Making Provenance Work for You

by Barbara Lerner, Emery Boose, Orenna Brand, Aaron M. Ellison, Elizabeth Fong, Matthew K. Lau, Khanh Ngo, Thomas Pasquier, Luis Perez, Margo Seltzer, Rose Sheehan, Joseph Wonsil

Abstract To be useful, scientific results must be reproducible and trustworthy. Data provenance—the history of data and how it was computed—underlies reproducibility of, and trust in, data analyses. Our work focuses on collecting data provenance from R scripts and providing tools that use the provenance to increase the reproducibility of and trust in analyses done in R. Specifically, our "End-to-end provenance tools" ("E2ETools") use data provenance to: document the computing environment and inputs and outputs of a script's execution; support script debugging and exploration; and explain differences in behavior across repeated executions of the same script. Use of these tools can help both the original author and later users of a script reproduce and trust its results.

Introduction

In today's data-driven world, an increasing number of people are finding themselves needing to analyze data in the course of their work. Often these people have little or no background or formal coursework in programming and may think of it solely as a tedious means to an interesting end. Writing scripts to work with data in this way is often exploratory. The researcher may be writing a script to produce a plot that enables visual understanding of the data. This understanding might then lead to a realization that the data need to be cleaned to remove bad values, and statistical tests need to be performed to determine the strength or trends of relationships. Examining these results may raise more questions and lead to more code. This type of exploratory programming can easily lead to scripts that grow over time to include both useful and irrelevant code that is difficult to understand, debug, and modify.

Creating a script and successfully running it once to analyze a dataset is one thing. Reproducing it later is another thing entirely. We might expect that re-running a script and reproducing a data analysis should be a simple matter of rerunning a program or script on the same data, but it is rarely that simple. Anyone who has tried to retrieve the version of the data and scripts used to produce the results presented in a paper will likely appreciate how difficult this can be. Data and scripts can be modified or lost. But even if care is taken to save the scripts and data, new versions of programming languages, libraries and operating systems may make scripts behave differently or be unable to run at all. In an ideal world, everything would be backwards-compatible, but in reality, what ran last week often doesn't run next week. It can be difficult to determine what went wrong, especially if programming is an occasional activity. The National Academy of Sciences report on Reproducibility and Replicability in Science (National Academies of Sciences, Engineering, and Medicine, 2019) describes at length the challenges associated with computational reproducibility of scientific results.

Motivated by an interest in supporting reproducibility of R scripts, we developed a package called **rdtLite** to collect data provenance containing a record of a script's execution and the environment in which it was executed (Lerner et al., 2018). Having done that, we then realized that the wealth of information contained in the data provenance could serve other purposes as well. This led to the development of End-to-End Provenance Tools ("E2ETools"): an evolving set of R packages that use data provenance to help users save workable copies of their data and scripts, debug them, understand how data and results of analyses were derived, discover what has changed when a script stops working, and reproduce prior results.

What is data provenance?

Provenance is the history of creation, ownership, chain-of-custody, and location of an object. In its original and still most-frequently used sense, provenance is used to authenticate and trace the legitimate ownership of a work of art; it confers, creates, or adds value to the work itself. But provenance can be constructed, identified, or traced for any object, including data (Becker and Chambers, 1988). Data provenance is analogous to provenance of a work of art in that it includes the history of a datum or entire dataset from the point at which it was collected (by a person or sensor), created (by a computational process), or derived (from other data). Data provenance also confers or adds value—as trustworthiness—to data, but data provenance can do more: it can be used to reproduce computational analyses and validate scientific conclusions.

More precisely, data provenance is the history of a data item ("datum") or a dataset ("data"); it describes **how** the datum or data came to be in its present state. Our E2ETools focus on *language-level provenance*: how data are created and manipulated by a programming language such as R during

```
# Load the mtcars data set that comes with R
data(mtcars)

# All the cars
allCars.df <- mtcars

# Create separate data frames for each number of cylinders
cars4Cyl.df <- allCars.df[allCars.df$cyl == 4, ]
cars6Cyl.df <- allCars.df[allCars.df$cyl == 6, ]
cars8Cyl.df <- allCars.df[allCars.df$cyl == 8, ]

# Create a table with the average mpg for each # cylinders
cylinders = c(4, 6, 8)
mpg = c(mean(cars4Cyl.df$mpg), mean(cars6Cyl.df$mpg), mean(cars8Cyl.df$mpg))
cyl.vs.mpg.df <- data.frame (cylinders, mpg)

# Plot it
plot(cylinders, mpg)
```

Listing 1: Source code for mtcars_example.R. This code is used to demonstrate the lineage traces provided by the debug.lineage function as described in the text

the execution of a script or program. Provenance is also referred to in other computing contexts. For example, data provenance can be used to understand results of queries to a database or to the processes that were used to create or modify a file. In the remainder of this paper, however, when we say "provenance" or "data provenance", we specifically mean language-level provenance.

We associate three types of information with provenance: environment information, coarse-grained information, and fine-grained information. *Environment information* includes information about the computing environment in which the script was executed. This includes information such as the operating system version, the R version, and the versions of the R libraries used, as each of these may play a role in understanding the details of how a script behaves. *Coarse-grained information* includes the source code of the script(s), the data input to the script, the data output by the script, and plots produced by the script. *Fine-grained information* includes an execution trace. Specifically, for each line of the script that is executed, fine-grained information includes the data used on that line and any data computed by, or object created by, that line. Our E2ETools can use this fine-grained information to help a user understand exactly how any data value or object in the script was computed or derived.

A first example

Consider this simple example, 'mtcars_example.R', that loads in the 'cars' dataset and plots miles per gallon (mpg) as a function of the number of cylinders (cylinders) (Listing 1).

