
<td>Mean</td>
<td>S.D.</td>
<td>Mean</td>
<td>S.D.</td>
</tr>
<tr>
<td>CXXR</td>
<td>302.04</td>
<td>3.08</td>
<td>1220.77</td>
<td>3.92</td>
<td>576.01</td>
<td>5.07</td>
</tr>
<tr>
<td>PA-CXXR</td>
<td>305.93</td>
<td>2.62</td>
<td>1264.80</td>
<td>0.00</td>
<td>620.63</td>
<td>4.01</td>
</tr>
<tr>
<td>Overhead</td>
<td>1.29%</td>
<td>3.61%</td>
<td>7.75%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 3.6.3 PROV Characterisation

In CXXR, the provenance of a binding state is represented by a Provenance object. Associated to the Provenance is a Parentage, which comprises its parents—the Provenances of those bindings that were read prior to the binding’s creation during the course of evaluating a top-level expression.

This representation involves the interconnection of Provenances by a parent relation, and many Provenances may arise from and therefore be attributed to a single top-level expression evaluation. The W3C PROV ontological view of provenance as introduced in Section 1.2.6 has at its core interrelations between entities and activities.

Provenance in CXXR can be characterised in W3C PROV terms by considering the evaluation of a top-level command to be analogous to a PROV activity, and a binding to be a PROV entity. Therefore if expression evaluation E reads binding B1 and then writes binding B2, it would be said that E used B1 and B2 wasGeneratedBy E.

A graphical representation of this characterisation for the example given in Listing 3.15 is shown in Figure 3.15 and described in PROV-N notation by Listing 3.23.

Section 4.5 will describe how W3C PROV information can be automatically extracted from CXXR session information.

---

**Listing 3.23: Example CXXR session represented in PROV-N**

```plaintext
1 document
2 default <http://cxxrexample.org/>
3
4 entity(one)
5 entity(two)
6 entity(three)
7 entity(sq)
```
CHAPTER 3. PROVENANCE IN CXXR

Figures 3.15: Example CXXR session as depicted in PROV.

```plaintext
8  entity(four)
9  entity(nine)
10
11 activity(expr1, [expr="one <- 1"])
12 activity(expr2, [expr="two <- one + one"])
13 activity(expr3, [expr="three <- 3"])
14 activity(expr4, [expr="sq <- function(x) x * x"])
15 activity(expr5, [expr="four <- sq(two)"])
16 activity(expr6, [expr="nine <- sq(three)")
17
18 wasGeneratedBy(one, expr1)
19 wasGeneratedBy(two, expr2)
20 wasGeneratedBy(three, expr3)
21 wasGeneratedBy(sq, expr4)
22 wasGeneratedBy(four, expr5)
23 wasGeneratedBy(nine, expr6)
24
25 used(expr2, one)
26 used(expr5, sq)
27 used(expr5, two)
28 used(expr6, sq)
29 used(expr6, three)
30 endDocument
```

3.6.4 Further Work

At present, there are no records made of details pertaining to the session that would be necessary in order to answer such provenance questions as 6. In particular: who is in control of the session (and is therefore responsible for causing a binding to be in a particular
state), or in what environment the binding state occurred. Such information regarding the user that might be of interest could include: user name or login; full-name; e-mail address. Information regarding the environment could include: CXXR version; loaded packages (and their versions); operating system and its version; host name of computer. One strategy for modelling these details would be to represent them as PROV Agents.
Chapter 4

Serialisation

4.1 Introduction

Serialisation is the process of transforming data structures that constitute object or program state into a format suitable for persistent storage in a file or transmission on a network, with the intention that this data may later undergo the reverse of this process—deserialisation—to restore data or program state in either the same or a different environment.

In this way, data analysis sessions may be saved and later resumed. Data that persists across more than one session will be referred to as cross-session.

This chapter addresses research goals (Section 1.5.1) number 4 by introducing cross-session provenance-awareness, and number 5 by considering how to enable interoperability of provenance captured by CXXR.

This chapter will:

• Re-cap the use case for cross-session provenance awareness;

• Look at emerging standards for serialisation of provenance;

• State the design objectives of a cross-session provenance-aware CXXR;

• Describe how this facility has been implemented;

• Describe how serialised provenance information can be processed to offer interoperability.

CXXR has inherited serialisation and deserialisation facilities from CR. CR’s facilities for serialisation and deserialisation of data objects offer a variety of options: from which objects are serialised—one object per file, numerous objects, or the entire session; to where
the serialisation occurs—a flat file, a compressed file, or even a HTTP/FTP connection; and also file format—ASCII, Binary or XDR (a big-endian format). An example of how cross-session CR looks is given in 4.1.

Listing 4.1: Cross-session R session example

```r
> sq <- function(x) x*x
> three <- 3
> nine <- sq(three)
> save.image(file="cas.RData")
> q()
```

4.1.1 Use Case

The motivating use case for cross-session provenance-awareness was outlined in Section 1.5, which describes a scenario in which a user returns to an analysis that was conducted some time ago for which there were no contemporaneous notes made, therefore the user wishes to ask provenance questions of the session to determine how the present data objects arrived in their respective states. A likely variation on this scenario is when a user is in receipt of an analysis conducted by a third-party and has a similar desire to ask provenance questions.

The provenance questions to be answered in this scenario remain the same as those defined in Section 3.1.

At the moment, only the data is serialised by the serialisation facility CXXR inherited from CR—attributed provenance information is not included.
4.1.2 Serialisation of Provenance

As introduced in Chapter 1, the principal objective of the W3C Provenance Working Group was to define a provenance interchange language (PIL) and accordingly publish W3C Recommendations to this effect. The Recommendation describing the central data model for representation of provenance information is PROV-DM [79].

Being a conceptual data model, PROV-DM does not prescribe—or describe—a format for the serialisation of provenance information. Three separate publications were made relating to serialisation: PROV-O: the PROV Ontology [64], which expresses PROV-DM in OWL2 Web Ontology Language, allowing mapping of PROV-DM to RDF (the serialisation of which may occur to several formats, principally XML); PROV-N: the Provenance Notation [80], which is serialisation of provenance intended for human consumption; and PROV-XML [112], which describes an XML schema allowing instances of PROV-DM to be serialised as XML.

The Working Group permitted a Member Submission PROV-JSON [52], which specifies a serialisation format for the representation of PROV-DM as JSON (JavaScript Object Notation), and focusses its attention on allowing the interchange of provenance information between web services and clients.

4.2 Design

Serialising the bindings belonging to a CXXR session necessitates saving the state of each binding, including its related attributes: the symbol, value, etc. and their attributes, and so on until every object that is required to restore the session has been saved.

4.2.1 Interpreter State

There is an important distinction to be made between certain types of objects in CXXR: those that are session-specific and those that are session-independent.

The former category comprises objects that are used to represent aspects of the interpreter’s state, which is established at the beginning of the session, and remains for the session’s duration. This type of object is not directly transferable from one session to another for reasons that are best illustrated by way of example.

By the time a user operating a CXXR session S1 decides to load into it a previously-saved session S0 the state of S1 has already been established. One such element of state is the global environment, which is among the first objects to be created on initialisation.
of the interpreter, and to which there are countless references from other objects. These
references would be broken and cause corruption of the interpreter state if, when S0 is
loaded, the global environment from S0 were to simply replace that of S1. Similarly,
of the objects in S0, any references to the global environment would not be well-served
notionally by loading the global environment of S0 into S1 without having it replace the
global environment.

More generally: if an object O refers to some aspect of interpreter state S, the salient
notion is that O makes a reference to the S of the session in which it exists, and not the
specific instance of S in the session in which it was created.

These objects are therefore termed session-specific as they cannot feasibly be trans-
planted from one session to another without either losing their semantic significance or
invalidating aspects of interpreter state. All other objects are session-independent.

Session-specific objects

The session-specific objects are currently:

- Two instances of Environment: the global environment and the base environment;
- All instances of Symbol;
- All instances of CachedString.

The global and base environments are established during the initialisation of the in-
terpreter and are fixed for the lifetime of the session. An instance of the Symbol class is
a necessarily unique textual representation of an identifier name; similarly CachedString
does likewise for arbitrary strings. For example, if one creates a binding in the global
environment to the symbol seq, this binding will reference precisely the same instance of
Symbol as the binding to the standard library function defined in the base environment.

When it comes to serialising an object of this type, it is not the referenced object itself
that needs to be preserved; rather the fact that it is a reference to some object—it should
therefore be serialised by reference.

Session-independent objects

All other objects are session-independent as their state within one session can be preserved
and subsequently loaded into a new session without any harmful side-effects.

The information that should be preserved for a session-independent object is its value—
it should therefore be serialised by value.
4.2.2 Design Objectives

The design considerations that influence the approach taken here are as follows:

- **Interpreter State.** As outlined above, the separation between serialise-by-value and serialise-by-reference objects needs to be handled;

- **Output Format.** This should be flexible and provided by an opaque ‘Archive’ type whereby the code that initiates serialisation of a data item should not be aware of how, to where, or in what format the data will actually be stored;

- **Versioning.** The serialisation process should allow for changes to attributes as the interpreter internals evolve;

- **Code Encapsulation.** CXXR promotes an ethos of extensibility, and affords developers the ability to introduce easily new data types into the RObject hierarchy. The serialisation/deserialisation functionality needs to be encapsulated with the code and attributes to which it pertains;

- **C++ features.** A robust solution should cope with the features of C++ that CXXR employs: inheritance, pointers, templates, smart pointers, managing object instances.

4.2.3 Algorithms

The two highest-level operations are serialisation and deserialisation whose functions here are to respectively save all of the bindings in the global environment, and load saved bindings into the current global environment. These two operations are outlined in Algorithms 4.1 and 4.2. They each utilise an algorithm to import bindings from the frame of one environment to another (Algorithm 4.3).

**Algorithm 4.1** CXXR session serialisation algorithm

1: procedure SERIALIZE
2: \( A \leftarrow \text{NEW OUTPUT ARCHIVE} \)
3: \( E \leftarrow \text{NEW ENVIRONMENT} \)
4: \( \text{IMPORT(R\_GlobalEnv.frame, E.frame)} \quad \triangleright \text{Import bindings from global env.} \)
5: \( A.\text{serialize}(E) \)
Algorithm 4.2 CXXR session deserialisation algorithm

1: \textbf{procedure} \textsc{Deserialize} \\
2: \hspace{1em} A \leftarrow \text{NEW INPUT ARCHIVE} \\
3: \hspace{1em} E \leftarrow A.\textsc{deserialise} \\
4: \hspace{1em} \textsc{Import}(E.frame, R\_GlobalEnv.frame) \quad \triangleright \text{Import bindings to global env.}

Algorithm 4.3 Import bindings algorithm

1: \textbf{procedure} \textsc{Import}(srcFrame, destFrame) \\
2: \hspace{1em} \textbf{for all} \ (srcSymbol, srcBinding) \in srcFrame \ \textbf{do} \\
3: \hspace{2em} destBdg \leftarrow \text{destFrame.\textsc{obtainBinding}}(srcSymbol) \quad \triangleright \text{Locate binding} \\
4: \hspace{2em} destBdg.m\_provenance \leftarrow srcBdg.m\_provenance \quad \triangleright \text{Set Provenance} \\
5: \hspace{2em} destBdg.m\_value \leftarrow srcBdg.m\_value \quad \triangleright \text{Set value}

Session-(in)dependent Objects

The handling of session-dependent objects described in the previous section is detailed in Algorithms 4.4, 4.5, 4.6, and 4.7. To signify references to particular session-dependent environments—global and base—and special symbols—unbound value, missing argument, and restart token—integer identifiers are used.

4.3 Implementation

C and C++ offer no native features for handling of serialisation.

Consequently, CR defines its own mechanisms for handling serialisation and deserialisation, which have so far been inherited by CXXR.

The serialisation facilities introduced here will utilise the \texttt{boost::serialization} library, which the following section will introduce and explain why this library satisfies the design criteria described above.

4.3.1 \texttt{boost::serialization}

Boost is a collection of peer-reviewed, highly-portable, C++ libraries that cover a vast range of applications [1]. Boost has an emphasis on creating libraries that are highly compatible the C++ standard library. Such has been its success, its libraries have been adopted as part of the most recent C++11 standard, and many of the concepts, classes and functions introduced in its libraries are increasingly being incorporated into the C++ Committee’s Library Technical Reports for consideration for inclusion in future definitions.
Algorithm 4.4 Object Serialisation/Deserialisation algorithms

1: procedure SerialiseObject(O, A) ▷ Serialise object O to archive A
2: \( T \leftarrow \text{TypeOf}(O) \)
3: \( A.\text{serialize}(T) \)
4: if \( T = \text{`Environment'} \) then
5: \( \text{SaveEnvironment}(O, A) \)
6: else if \( T = \text{`Symbol'} \) then
7: \( \text{SaveSymbol}(O, A) \)
8: else if \( T = \text{`CachedString'} \) then
9: \( \text{SaveCachedString}(O, A) \)
10: else
11: \( A.\text{serialize}(O) \)
12: procedure DeserialiseObject(A) ▷ Deserialise an object from archive A
13: \( T \leftarrow A.\text{deserialise} \)
14: if \( T = \text{`Environment'} \) then
15: \( O \leftarrow \text{LoadEnvironment}(A) \)
16: else if \( T = \text{`Symbol'} \) then
17: \( O \leftarrow \text{LoadSymbol}(A) \)
18: else if \( T = \text{`CachedString'} \) then
19: \( O \leftarrow \text{LoadCachedString}(A) \)
20: else
21: \( O \leftarrow A.\text{deserialise} \)

of the C++ language standard.

Boost’s libraries comprise libraries that are header-only such as \texttt{boost::circular\_buffer} and \texttt{boost::graph} that need to only be \#included; and those that require compilation into library objects, against which programs that wish to utilise their functionality must be linked. Examples of this type include \texttt{boost::regex}, \texttt{boost::thread} and \texttt{boost::serialization}.

CXXR already makes use of Boost elsewhere—specifically \texttt{boost::regex}, although \texttt{regex} is now part of C++11’s standard library—so there is in fact no additional dependency on a separate third-party library collection being introduced here.

This section will introduce the key features of \texttt{boost::serialization}, and how these are particularly well-suited to the requirements for serializing CXXR’s object hierarchy and provenance information.
CHAPTER 4. SERIALISATION

Algorithm 4.5 Serialise/Deserialise Environment

1: procedure SaveEnvironment(E, A)  ▷ Serialise E to archive A
2: if $E = R\_GlobalEnv$ then
3:     $EnvType \leftarrow 1$
4: else if $E = R\_BaseEnv$ then
5:     $EnvType \leftarrow 2$
6: else
7:     $EnvType \leftarrow 0$  ▷ All other environments
8:     A.serialize($EnvType$)
9: if $EnvType = 0$ then
10:    A.serialize($E$)
11: procedure LoadEnvironment(A)  ▷ Deserialise Environment from archive A
12: $EnvType \leftarrow$ A.deserialise
13: if $EnvType = 1$ then
14:     return $R\_GlobalEnv$
15: else if $EnvType = 2$ then
16:     return $R\_BaseEnv$
17: else
18:     $E \leftarrow$ A.deserialise
19: return $E$

Archives

An Archive is considered to be any stream of bytes that may have be embodied in any underlying file format, and comprises a complementary pair of interfaces for access: one each for data output and input. Boost provides specimen Archive types to cater for many needs: text, XML, and binary; however, the user is not restricted to these formats and may instead opt to extend Archive into some other bespoke format. This mechanism allows an application to treat any Archive in a unified manner and therefore be entirely format-agnostic, thereby not needing to be concerned with the fundamentals of how the data is being represented at a low level. The Archive interface is designed with the intention that it can easily be extended. It would therefore be possible to create an Archive that enabled representation of provenance information as PROV-O, RDF or even PROV-N.

Archives are constructed with either an \texttt{std::istream} for an input archive, or \texttt{std::ostream} for an output archive. These could typically be \texttt{std::ifstream} and
Algorithm 4.6 Serialise/Deserialise Symbol

1: procedure SaveSymbol(S, A) ▷ Serialise S to archive A
2: if $S = \text{R\_MissingArg}$ then ▷ Special Symbols
3: $\text{SymType} \leftarrow 1$
4: else if $S = \text{R\_RestartToken}$ then
5: $\text{SymType} \leftarrow 2$
6: else if $S = \text{R\_UnboundValue}$ then
7: $\text{SymType} \leftarrow 3$
8: else
9: $\text{SymType} \leftarrow 0$ ▷ All other symbols
10: A.serialize(\text{SymType})
11: if $\text{SymType} = 0$ then
12: A.serialize(S.toString())
13: procedure LoadSymbol(A) ▷ Deserialise Symbol from archive A
14: $\text{SymType} \leftarrow \text{A.deserialise}$
15: if $\text{SymType} = 1$ then
16: return $\text{R\_MissingArg}$
17: else if $\text{SymType} = 2$ then
18: return $\text{R\_RestartToken}$
19: else if $\text{SymType} = 3$ then
20: return $\text{R\_UnboundValue}$
21: else
22: string $\leftarrow \text{A.deserialise}$
23: return Symbol.obtain(string)

\texttt{std::ofstream} respectively for input from and output to files. Again this abstraction from underlying streams allows Archives to be constructed from streams that do not necessarily exist as files on a filesystem.

**Serializable classes**

A class is considered to be **serializable** if it either (i) implements an appropriate \texttt{serialize} member method:
Algorithm 4.7 Serialise/Deserialise CachedString

1: procedure `SaveCachedString`(C, A)  \(\triangleright\) Serialise C to archive A
2: \hspace{1em} A.serialise((C.string, C.encoding))
3: procedure `LoadCachedString`(A)  \(\triangleright\) Deserialise CachedString from archive A
4: \hspace{1em} (string, encoding) \(\leftarrow\) A.deserialise()
5: \hspace{1em} return CachedString.Obtain(string, encoding)

```
template<class Archive>
void serialize(Archive &ar, const unsigned int version);
```

or (ii) is provided with an appropriate free-standing `serialize` function, in accordance with the following prototype:

```
template<class Archive>
inline void serialize(
    Archive& ar,
    MyClass& t,
    const unsigned int file_version
) {
    [...]  
}
```

The former method ensures very tight code encapsulation, as the serialisation-related code is contained as part of the class definition. In cases where modifying the class definition is not possible, the latter method allows these class types to be made `Serializable`.