The following commands run the script, collect its provenance, and produce a textual summary of the provenance.

```
library(rdtLite)
prov.run("mtcars_example.R")
prov.summarize()
```

The provenance summary is shown in Listing 2. The environment information (lines 3–18) reports details of the computing environment in which the script was executed, such as the processor and operating system on which it ran and the version of R and R libraries used. The coarse-grained information (lines 20–36) identifies the location in the file system of the script, the input dataset, and the plot produced. The fine-grained information, which is not displayed by `prov.summarize()` but is accessible via other tools, indicates the input and output data for each line of code executed, linking them together so that one can see how the values computed in one statement are used in later statements. For example, the provenance debugger can use fine-grained information to display everything that is derived from a variable.

```
library(provDebugR)
prov.debug()
debug.lineage("cars4Cyl.df", forward = TRUE)
```

The resulting output displays the line numbers and code for everything computed, either directly or indirectly, from `cars4Cyl.df` .

```
Var cars4Cyl.df
8:    cars4Cyl.df <- allCars.df[allCars.df$cyl == 4, ]
```

```
PROVENANCE SUMMARY for mtcars_example.R

ENVIRONMENT:
Executed at 2022-07-28T13.52.25EDT
Total execution time was 1.516 seconds
Script last modified at 2022-07-22T10.41.25EDT
Executed with R version 4.2.1 (2022-06-23)
Platform was x86_64, darwin17.0
Operating system was macOS Catalina 10.15.7
User interface was 2022.02.3+492 Prairie Trillium (desktop)
Document converter was 2.2.1 @ /usr/local/bin/pandoc
Provenance was collected with rdtLite1.4
Provenance is stored in /Users/blerner/tmp/prov/prov_mtcars_example
Hash algorithm is md5

LIBRARIES (loaded by script):
None (see notes below)

SCRIPTS:
1[:] /Users/blerner/Documents/Process/DataProvenance/Papers/RJournal/scripts/examples/
    mtcars_example.R

PRE-EXISTING:
None

INPUTS:
1[:] /Library/Frameworks/R.framework/Versions/4.2/Resources/library/datasets/data/Rdata.rds

OUTPUTS:
1[-] /Users/blerner/Documents/Process/DataProvenance/Papers/RJournal/scripts/dev.off.11.pdf

CONSOLE:
None

ERRORS & WARNINGS:
None

NOTES: Files are listed in the order of execution (script 1 = main script).
The status of each file in its original location is marked as follows:
File unchanged [:], File changed [+], File missing [-], Not checked [ ].
Copies of original files are available on the provenance directory.

Libraries loaded by the user's script at the time of execution are displayed.
Note that some libraries may have been loaded before execution. Use details =
TRUE to see all loaded libraries along with script, file, and message details.
```

Listing 2: Provenance summary for mtcars_example.R, showing the environment in which the script was executed, identifying the script, input and output files, and any errors or warnings encountered when the script was executed.

```
14:   mpg = c(mean(cars4Cyl.df$mpg), mean(cars6Cyl.df ...
15:   cyl.vs.mpg.df <- data.frame (cylinders, mpg)
18:   plot(cylinders, mpg)
NA:   mtcars_example.R
```

Alternatively, a modified version of the same command

```
debug.lineage("cars4Cyl.df")
```

shows the lines of code that lead to the value for `cars4Cyl.df` being computed.

```
Var cars4Cyl.df
2:   data(mtcars)
5:   allCars.df <- mtcars
8:   cars4Cyl.df <- allCars.df[allCars.df$cyl == 4, ]
```

Having seen an introductory example of some things the E2ETools can do, we now turn to a more detailed discussion of each tool.

The end-to-end provenance tools

The E2ETools consist of three types of packages:

- A package to collect provenance: **rdtLite**;
- Packages that process data provenance to provide information to the user about a particular script and its execution: **provSummarizeR, provDebugR, provViz,** and **provExplainR**;

- Packages to enable tool developers to more easily use data provenance: **provParseR** and **provGraphR**.

We describe each of these packages, beginning with provenance collection. All the tools described are available on CRAN.

Collecting provenance with rdtLite

The rdtLite package collects provenance from R scripts as they execute.[1] rdtLite captures provenance data from both scripts and interactive console sessions. To capture provenance for a script, the user runs the script using the `prov.run` function.

```
library(rdtLite)
prov.run("script.R")
```

To collect provenance for an interactive session, the user begins the session with the `prov.init` function and concludes it with `prov.quit`.

```
library(rdtLite)
prov.init()
data <- read.csv("mydata.csv")
plot(data$x, data$y)
prov.quit()
```

rdtLite collects information about each file or URL read by the script, each file written by the script, and each plot created by the script. In addition, it records an execution trace of the top-level R statements. This trace identifies the statement executed. It records any variables set or used by the statement. When a variable is set, it records the type of the value, including its container (such as vector, data frame, etc.), dimensions, and class (e.g., character, numeric). If the container is a vector of length 1, rdtLite records its data value, embedded in the provenance (which is stored in a JSON file). rdtLite can save the values of larger containers in separate snapshot files. The user controls how much data to save using the `snapshot.size` parameter in `prov.init` and `prov.run`. The default is to not save snapshots. rdtLite also records any warning or error messages generated when the statement is executed. To capture similar information about scripts that are included using the `source` function, calls to `source` must be replaced with calls to `prov.source`.

The provenance is stored in a JSON file using a format that extends the PROV-JSON standard (W3C, 2014).[2] The extended format provides structured information about fine-grained provenance, such as a list of libraries used, a mapping from functions called to the libraries from which they came, script line numbers, and data values and their types. More information about the extended JSON format is provided in the Appendix.

The JSON file is stored in a provenance directory that also contains copies of all input and output files and the R scripts executed. By default, the provenance data is stored in the R session temporary directory, but the user can change this location either at the time that `prov.run` or `prov.init` is called or by setting the `prov.dir` option, for example, in the .Rprofile file.

Upon completion of a script called with `prov.run`, or after a call to `prov.quit`, rtdLite creates and populates a directory named either '`prov_script`', where '`script`' is the name of the script file, or '`prov_console`' for an interactive session. The directory will contain:

- 'prov.json' - the JSON file containing the fine-grained provenance
- 'data' - a directory containing copies of input and output files, URLs, plots created, and snapshot files.
- 'scripts' - a directory containing a copy of the scripts for which provenance was collected.

The rdtLite default is to overwrite this information if the same script is executed again or if `prov.init` is used again in a console session. However if the `overwrite` parameter is set to FALSE, the provenance is stored in a unique, time-stamped directory, allowing provenance from multiple executions to be analyzed and compared.

Using provenance

Having the provenance is extremely valuable, but it is not particularly usable without tools that read the provenance and provide *information* or enable *reproducibility*. We next describe four tools that

[1]rdtLite is a simplified version of RDataTracker (Lerner and Boose, 2014b; Lerner et al., 2018).