C++ primitive types are also considered `Serializable`.

Serializing members

`Archive` types overload the operators `<<` and `>>` for the respective purposes of writing to an output archive, and reading from an input archive. `Archive` types also polymorphically overload the binary operator `&`, so that it may be used in place of `<<` for an output archive, and `>>` for an input archive.

This enables the `serialize` member method to handle both serialisation and deserialisation with the same code, such as that shown in the example Listing 4.2.

Listing 4.2: Example definition of a class that de/serialises its member variables
Serializing Pointers

It is possible to serialise an object through a `pointer` to that object, for example a class that has a `Club*` member might serialise it as follows:

```cpp
class Player {
private:
    Club *club;

    template<class Archive>
    void serialize(Archive& ar, const unsigned int version) {
        ar & club;
    }
};
```

It is of course possible that an object is referenced by more than one pointer, for instance more than one `Player` will play for the same `Club`. In such a case, an instance will be only be recorded once in the archive. During deserialisation, a new object will be created into which its contents are loaded. Any subsequent deserialisation of pointers to this object will not result in the creation of duplicate instances of it; instead these pointers will be set to reference the pre-existing instance.

Polymorphic pointers are also handled in this fashion and will be discussed later.
Constructing Archives

As previously mentioned, an Archive is used for either input from or output to some particular format, typically to a file. In the case of the standard Boost specimen Archives, whether they are for input or output is indicated by their name, such as the XML Archives `xml_iarchive` for input and `xml_oarchive` for output, which like all Boost Archives require a corresponding file stream. Listing 4.3 exemplifies how XML input and output Archives may be created with the file `serialize.xml` and used to serialise and deserialise an object. In the case of XML archives, this process has one additional step to attribute a name to the XML tag of the object being serialised. This example uses a `boost` macro that simply generates a name based on that of the class.

```
1 Club *club;
2 ...
3 ofstream ofs("serialize.xml");
4 boost::archive::xml_oarchive oa(ofs);
5 oa << BOOST_SERIALIZATION_NVP(club); // Serialize 'club' to XML archive
6 ...
7
8 ifstream ifs("serialize.xml");
9 boost::archive::xml_iarchive ia(ifs);
10 ia >> BOOST_SERIALIZATION_NVP(club); // Deserialise XML archive to 'club'
```

Versioning

A particularly valuable feature of Boost::serialization’s approach to devolution of serialisation responsibility to each class, is that each class is able to evolve independently: there is no overall, application-wide version of the serialisation format, instead versioning is performed on a class by class basis. This is supported by use of the version parameter of the serialize method. When an archive incorporates a serialised instance of a class, the archive also records which serialisation version number of this class was used. This means that as a class changes over time, for instance when it incorporates new member fields, it can increment its serialisation version number to reflect the change in its own format, while at the same time it retains its ability to accommodate deserialising its previous versions.
It does this by determining **at the time it restores its data** what constituents pertain to the version with which it is presented.

**Splitting Save/Load**

The examples shown so far have presented serialisation and deserialisation as operations that differ only in the direction the data is flowing—everything serialised is deserialised in the same order. This has enabled both operations to be represented by the same `serialize` method, with the context-sensitive operator `&`. In the case of **multiple versioning**, these operations are unlikely to remain identical: while serialisation will occur to only one format that represents the current version of a class, deserialisation will need to account for all of the previous versions of a class. In that case, the solution is to **split** save and load into separate methods.

This requires that the `serialize` method is defined as follows to announce to the library that the save and load operations will be handled separately:

```cpp
define serialize(Archive& ar, const unsigned int version) {
        boost::serialization::split_member(ar, *this, version);
    }
```

The class then needs to implement methods with the following signatures:

```cpp
void save(Archive& ar, const unsigned int version) const;
void load(Archive& ar, const unsigned int version);
```

Listing 4.4 shows the previously-encountered example class `Club` has since been augmented with an additional member field `homeground`, and how this is class uses split save/load members to cater for this difference in versions.

Listing 4.4: Split save and load from `Club` serialize

```cpp
#include <boost/serialization/split_member.hpp>
#include <boost/serialization/version.hpp>

class Club { [...] constructor omitted [...] }
private:
    friend class boost::serialization::access;

    std::string officialname;
    int yearfounded;
    Ground* homeground;
```
template<class Archive>
void serialize(Archive& ar, const unsigned int version) {
    boost::serialization::split_member(ar, *this, version);
}

template<class Archive>
void save(Archive& ar, const unsigned int version) const {
    ar << officialname;
    ar << yearfounded;
    ar << homeground;
}

template<class Archive>
void load(Archive& ar, const unsigned int version) {
    ar >> officialname;
    ar >> yearfounded;
    if (version > 0)
        ar >> homeground;
}

BOOST_CLASS_VERSION(Club, 1);

Inheritance

A facility for serialising the superclass of a class is defined in header base_object.hpp and is invoked as illustrated in Listing 4.5;

Listing 4.5: Serialize a superclass

```cpp
#include <boost/serialization/split_member.hpp>
#include <boost/serialization/base_object.hpp>

class FootballClub : public Club {
    [
    private:
        friend class boost::serialization::access;
    Division* division;
}
```
This enables subclass A to pass the serialisation/deserialisation operation to its superclass B. However this does not cater for the case where an object of dynamic type A is being serialised via a pointer or reference of static type B, in which case it is necessary to downcast from B to A.

boost::serialization handles the serialisation of polymorphic pointers like this automatically, through a process of registration of derived classes. In the case that A is a derived class of B and it is intended that objects of type A are to be serialised through polymorphic pointers of type B, then A must first be registered using one of a few macros defined in boost/serialization/export.hpp. Our exemplar derived class FootballClub could be registered as follows:

```
BOOST_CLASS_EXPORT_KEY(FootballClub);
```

The following example shows how this registration enables a Player’s club, stored in a variable whose static type is Club and whose dynamic type is FootballClub, to be serialised/deserialised.
class Player {
    Club* club;

    template<class Archive>
    void serialize(Archive& ar, const unsigned int version) {
        ar & club;
    }
}

In particular, the type of the object instantiated when `club` is deserialised will be `FootballClub`.

### 4.3.2 Provenance Containers

**Provenance**

The serialisation functionality within class `Provenance` is split into a save/load pair to enable the necessary house-keeping to be performed when reconstructing a `Provenance` object.

The central data structure for provenance information is the class `Provenance`, whose objects are related to each other by way of a `Parentage` class (introduced in Section 3.4.2). The serialisation method for the `Provenance` class is shown in Listing 4.6, in which the data types of non-primitive variables have been annotated in comment. It is responsible for serialising its six member fields: two fields comprising the `timestamp` associated with its creation time; a `GCEdge` connecting to the `expression` associated with this provenance object; a `GCEdge` connecting to the `symbol` with which this provenance object is associated; a pointer to a `Parentage` object; and the `position` within that parentage.

**Listing 4.6: The `Provenance::save` method**

```
template <class Archive>
void save(Archive & ar, const unsigned int version) const {
    ar << boost::serialization::base_object<GCNode>(*this);
    ar << m_timestamp.tv_sec;
    ar << m_timestamp.tv_usec;
    ar << m_expression;  // GCEdge<Expression>
    ar << m_parentpos;
    ar << m_symbol;      // GCEdge<Symbol>
}
The deserialisation method for the Provenance class is shown in Listing 4.7. It rein-
states the class member fields from the archive, and also does some necessary house-keeping
activities:

- Establish an empty set of children;
- Increment the reference count of the associated parentage;
- Announce its birth to its parents.

Listing 4.7: The Provenance::load method

```cpp
template <class Archive>
void load(Archive & ar, const unsigned int version) {
    ar >> boost::serialization::base_object<GCNode>(*this);

    ar >> m_timestamp.tv_sec;
    ar >> m_timestamp.tv_usec;
    ar >> m_expression;
    ar >> m_parentpos;
    ar >> m_symbol;
    ar >> m_parentage;
    m_children=new Set();
    m_parentage->incRefCount();
    announceBirth();
}
```

Parentage

Similarly to Provenance, the serialisation and deserialisation methods of Parentage are
asymmetrical and so require splitting into save/load pairs.

As illustrated in Listing 4.8 saving a Parentage object consists of firstly saving its
size, and then iteratively saving each of its constituent Provenance objects.

Listing 4.8: The Parentage::save method
CHAPTER 4. SERIALISATION

110

1 template<class Archive>
2 void save(Archive & ar, const unsigned int version) const {
3 unsigned int sz=size();
4 ar << sz;
5 for (unsigned int i=0;i<sz;i++) {
6     Provenance *p=at(i);
7     ar << p;
8 }
9 }

10 Loading a Parentage object is a little more involved: its load method firstly retrieves from the archive the size of the Parentage in order to determine how many Provenance objects reside in the archive and are to due to be read. Each of the Provenance objects is then retrieved from the archive, exposed to the garbage collector, and pushed into the parentage. The relevant code is given in Listing 4.9.

Listing 4.9: The Parentage::load method

1 template<class Archive>
2 void load(Archive & ar, const unsigned int version) {
3 unsigned int sz;
4 ar >> sz;
5 for (unsigned int i=0;i<sz;i++) {
6     Provenance *p;
7     ar >> p;
8     GCNode::expose(p);
9     pushProvenance(p);
10 }
11 }

Frame

The frame, as introduced in Section 2.2.6, is the component of an environment which maps symbols to bindings, which in turn have an associated value. In order to incorporate the bindings of one frame into another, a method import has been introduced to Frame. The definition of this method in its derived class StdFrame is shown in Listing 4.10, and operates as described in Algorithm 4.3.

Listing 4.10: The StdFrame::import method
4.3.3 User-Level Functions

A new R-level function \texttt{bserialize} has been introduced to allow this functionality to coexist with existing R functions for serialisation. The \texttt{bserialize} function implements the functionality of Algorithm 4.1.

The complementary R-level function for deserialisation that has been introduced is \texttt{bdeserialize}. The \texttt{bdeserialize} function implements the functionality described in Algorithm 4.2.

4.3.4 Session-dependent Objects

A strategy for handling serialisation/deserialisation of CXXR’s session-dependent objects, which were introduced in Section 4.2.1, was described by algorithms in Section 4.2.3.

The implementation of the strategy described in Algorithm 4.4 takes advantage of the memory management technique that GCNodes use to refer to each other—the GCEdge. At the time of its serialisation, a GCEdge inspects its target to determine its type and if it might need to serialise it by reference, in which case it serialises an edge type identifier and marshals control to the appropriate dedicated method for serialising the object by reference. Otherwise, it serialises the target by value.

Deserialisation, as described in Algorithm 4.2 works as the converse of this process, and uses the edge type identifier to determine whether the object needs to be deserialised by value or by reference, and retargets the GCEdge accordingly.

This functionality is implemented in GCEdge’s base class, GCEdgeBase.
GCEdgeBase

GCEdgeBase defines an enumerated data type EdgeSerializationType that will be used as an edge type identifier:

```cpp
enum EdgeSerializationType {
    OTHEREDGE = 0, SYMBOLEDGE,
    CACHEDSTRINGEDGE, ENVIRONMENTEDGE
};
```

A member method shown in Listing 4.11 is introduced to disambiguate the type of a GCEdgeBase’s target, and resolve it to one of the given values in EdgeSerializationType.

```cpp
GCEdgeBase::EdgeSerializationType GCEdgeBase::serializationType() const {
    if (!m_target) return OTHEREDGE;
    switch (typeid(*m_target)) {
        case typeid(Symbol): return SYMBOLEDGE;
        case typeid(CachedString): return CACHEDSTRINGEDGE;
        case typeid(Environment): return ENVIRONMENTEDGE;
    }
    return OTHEREDGE;
}
```

Serialisation

The serialisation method of GCEdgeBase as shown in Listing 4.12. Firstly the edge type identifier of the current GCEdgeBase is determined and written to the archive.

```cpp
template<class Archive>
void save(Archive & ar, const unsigned int version) const {
    EdgeSerializationType type=serializationType();
    ar << type;
    switch(type) {
        case CACHEDSTRINGEDGE:
            saveCachedString(ar, m_target);
    }
}
```
CHAPTER 4. SERIALISATION

Following that, different methods are called depending upon the type of the target:

**CachedString** : Saves to the archive an `std::pair` comprising: an `std::string` representation of the cached string; and an internal encoding value.

**Environment** : In order to identify special environments—namely the global and base environments—an identifier based on an enumerated data type `EnvironmentSerializationType` is first saved to the archive, and if the environment is not deemed special, then in addition its contents is written to the archive, as illustrated in Listing 4.13.

**Symbol** : The handling of symbols is similar to that of environments as it too has to consider representing its special values: `R_MissingArg`, `R_RestartToken` and `R_UnboundValue`. An identifier representing which (if any) of these cases pertains to the current symbol is written to the archive, and if a special value is not being represented, then an `std::string` representation of the symbol is also saved to the archive.

**Other targets** are simply written to the archive. This will invoke the relevant `serialize` method—either a member of the appropriate class (which almost certainly will be derived from `GCNode`) or a free-standing method pertaining to the appropriate class.

```
Following that, different methods are called depending upon the type of the target:

**CachedString** : Saves to the archive an `std::pair` comprising: an `std::string` representation of the cached string; and an internal encoding value.

**Environment** : In order to identify special environments—namely the global and base environments—an identifier based on an enumerated data type `EnvironmentSerializationType` is first saved to the archive, and if the environment is not deemed special, then in addition its contents is written to the archive, as illustrated in Listing 4.13.

**Symbol** : The handling of symbols is similar to that of environments as it too has to consider representing its special values: `R_MissingArg`, `R_RestartToken` and `R_UnboundValue`. An identifier representing which (if any) of these cases pertains to the current symbol is written to the archive, and if a special value is not being represented, then an `std::string` representation of the symbol is also saved to the archive.

**Other targets** are simply written to the archive. This will invoke the relevant `serialize` method—either a member of the appropriate class (which almost certainly will be derived from `GCNode`) or a free-standing method pertaining to the appropriate class.

```

Listing 4.13: The `saveEnvironment` method
```

```
```
CHAPTER 4. SERIALISATION

Deserialisation of GCEdgeBase is the logical reverse of the serialisation process, and is described in Listing 4.14. First the edge type identifier is loaded from the archive and is then used to determine the appropriate handling routine. Each of the methods load{CachedString,Environment,Symbol} returns a pointer which is used to retarget the current GCEdgeBase.

Listing 4.14: The GCEdgeBase::load method
4.4 Evaluation

This section demonstrates how the serialisation/deserialisation facility of Provenance-Aware CXXR is used, and how provenance questions (Section 3.1) can be answered of data objects that have been restored from a previously-saved session.

4.4.1 Illustrative Example

In the interest of brevity of XML output, this example is necessarily short:

```r
> myVar <- "Hello, XML Serialization"
> myVar <- paste0(myVar, "!")
> myVar
[1] "Hello, XML Serialization!"
> bserialize()
> q()

This session can later be recalled as follows:

```r
> bdeserialize()
> ls()
[1] "myVar"
> myVar
[1] "Hello, XML Serialization!"
> pedigree("myVar")
[[1]]
myVar <- "Hello, XML Serialization"
[[2]]
myVar <- paste0(myVar, "!")
```

The corresponding XML output for only the first top-level command can be found in Appendix B.

4.4.2 Real-World Example

The by now familiar Peng AQA analysis was serialised after being performed in a CXXR session. The serialisation resulted in an XML file of size 3.5GB, comprising approximately 155 million XML elements. Being a text-based format, this XML file is highly compressible. The GNU *tar* program, for instance, is able to compressed the XML file to an
altogether more reasonable 23MB using gzip compression.

It is possible to recall the session thus:

```r
> bdeserialise()
> ls()
```

```
[1] "both"  "both.county"  "both.id"  "cnames"
[5] "cnt0"  "cnt1"  "dates"  "dates0"
[9] "dates1"  "missing.months"  "negative"  "pm0"
[13] "pm0sub"  "pm1"  "pm1sub"  "rng"
[17] "site0"  "site1"  "tab"  "x0"
[21] "x0sub"  "x1"  "x1sub"
```

And ask provenance questions of individual objects that have been restored:

```
> provenance(x1sub)
```

```
$command
x1sub <- pm1sub$Sample.Value

$symbol
x1sub

$timestamp
[1] "21/07/14 12:49:57.855978"

$parents
[1] "pm1sub"

$children
[1] "rng"
```

Or question the sequence of commands used to generate all the objects:
> pedigree(ls())

pm0 <- read.table("pm25_data/RD_501_88101_1999-0.txt", comment.char = ",", 
                   header = FALSE, sep = "|", na.strings = """)
cnames <- readLines("pm25_data/RD_501_88101_1999-0.txt", 1)
cnames <- strsplit(cnames, "|", fixed = TRUE)

names(pm0) <- make.names(cnames[[1]])
x0 <- pm0$Sample.Value

pm1 <- read.table("pm25_data/RD_501_88101_2012-0.txt", comment.char = ",", 
                   header = FALSE, sep = "|", na.strings = """)
names(pm1) <- make.names(cnames[[1]])
x1 <- pm1$Sample.Value

negative <- x1 < 0
dates <- pm1$Date
dates <- as.Date(as.character(dates), "%Y%m%d")

missing.months <- month.name[as.POSIXlt(dates)$mon + 1]
tab <- table(factor(missing.months, levels = month.name))
site0 <- unique(subset(pm0, State.Code == 36, c(County.Code, 
                  Site.ID)))
site1 <- unique(subset(pm1, State.Code == 36, c(County.Code, 
                  Site.ID)))
site0 <- paste(site0[, 1], site0[, 2], sep = ".")
site1 <- paste(site1[, 1], site1[, 2], sep = ".")
both <- intersect(site0, site1)

pm0$county.site <- with(pm0, paste(County.Code, Site.ID, sep = "."))

pm1$county.site <- with(pm1, paste(County.Code, Site.ID, sep = "."))
cnt0 <- subset(pm0, State.Code == 36 & county.site %in% both)
cnt1 <- subset(pm1, State.Code == 36 & county.site %in% both)
both.county <- 63
both.id <- 2008

pm1sub <- subset(pm1, State.Code == 36 & County.Code == both.county & 
                  Site.ID == both.id)

pm0sub <- subset(pm0, State.Code == 36 & County.Code == both.county & 
                  Site.ID == both.id)
dates1 <- as.Date(as.character(pm1sub$Date), "%Y%m%d")
x1sub <- pm1sub$Sample.Value
dates0 <- as.Date(as.character(pm0sub$Date), "%Y%m%d")
x0sub <- pm0sub$Sample.Value
rng <- range(x0sub, x1sub, na.rm = T)
4.5 Provenance Interchange

This chapter has so far detailed serialisation of a CXXR session in which provenance information attributed to data items within that session is preserved alongside the data to which it pertains, so that at a later time the session may be restored, complete with data and provenance.

This is only one potential use of provenance information serialised from CXXR. This section will describe how it is possible to enable interoperability of provenance by extracting from a saved CXXR session, provenance information in the W3C PROV-O format described in Section 1.2.6.

4.5.1 Design

By serialising to XML using Boost, CXXR allows its saved sessions to be processed as regular XML documents. One particular advantage offered by these means is Boost’s attribution of identifiers to each type of C++ class encountered as well as each instance of an object. It is therefore practical to reconstruct from XML the graph of objects (i.e. instances of C++ classes) that compose the CXXR session with provenance information, so that this may then be written in some other format. PROV-O is an OWL2 Web Ontology Language (OWL2) ontology expressing the PROV Data Model, whose namespace is defined as http://www.w3.org/ns/prov# and—as is common practice—this namespace will herein be bound to the prefix prov. As an OWL2 ontology, PROV-O can be serialised in a variety of syntaxes including but not limited to RDF/XML, OWL/XML, and Turtle. Turtle is something of a de facto standard syntax for representing PROV-O; therefore it is the chosen syntax into which CXXR provenance will be transformed.

Classes

To recap Section 1.2.6: a prov:Entity class is analogous to a CXXR::Provenance—a representation of a binding state—and a prov:Activity class is analogous to a top-level expression evaluation—subsequently represented in this implementation by class CXXR::CommandChronicle, which implements the combined concepts of parentage and expression.

Properties

A CommandChronicle also comprises references to those objects that were read in the course of its expression evaluation, each of which can be said—in PROV terms—to have
been used by the Activity. The `prov:wasGeneratedBy` attribute of a `prov:Entity` is used to represent a Provenance reference to a CommandChronicle.

**Labels**

To assist with human-readability, both `prov:Entity` and `prov:Activity` may have `label` attributes, which are expressed in PROV-O using the `rdfs:label` property. The label of a CommandChronicle will be a string representation of the expression; the label of a Provenance will be the string representation of the symbol of the binding to which it pertains.

**Identifiers**

Each of the resources described (i.e. `prov:Activity`s and `prov:Entity`s) will necessarily be given identifiers: Universal Resource Identifiers (URIs) whose namespace will be `http://cs.kent.ac.uk/projects/cxxr#`. An RDF statement is in the form of a `subject-predicate-object` Triple. The identifiers here are used to reference resources in the subject and object components of statements. In many cases the predicates will be from the `prov` or `rdfs` namespaces. At present, CXXR does not attribute identifiers to data objects, or the session itself and this will be discussed further in Section 4.5.4; however, for the purpose of constructing valid PROV-O, these identifiers can be automatically generated during conversion from XML.

**XML Parsing**

As previously alluded to in Section 4.4, the XML document produced by CXXR session serialisation is verbose; the result of running the example given in the previous section during which only one assignment is performed, is an XML document consisting of 113 lines and the length of the output file grows considerably with each additional statement evaluated in the session. An analysis of modest size could quite feasibly produce an XML serialisation of several gigabytes, which when loaded by an XML parsing library will further incur a considerable memory overhead; therefore, it is not practical to implement an XML parsing strategy that would necessitate an in-memory representation of the entire XML tree. An alternative to this is to employ a stream (or iterative) parsing approach, in which events such as `node start` or `node end` are processed. This technique affects a depth-first traversal of the XML tree, and this design choice allows flexibility in which selecting which elements forming sub-trees are retained in memory.
Listing 4.15 shows the first few lines from an XML-serialised CXXR session. The events encountered while parsing this extract would be:

- Event: *start*; Element tag: `boost_serialization`
- Event: *start*; Element tag: `env`
- Event: *start*; Element tag: `RObject`
- Event: *start*; Element tag: `GCNode`
- Event: *end*; Element tag: `GCNode`
- Event: *start*; Element tag: `type`
- Event: *end*; Element tag: `type`
- Event: *start*; Element tag: `m_attrib`
- Event: *end*; Element tag: `m_attrib`
- Event: *end*; Element tag: `RObject`
- Event: *start*; Element tag: `envtype`
- Event: *end*; Element tag: `envtype`

Listing 4.15: XML extract to illustrate parsing events

```xml
<boost_serialization signature="serialization::archive" version="10">
  <env class_id="1" class_name="CXXR::Environment" tracking_level="1" version="0">
    object_id="_0">
  <RObject class_id="2" tracking_level="0" version="0">
    <GCNode class_id="0" tracking_level="1" version="1" object_id="_1"></GCNode>
    <type>4</type>
    <m_attrib class_id="-1"></m_attrib>
  </RObject>
  <envtype>6</envtype>
</boost_serialization>
```

The classes whose contents we wish to inspect are:

- `CXXR::Provenance` for information on the binding: the timestamp of its creation; the symbol it bound; and its related `CommandChronicle`.
• `CXXR::Symbol` for a string representation of the Symbol

• `CXXR::CommandChronicle` for the expression that was evaluated and a list of parent Provenances.

These are the classes of interest.

Figure 4.1 illustrates where these elements occur within the graph of an XML-serialised session.

Figure 4.1: A graphical depiction of the XML elements of a serialised CXXR session, annotated to show those elements of interest

The way in which an XML element can be disambiguated to determine which C++ class it represents is by inspecting its `class_id` attribute. The first occurrence of a particular class within an XML document is signified by the presence of a `class_name` attribute. Therefore when a class of interest is first encountered, its ID is noted so that subsequent elements of interest can be identified by the value of their `class_id` attribute.

Elements occurring between the opening and closing tags of an element of interest are retained in memory so that their values can be inspected upon reaching the closing tag.

This is facilitated by:

• Maintaining a count, which is incremented upon encountering the start of an element
of interest, and decremented on encountering an end;

- A node stack onto which elements within an element of interest are pushed. When the count is decremented and reaches 0, this signifies leaving a subtree of interest, and so the node stack can be unwound and each of its elements’ memory allocations released;

- Further performance-enhancing control is obtained by inhibiting all recording of a given element tag. For example, the value of a xenogenous binding may comprise several million elements that are not required, so inside a CXXR::Provenance element, an inhibitor on the <m_value> tag can be established to prevent values within that element being retained.

Similarly to its handling of classes, Boost serialisation assigns an identifier to each object (i.e. instance of a class) in an object_id attribute of its first occurrence. Subsequent occurrences of an object will use this identifier in their object_reference_id attributes.

The stages of extracting in RDF, provenance information from a CXXR session serialised as XML are as follows:

1. Parse the XML document and iteratively process its elements to derive collections of symbols, provenances and command chronicles, indexed by their object_ids;

2. Dereference the relationships between these collections formed by object_id_references;

3. Construct an RDF representation of the data comprised within the collections, expressed using appropriate terms from PROV-O, RDFS, FOAF vocabularies;

4. Serialise the RDF graph.

This is illustrated by an activity diagram shown in Figure 4.2.
4.5.2 Algorithm

The outline of the iterative parsing approach to processing a saved CXXR session into PROV-O is described in Algorithm 4.8.

Activity diagrams for the node start and node end portions of Algorithm 4.8 are shown respectively in Figures 4.3 and 4.4.

Callback Handlers

The ‘start’ handlers for CXXR::Symbol and CXXR::CommandChronicle are empty, while that for CXXR::Provenance establishes an inhibitor for the <m_value> tag to avoid entirely loading those XML elements that represent the value of xenogenous bindings:

\[\text{inhibitor\_queue.append("m\_value")}\]

The ‘end’ handler for each class reads the values of relevant XML elements using XPath queries, and stores these in an associative container with key object_id. In the case of a symbol, only its string representation is stored, while for Provenance and Command-Chronicle a composite structure is used to contain the various attributes. End handler algorithms for class types CXXR::Symbol, CXXR::CommandChronicle, CXXR::Provenance are respectively shown in Algorithms 4.9, 4.11, and 4.10.

4.5.3 Implementation

The above has been implemented in Python, using the lxml library [12] for its iterparse method for “parsing XML into a tree and generates tuples (event, element) in a SAX-like fashion”. For RDF output the rdflib library [29] has been used for its ability to construct and serialise RDF graphs in numerous formats including RDF/XML, N3, NTriples and Turtle.

The source code of the implementation is given in [104].
Algorithm 4.8 The cxxr2prov algorithm

1: procedure cxxr2prov(input_file)

2: interest['CXXR :: Symbol'] ← (symbol_start, symbol_end)

3: interest['CXXR :: Provenance'] ← (prov_start, prov_end)

4: interest['CXXR :: CommandChronicle'] ← (chron_start, chron_end)

5: classes ← empty dictionary

6: inhibitor_queue ← []

7: inhibitors ← []

8: node_stack ← []

9: interest_count ← 0

10: for all (event, elem) ∈ iterativeparse(input_file) do

11:     if event = 'start' then

12:         if count(inhibitors) > 0 then continue

13:             if 'class_name' ∈ elem.attrib then ▷ First encounter of class

14:                 class_name ← elem.attrib['class_name']

15:                 class_id ← elem.attrib['class_id']

16:                     if class_name ∈ interest then ▷ Is this a class of interest?

17:                         handler ← interest[class_name]

18:                         classes[class_id] ← handler ▷ Associate callbacks with class ID

19:             if elem.tag ∈ inhibitor_queue then

20:                 inhibitor_queue.remove(elem.tag)

21:                 inhibitors.add(elem.tag) continue

22:         if interest_count > 0 or elem of interest then

23:             node_stack.push(elem)

24:     if elem of interest then

25:         interest_count ++

26:         class_id ← elem.attrib['class_id']

27:             Call classes[class_id][0](elem) ▷ Start handler for class
else if event = ‘end’ then
  if elem.tag ∈ inhibitors then
    inhibitors.remove(elem.tag) continue
  if count(inhibitors) > 0 then
    elem.clear() continue
  if elem of interest then
    class_id ← elem.attrib[“class_id”]
    Call classes[class_id][1](elem) ▷ End handler for class
    interest_count − −
    if interest_count == 0 then
      while count(node_stack do
        e ← node_stack.pop()
        e.clear()
      if interest_count == 0 then ▷ Not currently recording
        elem.clear()
**process node start**

- «decisionInput»
  - count(inhibitors) > 0

- «decisionInput»
  - 'class_name' in elem.attributes

- «decisionInput»
  - elem.tag in inhibitor queue

- «decisionInput»
  - interest_count > 0 or elem is of interest

- «decisionInput»
  - elem is of interest

**Figure 4.3: Activity diagram showing processing of XML 'start node' event**
For each element on node stack:
pop and clear

Figure 4.4: Activity diagram showing processing of XML ‘end node’ event
Algorithm 4.10 cxr2prov: provenance_stop handler

Require: elem

\[ prov\_id ↩ elem["object\_id"] \] \hspace{1cm} \text{▷ Provenance ID}

\[ chron ↩ elem.xpath("child :: chronicle") \] \hspace{1cm} \text{▷ CommandChronicle ID}
\[ chron\_id ↩ chron.get("object\_id" or "object\_id\_reference") \]

\[ sym ↩ elem.xpath("child :: symbol") \] \hspace{1cm} \text{▷ Symbol ID}
\[ sym\_id ↩ sym.get("object\_id" or "object\_id\_reference") \]

\[ sec ↩ \text{int}(elem.xpath("child :: sec/text()")) \] \hspace{1cm} \text{▷ Timestamp}
\[ usec ↩ \text{int}(elem.xpath("child :: usec/text()")) \]

\[ prov ↩ \text{Provenance}(sym\_id, chron\_id, sec, usec) \]
\[ provenances[prov\_id] ↩ prov \]

Algorithm 4.11 cxr2prov: chronicle_stop handler

Require: elem

\[ chron\_id ↩ elem["object\_id"] \] \hspace{1cm} \text{▷ CommandChronicle ID}

\[ cmd ↩ elem.xpath("child :: str\_command/text()") \] \hspace{1cm} \text{▷ Expression}

\[ parents ↩ [] \] \hspace{1cm} \text{▷ Parents}

\textbf{for all} parent ∈ elem.xpath("child :: parent") \textbf{do}

\[ parent\_id ↩ parent.get("object\_id" or "object\_id\_reference") \]
\[ parents.append(parent\_id) \]
\[ chronicles[chron\_id] ↩ Chronicle(cmd, parents) \]
4.5.4 Evaluation

Example

Listing 4.16 shows a trivial R session for demonstrating the use of cxxr2prov. Once evaluated under CXXR and serialised to XML as the file trivial.xml, it may be processed with cxxr2prov as shown in Figure 4.5 to store the resultant PROV-O in RDF/Turtle as file trivial.ttl.

Listing 4.16: Trivial R examplar for cxxr2prov

```
1 sq <- function(x) x * x
2 one <- 1
3 two <- one + one
4 three <- two + one
5 nine <- sq(three)
```

Figure 4.5: Invocation and (verbose) output of cxxr2prov

The resultant file trivial.ttl may be supplied to other provenance-aware systems such as PROV-O-Viz [49], which is designed to visualise provenance in a Sankey Diagram. The output of this is shown in Figure 4.6.

The purpose of this investigation was to satisfy Research Goal 5 (Section 1.5.1), to understand how provenance information recorded in CXXR can be enabled for interoperability with other provenance-aware systems. It has been demonstrated that it is possible using the given approach to express automatically an XML-serialised Provenance-Aware CXXR session in terms of W3C PROV, and that as such, provenance information gathered by CXXR can be used in other provenance-aware systems.
CHAPTER 4. SERIALISATION

Performance

For testing the performance of cxxr2prov, R.D. Peng’s ‘Air Quality Audit’ analysis (Appendix C) was selected as an example of a real-world data analysis that consists of a moderate number of top-level expressions (50), and utilises reasonably large datasets (approximately 2.5 million rows x 29 columns).

This analysis was performed under CXXR and the session then serialised, resulting in an XML document of size 3.5GB and containing 155,593,443 XML elements. On the development machine used\(^1\) the cxxr2prov process took on average 14 minutes, 15 seconds to complete while crucially experiencing a peak memory consumption of only 425 MB. This represents a substantial improvement in memory consumption when compared to tests conducted without inhibitors, in which case the process would consume in excess of 15GB before being terminated by the user who deemed this to be having an unreasonable effect on system performance.

In respect of execution time: it takes over 8 minutes to simply parse the sample XML file in Python with lxml while trapping ‘start’ and ‘end’ events but without performing any further action of any kind. For these reasons the performance of the cxxr2prov algorithm is considered to be satisfactory.

Future Work

This work herein describes the current ability to express provenance collected during a CXXR session in terms of W3C PROV-O. In order for the interoperability of this

\(^1\)Intel Core2 Quad Q6600 @ 2.4GHz; 8GB RAM; 2 x 7200RPM SATA HDDs in RAID-0; Debian 7.6 (Wheezy)
provenance to be more robust, it would be necessary for CXXR to attribute to each resource—i.e. Provenance, CommandChronicle—a unique identifier, or UUID, to which that resource can be persistently referred. For further reducing ambiguity of UUIDs, it would also be desirable to have CXXR attribute each session with an identifier. This would also enable provenance information regarding the session itself to be recorded, such as the CXXR version, the name of the user operating the session, details of loaded packages, so that provenance questions involving these details—such as question 6 in Section 3.1—may be answered.
Chapter 5

Further Provenance

Chapter 3 describes a view of provenance and design for modelling and capturing provenance that is generally applicable to most use cases of R; however, there exist use cases where this design does not apply and provenance questions are unanswerable, either at all or to such a degree that would be considered sufficient.

This chapter will consider such use cases of R and aspects of the (CXX)R interpreter in which provenance-awareness is not adequately catered for by the design described in Chapter 3, and propose new designs to address the use cases and scenarios.

5.1 Expressions from Outside

So far we have only considered the effects of evaluating expressions that were issued by the user as top-level commands in a CXXR session. CXXR, like R, instruments a mechanism for reading and evaluating expressions from outside the usual console interface, taking its input from files, connections or the standard input. The function for accomplishing this is R’s source function.

This section will discuss an alternative design for capturing provenance of the source function to refine its provenance-awareness in accordance with research goal 3 (Section 1.5.1). The outline for the following section is as follows:

- Introduce source and describe its usage,
- Describe alternative views of provenance collected in source,
- Describe the use case for an alternative view of source’s provenance to that suggested in Chapter 3,
- Show the design for enabling a different granularity of provenance collection,
• Show the implementation of the design in CXXR,
• Evaluate the approach taken.

5.1.1 Introduction

R’s source function can trace its origins back to S, where it was one of a number of ways in which expressions could be read from a file and then evaluated [9].

Listing 5.1: example.R file contents

<table>
<thead>
<tr>
<th>No.</th>
<th>Code</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x &lt;- date()</td>
<td># String representation of current date/time</td>
</tr>
<tr>
<td>2</td>
<td>y &lt;- rnorm(10)</td>
<td># Vector of 10 normally-distributed numbers</td>
</tr>
<tr>
<td>3</td>
<td>strs &lt;- paste(x, y, sep = &quot; &quot;)</td>
<td># Concatenate each element of vector x with vector y</td>
</tr>
</tbody>
</table>
| 4   | cat("Goodbye
") | # Print a farewell message |

The expressions given in the file example.R, whose contents is shown in Listing 5.1, would be read and evaluated by source by issuing the following top-level command:

```r
> source("example.R")
```

In its current form in R, source enables the interpreter to accept input from sources other than the standard input, such as a file, a URL, or a connection, and parse the received data line-by-line.

If all expressions are syntactically correct, then they are sequentially evaluated in either the local environment from within which source was invoked (i.e. from within the body of a function), or the global environment.

The parameters of source are given in Listing 5.2, and the commonly-used ones will now be briefly discussed. file is either the name of a connection, or a string containing the path to a file or URL from which to read. local allows the environment in which the expressions are to be evaluated to be defined. The default of logical FALSE indicates that the global environment should be used, while logical TRUE will evaluate the expressions in the environment from which source was called. The parameter may also be a specific environment. If echo is TRUE then each expression is printed after parsing, but before evaluation.

Listing 5.2: source function parameters

```r
source(file, local = FALSE, echo = verbose, print.eval = echo, verbose = getOption("verbose"),
```
5.1.2 Use Case

By maintaining the granularity of the top-level expression as originally set out in Chapter 3, any and all bindings states created during the course of evaluating a top-level `source` expression will have attributed to them this top-level expression.

This section considers the case where the user wishes to refine this granularity, to know precisely which expression inside the sourced file is responsible for the creation of a binding state, and to therefore provide more refined answers to provenance questions 1 and 5 (Section 3.1).

**Views of source**

When a binding state created in the course of evaluating a top-level `source` expression has only the invocation of `source` attributed to it as the expression responsible for its creation and not the specific expression within the sourced file whose evaluation resulted in its creation, we refer to this as a **black box** view of `source`.

In other words, in the established provenance record only the **input**—the `source` command and its parameters—and the **output**—the bindings that are created in the course of its evaluation—are known, and the internal details—the individual expressions—of the `source` command are unknown.

In contrast to this is the **white box** view of `source`, which instead attributes to a binding created in the course of evaluating a `source`, the specific expression from the sourced file that was responsible for its creation.

Consider evaluating the example given in Listing 5.1 with the command `source("example.R")`.

In the black-box view of `source`, all three bindings created—`x`, `y`, `strs`—would each have the expression `source("example.R")` recorded in their respective provenance records; whereas, in the white-box view `x` would be attributed to `x <- date()`, `y` would be attributed to `y <- rnorm(10)`, and `strs` would be attributed to `strs <- paste(x, y).`
5.1.3 Design

To refine the granularity to the level of individual expressions evaluated by source, an interception is required of expression evaluations invoked by source. Once such an expression has been trapped, it can be used to override the top-level expression that would otherwise be used by default in attribution of created binding states.

Class ProvenanceTracker is augmented, as shown in Figure 5.1, to include an attribute ‘current_expression’ whose state is set by the operation ‘setExpression’. At the appropriate time in the evaluation of source, ProvenanceTracker’s setExpression operation is called with the appropriate expression.

This desired behaviour is depicted in Figure 5.2.