[2]`https://github.com/End-to-end-provenance/ExtendedProvJson/blob/master/JSON-format.md`.

use provenance to help R programmers understand executions of their script. The **provSummarizeR** package provides a concise textual summary of an execution. The **provViz** package provides a graphical visualization of the provenance. The **provDebugR** package uses collected provenance to help programmers debug their code. The **provExplainR** package compares provenance from two executions to help the programmer understand changes between them. These applications exist in packages separate from **rdtLite** and would work equally well with provenance collected by other tools that produce the same JSON format.

provSummarizeR

The purpose of **provSummarizeR** is to produce a concise record of the environment in which a script was executed. This information could be particularly valuable when including a script and its results in a paper, or when sharing a script with a colleague. For an example, please see Listing 2 above. The summary includes the following information:

- The ENVIRONMENT section shows information about when the script was modified and executed, what version of R was used, what hardware and operating system were used, what R environment (such as RStudio) was used, what tool collected the provenance, where the provenance is stored, and what hash algorithm was used to store hash values for files used in the input and output of the script.
- The LIBRARIES section shows the libraries loaded by the script and their version numbers.
- The SCRIPTS section lists the main script and any scripts that are included in the execution of this script using the `source` or `prov.source` functions.
- The PRE-EXISTING section shows any variables where the script uses a value that was bound to the variable before the script started. This is a common R programming error that can lead to unexpected results if the script is run again in a different environment, where such a variable might have a different value or not be set at all.
- The INPUTS and OUTPUTS sections list all input and output files, the date they were last modified, and their hash values, using the hash algorithm shown in the environment section.
- The CONSOLE section shows any output sent to the console when the script executed.
- The ERRORS & WARNINGS section lists any errors or warnings that occurred when the script executed, including the number of the line that caused them.

In our own day-to-day work, we use **provSummarizeR** to document the processing of real-time meteorological and hydrological data at Harvard Forest. Data and plots of data captured in the past 30 days, including air temperature, precipitation, stream discharge, and water temperature, are updated and posted every 15 minutes.[3] Also posted at the same site are provenance summaries for the script execution that creates the plots.

There are three functions provided to generate summaries:

```
prov.summarize(details = FALSE)
prov.summarize.file(prov.file, details = FALSE)
prov.summarize.run(r.script, details = FALSE)
```

- `prov.summarize` produces a summary for the last provenance collected in the current R session.
- `prov.summarize.file` takes the name of a JSON file containing provenance and produces a summary from it.
- `prov.summarize.run` takes the name of a file containing an R script. It runs the script, collects its provenance, and produces a summary.[4]

By passing `TRUE` for the `details` parameter, the user can see more detail about some aspects of the provenance. In particular,

- The libraries section is divided into three parts. The first part shows the libraries loaded by the script. The second part shows the libraries that were loaded before the script starts. The third part shows the libraries loaded by the rdtLite code itself.
- The information about script, inputs, and output files includes modification date and hash value.
- The information about errors and warnings includes the line number on which each occurred.

The provViz and provDebugR tools described below provide a similar set of three functions: one to use the last provenance collected, one to use a specific JSON file, and one to run a script and use its provenance.

[3] https://harvardforest.fas.harvard.edu/met-hydro-stations
[4] All three functions have additional optional parameters. For details, see the online help page.

provViz

The **provViz** package allows visual exploration of script execution as shown in Figure 1. There are two types of nodes: data nodes and procedure nodes. Data nodes represent things such as variables, files, plots, and URLs. Procedure nodes represent executed R statements. An edge from a data node to a procedure node indicates that the statement represented by the procedure node uses the data represented by the data node. For example, the edge from data item, '`7-mpg`', to procedure node, '`9-plot(cylinders,mpg)`', indicates that `mpg` was used in the call to the `plot` function. Conversely, an edge from a procedure node to a data node indicates that the procedure produced the data, for example, by assigning to a variable or writing to a file. An edge between two procedure nodes represents control flow, indicating the order in which the statements were executed.

provViz also allows the user to view the graph and explore it to examine intermediate data values or input and output files and to perform lineage queries. The node colors indicate node type. Data nodes representing variables are purple. Files are tan. Orange nodes represent standard output, while red data nodes represent warnings and errors. Yellow nodes represent R statements. Green nodes come in pairs and represent the start and end of a group of R statements. Clicking on a green node reduces the set of statements between the matching '`Start`' and '`Finish`' nodes into a single node, which is useful for making large graphs more manageable.

To see everything that depends on the value of a variable at a particular point in the execution of the script, the user can right-click on the data node and select '`Show what is computed using this value`'. This will display a subgraph containing just the data and procedure nodes that are in the lineage of the data node, as shown in Figure 2, which shows the lineage of '`3-cars4Cyl.df`'. Notice that statements that do not use the value of `cars4Cyl.df`, either directly or indirectly, are not shown.

In addition to examining data values and tracing lineages as in this example, provViz supports the following ways of exploring the provenance:

- Viewing input and output data files
- Viewing plots created
- Viewing the source code for a node or the entire script
- Comparing R scripts
- Comparing provenance graphs
- Searching for nodes by name and type
- Sorting procedure nodes based on execution time

provViZ itself is a small R program that connects to a Java program called DDG Explorer (Lerner and Boose, 2014a), which does the actual work of creating and managing the display.

provDebugR

The **provDebugR** package provides debugging support by using the provenance to help users understand the state of their script at any point during execution. It provides command-line debugging capabilities, but one could imagine building a GUI on top of these functions to produce a friendly interactive debugging environment. By using provenance, ProvDebugR provides insight into the entire execution and creates a rich debugging environment that provides execution context not typically available in debuggers.

For example, consider a simple, but buggy script.

```
w <- 4:6
x <- 1:3
y <- 1:10
z <- w + y
y <- c('a', 'b', 'c')
xyz <- data.frame (x, y, z)
```

Running this script produces a warning and an error.

```
Error in data.frame(x, y, z) :
  arguments imply differing number of rows: 3, 10
In addition: Warning message:
In w + y : longer object length is not a multiple of shorter object length
```

Of course, with a short script like this, a user could simply step through the script one line at a time and examine the results, but for the purposes of demonstrating the debugger, imagine that this