```
ProvenanceTracker
- current_parentage : Provenance [*]
- current_expression : Expression
- seen : Provenance [*]
+ resetParentage (void) : void
+ resetSeen (void) : void
+ setExpression (Expression) : void
+ getExpression() : Expression = current_expression
+ writeMonitor (Binding) : void
+ readMonitor (Binding) : void
```

Figure 5.1: Class Diagram of ProvenanceTracker augmented to allow specification of current expression

A simplified\(^1\) source function is given in pseudocode in Algorithm 5.1, in which it has been augmented to instrument a call to ProvenanceTracker setExpression to override the current expression being evaluated. It is also necessary to reset the current parentage to match the granularity of this to that of the expression being evaluated.

```
Algorithm 5.1 Refined granularity of provenance in source
1: procedure SOURCE(file)
2:    exps ← parse(file)
3:    for all expr ∈ exps do
4:        PROVENANCETRACKER.setExpression(expr)
5:        PROVENANCETRACKER.resetParentage
6:    result ← eval(expr)
```

\(^1\)The focus here is on the file being sourced—extraneous arguments such as environment, encoding, verbosity of output have been omitted for greater clarity
Figure 5.2: Sequence diagram depicting REPL which has been augmented to override the top-level expression
5.1.4 Implementation

`source` is defined as an R-level function in the base package. Parts of the function relating to reading and evaluating expressions from a file are shown in Listing 5.3\(^2\), where it can be seen to first parse the input file, then iterate over the resulting list of expressions, evaluating each in turn with the function `eval.with.vis`. This implements some of the basic behaviour outlined in Algorithm 5.1.

Listing 5.3: Selected source code from R’s `source` function

```r
function (file, local = FALSE, echo = verbose, print.eval = echo,
    verbose = getOption("verbose"), prompt.echo = getOption("prompt"),
    max.deparse.length = 150, chdir = FALSE, encoding = getOption("encoding"),
    continue.echo = getOption("continue"), skip.echo = 0,
    keep.source = getOption("keep.source"))
{
    [...] exprs <- .Internal(parse(file, n = -1, NULL, "?", srcfile, encoding))
    Ne <- length(exprs)
    [...] for (i in 1L:Ne) {
        ei <- exprs[i]
        [...] yy <- eval.with.vis(ei, envir)
        [...] }
    [...] }
```

The function `eval.with.vis` referred to in this code extract is defined locally within `source`, and is simply a wrapper for the internal C function of the same name.

The `CXXR::ProvenanceTracker` class implements the following to reflect changes in design to ProvenanceTracker (Figure 5.1). The current_expression attribute is implemented as a static member field with the name `e_current`, as follows:

```c
RObject* ProvenanceTracker::e_current;
```

This variable is used in conjunction with the method `setExpression` (`RObject* expr`) to set explicitly the expression that should be attributed to the provenance of binding states created. The `ProvenanceTracker::setExpression` method is simply defined as:

\(^2[...]\) denotes lines omitted
To recap Chapter 3: when a binding is established, the write monitor method `ProvenanceTracker::writeMonitor` is called with the given binding as an argument. The write monitor creates a `Provenance` object and attributes it to the binding. This `Provenance` contains a reference to the expression that resulted in the creation of the binding. This expression is determined by the method `ProvenanceTracker::expression()`, shown in Listing 5.4, so that if an expression has been previously explicitly set by a call to `setExpression`, then it is returned; otherwise the default behaviour is to just return `R_CurrentExpr`. This behaviour implements the `ProvenanceTracker::getExpression` operation introduced to the design in Section 5.1.3.

### Listing 5.4: The `ProvenanceTracker::expression()` method

```c++
Expression* ProvenanceTracker::expression() {
    RObject* exp=R_CurrentExpr;
    if (!e_current)
        return static_cast<Expression*>(exp);

    if (TYPEOF(e_current)==EXPRSXP) {
        ExpressionVector* ev=static_cast<ExpressionVector*>(e_current);
        RObject* o = (*ev)[0];
        return static_cast<Expression*>(o);
    } else if (TYPEOF(e_current)==LANGSXP) {
        return static_cast<Expression*>(e_current);
    } else {
        return static_cast<Expression*>(exp);
    }
}
```

In order to instrument a white box version of `source` it is necessary to determine when an expression resulting from `source` is being evaluated, and then explicitly instruct the provenance tracker to attribute this expression to any resultant bindings. The approach described in Algorithm 5.1 is implemented not at the R level—for it would be unsafe to allow such access to the internals of the interpreter at such a level—instead, the changes are implemented inside the internal function responsible for handling expression evaluation: in Algorithm 5.1 this is denoted as `eval`, in the (CXX)R implementation it is denoted by...
eval.with.vis, a function whose call is handled by do_eval. We will look first at how these internal functions are called from the R level, and then how this arrangement was established.

A C function that has been compiled into the interpreter can be accessed from R code via one of two interfaces: the .Primitive interface, used by the primitive functions outlined in Section 2.2; and the .Internal interface. A call made via .Internal is necessarily wrapped in a closure by some R code, which provides greater transparency in its handling of named and default arguments, although at the expense of some performance overhead in constructing and evaluating the closure; whereas the .Primitive interface is more direct to better support more low-level operations, such as the language elements if, for, while and the like. One example of a function accessed via the .Internal interface is eval.with.vis used in R’s source. R maintains a function table, which—amongst other things—connects the name of the R function to the C function responsible for handling the call, and designates through which interface it may be called. A single C function may handle calls from more than one R internal or primitive function that are distinguished by a unique integer offset that gets passed as the second argument to the C function, which may inspect it using the function PRIMVAL.

According to the function table, the .Internal R function eval.with.vis is handled by the C function do_eval with an offset of 1. This is the point at which a call to eval.with.vis can be identified and whilst there is nothing to prevent any other function from making a call to .Internal(evalwithvis(...)), source is the only function in the standard R distribution to do so, therefore it is safe to deduce that a call to eval.with.vis originated from source.

do_eval has been augmented to check whether the R internal function being handled is an eval.with.vis—as opposed to eval that is also handled by this function—in which case it informs the provenance tracker that the current expression is the one passed from source to eval.with.vis. This is shown in Listing 5.5. The parentage is reset to begin a new parentage for this expression evaluation so as not to include things in this parentage that only pertain to the previous expression.

Listing 5.5: The C function do_eval

```c
SEXPtr attribute_hidden do_eval(SEXP call, SEXP op, SEXP args, SEXP rho)
{
    SEXP encl, x, xptr;
```
5.1.5 Evaluation

The objective of this investigation is to consider the granularity of provenance information collected during the course of evaluating an expression that invokes the `source` function. This relates to provenance questions 1 and 5. There were introduced two alternative views of provenance in `source`: the black box and the white box. The design was based around the white box view which captured a finer-grained provenance.

The relevant provenance questions can be seen to be answered in the following examples.

Example

Listing 5.6 shows `example.R` being evaluated with the white-box implementation of `source`. The effect of this implementation can firstly be seen in the parents of `strs` being correctly identified as `x` and `y`; and secondly, the pedigree of all of present bindings shows that each binding has been attributed to the expression in the file responsible for its creation.

Listing 5.6: Example of white-box `source` in action

```bash
> source("example.R")
Goodbye
> ls()
```
This behaviour is consistent even if `source` is evaluated from an expression derived from a call to `source`: if `liftExample.R` has as its contents:

```r
source("example.R")
```

The result of calling `source("liftExample.R")` is the same as simply calling `source("example.R")`.

If another expression is added after the `source` so that `liftExample2.R` has contents:

```r
source("example.R")
z <- y * 2
```

It can be seen from Listing 5.7 that this full transparency in expressions is at the expense of the ability to identify in which file the expression occurred.

```
LISTING 5.7: Example of white-box source within a source
```

Default `source` behaviour: Black box?

Without implementing the above features it would not be possible to obtain a white-box version of `source`; however, that is not to say that `source au naturel` is of the black-box variety. Unfortunately `source` displays by default some idiosyncratic behaviour that
results in no meaningful provenance-awareness.

As described in Algorithm 5.1, before being able to evaluate the expressions, `source` must first `parse` the input it receives from a file (or connection or standard input). It does this by calling `.Internal(parse(file,...))` (Line 8 in Listing 5.3).

Unfortunately, the `parse` function ultimately has a rather destructive side-effect: it overwrites `R_CurrentExpr`. It actually does this for each expression it parses (which in at least one of its other applications—parsing the top-level command as part of a REPL iteration—is perfectly acceptable).

The effect this has is that by the time the expressions are *evaluated* by `source`, `R_CurrentExpr` has already been made to point to an expression representing each line of `source`'s input, and so it remains pointing to the final one, which is attributed to any bindings created during subsequent evaluation:

```r
> source("example.R")
Goodbye
> ls()
[1] "strs" "x" "y"
> provenance(strs)$command
cat("Goodbye\n")
> provenance(ls())
cat("Goodbye\n")
cat("Goodbye\n")
cat("Goodbye\n")
```

In order to avoid this, the value of `R_CurrentExpr` needs to be captured sooner, for instance as soon as it is parsed by the REPL iteration. This gives the ‘expected’ result of a black-box view of `source`:
In this scenario there is no record of `str`'s parents—x and y—because their bindings were created in the same top-level command as `str`, so would appear in the `seen` set. In the white-box version this was not encountered because the parentage was reset prior to each expression evaluation (Line 16 in Listing 5.5).

Although it may be reasonable to attribute an individual binding to a `source` top-level command in isolation; when considering a collection of bindings (such as the `ls()` above), it is inconsistent with our reasoning of what constitutes a pedigree to repeat the `source` command like this: it should instead be the top-level commands that need to be evaluated to reconstruct the given bindings.

For the above reasons the black-box model presented here does not entirely accurately model the landscape of a session in a consistent way. To overcome these limitations would require a significant change in approach; the possibility of which will be touched on in the following section.

**Future Work**

While the white-box implementation has been shown to be adequate in some moderately simple cases, more sophisticated scenarios cannot be modelled as effectively.

If we consider the following session:

```r
> f <- function() {
+     source("example.R")
+     y + 5
+ }
> z <- f()
```

The first of these statements establishes a binding `f` to a function, whose body calls
source and then returns $y + 5$; the second makes a call to this function and binds the result to $z$.

This does not affect the provenance of the bindings created by the source—x, y and strs—but it does expose a similar issue to that encountered in the black-box implementation for $z$:

```
> provenance(z)
$command
  cat("Goodbye\n")

$symbol
  z

$timestamp

$parents
  [1] "y"

$children
  NULL
```

The expression to which $z$ has been attributed is that to which R_CurrentExpr referred at the time of the binding’s creation, which was set by the parsing routine as it went about its business.

Furthermore, the parentage that is attributed to $z$ is that which was established at the beginning of evaluating the final expression in the source. A more realistic parentage for $z$ would include $f$ as well, but were a single parentage to exist throughout the life-time of the top-level command—and not be reset for each expression in source—then $y$ would be omitted as part of the seen set.

In general: this cannot be modelled by maintaining only one parentage; in the same way that commands cannot be modelled as sequential either—these concepts need to be nested to accurately model this sort of scenario. For instance, in such a view, the binding resulting from the sourced expression $x <- \text{date}()$ should be attributed to both that expression and source("example.R").

At the moment we can express dependencies between bindings in the form of a parentage; what is needed here is a means to represent expression dependencies. This issue will be discussed further in Section 7.2.
5.2 Lazy Loading

5.2.1 Use Case

When a function is defined in the global environment, it is possible to determine which other bindings used it during their creation by inspecting the children of the function:

```r
> sq <- function (x) x * x
> four <- sq(2)
> nine <- sq(3)
> provenance(sq)$children
[1] "four" "nine"
```

Suppose we wanted to tell which bindings were created by using one of R’s standard functions such as `source` or `seq`. For instance, if the `seq` function were discovered to have been defective after it had been used in some calculation, one might wish to determine which bindings had used it and will need to be regenerated after it has been fixed; something like `provenance(seq)$children`. At present, only the global environment is instrumented with facilities for tracking provenance and since the functions of the standard R distribution reside in other environments—such as the `base environment` in the case of `source` and `seq`—interactions with their bindings are not recorded.

The research objective of this section is a constituent of Section 1.5.1’s research goal 3 and is to investigate how provenance question 4 (Section 3.1) can be answered in the above use case.

This section will:

- Introduce the (CXX)R concept of the `promise` for lazy evaluation,
- Look at how packages (Section 2.2.5) are loaded,
- Show how lazy evaluation is used to support lazy loading of functions in packages,
- Illustrate how the current design for provenance-awareness does not in the instance of lazy-loading create a representative provenance record,
- Present a modified design and its implementation.

5.2.2 Promises

A `promise`—internally designated as `SEXPTYPE PROMSXP`, and encapsulated in CXXR’s `Promise` class—is R’s facility for lazy evaluation, which enables the evaluation of some expression to be deferred until such a time as the result of its evaluation is required.
A promise comprises three components: an expression whose evaluation is deferred; an environment in which the expression should be evaluated; and a value, which is initially the symbol `R_UnboundValue` and used to store the result of the evaluation (a form of the memoization optimisation technique to ensure that an expression is evaluated at most once). When the value of a promise is required, it is said that its value is forced—in other words, the given expression is evaluated in the given environment, and this result is stored in the promise’s value field.

R has three types of function: builtin and special—collectively known as the primitives described in Section 2.2—and closures. A primitive function is internal to R—typically implemented in C—and is designated to be called via one of the two interfaces `.Internal` or `.Primitive`. Closures are the type defined in R using `function`.

R closures are a form of the general functional programming concept of function closures, which are composed of the following:

- **Formal Parameters.** A comma-separated list of formal parameters that are accepted by the function. A parameter may take one of three forms: a symbol to which the argument will be bound; `symbol = default` to define a default argument value; or `...`, a special symbol which when used as a formal argument denotes the function accepts multiple arguments. This is used in scenarios such as when the precise number of arguments is not known in advance, or when one function extends another without specifying the full list of arguments.

- **Body.** An R statement; or composition of multiple statements enclosed within braces.

- **Environment.** The environment that was active at the time of the function’s creation.

The rules for argument evaluation are determined by the type of the function. Builtin functions have their arguments evaluated before being passed to the function, while arguments to special functions are not evaluated prior to the function call.
Evaluation of a closure is a little more involved because the actual arguments need to be matched to the formal parameters, either by position or by name, and the closure may define default argument values that need to be applied if a parameter is unmatched. When a closure is evaluated, an environment known as its evaluation environment is created and populated with bindings for each of its formal parameters. The symbol of a binding in the evaluation environment is the name of the formal parameter, and the binding’s value is determined by a process of argument matching. Each actual argument is firstly encapsulated in a promise so that in the case that the argument is an expression, it would only be evaluated at the time that its value is required. For example in mystery(1+2) the expression 1+2 will only be evaluated if the result of its evaluation is required such as if mystery <- (x) x * x.

The body of the closure is then evaluated in the evaluation environment—as Section 5.4 will discuss in a bit more detail—in which if an argument’s values is required, the promise is forced.

This technique enables R to present a call-by-need evaluation strategy for function closures. Promises may also be, and indeed are, used outside of this scenario. Because a promise can encapsulate and defer evaluation of any arbitrary R expression, they can be used to support other facilities.

This section will now describe how promises allow R to perform lazy loading of its packages.

The R distribution comprises a number of standard packages including base, datasets, methods and utils packages (see Table 2.1 for the complete list). These are collectively known as the base packages, many of which—including those just mentioned—are loaded automatically at the outset of an R session (the full list of these is given in Figure 2.2). Any other packages—both those forming a part of the standard R distribution and add-on packages—are loaded as they are required.

When loading a package, such as the base package, two environments are defined: a namespace environment, named for instance namespace:base, contains the bindings from the package; and a package environment, such as package:base, which is populated by selected bindings from the package’s namespace known as its exports. It is the latter environment that gets attached to form part of the search path. This mechanism allows a package a degree of control over which of its bindings are exposed to public visibility and addressable from outside the package.

There is another, more subtle, distinction between the two environments: the enclosing environment of a package’s namespace is one created to hold explicit imports from other namespaces defined by
5.2.3 Lazy-Loading

R uses a **lazy-loading** strategy to load the *value* of an object introduced by a package at the time of its first use. When a package is loaded, its namespace is populated with bindings to the symbols defined in the package; however, the **value** of these bindings are not those intended for use: they are **promises**, which when evaluated will fetch the *actual* value from a database, to which the given symbol is then **rebound**. For example, Figure 5.4 describes how, when the base package is loaded, `seq`'s initial binding state is established with the value of a Promise `loadseq`.

5.2.4 Problem

This presents a problem to our strategy for collecting provenance, because the original binding being accessed is not the one that will persist in the package’s namespace. Suppose for example one wanted to use the `seq(from, to, by)` function for generating regular sequences defined in the **base** package to generate a simple sequence of integers between one and five:

```r
> x <- seq(from=1, to=5)
> x
[1] 1 2 3 4 5
```

We will assume that this is the first invocation of the `seq` function. This explanation follows the sequence diagram given in Figure 5.5. As with all symbols in the base environment, `seq` is initially bound to an unevaluated promise. This binding state is represented by S1. In evaluating the expression in which its symbol appears, the `seq` binding is read, and S1 is added to the current **parentage** and **seen** set. Its value is required for the evaluation of the expression, and being an unevaluated promise, its value gets **forced**. This evaluates the lazy-loading code encapsulated in the promise ‘loadseq’, causing the definition of the ultimately desired value—a closure, ‘seqclos’—of `seq` to be loaded from a database, which gets bound to the symbol `seq`. This creates a **new** binding state of `seq`, S2, which has as its parent S1—the first `seq` binding state—with which it registers itself as a child, and it is inserted into the seen set as per protocol for all written bindings.

---

the package, and the enclosing environment of *that* environment is the base environment. (The enclosing environment of a package’s (‘package’) environment depends on its order in the search path.) The enclosing environment of the base namespace is the global environment. The search path for a package `pkg` that begins in `namespace:pkg` next goes through its imports, then to the base namespace, then on to the standard search path beginning in the global environment (then through whatever package environments are attached, ultimately ending up at the `package:base` and `R_EmptyEnv`, as described in Section 2.2.6.
Initialising the base environment with vestigial \texttt{seq} binding to a Promise

The \textit{expression} attribute of this Promise is R code for lazy-loading—i.e., opening a file and reading from it the real value of \texttt{seq} and rebinding \texttt{seq}.

Figure 5.4: Sequence diagram depicting initial binding state of \texttt{seq} in base environment.
Lazily-Loading the \texttt{seq} closure: Provenance-Tracker

Environment base : Environment

\texttt{loadseq} : Promise

\texttt{force()} : Evaluator

\texttt{evaluate(\texttt{\{x \leftarrow seq(1,5)\}})}}

\texttt{result = [1,2,3,4,5]}

\texttt{x : Binding}

\texttt{assign(result)}

\texttt{writeMonitor(x)}

\texttt{X1 : Provenance}

\texttt{registerChild (X1)}

\texttt{setParentage(current)}

\texttt{add(X1)}

\texttt{setProvenance(X1)}

The inclusion of \texttt{"S2"} in the current parentage is prohibited because it already appears in \texttt{\textit{seen}}.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{sequence_diagram.png}
\caption{Sequence diagram depicting the first evaluation of \texttt{seq} resulting in its lazy-loading}
\end{figure}
Now that the *bona fide* closure is bound to `seq` in the base namespace, it now needs to be evaluated so that it may generate the sequence according to the given arguments. In doing this, the *new* binding of `seq` is read—binding state S2; however, S2 will not be recorded in the parentage because it was added to the seen set when it was written. Consequently, S2 is not been recorded in the parentage; therefore, when the binding `x` is written, the sole\(^4\) parent of `x` is S1—*the first* binding state of `seq` that existed only to lazily-load the real one—while the binding state S2—representing the *actual closure* that generated the current value of `x`—has not been recorded.

```r
> provenance(x)$parents
[1] "\.Options"        "getNamespace"       "as.name"
[4] "seq"              "__.S3MethodsTable__." "seq.default"
```

This is reflected in the set of `seq`’s children where we would like to—but won’t—see `x`, and the command that resulted in the binding of `seq` makes it plain when it was created:

```r
> provenance(seq)$children
NULL
> provenance(seq)$command
x <- seq(from = 1, to = 5)
```

Upon subsequent uses of the `seq` function it can be seen to exhibit the desired behaviour:

```r
> y <- seq(from = 1, to = 5)
> z <- seq(from = 5, to = 1)
> provenance(seq)$children
[1] "y" "z"
```

### 5.2.5 Design

This type of behaviour does not satisfy the requirements of the initial motivating scenario of being able to identify those bindings that were created by a (faulty) standard function.

In order to achieve the desired functionality it is necessary to *exclude* the original binding of a promise from the seen set. Algorithm 5.2 redefines the write monitor to enable control over whether a binding state is recorded in the seen set. At the time the write monitor is called when forcing the promise, i.e. in the Promise force operation, this new write monitor is called with the value *false* for its `beenSeen` argument.

---

\(^{4}\)The ‘sole’ binding with which we are presently concerned—there are some extraneous bindings for various bits of housekeeping that live in the base environment.
Algorithm 5.2 An updated write monitor to allow overriding of inclusion in the seen set

1: procedure ProvenanceTracker.writeMonitor(bdg : Binding, beenSeen : Boolean)

2: \[ P \leftarrow \text{newProvenance} \] \(\triangleright\) Create new binding state

3: \[ P.m\_expression \leftarrow \text{Current expression} \]

4: \[ P.m\_symbol \leftarrow \text{bdg.symbol} \]

5: \[ P.m\_parents \leftarrow p\_current \]

6: \[ P.m\_timestamp \leftarrow \text{Current Time} \]

7: \[ P.m\_children \leftarrow [] \]

8: for all parent \in P.m\_parents do \(\triangleright\) Register as child of each parent

9: \[ \text{parent.children.add}(P) \]

10: \[ \text{bdg.m\_provenance} \leftarrow P \]

11: if beenSeen then

12: \[ \text{p\_seen.add}(P) \]

5.2.6 Implementation

The modified write monitor given in Algorithm 5.2 is implemented as follows:

```cpp
void ProvenanceTracker::writeMonitor(const Frame::Binding &bind,
                                       bool beenSeen) {
  [...] if (beenSeen) {
    GCEdge<Provenance> tmp(prov);
    seen()->insert(tmp);
  }
}
```

A method `forcedPromise` is added to class `ProvenanceTracker`:

```cpp
void ProvenanceTracker::forcedPromise(const Frame::Binding & bdg) {
  writeMonitor(bdg, false);
}
```

The `forcedPromise` method is called at the point at which a promise is forced—in the `Frame::forcedValue` method shown in Listing 5.8. This will ensure that the promise being forced is excluded from the seen set.

Listing 5.8: The `Frame::forcedValue` method from `envir.cpp`
5.2.7 Evaluation

The design presented here allows the first invocation of a lazily-loaded closure in the base environment to be accurately attributed to a binding whose value depends upon it, as set out in the use case at the beginning of the section.

The following example illustrates how provenance question \(4\), “Which other objects used it during their creation?” can be answered of a lazily-loaded function in the base environment:

```r
> a <- seq(from = 1, to = 5)
> b <- seq(from = 5, to = 1)
> c <- seq(from = 1, to = 935, by=39)
> provenance(seq)$children
[1] "seq.default" "a"        "b"        "c"        ".Last.value"
```
5.3 Values from Outside

5.3.1 Use Case

The scenario described in Section 5.1 of reading and parsing expressions from a file is one example of how an R function, in that instance `source`, may result in an output that is not entirely dependent upon its input arguments. A function that modifies some aspect of external state—like `source`, which may for instance create bindings in the global environment—is said to exhibit side-effects. A function whose output is determined entirely by its input and has no side-effects is called a pure function, whose converse is naturally an impure function.

When pure functions are used exclusively, provenance question 5, “what was the sequence of commands that resulted in binding state S?”, can be answered by collating in order of evaluation the expressions that resulted in each ancestor binding state of S.

However, there is an implication for provenance of binding states derived from impure functions: if an impure function F is used to determine the value of binding state S or any ancestor state of S, then it is not possible to describe certainly the process that led to S using only the expressions that were evaluated.

One typical use case for this provenance question is to determine how a binding state can be recreated. By a slight refocussing of this provenance question we get: “How can S be recreated?” This section looks at how this question can be answered in situations that have the involvement of impure functions.

5.3.2 Xenogenesis

Many of the R functions encountered so far are pure: the value they return depends exclusively on the value of their arguments. Other functions are useful precisely because they have side-effects; in other words they modify some aspect of the interpreter state as well as or instead of returning a value. One example of this is pseudo random number generation for which the seed may be set with the `set.seed` function, which modifies the `.Random.seed` binding in the global environment. When a random number is generated, this binding is read and subsequently written, and because this all occurs through interactions with bindings, it occurs within the scope of Provenance-Tracked CXXR’s facilities as so far described. Any subsequent generation of a random number will depend on the seed, as illustrated in Listing 5.9.
CHAPTER 5. FURTHER PROVENANCE

Listing 5.9: Provenance tracking pseudo-RNG

```
> set.seed(1)
> x <- rnorm(10)
> x
[1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078 -0.8204684
[7]  0.4874291  0.7383247  0.5757814 -0.3053884
> pedigree("x")$commands
[[1]]
set.seed(1)
[[2]]
set.seed(1)
[[3]]
x <- rnorm(10)
```

However, it is not always the case that a function’s behaviour depends on either its arguments or some other aspect of the interpreter state. This occurs when a function receives some external influence such as reading from a file or database, accepting user input interactively, or because it calls non-R code via one of the foreign language interfaces (such as to C). Examples of R functions that behave in this way include: `scan` which reads data into a vector from a file (or connection) or console; `identify` reads the position of the graphical user interface (‘mouse’) pointer when the button is pressed and identifies, within a given dataset, which point is closest to the position of the pointer; `edit` launches a text editor in which the user can define the value of an R object, which when the text editor is closed, is returned by the `edit` function so that it may be bound to a symbol.

An example of using `edit` to define the function `sq` which squares its argument is given in Listing 5.10, the user interaction with the Vim text editor that was launched by the invocation of `edit()` is depicted in Figure 5.6.

Listing 5.10: Using `edit` to define a function `sq`.

```
> sq <- edit()
> sq
function (x) {
    x * x
}
```

This can be reflected only so far in the provenance record for the binding `sq`; it is attributed to the command `sq <- edit()` as this was the top-level command that gave rise to its existence. However, there is no record of the actual `substance` of the call to
edit—the text that was entered by the user—as it comes from outside the state of the interpreter and it is not subject to the provenance-tracking facilities.

For this reason, this binding is known as **xenogenous**: “caused by a foreign body”, and functions that give rise to these bindings are **xenogenetic**.

Therefore, unlike a regular binding it is not possible to define categorically the process for regenerating the value of a xenogenous binding solely in terms of its input artifacts (i.e. parents) and the top-level command that gave rise to it. If a xenogenetic function is evaluated (either directly or indirectly) during the course of evaluating a top-level command, any bindings created subsequently are considered to be xenogenous.

### 5.3.3 Design

It is not possible heuristically to determine whether a function is xenogenetic, because there is no particular, common way in which they will accept input. It is therefore necessary to modify each xenogenetic function to announce to the provenance tracker that a xenogenetic function has been evaluated, and that binding states created subsequently should be considered xenogenous.

In order to be able to recreate a xenogenous binding—as required by the motivating question—we elect to **preserve its value**. Therefore the value of a xenogenous binding may be later recalled using an R-level function and then utilised in subsequent expressions.

Figure 5.7 shows (only) the new attributes for representing a xenogenous binding state: xenogenous for indicating that a binding state is xenogenous, value for storing the value assigned to a xenogenous binding state; and new operations: isXenogenous which returns true iff the binding state is xenogenous, setXenogenous for attributing the value of a xenogenous binding to the Provenance, and value for accessing the value attribute.

The ProvenanceTracker maintains a **flag** to indicate whether or not a xenogenetic
Figure 5.7: Provenance class diagram showing new attributes and operations for xenogenesis

function has been evaluated so that it may inform a binding state upon its creation that it is xenogenous. The new attributes and operations involved with this are depicted in Figure 5.8. The simple operation `flagXenogenesis` is given in Algorithm 5.3.

Figure 5.8: Provenance Tracker class diagram showing new attributes and operations for xenogenesis

**Algorithm 5.3** The ProvenanceTracker `flagXenogenesis` operation

1: `procedure` **PROVENANCETracker.FLAGXENOGENESIS**

2:  `ProvenanceTracker.xenogenetic ← True`

Algorithm 5.4 describes what, in addition to its previous design, the write monitor must do to signify to a Provenance object that it represents a xenogenous binding state, whose current value is to be preserved.

The way in which these entities interact is depicted in Figure 5.9. Initially the occurrence of a xenogentic function is flagged to Provenance Tracker. In the evaluation of the top-level expression, any binding state subsequently created is declared xenogenous and its present value is recorded as part of its provenance record.

**5.3.4 Implementation**

This section follows an implementation approach suggested by Dr. A. R. Runnalls.
Figure 5.9: Sequence diagram depicting how evaluation of a xenogenetic function would flag ProvenanceTracker, and when a xenogenous binding state is created, how this is recorded and its value is retained.
Algorithm 5.4 Functionality added to write monitor to, depending on ProvenanceTracker state, declare a Provenance xenogenous and preserve the present value of the binding

1: \textbf{procedure} ProvenanceTracker.writeMonitor(bdg: Binding)
2: \hfill $P \leftarrow \text{new Provenance}$ \hfill $\triangleright$ Create new binding state
3: \hfill $[\ldots]$ \hfill $\triangleright$ Initialise fields of $P$ as previous
4: \hfill $bdg.m\_provenance \leftarrow P$
5: \hfill $\text{if ProvenanceTracker.xenogenetic then}$
6: \hfill $val \leftarrow \text{BDG.RAWVALUE}$
7: \hfill $P.m\_set\_xenogenous(val)$

The new design of the Provenance class (Figure 5.7) is implemented in the CXXR::Provenance class as shown in Listings 5.11 and 5.12.

Listing 5.11: Extract from class Provenance header file showing relevant additions to track provenance of xenogenous values

1 | class Provenance : public GCNode {
2 | public:
3 | $[\ldots]$ 4 | const RObject* value() const
5 | $\{$
6 | \hfill return $m\_value$;
7 | $\}$
8 | $[\ldots]$ 9 | bool isXenogenous() const
10 | $\{$
11 | \hfill return $m\_xenogenous$;
12 | $\}$
13 | $[\ldots]$ 14 | void setXenogenous(const RObject* value);
15 | $[\ldots]$ 16 | private:
17 | GCEdge<const RObject> $m\_value$;
18 | bool $m\_xenogenous$;
19 | $[\ldots]$ 20 |};

Listing 5.12: Extract from soure file of class Provenance, illustrating mutator method
void Provenance::setXenogenous(const RObject* value) {
  m_value = value;
  m_xenogenous = true;
}

In this implementation of the design, some of the attributes and methods of the ProvenanceTracker class are implemented by a class ProvenanceTracker::CommandScope, e.g. read and write monitors, and it is also this class that implements the design changes given in Figure 5.8.

The relevant extracts from the header file and source file for this class are given in Listings 5.13 and 5.14 respectively.

Listing 5.13: Extracts from header file for class ProvenanceTracker

```c++
#ifdef __cplusplus
namespace CXXR {

class ProvenanceTracker {
public:
  class CommandScope {
  public:
    [[...]]
    void flagXenogenesis() {
      m_xenogenetic = true;
    }
    [[...]]
  private:
    bool m_xenogenetic;
  };
  static void flagXenogenesis();
  [[...]]
}; // namespace CXXR

extern "C" {
  #endif // __cplusplus
  void flagXenogenesis();
  #endif // __cplusplus
}
```
A call to the method `ProvenanceTracker::flagXenogenesis()` (or its equivalent C wrapper) is appropriately inserted to the internal C++/C function which underlies each xenogenetic R function: for example `edit()`, `scan()`, `readLines()`, `readBin()`, `readChar()` and `load()`.

The augmentation of the write monitor design in Algorithm 5.4 is implemented as shown in Listing 5.15.

Listing 5.15: The write monitor `ProvenanceTracker::CommandScope::writeMonitor` which identifies xenogenous bindings

```cpp
void ProvenanceTracker::CommandScope::monitorWrite(const Frame::Binding &bdg) {
    const Symbol* sym = bdg.symbol();
    GCStackRoot<Provenance> prov(CXXR_NEW(Provenance(sym, m_chronicle))); 
    if (m_xenogenetic)
```
In this implementation the R function \texttt{pedigree(x)} traverses the graph of provenance objects formed by ancestors of bindings given in \textit{x}. An extract of the implementation of this function is shown in Listing 5.16. This function returns a list comprising five vectors named: \textit{commands}, \textit{timestamps}, \textit{symbols}, \textit{xenogenous}, and \textit{values}. Each member of the graph of provenances is interrogated and its details added to each of the vectors; in particular, if it is xenogenous then it sets its element of the \textit{xenogenous} vector to \texttt{TRUE} and sets the corresponding element of \textit{values} to the value recorded in the provenance.

Listing 5.16: Extract of code from function \texttt{do_pedigree} which underlies the R function \texttt{pedigree}