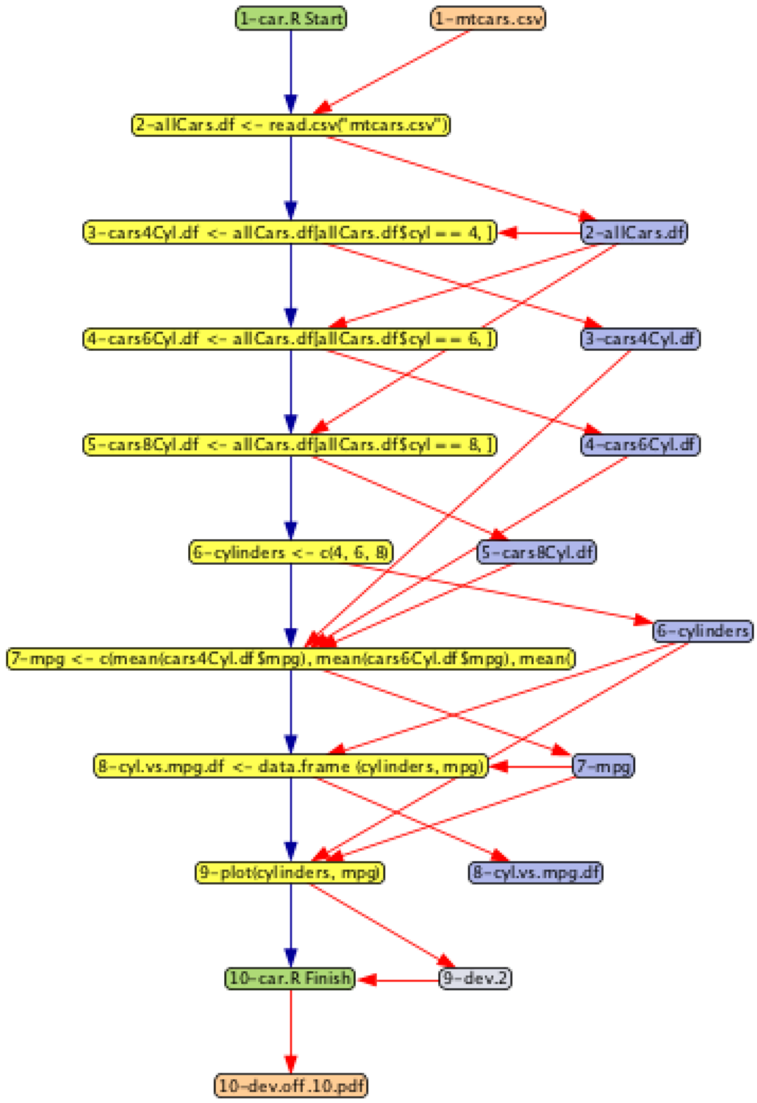

Figure 1: A provenance graph as displayed using provViz. Yellow nodes represent statements in the code, blue nodes represent variables, orange nodes represent files and green nodes mark the start and end of the script.

code is buried within a large script. The lines of code might not be consecutive as shown here, and it may even be difficult to determine what lines caused the reported errors.

The debugger provides some functions that are particularly helpful for understanding warning and error messages. For example, if the user needs help understanding where a warning came from, calling `debug.warning` with no arguments lists all the warnings; when called with a warning number, it displays the lines of code leading up to the warning.

```
> debug.warning()
Possible results:

1 In  w + y :  longer object length is not a multiple of shorter object length

Pass the corresponding numeric value to the function for info on that warning
> debug.warning(1)
Warning: In  w + y :  longer object length is not a multiple of shorter object length
        1:          w <- 4:6
        3:          y <- 1:10
        4:          z <- w + y
```

By omitting lines that do not contribute to the computations that lead to the warning, the R programmer should be able to find the problem more easily.

Similarly, the user can get information about what led up to an error using `debug.error`.

```
> debug.error()
Your Error: Error in data.frame(x, y, z): arguments imply differing number of rows: 3, 10

Code that led to error message:
        1:          w <- 4:6
        2:          x <- 1:3
        3:          y <- 1:10
        4:          z <- w + y
        5:          y <- c('a', 'b', 'c')
        6:          xyz <- data.frame (x, y, z)
```

The `debug.error` function has an optional logical parameter, `stack.overflow`. When set to `TRUE`, `debug.error` uses the stackexchange API to search Stack Overflow for posts about similar error messages. It lists the questions asked in the top six posts. The user can select one and a tab will open in the user's browser displaying the selected post.

Listing 3 shows a sample dialog using `debug.error`. Selecting 1 results in the user's browser going to the page displayed in Figure 3.[5] By scrolling down through answers to this question (not shown here), users will ideally obtain helpful information allowing them to solve their problem quickly.

A common cause of programming errors in R is caused by automatic type conversions as occurs here:

```
x <- 1
y <- 1:10
z <- 2
x <- x + y
if (x == 2) {
  print ("x is 2")
} else {
  print ("x is not 2")
}
```

Running this simple script produces this output.

```
Error in if (x == 2) { : the condition has length > 1
```

The programmer may be surprised or confused to get this warning message, as the assignment back to x may have been a mistake. Since R is a dynamically-typed language, there is no error at the time of the assignment, but only later when the value is used. The programmer can use `debug.variable` to quickly identify the type of x at each assignment

```
> debug.variable(x, showType=TRUE)
Var: x
```

[5] https://stackoverflow.com/questions/26147558/what-does-the-error-arguments-imply-differing-number-of-rows-x-y-mean

```
> debug.error(stack.overflow=TRUE)
Your Error: Error in data.frame(x, y, z): arguments imply differing number of rows: 3, 10

Code that led to error message:
  1:   w <- 4:6
2:   x <- 1:3
3:   y <- 1:10
4:   z <- w + y
5:   y <- c('a', 'b', 'c')
6:   xyz <- data.frame (x, y, z)

Results from StackOverflow:
[1] "What does the error \"arguments imply differing number of rows: x, y\" mean?"
[2] "ggplot gives \"arguments imply differing number of rows\" error in geom_point while it
     isn't true - how to debug?"
[3] "Checkpoint function error in R- arguments imply differing number of rows: 1, 38, 37"
[4] "qdap check_spelling Error in checkForRemoteErrors(val) : one node produced an error:
     arguments imply differing number of rows"
[5] "Creating and appending to data frame in R (Error: arguments imply differing number of
     rows: 0, 1)"
[6] "Caret and GBM: task 1 failed - \"arguments imply differing number of rows\""

Choose a numeric value that matches your error the best or q to quit:
```

Listing 3: The output of a call to debug.error, showing the titles of posts on Stack Overflow related to the error encountered in the script. The user can select an option to be taken to the corresponding Stack Overflow page.

```
        1:          1          x <- 1
  container dimension type
1 vector    1         numeric
        4:           2  3  4  5  6  7  8  9 10 11          x <- x + y
  container dimension type
2 vector    10        numeric
```

This shows that on line 4, x changed from a single element vector whose value was 1 to a 10-element vector containing the numbers 2 through 11.

Next, the programmer may want to find out why x became a vector. The `debug.lineage` function provides this information.

```
> debug.lineage(x)
Var x
        1:         x <- 1
        2:         y <- 1:10
        4:         x <- x + y
```

By showing the lines that led to x's value and type at line 4, we see the vector assignment to y in line 2, followed by the computation of x in line 4. Notice that line 3, the assignment to z, is not included in the lineage, since it played no role, either directly or indirectly in the value assigned to x. Ideally, by examining the provenance, the programmer realizes that the assignment should have been to y rather than to x.

An experienced R programmer may realize that unexpected type changes such as these can commonly lead to errors. Even if no error had been reported, they might want to check preemptively for type changes. This can be done by calling `debug.type.changes`, which reports all variables where the container, dimension, or type of value in the container have changed, showing just the values immediately before and after the type change.

```
> debug.type.changes()
The type of variable x has changed. x was declared on line 1 in debugScript4.R.
        1: debugScript4.R, line 4
                dimension changed to: 10
                from:                      1
                code excerpt: x <- x + y
```

The `debug.line` and `debug.state` functions allow the user to inspect variable values at specific lines in the code. The `debug.line` function shows the values of all variables used or modified on a specific line.

```
> debug.line(4)
Results for line(s): 4
```

```
4: x <- x + y
        Inputs:
              1. x   1
              2. y    1  2  3  4  5  6  7  8  9 10
        Outputs:
              1. x    2  3  4  5  6  7  8  9 10 11
```

The debug.state function shows the values that all variables have after execution of a specific line, showing the line number where the variable was set.

```
> debug.state(4)
Results for line(s): 4

Line 4
       4:        x         2  3  4  5  6  7  8  9 10 11
       2:        y         1  2  3  4  5  6  7  8  9 10
       3:        z        2
```

Earlier we showed the debug.lineage function that shows the user how a particular value was computed. That was an example of **backward lineage** or **ancestry**, because it starts with a variable and goes back in time to show all the computations on which a variable depends. The debug.lineage function can also display **forward lineage** to show how a value is used, i.e., all the subsequent computations that depend on it. This is particularly helpful in identifying all the information that might be affected by a programmatic change or modification to an input file.

```
> debug.lineage(x, forward = TRUE)
Var x
       1:         x <- 1
       4:         x <- x + y
       5:         if (x == 2) {
```

Note that by using provenance, provDebugR is able to display information about the execution state of the script at different points in its execution without the need to set breakpoints or insert print statements and re-run the script. This is particularly helpful for stochastic processes where the output might vary on each execution, causing some bugs to be challenging to track down.

provExplainR

Whereas **provSummarizeR** provides a summary of a single script execution, **provExplainR** goes a step further and provides a textual description of the difference between two script executions. If two executions of a script produce different outputs, **provExplainR** can be used to expose differences. This can be helpful when returning to work on an old script, when porting a script to a new environment, or when inheriting a script from someone else.

The prov.explain function reads two provenance directories and identifies differences in the computing environment, the input data, the versions of R or its libraries, and/or the main and sourced scripts.

```
prov.explain(
  dir1 = "prov_factorial_2021-03-31T12.01.36EDT",
  dir2 = "prov_factorial_2021-04-26T16.34.16EDT")
```

Results are displayed in the console (Listing 4).

The prov.diff.script function can be used to identify differences between two scripts.

```
prov.diff.script(
  dir1 = "prov_MyScript_2019-08-06T15.59.18EDT",
  dir2 = "prov_MyScript_2019-08-21T16.25.58EDT")
```

This function uses the **diffobj** package to identify and display differences (Figure 4).

We are planning to extend the functionality of provExplainR so that it also helps the programmer understand the impact of any reported changes by identifying where the behavior of the two executions start to differ. We expect this will help the programmer understand more specifically why the script is behaving differently. For example, if the line of code where changes first appear involves calling a function from an updated library, the programmer will likely want to understand better what changed with the new version of the library.

```

You entered:
dir1 = prov_factorial_2021-03-31T12.01.36EDT
dir2 = prov_factorial_2021-04-26T16.34.16EDT
SCRIPT CHANGES: The content of the main script factorial.R has changed
Run prov.diff.script to see the changes.
### dir1 main script factorial.R was last modified at: 2021-03-31T11.58.03EDT
### dir2 main script factorial.R was last modified at: 2021-03-31T11.58.21EDT

LIBRARY CHANGES:
Library version differences:
      name dir1.version dir2.version
      base        4.0.0        4.0.5
  datasets        4.0.0        4.0.5
   ggplot2        3.3.2        3.3.3
  graphics        4.0.0        4.0.5
 grDevices        4.0.0        4.0.5
   methods        4.0.0        4.0.5
     stats        4.0.0        4.0.5
     utils        4.0.0        4.0.5

Libraries in dir2 but not in dir1:
No such libraries were found
Libraries in dir1 but not in dir2:         name version
        dplyr   1.0.0
   provDebugR     1.0
 provExplainR     1.0


INPUT FILE CHANGES:
No input files were found in dir 1
No input files were found in dir 2

ENVIRONMENT CHANGES: Value differences:
Attribute: language version
### dir1 value: R version 4.0.0 (2020-04-24)
### dir2 value: R version 4.0.5 (2021-03-31)
Attribute: scriptHash
### dir1 value: c6b976a5ba662833323d56543817671b
### dir2 value: 426ecf01ebab431cdcbb000a20c3e273
Attribute: total elapsed time
### dir1 value: 1.483
### dir2 value: 1.752
Attribute: working directory
### dir1 value: /Users/blerner/Documents/Process/DataProvenance/workspace/provExplainR/
     testscripts/factorial-1
### dir2 value: /Users/blerner/Documents/Process/DataProvenance/workspace/provExplainR/
     testscripts/factorial-2
Attribute: provenance directory
### dir1 value: /Users/blerner/tmp/prov/prov_factorial_2021-03-31T12.01.36EDT
### dir2 value: /Users/blerner/Documents/Process/DataProvenance/workspace/provExplainR/inst/
     testdata/prov_factorial_2021-04-26T16.34.16EDT
Attribute: provenance collection time
### dir1 value: 2021-03-31T12.01.36EDT
### dir2 value: 2021-04-26T16.34.16EDT

PROVENANCE TOOL CHANGES: Tool differences: No differences have been detected
```

Listing 4: Output from prov.explain describing the differences found in the provenance of two executions of factorial. In this case, the significant differences are differences in the factorial script, the library versions, and the version of R

Developing new provenance-based tools

In addition to end-user tools as described above, we have also made available packages intended for programmers interested in developing their own tools incorporating provenance information.

provParseR

The **provParseR** package parses the JSON provenance and provides a convenient API to access portions of the provenance. To get started the tool developer calls the `prov.parse` function.

```
prov.parse(prov.input, isFile = TRUE)
```

The `prov.input` parameter is a string that can either be the path to a JSON file containing provenance or it can be a string containing the provenance. The second parameter (`isFile`) is used to disambiguate these cases. The default assumption is that `prov.input` is the path to a file. This function returns an object whose class is `ProvInfo`. The remaining functions provided by **provParseR** are getters that are passed a `ProvInfo` object and return information, typically a data frame containing that portion of the provenance.

For example, `get.input.files` returns a data frame containing a subset of the data nodes that correspond to files read by the script. The data frame that is returned includes the following information:

- id - a unique id
- name - the file name
- value - the path to a saved copy of the file
- hash - the hash value of the file
- location - the path to the original file

The `get.environment` function returns a data frame including information about the execution environment, such as the architecture and operating system on which the script was executed, the version of R, and the modification and execution times of the script.