```c++
SEXP attribute_hidden do_pedigree (SEXP call, SEXP op, SEXP args, SEXP rho) {
  ...
  /* 'provs' is a Provenance::Set of Provenance objects */
  Provenance::Set* ancestors = Provenance::ancestors(provs);
  GCStackRoot<ListVector> ans(CXXR_NEW(ListVector(5)));
  // Assemble result:
  {
    size_t n = ancestors->size();
    GCStackRoot<ListVector> commands(CXXR_NEW(ListVector(n)));
    GCStackRoot<RealVector> timestamps(CXXR_NEW(RealVector(n)));
    GCStackRoot<ListVector> symbols(CXXR_NEW(ListVector(n)));
    GCStackRoot<LogicalVector> xenogenous(CXXR_NEW(LogicalVector(n)));
    GCStackRoot<ListVector> values(CXXR_NEW(ListVector(n)));
    size_t i = 0;
    for (Provenance::Set::iterator it = ancestors->begin();
      it != ancestors->end(); ++it) {
      const Provenance* p = *it;
      (*commands)[i] = const_cast<RObject*>(p->command());
      (*timestamps)[i] = p->timestamp();
      (*symbols)[i] = const_cast<Symbol*>(p->symbol());
      (*xenogenous)[i] = FALSE;
    }
  }
```

```c++
  m_chronicle->writeBinding(prov);
```

```c++
  CXXR::Frame::Binding& ncbdg = const_cast<CXXR::Frame::Binding&>(bdg);
  ncbdg.setProvenance(prov);
  // Maybe ought to clone value
  prov->setXenogenous(bdg.rawValue());
```
5.3.5 Evaluation

Example

Suppose one decided to revisit a mini-example encountered previously, say Listing 3.15, but instead of defining the function at the command line, you use the edit() function:

```r
> sq <- edit()
> three <- 3
> nine <- sq(three)
```

The value of the binding to `sq` was obtained via the `edit()` function, which opens a text editor to accept user input; in this instance a function definition was supplied. The `pedigree` function allows for the interrogation of provenance information of a binding such as `sq`, and it will inform us that the binding is xenogenous and provide the value that was bound to it:
The output of `pedigree` is quite simple in the case of interrogating a single binding; Listing 5.17 shows how the pedigree function provides provenance information pertaining to multiple bindings, in particular `nine` and all of its ancestors.

Listing 5.17: Provenance interrogation using the `pedigree()` function. Illustrates the way in which corresponding list elements describe a particular binding.
Discussion

One of the primary drawbacks of this method is that a degree of manual intervention is required both to identify xenogenetic functions and to modify their definitions. If a new R function were to be added to part of the standard R distribution, it would need to be manually assessed to see if it ought to be considered xenogenetic and if so, its definition modified to announce this fact to the provenance tracker. A further issue concerns third-party packages, the functions of which may exhibit xenogenesis via some means which has not been instrumented to declare their xenogenicity.

A further issue surrounding this method of handling xenogenesis is that of granularity. Suppose one were to issue the top-level expression `{ a <- edit(); x <- 1 }`, then despite not actually resulting from a call to a xenogenetic function, x will be flagged as xenogenous, as shown in Listing 5.18. This behaviour is due to the level of granularity being fixed and limited to only the top-level expression.
Listing 5.18: Example of granularity issue when R code is defined within a code block

```r
> { 
+ a <- edit()
+ x <- 1
+ }
> pedigree("x")
$commands
$commands[[1]]
{
+ a <- edit()
+ x <- 1
}

$timestamps

$symbols
$symbols[[1]]
x

$xenogenous
[1] TRUE

$values
$values[[1]]
[1] 1
```

The granularity issue encountered here is not unique to this scenario; indeed it echoes issues touched upon in Sections 5.1 and 5.4 of this chapter. This issue of granularity will be discussed further in Section 7.2.

### 5.4 Functions with State

R allows for the creation of functions, or more precisely function closures, that exhibit and maintain local state that persists across invocations, in a manner similar to static local variables in a C function. In such cases, it is presently not possible to answer to provenance questions because the local state falls outside the scope of the provenance
tracking facilities described to this point.

This section will:

- Explain how function closures are evaluated,
- Describe how a by-product of closure evaluation, the evaluation environment, can be used to make state persistent,
- Describe the software design for handling the recording of provenance in this scenario,
- Show the implementation of this design in CXXR,
- Evaluate the approach taken and discuss the implications of this scenario for the view of provenance developed.

5.4.1 Introduction

To summarise the definitions given in Section 3.1: An R environment comprises a frame, which is a collection of bindings that map symbols to values; and a reference to an enclosing environment. When a binding is sought in an environment but cannot be located, then the chain formed by enclosing environments is recursively searched.

The concept of R’s closure is described in Section 5.2. In brief: a closure combines (i) a list of formal parameters; (ii) a body consisting of R code; and (iii) an environment—the environment in which the statement that created the closure was evaluated. A most simple definition of a closure is:

```r
> f <- function() { "I'm a closure" }
```

A closure is evaluated by giving the name of the symbol to which it is bound, followed by a list of arguments enclosed in brackets (e.g. `f()`). It is possible to examine the environment of a closure using the `environment` function:

```r
> f()
[1] "I'm a closure"
> environment(f)
<environment: R_GlobalEnv>
> environment(seq)
<environment: namespace:base>
```

During the evaluation of a closure, a new environment is created—the evaluation environment, whose enclosing environment is the environment of the closure. The evaluation environment is initially populated with the unevaluated promises of the arguments,
and as evaluation of the closure body continues, any local variables created are bindings established in the evaluation environment.

Consider the following example which corresponds to evaluation of ‘A’ in Figure 5.10:

```r
> sq <- function (x) {
+   rc <- x * x
+   rc
+ }
```

Evaluating this statement at the top-level (i.e. in the global environment) constructs a closure, whose environment attribute is the global environment; body is ```rc <- x * x; rc;``` and formal arguments is a list composed solely of ```x``` without a default value.

An evaluation of the ```sq``` function will cause the creation of an evaluation environment, initially populated with a binding to symbol ```x``` , which gets matched to the actual argument supplied wrapped in a promise. The use of ```x``` in the RHS of the assignment operation will cause the promise referent of ```x``` to be forced and its value then used in the multiplication operation, the result of which is bound to symbol ```rc``` in the evaluation environment by the assignment operation.

Figure 5.10’s evaluation of statement ‘B’ supposes that ```sq(3)``` is evaluated. At the point at which the closure returns, the evaluation environment that was constructed is populated with bindings to symbols ```x``` and ```rc``` . The function returns the value of the binding ```rc``` and then the evaluation environment is discarded.

### 5.4.2 Functions with State

As Figure 5.10 shows, the evaluation environment gets discarded following evaluation of the closure: it has fulfilled its purpose and it is no longer required. We can more formally define the requirement of an object, such as the evaluation environment ```evEnv``` in the diagram, in respect of whether it is referenced by any other objects. In practice, CXXR’s reference-counted garbage collector will ensure that an object without references is destroyed.

It is not, however, necessarily the case that the evaluation environment is discarded: it is possible to retain the evaluation environment by attributing it as the environment of a closure. This enables the evaluation environment to ```persist``` beyond the end of a closure evaluation, and indeed beyond the end of a top-level expression evaluation.

By attributing a persistent environment to a closure, this closure is able to exhibit state.\(^5\)

\(^5\)Strictly speaking: the bindings within this environment would be known in functional programming
Creation of a Closure and its subsequent evaluation

\[ \text{sqB} : \text{Binding} \]
\[ \text{glEnv} : \text{Environment} \]
\[ \text{sqC} : \text{Closure} \]
\[ \text{params} : \text{PairList} \]
\[ \text{args} : \text{ArgList} \]
\[ \text{x} : \text{Binding} \]
\[ \text{rc} : \text{Binding} \]

\[ A : \text{sq} \leftarrow \text{function}(\text{x}) \]
\[ \text{rc} \leftarrow \text{x} \times \text{x} \]
\[ \text{rc} \]

\[ B : \text{sq}(3) \]

Figure 5.10: Sequence diagram of creating and subsequently evaluating a Closure
Listing 5.19: The ‘counter’ example

```r
> makecounter <- function() {
+   count <- 0
+   function() { # ‘F’
+     count <<- count + 1
+     count
+   }
+ }

> counter <- makecounter()
> counter()
[1] 1
> counter()
[1] 2
> counter()
[1] 3
```

In the ‘counter’ example shown in Listing 5.19, `makecounter` is a function that is able to create counters such as `counter`, which are functions that utilise local state to maintain a count, and whose each invocation increments and returns the value of its count.

The operator `<<-` seen in the function body is used to perform a non-local assignment to the symbol given as its left-hand-side operand. Unlike regular assignment which will rebind (or should a binding not exist, create a new binding of) the symbol in the environment in which the expression is being evaluated, non-local assignment will instead traverse the chain of enclosing environments starting in the environment in which the assignment is evaluated, in search of a binding to the given symbol. If such a binding exists then it will be rebound to the value given on the right-hand-side operand. If no such binding is found then the assignment takes place in the global environment.

What follows is a description of the example depicted by the sequence diagram in Figure 5.11. `makecounter` is bound in the global environment to a closure C that accepts no parameters, and whose environment is the global environment, by virtue of being defined from the command line. This state is depicted in Figure 5.12 and the sequence of actions which led it it can be seen in the evaluation of ‘A’ in Figure 5.11.

The body of the closure C bound to `makecounter (i)` initialises the variable `count` to circles as ‘free’ variables—they are non-local to the actual body of the closure; where local may be defined as the evaluation environment for the particular evaluation of a closure.

And the binding is not locked, i.e. its value may be modified. This provides a safeguard in the protection of bindings in attached packages, which may be identified in a process such as non-local assignment.
Defining `makecounter()` and evaluating it:

```r
makecounter <- function() {
  count <- 0
  function () {
    count <<- count + 1
    count
  }
}
```

Figure 5.11: Sequence diagram of establishing the `makecounter` closure, and then evaluating it and binding the result (another closure) to `counter`
Figure 5.12: Class diagram depicting interpreter state following creation of `makecounter` zero (in the evaluation environment created for the evaluation of `makecounter()`); and (ii) returns a closure F, whose environment is the evaluation environment. Therefore if F is bound to a symbol in some persistent environment (such as the global environment) then the evaluation environment used in the evaluation of `makecounter` is necessarily preserved, containing the binding `count` created during the course of evaluating C’s body. This state is depicted in Figure 5.13, and its preceding actions shown in the evaluation of ‘B’ in Figure 5.11.

The following is a description of the evaluation of ‘C’ in Figure 5.11. During evaluation of F (such as the calls to `counter()` in Listing 5.19) an evaluation environment E2 is created, whose enclosing environment is E, the environment of closure F. The expressions constituting F’s body are evaluated in E2, of particular interest is the assignment `count < = count + 1`. The RHS expression `count + 1` will be evaluated in E2 in which the search for a binding to symbol `count` will begin and then proceed through the chain of enclosing environments. It will of course be found in E2’s immediately enclosing environment—E, the environment of closure F—in which same environment the non-local assignment operation will locate and rebind the `count` symbol given as its LHS operand. The final line of F’s body simply returns the value of `count` (which again will be found via the search path to exist in the immediately enclosing environment of E.)

When an environment E that was initially constructed as the evaluation environment for a closure but has been persisted because the evaluation resulted in the creation of another closure whose environment is E, E will be referred to as a local environment.

These local environments fall outside of the scope of the design for recording provenance.
Figure 5.13: Class diagram depicting interpreter state following creation of `makecounter` and binding the result of its evaluation to `counter` described in Section 3.2, which monitors bindings in only the global environment.

This will now be exemplified. Listing 5.20 performs the construction of a `counter` as in the previous example, but instead of simply printing the result of the calls to `counter()`, they are respectively assigned to variables `x`, `y` and `z`.

Listing 5.20: The ‘counter’ example, augmented to assign results to variables

```r
> makecounter <- function() {
+ count <- 0
>  } function() {
+ count <<- count + 1
+ count
+ }
> counter <- makecounter()
> x <- counter()
> y <- counter()
> z <- counter()
> x
[1] 1
```
Evaluating a closure with state

```haskell
counter = counter()

evaluate (C, glEnv)
  findBinding("counter")
  value()

apply([], glEnv)
```

No such binding exists in E2, so the search passes to its closing environment E

```
invoke([], E)
```

```
create(E)
```

```
findBinding("count")
```

```
assign([1])
```

```
value()
```

```
findBinding("count")
```

```
[1]
```

```
[1]
```

Figure 5.14: Sequence diagram of evaluating the `counter` closure
The first time \texttt{counter()} is evaluated, the value of the \texttt{count} variable is incremented to 1, returned, and assigned to \texttt{x}. The second time \texttt{counter()} is evaluated, the value of \texttt{count} is incremented to 2, returned, and assigned to \texttt{y}. Similarly its third invocation results in \texttt{count} being incremented to 3 and assigned to \texttt{z}. Clearly the expressions that involve assignments to \texttt{y} and \texttt{z} have been influenced by the previous invocations of \texttt{counter()}. However, as shown in Listing 5.21, this fact has not been recorded.

Listing 5.21: Result of the ‘counter’ example illustrating the omission of provenance tracking in local environments

```r
> pedigree("y") # commands
[[1]]
makecounter <- function() {
  count <- 0
  function() {
    count <<- count + 1
    count
  }
}
[[2]]
counter <- makecounter()
[[3]]
y <- counter()

> pedigree("z") # commands
[[1]]
makecounter <- function() {
  count <- 0
  function() {
    count <<- count + 1
    count
  }
}
```

This behaviour occurs because there is no record of the binding `count` being read or written; only those in the global environment—`counter` is read, and `y` and `z` are written. During a single invocation of `counter()`, `count` will firstly be read, and then written.

### 5.4.3 Design

To overcome this it is necessary to instrument facilities for tracking provenance within local environments. In the context of the given example, the read of the binding to symbol `count` needs to be recorded, and when a new binding to symbol `count` is written it is attributed the previous binding of `count` as a parent.

The difficulty lies in determining which environments, or more specifically frames, should be instrumented for provenance-tracking facilities.

Environments and their frames are created frequently during the course of evaluating an expression and these frames may not necessarily be persisted for any particular purpose, and so should not necessarily be instrumented for provenance-tracking. This is a performance consideration, that consequently informs the design to track provenance in only those frames that survive the top-level expression evaluation. A CXXR-specific refinement of this practice is to perform a lightweight garbage collection at the end of the top-level expression, to further ensure that inaccessible frames are destroyed.

Figure 5.15 gives a sequence diagram to describe frame monitoring and Figure 5.16 shows the new static member attributes and operations on class Frame. Class Frame is instrumented with the ability to track the creation of each instance of Frame in a set. The idea of frame monitoring is introduced to mean that the creation and destruction of Frame instances should be recorded. At the beginning of a top-level expression frame monitoring is enabled (Algorithm 5.5), and it is disabled at the end of the top-level expression (Algorithm 5.6). When a frame is constructed, it is said to be registered whereby it is added to the set; conversely, a frame undergoing destruction is deregistered and is hence removed from the set.

### 5.4.4 Implementation

Listing 5.22 shows what is added to class Frame to implement the above design.
CHAPTER 5. FURTHER PROVENANCE

Figure 5.15: Sequence diagram showing frame monitoring

Figure 5.16: Class diagram showing new attributes and operations in class Frame

Algorithm 5.5 The Frame enableFrameMonitoring operation

1: procedure Frame.enableFrameMonitoring
2:    frame_monitoring ← true
3:    SET.CLEAR
Algorithm 5.6 The Frame disableFrameMonitoring operation
1: procedure Frame.disableFrameMonitoring
2:  frame_monitoring ← false
3:  gclite ▷ This is CXXR-specific performance consideration
4:  for all $F \in \text{set}$ do
5:    F.enableReadMonitoring
6:    F.enableWriteMonitoring

Listing 5.22: Additions to definition of class Frame to enable it to maintain a collection of its instances

```c++
class Frame : public GCNode {
private:
  class Comparator {
    public:
      bool operator() (const Frame* lhs,
                      const Frame* rhs) const {
        return (lhs < rhs);
      }
  }
  typedef std::set<const Frame*, Comparator> Set;

friend class SchwarzCounter<Frame>;
static bool s_frame_monitoring;
static Set* s_set;
[...]
public:
  static void cleanup();
  static void initialize();

  void deregisterFrame() {
    s_set->erase(this);
  }

  void registerFrame() {
    s_set->insert(this);
  }
};
```
static void enableFrameMonitoring(bool on);

namespace {
  CXXR::SchwarzCounter<CXXR::Frame> frame_schwarz_ctr;
}

The class defines a private type `Set` as an alias of a C++ `std::set` to contain pointers to `Frame`. A static member field `s_set` is defined of this type, whose purpose will be to maintain a collection of `Frame` objects created during the course of evaluating a top-level expression.

Class `Frame` already defined static member fields—the read and write monitors—but these do not require any special initialisation (their combined declaration.initialisation to NULL is sufficient) so this class did not previously require use of a Schwarz counter to ensure correct initialisation of its static members. Class `Frame` now has a friend class `CXXR::SchwarzCounter<Frame>` and declares an instance of this type as `frame_schwarz_ctr` in an anonymous namespace. The new member field `s_set` is initialised to an empty `Frame::Set` in the `initialize()` method that is invoked necessarily by `SchwarzCounter<Frame>`:

```cpp
void Frame::initialize() {
  s_set = new Set();
  s_frame_monitoring = false;
}
```

Similarly the `cleanup` method required by `SchwarzCounter` destroys `s_set`:

```cpp
void Frame::cleanup() {
  delete s_set;
}
```

The boolean member field `s_frame_monitoring` is used to signify whether frame monitoring is enabled (true) or disabled (false). This field is modified by the method `enableFrameMonitoring(bool on)` which will be discussed later.

Whenever a `Frame` is created and frame monitoring is enabled, it needs to be registered in the set `s_set`; and when it is destroyed, it must be deregistered. The member method `registerFrame` is used to register the instance of `Frame` on which it is called, by adding it to `s_set`. Conversely `deregisterFrame` deregisters from `s_set` the frame on which it is called. They are respectively called from the constructor and destructor of `Frame`, which are shown in Listings 5.23 and 5.24 respectively.
Listing 5.23: The constructor of Frame modified to register this Frame instance with the set of frames

```cpp
Frame()
: m_cache_count(0), m_locked(false),
  m_read_monitored(false), m_write_monitored(false)
{
  if (s_frame_monitoring)
    registerFrame();
}
```

Listing 5.24: The destructor of Frame modified to deregister this Frame instance with the set of frames

```cpp
~Frame()
{
  if (s_frame_monitoring)
    deregisterFrame();
  statusChanged(0);
}
```

The method `enableFrameMonitoring(bool on)` implements both design operations `enableFrameMonitoring` (Algorithm 5.5) and `disableFrameMonitoring` (Algorithm 5.6). Firstly, it sets the field `s_frame_monitoring` to its argument `on`, which informs the `{de,}registerFrame` methods as to whether they should perform any action. Secondly, if frame monitoring is being enabled (i.e. its `on` argument is true) at the start of a top-level expression then it clears `s_set` in anticipation of being populated with Frames created during evaluation of the top-level expression; if frame monitoring is being disabled at the end of a top-level expression, then a light garbage collection will be performed and each surviving frame will have its read and write monitors enabled. This method is given in Listing 5.25.

Listing 5.25: The Frame::enableFrameMonitoring(bool) method

```cpp
void Frame::enableFrameMonitoring(bool on)
{
  if (on) {
```
The points at which frame monitoring is enabled and disabled are respectively before and after evaluation of a top-level expression as illustrated in Figure 5.15. This is introduced in the `Rf_ReplIteration` function which handles the Read-Evaluate-Print Loop shown in Listing 5.26.