Two functions provide information about the R libraries used. The `get.libs` function returns the name and version of each library, and whether it was loaded by the script, loaded before the script ran, or loaded by rdtLite code. The `get.func.lib` function returns the name of each function called from a library and the library from which it came.

Other functions provide information about the R statements executed and the edges between nodes. See the package's help page for a complete list of the functions and what they do.

The provSummarizeR, provDebugR and provExplainR tools all use provParseR to extract the information they need from the JSON file.

provGraphR

The **provGraphR** package provides an API that allows a tool developer to make lineage queries over provenance, as provDebugR does. To get started, the tool developer calls the `create.graph` function.

```
create.graph(prov.input = NULL, isFile = TRUE)
```

The `create.graph` function uses the **igraph** package to calculate an adjacency matrix representation of the graph. The value returned by `create.graph` can be used as an argument to the `get.lineage` function to perform lineage queries. As with `prov.parse`, the default behavior is for `prov.input` to be the path to a JSON provenance file and for `isFile` to be TRUE. Alternatively, `prov.input` can be a string containing JSON provenance if `isFile` is `FALSE`.

The `get.lineage` function computes either backward or forward provenance.

```
get.lineage(adj.graph, node.id, forward = FALSE)
```

Its `node.id` parameter is the unique id assigned to each node in the graph. Using parser functions, such as `get.input.files`, `get.output.files`, `get.variables.set`, and `get.variables.used`, a tool developer can find the id of a file or variable and then obtain its lineage.

These functions provide information about how input data is used or how the values stored in an output file or a plot were computed. The return value is a vector of node ids identifying the nodes in the lineage. The functions return complete lineage, so backward provenance traces back to input files or constants, while forward lineage traces to output. This function underlies the various trace and lineage functionality provided in **provDebugR**.

Limitations

There are two techniques used to capture provenance, each with its own limitations.

First, provenance information concerning files that are read or written is done by using R's `trace` function. Specifically, we trace the low-level I/O functions provide by R, such as `writeLines`, `write.table`, `readLines`, and `read.table`, as well as I/O functions from the **vroom** package. We also trace plotting functions provided by the **grDevices** package, like `pdf`, and functions from the **ggplot2** package, like `ggsave`. Any I/O function built on top of any traced functions will effectively be traced. However, I/O functions that instead use an external library to do the actual I/O will not be traced. It is not difficult to add new functions to trace, but it requires a modification to **rdtLite** for that to happen.

Second, statement-level provenance is captured by parsing each statement to find the variables used and set and then executing the statement to capture the values of variables that are modified. Each top-level statement is executed atomically. As a result, an if-statement, loop, or a function call is executed as a unit. While I/O information is captured internally to these, provenance at the level of variables is not captured on a line-by-line basis internally to these programming constructs. Provenance collection slows down the execution of scripts, and collecting more detailed provenance seems prohibitive, although it does limit the usefulness of **provDebugR**, in particular.

For a similar reason, a statement that uses the pipe operator is also executed as a unit. The variables used within pipes, and the final value computed by a statement that uses pipes is captured. However, the intermediate values passed through the pipe are not captured.

rdtlite may misidentify some expressions as variables when non-standard evaluation is used. For example, in the statement

```
cars6Cyl.df <- subset(allCars.df, cyl == 6)
```

`cyl` is not a variable, but rather the name of a column in the `allCars.df` data frame. In order to know that `cyl` is not a variable, **rdtLite** would need to know how the `subset` function evaluates its parameters. There is no general purpose way of determining this, Handling this situation would require creating a list of known functions and which parameters use non-standard evaluation. **rdtLite** does not do this currently.

Finally, rdtLite captures values associated with R's base types. However, it has not been extensively tested with the various class systems supported by R.

Related work

There are many systems that collect provenance and several excellent survey papers on provenance systems (Freire et al., 2008; Herschel et al., 2018; Pimentel et al., 2019). Provenance collection is common in workflow systems where it is built directly into the execution environment, such as in Kepler (Altintas et al., 2006), VisTrails (Koop et al., 2013), and Taverna (Missier et al., 2008). Of particular interest is the work of de Oliveira et al. (2014) who use provenance to debug long-running workflows, and Why-Diff (Thavasimani et al., 2019) which compares provenance of multiple workflow executions to find differences. Provenance collection in programming languages is much less common, with the exception of the NoWorkflow (Murta et al., 2014) implementation for Python.

There has been previous work on collecting provenance for R. Much of this work collects provenance at the level of files. **rctrack** (Liu and Pounds, 2014) uses R's `trace` function to record information about files read and written and the computing environment. It saves copies of data files and scripts with the goal of being able to reproduce a computation. Similarly, **recordr** (Slaughter et al., 2018) records information about files read and written and the computing environment. It can also save copies of those files.

The **CodeDepends** (Lang et al., 2019), **trackr** (Becker et al., 2017), and **histry** (Becker et al., 2017) packages coordinate to provide insights and records of code execution similar to how **rdtLite** and its associated tools work. The techniques used to collect provenance and the functionality built on top of the collected provenance are different, however. The **CodeDepends** package collects dependency information from R code based on static analysis of the code, rather than through execution. The **histry** package tracks expression evaluation and weaving as with RMarkdown. The **trackr** package (Becker et al., 2017) captures the provenance of plots created by a script. Metadata about how a plot is created comes from the dependencies and provenance gathered by **CodeDepends** and **histry**. The plots can later be discovered by performing searches on the metadata.

adapr (Gelfond et al., 2018) stores hash values of data files with the R code in a GitHub repository. They assume the data themselves are stored elsewhere. Their goal is to be able to confirm that data match the data used by the code. If the data are modified, the modification will be observable, but the original data cannot be restored by **adapr**.

While these R provenance systems collect valuable information useful for archiving data provenance, they do not produce the fine-grained provenance needed for debugging. In contrast, **CXXR** (Silles and Runnalls, 2010; Runnalls and Silles, 2012) computes fine-grained provenance using a modified R interpreter where the read-eval-print loop is modified to collect provenance. The collected provenance is available interactively but is not stored persistently. This type of provenance can be helpful for debugging but does not support archiving the provenance.

In contrast to these, **rdtLite** saves information persistently about file inputs and outputs that is useful for archival purposes and saves fine-grained provenance useful for debugging. The E2ETools also build on top of this provenance to provide useful functionality to the user and provide building blocks to enable more tools to be built. Since the JSON provenance format is language-agnostic, the same provenance tools should be usable for different programming languages, and we are currently working on supporting Python by translating provenance collected by noWorkflow (Murta et al., 2014) into the E2ETool JSON format.