```
Listing 5.26: Outline of `Rf_ReplIteration` function with control of frame monitoring

```
5.4.5 Example

The result of evaluating the ‘counter’ example given in Listing 5.20 with provenance-tracking of local environments enabled can be seen by inspecting the pedigree of $z$, as shown in Listing 5.27. The pedigree of $z$ includes all of those previous calls to `counter()` which are crucial for accounting for the derivation of $z$ but were previously not included.

Listing 5.27: Result of the ‘counter’ example illustrating the inclusion of provenance tracking in local environments

```r
> pedigree("z")$commands
[[1]]
makecounter <- function() {
  count <- 0
  function() {
    count <<- count + 1
    count
  }
}
[[2]]
counter <- makecounter()
[[3]]
x <- counter()
[[4]]
y <- counter()
[[5]]
z <- counter()
```

This method requires a careful review of the way in which provenance is characterised in this system, as this method presents a perhaps unexpected nuance, as illustrated in Listing 5.28.

Listing 5.28: Illustration of side-effect of local environment provenance tracking

```r
> pedigree("counter")$commands
[[1]]
makecounter <- function() {
```
As can be seen, the three calls to `counter()` do not appear in its pedigree. Despite
the rebounding of the `count` symbol to different values in the local environment, there is no
record of this having any effect on `counter`. This is because `counter` itself has not been
rebound, because its own value has not been altered—it still points to the original closure.
Furthermore, the closure itself hasn’t been altered either; the only change that has taken
place is to a binding within the closure’s environment.

For this to be effective on `counter`, it would be necessary to regard any modification
to `count` to represent a change in the state of `counter`.

To consider the state of `counter` to not strictly depend on `count` seems, on the one
hand, counter-intuitive: the expectation of a provenance record is that it ought to con-
tain the complete sequence of commands that resulted in an item reaching a given state;
however, on the other hand, to consider `counter`’s state independent to that of `count
does remain faithful to our definition of a binding state’s provenance: operating at the
granularity of bindings, without there being a new binding created, there is nothing that
need be reflected in the provenance record.

Or more generally: if `B` is a binding in the global environment to a closure `F` whose
environment is `E`, the state of the bindings in `E` are not considered to be part of the state
of `B`.

The consequence of this is that it is not possible to reconstruct `B` from its pedigree;
however, in the alternative, it would be necessary to rule that `B` should be characterised
as being dependent upon `E`—either some element thereof or in its entirety—which raises
conceptual questions, as well as considerable practical difficulties.

Addressing the latter first: it would be practically impossible to implement this func-
tionality in CXXR. The mechanism by which environments enclose each other is strictly
unidirectional—it is not possible to traverse from an outer environment to an inner en-
vironment. It also challenges the concept of a binding’s ancestry—the other bindings on
which it depends—being defined as those things that were read before it was written in
the course of evaluating a top-level expression. In order to attribute changes to bindings in $E$ as representative of a change in $B$ then it would be necessary to modify or offer exception to this rule. It is not accurate to identify all bindings in $E$ as necessarily being precursors of $B$, but nor is it practical to identify precisely those that are.

Whether this is a satisfactory representation is a matter of perspective. At this stage, it is necessary to accept this as simply a limitation of the given approach. Section 5.4.6 of this chapter will discuss in some detail potential approaches to overcoming this limitation.

### 5.4.6 Discussion

To address this problem further, it would be necessary to move away from the definition of binding’s ancestry as being the sole determinant of its state. During the course of evaluating a top level expression those bindings read prior to the writing of a binding are considered to be its parents, which enables an ancestry to be established by tracing through the generations of parentage. Presently it is only these ancestors that are considered to have been influential in deriving a binding’s present state. If we wish to consider changes to bindings which are not directly incorporated in a binding’s ancestry as having exerted any influence over it, then this presents a challenge in how to conceptually model this behaviour while working within reasonable technical constraints.

A naive policy might attempt an approximation of this by assuming that an invocation of a closure bound to $B$ may have caused its state to change, and so should be considered to be a new version of $B$. One immediate problem with this approach is that a single closure might be referred to by two bindings, such as in $c_1 <- c_2 <- \text{makecounter}()$, in which case calls to $c_1()$ should, but would not, be recorded in the provenance of $c_2()$ (and vice versa.)

There are three issues surrounding this problem.

- What constitutes a change in $B$? If $E$ is some environment referred to by $B$, either directly or indirectly through some chain of environments, then how should changes to the bindings in $E$ be considered as influential on $B$.

- Is it possible to categorically determine which bindings in $E$ have been of influence to $B$? Identifying those items in $E$ whose alteration should be regarded as a change in the state of $B$.

- How should these changes be reflected in the state $B$? Does a change to some member of $E$ constitute a new $B$?
What constitutes a change in B?

This has previously been straightforward to define as the creation of a binding B. If the new binding B had been derived from the previous binding of B then this would be recorded in its parentage.

What changes in E determine B?

Where E is a local environment of a closure to which B is bound, B could conceivably take the following values:

- All items in E. Consider a change to any binding in E to represent a change in B. This may seem conceptually sound in the toy example, but this cannot borne out into more sophisticated scenarios. One immediate problem is that it is possible to manually modify the environment of the closure from E1 to E2: `environment(B) <- E2`. At which point E1 is no longer of any interest, but also everything in E2 should not necessarily be considered a determinant in the value of B, as this environment may well be populated with innumerable extraneous bindings.

- Bindings that were present in E at the time of B’s creation. This is still subject to the problem outlined in the previous point, some practical considerations of which include:

  - The process of modifying the environment of a closure from E1 to E2 would need to be augmented to apply those monitors on bindings in E1 to those of the same symbol in E2.

  - It is possible that a relevant binding B in E1 does not presently exist in E2. This introduces the problem of monitoring bindings which do not yet exist. The monitoring on B in E1 would need to be instrumented as soon as it was created.

- Bindings in E on which the return value of the closure referred to by B depends. The closure body may have no single consistent return value, and therefore closure evaluation would need special monitoring.

How are changes in E reflected in B?

There is no clear characterisation of how a state changes in E should be attributed to, or reflected in, the state, or provenance record, of B. One pertinent question is: should a change in E constitute a new binding of B?
If this were the case, then it might cause problems of repetition. Using the counter example, suppose that an invocation of \texttt{counter()} were to cause a \textit{new}—or perhaps only \textit{apparently new}—binding of \texttt{counter}, due to the change in state of its local environment. Following this approach, it is difficult for one to reason about \texttt{x <- counter()} as before: it would now involve \textit{two} bindings (one of \texttt{counter}) and one of \texttt{x}.

One potential direction of exploration for this could be instead of—as is currently done—attributing \texttt{count} as a parent of \texttt{x}, have \texttt{count} as a parent of the \textit{new} generation of \texttt{counter}, and have that \texttt{counter} as a parent of \texttt{x}.

\section*{Further Work}

The principal direction in which further work should be conducted is in relation to the idea of nested command scopes—i.e. attributing binding states at a finer granularity than just the top-level expression—to which Section \ref{sec:provenance} alluded, and what effect might be achieved if this approach were applied to local environments.

Another candidate for further attention would be enhancing the R functions for exploring provenance—namely \texttt{provenance} and \texttt{pedigree}—to better denote and interrogate the provenance of bindings in environments other than the global environment. Establishing an intuitive means of referring to an arbitrary environment is not trivial; the way in which R does this presently is by address:

\begin{verbatim}
> counter <- makecounter()
> environment(counter)
<environment: 0x7f8dbb7a3f40>
\end{verbatim}

Which would just about qualify as a decent starting point, but ultimately a more descriptive handle for environments would be preferable—something that would give a useful qualification to symbols such as these:

\begin{verbatim}
> provenance(counter)$children
[1] "count" "count" "count" "count"
\end{verbatim}
Chapter 6

Reproducible Research

Reproducible research, as introduced in 1, is a movement towards enabling third-party repetition of a scientific process described in a published work. This involves making available the data and code used to arrive at some result, along with the result itself. It was the second objective of this thesis to consider what Provenance-Aware CXXR can contribute to this field. This chapter will contend that the first step in making research reproducible is by accurately establishing provenance, and describe the ways in which Provenance-Aware CXXR—and more generally how provenance-aware software—can support reproducible research.

By recording the derivation of a data artifact, its derivation may be repeated to allow, in the context of a scientific application for instance, the results to be reproduced, perhaps independently to provide verification. In making the code and data that produced the results available alongside the results, the process becomes transparent and enables third-parties to repeat and either verify or, should this not prove to be possible, potentially identify and correct issues in the process. This of course is not possible while there exists to any degree opaqueness.

6.1 Provenance as the means to Reproducible Research

Just as it has been stated that it is by its metadata that data may be considered to be good, this may be similarly applied to the results of an analysis—the process which led to the results is crucial to adequately determine the quality of those results.

As the objective of reproducible research is to enable the repetition of some scientific process, such as a statistical data analysis, obviously a prerequisite of this is to record the process in the first place. By using provenance-aware software then this procedure
is automatically taken care of on behalf of the user, who need not worry about manual methods for recording and documenting the process, which may be ad-hoc, inconsistent, incomplete and—as is anything exposed to human involvement—prone to human error and should be considered generally unreliable. By automating this process, the reliability of the process documentation is ensured.

While simple replication of an analysis with original input data may only validate it with respect to claims made within those original confines, by recording and distributing the process documentation it becomes possible to repeat an analysis using different input data to further prove or disprove claims. This repeatability may also be required in an instance where some input data has been identified as being corrupt. Provenance-aware software should be able to identify which other data has been influenced by or directly derived from the suspect data, and regenerate these data in accordance with the original process.

The accountability of an analysis can be enhanced by accurate provenance documentation, as it becomes possible to validate what actually took place with respect to what was purported to have taken place in a descriptive text or article. Spotting any discrepancy in the accounts becomes trivial, and this for instance provides a means for reviewers of articles submitted to journals to verify what is described within those articles.

There is a considerable overlap between the objectives of the previously somewhat distinct fields of provenance and reproducible research, and it is clear that the rigorous application of provenance-aware computing methods is essential to enable and support reproducible research.

6.2 Reproducible Research in R

There are a few means to accomplish reproducibility within R, and these can be broadly classified into those based upon principals of literate programming, and those that are not.

6.2.1 Literate Programming

R’s standard library package “utils” includes a function Sweave, which facilitates the literate programming of R code [65]. As with literate programming in general, an Sweave document is written in some mark-up language such as \LaTeX{} or HTML, into which sections of R code are interleaved. When the document is processed by the Sweave function, each section of R code is replaced by the result of its evaluation, which may for instance be numerical values, tables, or graphical figures. Sweave utilises the Noweb format [94],
which offers a simple syntax for demarcating “code chunks” within the mark-up format and typically uses file extensions .Rnw and .Snw.

The two literate programming processes, weave and tangle, are implemented as Sweave and Stangle. When the user wishes to transform an Sweave document such as example.Rnw shown in Listing 6.2.1 into the human-readable document, the weave process would be invoked as follows: > Sweave("example.Rnw"). The result of this process is the \LaTeX file example.tex, which may be processed into a PDF directly from R using tools::texi2pdf("example.tex"), the result of which is shown in Figure 6.1.

One potential barrier to this method is the prerequisite experience of the user in some mark-up language such as \LaTeX or HTML to write the text portion of their document. An alternative is provided by the R package odfWeave [38], which allows users to instead use the OpenOffice word processor to write the text portion.

One further enhancement to sweave is Peng’s cacheSweave package, which enables the caching of the results of expressions evaluated during the course of weaving a document [91]. This overcomes Sweave’s default mode of operation whereby every section of code will be evaluated every time the document is processed, which can be a time and resource consuming process for lengthy analyses.

The knitr package [120] incorporates features of Sweave, cacheSweave and other R packages to produce a more powerful and transparent engine for dynamic report generation.
In this example we will generate a vector of 100 normally-distributed data points, whose mean we would expect to be around 0:

```r
> x <- rnorm(100, mean=0, sd=1)
> mean(x)
[1] 0.1104283
```

A histogram of the values:

```r
> hist(x)
```

![Histogram of x](example.pdf)

Figure 6.1: example.pdf generated using Sweave
Building on the literate programming paradigm, Gentleman and Temple Lang contend that the separation of a textual description of an analysis from the data and code on which it is founded often relegates the latter to an appendix and such disjoint treatment constitutes an obstacle to a user wishing to repeat the analysis.

For this reason, Compendium is proposed to be “one or more self-contained dynamic documents”, which is “an ordered composition of code chunks and text chunks that describe and discuss a problem and its solution” [40]. A compendium comprises dynamic documents, from which views may be transformed, with any other required data and auxiliary software.

The Compendium authors identify limitations in the data capture process during creation of a compendium. This includes the scenario in which data is necessarily anonymised, and defines the scope of the compendium may only reasonably exist from a certain starting point. The capture of data at a finer granularity is described as “practical problem, not a “conceptual one”.

### 6.2.2 Non-literate Programming

The cacher R package introduced in Section 1.4.7 describes a means by which an analysis in R may have its contents (both the sequence of expressions and their resultant objects) recorded in a ‘cache’, which may then be distributed to other users who may wish to repeat, verify or otherwise interact with it.

It achieves this by parsing a file containing R expressions and then supervising the evaluation of each expression. Each expression is attributed an identifier which is the SHA-1 digest of: the expression; the expression history (a string vector composed of all expressions preceding this expression); and the name of the source file. If, according to the identifier, there exists a previously cached result for this expression, then this cached result (i.e. R object(s)) is (lazily) loaded; otherwise evaluate the expression in a temporarily constructed environment \( E \). Prior to evaluation of an expression a list bindings in the global environment is saved and later compared with a list of bindings in the global environment after expression evaluation. These new bindings in the global environment along with those bindings within environment \( E \) are cached in the database against a key of the expression identifier. Finally, the bindings in \( E \) are recreated in the global environment, to mimic ordinary evaluation at the command prompt.

All objects are necessarily cached (with the exception of connection objects) and this
certainly offers performance advantages in some cases, for example where the analysis is simply being repeated without modification. However the number of these cases is quite small and it is very easy to invalidate cached results and require every expression to be evaluated. Because an expression is identified in part by the sequence of expressions preceding it, any change to this sequence—even one which is of no influence to any subsequent expression—will immediately invalidate the subsequent cached results.

For example consider the following file used as input to `cacher()`:

Listing 6.1: Trivial R Code for `cacher()` example