Conclusions and Future Work

Data provenance contains a wealth of information. Although provenance initially was thought of as documentation to bolster trust in the data, it has many uses beyond that. In particular, fine-grained provenance offers rich opportunities to develop tools that can be helpful for debugging, learning how a script works, maintaining scripts, and porting scripts to new environments.

Reproducibility as a Service (RaaS) (Wonsil, 2021), a web-based reproducibility tool, strongly benefits from collecting and using provenance data. This tool automatically constructs a computational environment in a Docker container for a given set of R scripts and the data they analyze. It then executes all the scripts, collecting provenance with **rdtLite** and saving all the results to a Docker image. The resulting provenance currently allows RaaS to build a report for its users and situates it perfectly to use the E2ETools in the future. For example, it could use **provSummarizeR** to generate its reports. If researchers want to compare the RaaS execution to their initial execution on their machine, RaaS could integrate **provExplainR** for easy comparisons. Finally, RaaS could also incorporate **provDebugR** to allow users to step through the execution of the scripts entirely within their browser without needing an R session or even downloading the data.

Our collaborators have used a variant of **provDebugR** to explore asynchronous collaboration between data scientists. This variant, called the Multilingual Provenance Debugger (MPD) (Yoo et al., 2021), is not tied to the R language. Instead, it works on provenance for any language that exports to the same PROV-JSON format as **rdtLite**. An experimental feature in MPD allows users to record and annotate a debugging session as a trace to send to another collaborator, who can replay the trace step-by-step or view the whole session as a pretty-printed markdown file. We could implement similar features in **provDebugR** and extend it to include a visualization component.

Finally, another avenue for future work is the semi-automatic generation of model cards, an artifact that Mitchell et al. (2019) proposed to increase transparency for machine-learning models. One of our current collaborations includes contributions to the open-source Tribuo machine-learning library (Pocock, 2021), which contains a built-in provenance collection system focused on machine-learning provenance. Using the provenance that Tribuo generates, our collaborators built a feature to automatically generate the technical details for model cards and provide support for annotations to supplement the data on the card. We can bring a variant of this feature back into the R ecosystem as an extension of **provSummarizeR**, either directly for machine learning in R or, more generally, to build an 'analysis card' or 'script card.' As these ongoing projects demonstrate, collecting provenance is just the beginning. Developing software that builds on collected provenance to support reproducibility, understanding, and enhancement of software is the long-term goal of this work.

Acknowledgements

This work was supported by NSF grants DEB-1237491, DBI-1459519, and SSI-1450277, the Charles Bullard Fellowship program at Harvard University, and a faculty fellowship from Mount Holyoke College. This paper is a contribution of the Harvard Forest Long-Term Ecological Research (LTER) program.

The authors acknowledge intellectual contributions from the following students: Shaylyn Adams, Vasco Carinhas, Marios Dardas, Andrew Galdunski, Connor Gregorich-Trevor, Nicole Hoffler, Jennifer Johnson, Siqing (Alex) Liu, Erick Oduniyi, Antonia Oprescu, Luis Perez, Moe Pwint Phyu, Katerina Poulos, Garrett Rosenblatt, Cory Teshera-Sterne, Sofiya Toskova, Morgan Vigil, and Yujia Zhou.

Appendix: Extended Prov JSON Format

The provenance collected by **rdtLite** uses a JSON format that extends the Prov JSON format defined by W3C W3C (2014). The W3C Prov JSON format was designed to capture workflow involving multiple activities with information flowing between them. An activity might be performed by a piece of software, or by a person. The detailed provenance captured by **rdtLite** has activities that are at the level of R statements, with the data being files and variables. The extensions use the same schema as defined by W3C, encoding the provenance data as described below.

Prov JSON has three types of elements: entities, agents, and activities. In the extended JSON used by rdtLite, information about data, libraries, and functions, as well as the runtime environment are encoded as entities. The tool used to collect the provenance is encoded as an agent. Information about statements is encoded as activities.

Prov JSON provides many types of relationships. In the extended JSON, just four of these are used. The `wasInformedBy` relationship is used to represent edges connecting statement elements. Specifically, these edges capture control flow information. The `wasGeneratedBy` relationship connects a statement element to the data elements that it generates, such as a variable that is modified, or a file that is output. The `used` relationship is used to connect a data element to the statement elements that uses the data, such as a variable used within a statement or a file input by a statement. The `used` edge also is used to record what functions are used by each statement. The `hadMember` relationship records which library each function comes from.

For more details on this format, see `https://github.com/End-to-end-provenance/ExtendedProvJson/blob/master/JSON-format.md`.

Bibliography

I. Altintas, O. Barney, and E. Jaeger-Frank. Provenance collection support in the Kepler scientific workflow system. In *Proceedings of the International Provenance and Annotation Workshop*, pages 118–132, Chicago, May 2006. Springer-Verlag. [p13]

G. Becker, S. E. Moore, and M. Lawrence. trackr: A framework for enhancing discoverability and reproducibility of data visualizations and other artifacts in r, 2017. URL `https://arxiv.org/abs/1706.04440`. [p13]

R. A. Becker and J. M. Chambers. Auditing of data analyses. *SIAM Journal of Scientific and Statistical Computing*, 9:747–760, 1988. [p1]

D. de Oliveira, F. Costa, V. Silva, K. Ocaña, and M. Mattoso. Debugging scientific workflows with provenance: Achievements and lessons learned. In *in Proceedings of the 29th SBBD*, pages 67–76, Brazil, October 2014. [p13]

J. Freire, D. Koop, E. Santos, and C. T. Silva. Provenance for computational tasks: A survey. *Computing in Science and Engineering*, 10(3):11–21, May/June 2008. [p13]

J. Gelfond, M. Goros, B. Hernandez, and A. Bokov. A system for an accountable data analysis process in R. *The R Journal*, 10(1):6–21, July 2018. [p13]

M. Herschel, R. Diestelkämper, and H. B. Lahmar. A survey on provenance: What for? what form? what from? *VLDB Journal*, 2018. [p13]

D. Koop, J. Freire, and C. T. Silva. Enabling reproducible science with vistrails. In *First Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE1)*, Denver, CO, November 2013. [p13]

D. T. Lang, R. Peng, D. Nolan, and G. Becker. CodeDepends. `https://github.com/duncantl/CodeDepends`, 2019. [Online; accessed 19-July-2022]. [p13]

B. Lerner, E. Boose, and L. Perez. Using introspection to collect provenance in R. *Informatics*, 5(12), 2018. URL `http://www.mdpi.com/2227-9709/5/1/12/htm`. [p1, 4]

B. S. Lerner and E. R. Boose. Poster: RDataTracker and DDG Explorer — capture, visualization and querying of provenance from R scripts. In *Proceedings of the International Provenance and Annotation Workshop*, Cologne, Germany, June 2014a. [p6]