```
1 one <- 1
2 three <- 3
```

If one were to add as the second expression: `two <- 2`, then because the sequence of expressions preceding `three <- 3` has been altered, the cached result of this expression (and likewise all following expressions) will no longer be considered valid and its evaluation will be necessitated.

A more strict definition of the history of an expression E would identifying only those expressions that affected any of the bindings used as input to the evaluation of E; but without provenance-tracking facilities this is impossible within CR or its packages.

The global environment is considered as a special case, and only bindings created in the temporary environment and the global environment are considered to have resulted from the evaluation of an expression. This is to handle one case of expressions with side effects, in this instance where bindings in the global environment are modified. Other special cases handle those expressions which do not create or modify any bindings at all (either in the temporary environment or global environment), such as when output is to a graphics device (as in a call to `plot`), or when a call to `attach()` alters the search path. These expressions are deemed uncachable and their evaluation is necessarily forced and this property is recorded in the database.

### 6.3 Reproducible Research in CXXR

The literate programming approaches available within R allow its users to create documents that incorporate elements designed for human consumption, such as the manuscript for a paper that includes results; as well as elements designed for computer consumption such as the data and code used to derive the included results. This methodology has the
advantage of encapsulating some code contiguous with its natural language description, so that when presented with the document, one would be able to (a) repeat the code-chunks; (b) verify the results against those in the original paper; and (c) corroborate the actual process of deriving the results with the natural language description of the process.

However, for an author who wishes to publish a result, this method does not provide the means to systematically establish the process that derived that result. The process of producing a literate programming document is conducted in vitro. The author must rely on his or her own ad-hoc method to maintain a record of the analysis that took place and what elements of that analysis must be included in the document to form a sufficient account of it.

In order to achieve this in a robust manner, it is necessary for the provenance of data artifacts to be recorded and persisted. As has been herein demonstrated, Provenance-Aware CXXR performs this crucial aspect of housekeeping on the user’s behalf and better facilitates this aspect of reproducible research than the present approaches. Implementing rigorous provenance-tracking facilities to software enables it to automatically document its processes in a manner that is systematic, accurate and consistent, and it is ultimately these characteristics that must be satisfied if the process is to be reproduced. In particular it is intended that by providing such functionality, Provenance-Aware CXXR can complement the popular literate programming in R approach.

Furthermore, as a result of its serialization/deserialization facility, Provenance-Aware CXXR offers a means to encapsulate for distribution data resulting from an analysis along with its provenance, therefore including the original source data and the entire sequence of commands responsible for deriving the resultant data. From this emerge advantages over a provenance-unaware system such as cacher. The availability of accurate provenance information allows for the identification of the ancestors of a data artifact and therefore allows the analysis to be easily and efficiently repeated with different input data. When an input, such as a dataset, or parameter value or even an entire expression is modified, it is possible using the provenance information recorded by P-A CXXR to identify and deterministically evaluate only those expressions whose evaluation will be affected by the change, as opposed to evaluating each and every (potentially time and resource consuming) extraneous expression. Therefore this approach facilitates not only repetition of the original analysis, but also by facilitating the application of a given process to another set of input data, a degree of independent verification.
Chapter 7

Conclusions

The research presented in this thesis has sought to achieve the objectives set out in Section 1.5.1. The first of these objectives was to understand how and the extent to which facilities for recording, preserving and querying provenance information can be introduced to the CXXR implementation of the R statistical language and environment. This objective was then split into separate research goals.

Chapter 1 sets out the motivation for this work, and the typical use cases in which provenance questions arise. Owing partially to its heritage as a spiritual descendent of S, CXXR is used quite typically for conducting exploratory data analysis, which gives rise to particular provenance questions.

In recent years, the field of provenance in computing has received increasingly wide interest, no better exemplified than by the success of IPAW conferences and such a mature standard as W3C’s PROV specification. While there are documented accounts of software being adapted to become provenance-aware, such facilities have not been incorporated to an environment to support an interpreted language such as R. Chapter 2 explains why the (C)R interpreter itself is not well-suited to exploratory implementation of functionality, and that this is the main motivation behind CXXR, which provides the necessary opportunity to introduce provenance tracking to the R language and environment.

The field of reproducible research has also seen a remarkable increase in interest in recent years. As introduced in Section 1.4, there are strong motivators for this: researchers and publishers alike are increasingly viewing the inability to reproduce published results as being a significant barrier to progress, as well as potentially damaging to the credibility of all concerned. As Chapter 6 identifies: the reproducible research effort has long been conducted without due consideration to the provenance of data.

No more so is this typified than in the R packages intended to facilitate reproducible
research. Some of these may go a long way towards solving certain issues of reproducibility such as logging commands, caching results, and packaging things for easy distribution; while other means such as literate programming involve manually annotating a process—all in a fashion that promotes tight encapsulation of source code and results—in a slightly *in vitro* manner: there is no accommodation for recording the *live* exploratory component of an analysis. There remains a conspicuous omission in these methods for reproducible research in R: how can we categorically identify the process that led to a particular object?

The features designed in this thesis demonstrate that, in general, interactions with R objects can be monitored, recorded and distilled into provenance information about R objects that can be queried by the user. This thesis has also shown it is possible to preserve this provenance information along with the R object to which it pertains. As discussed in Chapter 5, there are aspects of the R language as well as the CR interpreter that present edge cases that are not adequately catered for by the general algorithm, and these have received special attention. During these investigations, it has become apparent that there is scope for improvement to the methods employed. One such area will be discussed in Section 7.2.

### 7.1 Contributions

This thesis presents in Chapters 3, 4, and 5 original research to investigate an approach to introducing facilities for recording, preserving, and querying provenance in the CXXR implementation of the R statistical language and environment.

Chapter 3 describes the motivating provenance questions that a provenance-aware CXXR should be capable of answering (research goal 2). A view of provenance in CXXR is described that pertains to the *binding*, at a granularity of the *top-level expression*. A design for capturing provenance in CXXR takes into consideration the read-evaluate-print-loop strategy, and introduces *monitors* to capture read and write operations performed on bindings during the evaluation of a top-level expression. Other considerations that inform the design presented at this stage are maintaining the granularity of the top-level expression in loops and how the provenance information is queried by the user by an *in-interpreter* interface (research goal 3). With respect to a real-world R exploratory data analysis, this chapter also presents a performance analysis of the provenance-tracking facilities in CXXR, and demonstrates how the provenance questions can be answered (research goal 6).
The work presented in Chapter 4 gives an account of how a serialisation facility has been engineered into CXXR to allow objects within a session to be saved to a file along with their provenance information, all of which can be restored to a subsequent session by loading the file. This enables the user to ask provenance questions of objects created in previous sessions, and also enable newly created objects to include in their pedigree, the pedigree of objects used in their creation, even if these objects were created in a previous CXXR session (research goal 4). Chapter 6 describes how, further to simply providing general utility, the provenance-awareness introduced to CXXR along with its serialisation capabilities may be used to support reproducible research. This chapter also addresses how provenance information recorded in CXXR can be enabled for interoperability (research goal 5).

Chapter 5 addresses a number of scenarios that require special design considerations in order that sufficient provenance information is recorded to provide answers to provenance questions in these instances (research goals 3). Much of the focus of this chapter is on recording provenance of expressions evaluated outside the main read-evaluate-print loop, including the expressions themselves (in the case of Sections 5.1 and 5.2) as well as functions which accept input from outside means (Section 5.3). The sections of this chapter—to some extent independently—conclude that in order to further refine the provenance collection, some flexibility of granularity is required.

### 7.2 Further Work

#### 7.2.1 Provenance-Aware CXXR

As suggested by the previous section, the principal candidate area for conducting further work in is that of granularity, and in particular, to investigate the effects of tracking provenance at more than one level.

The example originally encountered in Section 5.1.3 included nested invocations of source, beginning with source("liftExample2.R"), where liftExample2.R’s contents was:

```r
source("example.R")
z <- y * 2
```

And example.R contained:
At present we can adequately express the relationships between the *bindings* through the parentage relation as shown in Figure 7.1(a) one of the challenges remaining is to adequately express the relationships between the *expressions*. A potential expression relationship graph for the above is depicted in Figure 7.1(b).

```
x <- date()
y <- rnorm(10)
strs <- paste(x, y)
```

This issue concerns the **granularity** at which an expression is attributed to a binding it created. Presently, it is only possible to record the top-level expression that gives rise to a binding state—with the exception of the handling of `source` given in Section 5.1.

One potential design for the handling of this would be to trace the *nesting* of expression evaluations, so that the granularity is not fixed at only the top-level expression and, allowing individual binding states to be attributed to the sub-expressions from whose evaluation they resulted.

It is possible that such an approach would alleviate the difficulties of handling `source` calls, functions with local environments, and xenogenesis.

Another area for potential exploration might be whether serialisation could occur natively to a format compatible with PROV-DM, and how CXXR can operate on provenance information generated by other software.

Finally, further user interfaces to provenance information in CXXR could also be explored. In particular, some form of graph visualisation could be valuable. However, R does not have a native package for graph visualisation, and its visualisation facilities may not provide adequate interactivity for exploring a large provenance graph, but there is definite scope for investigation.

It is intended that the work presented in this thesis forms a strong foundation for further exploration into provenance-awareness in CXXR and software that exhibits similar characteristics—such as mutable binding variables; REPL-based expression evaluation; lazy-evaluation; closure evaluation—with a view that provenance-aware software has a
vital role to play in furthering reproducible research.

7.2.2 Reproducible Research

Provenance-Aware CXXR lays the foundation for facilitating reproducible research. Further areas of exploration might involve the addition of R functions to, for instance, determine whether a binding is out of date with respect to its ancestors, in which case it may be regenerated; and to replay an analysis with different input data or parametrised computations.

The environment mechanism could be used to support this: if a user wished to regenerate a binding (or set of bindings) using an ‘input’ binding with a different value, P-A CXXR could construct a new environment, and using its provenance for the output bindings, determine which bindings are duplicated into the new environment, and which expressions should be evaluated in order to generate the desired output.

This sort of functionality could have implications for the way in which historical bindings (perhaps in the case where symbols are, over time, used to bind different values) are denoted or referred to.

Reproducible research in areas in which many R users work may involve sensitive data that requires anonymisation. This is a scenario that presents problems to provenance-aware systems in general as well as to P-A CXXR.
Bibliography


[58] John P A Ioannidis, David B Allison, Catherine A Ball, Issa Coulibaly, Xiangqin Cui, Aedin C Culhane, Mario Falchi, Cesare Furlanello, Laurence Game, Giuseppe


[77] Luc Moreau and Ian Foster, editors. *Provenance and Annotation of Data*, volume 4145 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006. 6


[91] Roger D. Peng and with contributions from Tobias Abenius. cacheSweave: Tools for caching Sweave computations, 2012. R package version 0.6-1. 189


[99] Satya Sahoo, Paul Groth, Olaf Hartig, Simon Miles, Sam Coppens, James Myers, Yolanda Gil, Luc Moreau, Jun Zhao, Michael Panzer, and Daniel Garijo.
BIBLIOGRAPHY


Appendix A

Exploring R

A.1 Operators in R

The base package in R contains, among other things, primitive functions. A list of these may be retrieved as shown in Listing A.1. And they may be divided into their two types, builtin and special as shown in Listing A.2.

Listing A.1: Extract a list of primitive functions from R-2.15.1

```r
> obs <- ls("package:base", all.names=TRUE)
> prims <- sapply(obs, function(x) is.primitive(get(x)))
> primFunctions <- obs[prims]
> primFunctions

[1] 
[4] 
[7] 
[10] 
[13] 
[16] 
[19] 
[22] 
[25] 
[28] 
[31] 
[34] 
[37] 
[40] 
[43]

```

210
APPENDIX A. EXPLORING R

Listing A.2: Distinguish between `builtin` and `special` primitive functions

```r
> funTypes <- split(primFunctions, 
sapply(primFunctions, function(x) typeof(get(x))))
> names(funTypes)
[1] "builtin" "special"
> sapply(funTypes, length)
builtin special
166 35
> funTypes$special
[1] ".Internal" "," ",[" ",][<-" ",]<-
[6] "." ".O" ".&" ".<-" ",<=-
[11] ".=" ",|" ",-" ",\$" ",\$<-
[16] "break" "call" "expression" "for" "function"
[21] "if" "log" "missing" "next" "on.exit"
[26] "quote" "rep" "repeat" "return" "round"
[31] "signif" "substitute" "switch" "UseMethod" "while"
> funTypes$builtin
[1] "+" ",." ",|"
[4] ",!=" ",.C" ",.cache_class"
[7] ",.Call" ",.Call.graphics" ",.External"
[10] ",.External.graphics" ",.Fortran" ",.Primitive"
[13] ",.primTrace" ",.primUntrace" ",.subset"
[16] ",.subset2" ",(" ",+"
[19] ",/" ",\" ",\%\%"
[...]
```
Appendix B

XML Serialization

Given the following R session:

```r
> myVar <- "Hello, XML Serialization"
> bserialize()
```

The following Listing details the XML output. This is a necessarily brief session, for the sake of space. If one were to append "!" (using the R function `paste`) to `myVar`, with the added Provenance information this file more than doubles in length (244 lines as opposed to 113).

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE boost_serialization>
<boost_serialization signature="serialization::archive" version="10">
<env class_id="1" class_name="CXXR::Environment" tracking_level="1" version="0" object_id="_0">
<RObject class_id="2" tracking_level="0" version="0">
<GCNode class_id="0" tracking_level="1" version="1" object_id="_1"></GCNode>
$type>4</type>
<m_attrib class_id="-1"></m_attrib>
</RObject>
<envtype>6</envtype>
<m_enclosing class_id="-1"></m_enclosing>
<Frame class_id="4" tracking_level="0" version="0">
<GCNode object_id="_3"></GCNode>
<locked>0</locked>
</Frame>
<symbol class_id="5" class_name="CXXR::Symbol" tracking_level="1" version="0" object_id="_4">
<RObject>
<GCNode object_id="_5"></GCNode>
$type>1</type>
<m_attrib class_id="-1"></m_attrib>
</RObject>
symtype>0</symtype>
<name>myVar</name>
<m_atrib class_id="-1"></m_atrib>
</RObject>
<envtype>6</envtype>
<m_enclosing class_id="-1"></m_enclosing>
<Frame class_id="3" class_name="CXXR::StdFrame" tracking_level="1" version="0" object_id="_2">
<numberOfBindings>1</numberOfBindings>
<symbol class_id="5" class_name="CXXR::Symbol" tracking_level="1" version="0" object_id="_4">
<RObject>
<GCNode object_id="_5"></GCNode>
$type>1</type>
<m_atrib class_id="-1"></m_atrib>
</RObject>
<symtype>0</symtype>
<name>myVar</name>
<m_atrib class_id="-1"></m_atrib>
</Frame>
<symbol class_id="6" class_name="CXXR::StringVector" tracking_level="1" version="0" object_id="_6">
<size>1</size>
<VectorBase class_id="8" tracking_level="0" version="0" object_id="_7"></VectorBase>
<RObject object_id="_7"></RObject>
$type>16</type>
</symbol>
```

212
<m_attrib class_id="-1"></m_attrib>
</RObject>
<m_truelength>1</m_truelength>
</VectorBase>
<numnas>0</numnas>

<item class_id="9" class_name="CXXR::String" tracking_level="1"
version="0" object_id="_8">
<RObject>
<GCNode object_id="_9"></GCNode>
?type>9</type>
<m_attrib class_id="-1"></m_attrib>
</RObject>
<isna>0</isna>
<string>Hello, XML Serialization</string>
<m_encoding>0</m_encoding>
</item>
</m_value>

<m_provenance class_id="10" class_name="CXXR::Provenance" tracking_level="1"
version="0" object_id="_10">
<symbol class_id_reference="5" object_id_reference="_4"></symbol>
<chronicle class_id="11" class_name="CXXR::CommandChronicle" tracking_level="1"
version="0" object_id="_11">
<command class_id="12" class_name="CXXR::Expression" tracking_level="1"
version="0" object_id="_12">
<ConsCell class_id="13" tracking_level="0" version="0">
<RObject>
<GCNode object_id="_13"></GCNode>
?type>6</type>
<m_attrib class_id="-1"></m_attrib>
</RObject>
<m_tag class_id="-1"></m_tag>
<m_car class_id_reference="5" object_id="_14">
<RObject>
<GCNode object_id="_15"></GCNode>
?type>1</type>
<m_attrib class_id="-1"></m_attrib>
</RObject>
<m_tag class_id="-1"></m_tag>
<symtype>0</symtype>
</name>&lt;-</name>
</m_car>
<m_tail class_id="14" class_name="CXXR::PairList" tracking_level="1" version="0"
object_id="_16">
<ConsCell>
<RObject>
<GCNode object_id="_17"></GCNode>
?type>2</type>
<m_attrib class_id="-1"></m_attrib>
</RObject>
<m_tag class_id="-1"></m_tag>
<m_car class_id_reference="5" object_id_reference="_4"></m_car>
<m_tail class_id_reference="14" object_id="_18">
<ConsCell>
<ConsCell>
<RObject>
<GCNode object_id="_19"></GCNode>
?type>2</type>
<m_attrib class_id="-1"></m_attrib>
</RObject>
<m_tag class_id="-1"></m_tag>
<m_car class_id_reference="7" object_id_reference="_6"></m_car>
<m_tail class_id="-1"></m_tail>
</ConsCell>
</ConsCell>
</m_tail>
</m_tail>
</ConsCell>
</command>
<GCNode object_id="_20"></GCNode>
<size>0</size>
</chronicle>
<GCNode object_id="_21"></GCNode>
<sec>1387796556</sec>
<usec>508568</usec>

<m_num_parents>0</m_num_parents>
<m_value class_id="-1"></m_value>
<m_xenogenous>0</m_xenogenous>
</m_provenance>
<m_origin>0</m_origin>
<m_active>0</m_active>
<m_locked>0</m_locked>
</m_frame>
<m_single_stepping>0</m_single_stepping>
<m_locked>0</m_locked>
</env>
</boost_serialization>
Appendix C

Air Quality Analysis

Here is R.D. Peng’s air quality analysis in R.

```r
pm0 <- read.table("pm25_data/RD_501_88101_1999-0.txt", comment.char = "#", header = FALSE, sep = "|", na.strings = "")
dim(pm0)
head(pm0[, 1:13])
cnames <- readLines("pm25_data/RD_501_88101_1999-0.txt", 1)
cnames <- strsplit(cnames, "|", fixed = TRUE)
names(pm0) <- make.names(cnames[[1]]) ## Ensure names are properly formatted
head(pm0[, 1:13])
x0 <- pm0$Sample.Value
summary(x0)
mean(is.na(x0)) ## Are missing values important here?

pm1 <- read.table("pm25_data/RD_501_88101_2012-0.txt", comment.char = "#", header = FALSE, sep = "|", na.strings = "")
names(pm1) <- make.names(cnames[[1]])
x1 <- pm1$Sample.Value
boxplot(log2(x0), log2(x1))
summary(x0)
summary(x1)

negative <- x1 < 0
mean(negative, na.rm = T)

dates <- as.Date(as.character(pm1$Date), "%Y%m%d")
missing.months <- month.name[as.POSIXlt(dates)$mon + 1]
tab <- table(factor(missing.months, levels = month.name))
round(100 * tab/sum(tab))

site0 <- unique(subset(pm0, State.Code == 36, c(County.Code, Site.ID)))
site1 <- unique(subset(pm1, State.Code == 36, c(County.Code, Site.ID)))
site0 <- paste(site0[, 1], site0[, 2], sep = ".")
site1 <- paste(site1[, 1], site1[, 2], sep = ".")
str(site0)
str(site1)
both <- intersect(site0, site1)
print(both)

## Find how many observations available at each monitor
pm0$county.site <- with(pm0, paste(County.Code, Site.ID, sep = "."))
pm1$county.site <- with(pm1, paste(County.Code, Site.ID, sep = "."))
cnt0 <- subset(pm0, State.Code == 36 & county.site %in% both)
cnt1 <- subset(pm1, State.Code == 36 & county.site %in% both)
sapply(split(cnt0, cnt0$county.site), nrow) ## 1999
both.county <- 63
both.id <- 2008

## Choose county 63 and side ID 2008
pm1sub <- subset(pm1, State.Code == 36 & County.Code == both.county & Site.ID == both.id)
both.date <- as.Date(as.character(pm1sub$Date), "%Y%m%d")
```

215
x1sub <- pm1sub$Sample.Value
dates0 <- as.Date(as.character(pm0sub$Date), "%Y%m%d")
x0sub <- pm0sub$Sample.Value

## Find global range
rng <- range(x0sub, x1sub, na.rm = T)
par(mfrow = c(1, 2), mar = c(4, 5, 2, 1))
plot(dates0, x0sub, pch = 20, ylim = rng, xlab = "", ylab = expression(PM[2.5] * " (\mu \text{ g/m}^3\) "))
abline(h = median(x0sub, na.rm = T))
plot(dates1, x1sub, pch = 20, ylim = rng, xlab = "", ylab = expression(PM[2.5] * " (\mu \text{ g/m}^3\) "))
abline(h = median(x1sub, na.rm = T))