B. S. Lerner and E. R. Boose. RDataTracker: Collecting provenance in an interactive scripting environment. In *Proceedings of 6th USENIX Workshop on the Theory and Practice of Provenance (TaPP '14)*, Cologne, Germany, June 2014b. [p4]

Z. Liu and S. Pounds. An R package that automatically collects and archives details for reproducible computing. *BMC Bioinformatics*, 15(138), 2014. [p13]

P. Missier, S. Embury, and R. Stapenhurst. Exploiting provenance to make sense of automated decisions in scientific workflows. In *Provenance and Annotation of Data and Processes: Second International Provenance and Annotation Workshop, IPAW 2008*, number 5272 in Lecture Notes in Computer Science, pages 174–185, Salt Lake City, Utah, June 2008. Springer-Verlag. [p13]

M. Mitchell, S. Wu, A. Zaldivar, P. Barnes, L. Vasserman, B. Hutchinson, E. Spitzer, I. D. Raji, and T. Gebru. Model cards for model reporting. In *Proceedings of the conference on fairness, accountability, and transparency*, pages 220–229, 2019. [p14]

L. Murta, V. Braganholo, F. Chirigati, D. Koop, and J. Freire. noworkflow: Capturing and analyzing provenance of scripts. In *Proceedings of IPAW 2014*, Cologne, Germany, June 2014. [p13, 14]

National Academies of Sciences, Engineering, and Medicine. *Reproducibility and Replicability in Science*. National Academies Press, Washington, DC, 2019. [p1]

J. F. Pimentel, J. Freire, L. Murta, and V. Braganholo. A survey on collecting, managing, and analyzing provenance from scripts. *ACM Comput. Surv.*, 52(3), June 2019. [p13]

A. Pocock. Tribuo: Machine learning with provenance in java, 2021. URL `https://arxiv.org/abs/2110.03022`. [p14]

A. Runnalls and C. Silles. Provenance tracking in R. In *Proceedings of the 4th International Conference on Provenance and Annotation of Data and Processes*, IPAW'12, pages 237–239, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-34221-9. doi: 10.1007/978-3-642-34222-6_25. [p14]

C. A. Silles and A. R. Runnalls. Provenance-awareness in R. In *Proceedings of the 3rd International Conference on Provenance and Annotation of Data and Processes*, pages 64–72, 2010. [p14]

P. Slaughter, M. B. Jones, C. Jones, and L. Palmer. recordr, 2018. URL `https://github.com/NCEAS/recordr`. [p13]

P. Thavasimani, J. Cała, and P. Missier. Why-diff: Exploiting provenance to understand outcome differences from non-identical reproduced workflows. *IEEE Access*, 2019. [p13]

W3C. The PROV-JSON Serialization. `https://openprovenance.org/prov-json/`, 2014. [Online; accessed 18-July-2022]. [p4, 15]

J. Wonsil. *Reproducibility as a service*. PhD thesis, University of British Columbia, 2021. URL `https://open.library.ubc.ca/collections/ubctheses/24/items/1.0398221`. [p14]

J. Yoo, A. Li, and J. Wonsil. Multilingual provenance debugger, 2021. URL `https://github.com/jyoo980/MultilingualProvenanceDebugger`. [p14]

Barbara Lerner
Computer Science Department
Mount Holyoke College
South Hadley, MA 01075
United States of America
`blerner@mtholyoke.edu`

Emery Boose
Harvard Forest
Harvard University
Petersham, MA 01366
United States of America
`boose@fas.harvard.edu`

Orenna Brand
Columbia University
New York, NY 10027
United States of America
`o.brand@columbia.edu`

Aaron M. Ellison
Sound Solutions for Sustainable Science
Boston, MA 02135
United States of America
aaron@ssforss.com

Elizabeth Fong
Computer Science Department
Mount Holyoke College
South Hadley, MA 01075
United States of America
fong22e@mtholyoke.edu

Matthew Lau
Sustainable Community Food Systems Program
Division of Social Sciences
University of Hawaii West Oahu
91-1001 Farrington Hwy, Kapolei, HI 96707
United States of America
mklau3@hawaii.edu

Khanh Ngo
Mount Holyoke College
South Hadley, MA 01075
United States of America
ngo22k@mtholyoke.edu

Thomas Pasquier
Department of Computer Science
University of British Columbia
2366 Main Mall #201, Vancouver, BC V6T 1Z4
Canada
tfjmp@cs.ubc.ca

Luis A. Perez
Harvard College[6]
Massachusetts Hall
Cambridge, MA 02138
United States of America
nautilik@deepmind.com

Margo Seltzer
Department of Computer Science
University of British Columbia
2366 Main Mall #201, Vancouver, BC V6T 1Z4
Canada
mseltzer@cs.ubc.ca

Rose Sheehan
Mount Holyoke College
South Hadley, MA 01075
United States of America
sheeh22r@mtholyoke.edu

Joseph Wonsil
Department of Computer Science
University of British Columbia
2366 Main Mall #201, Vancouver, BC V6T 1Z4
Canada
jwonsil@cs.ubc.ca

[6]Primary contributions while at Harvard College. Now at DeepMind.

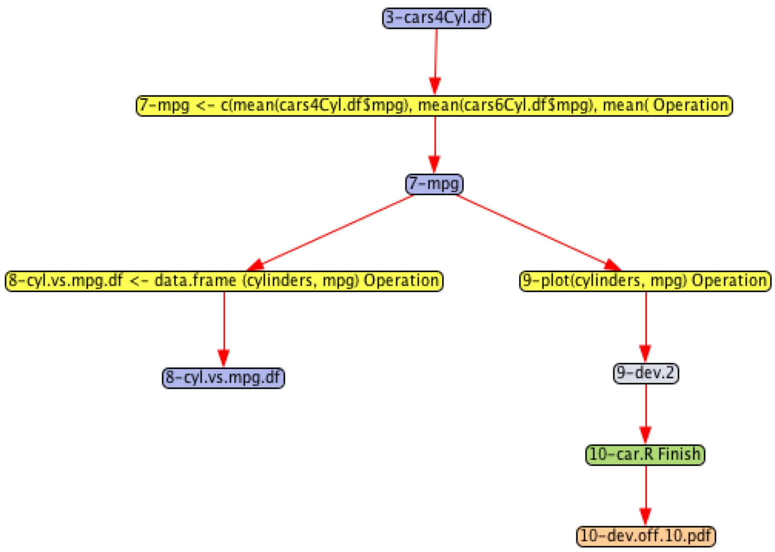

Figure 2: Displaying the Lineage of 3-cars4Cyl.df

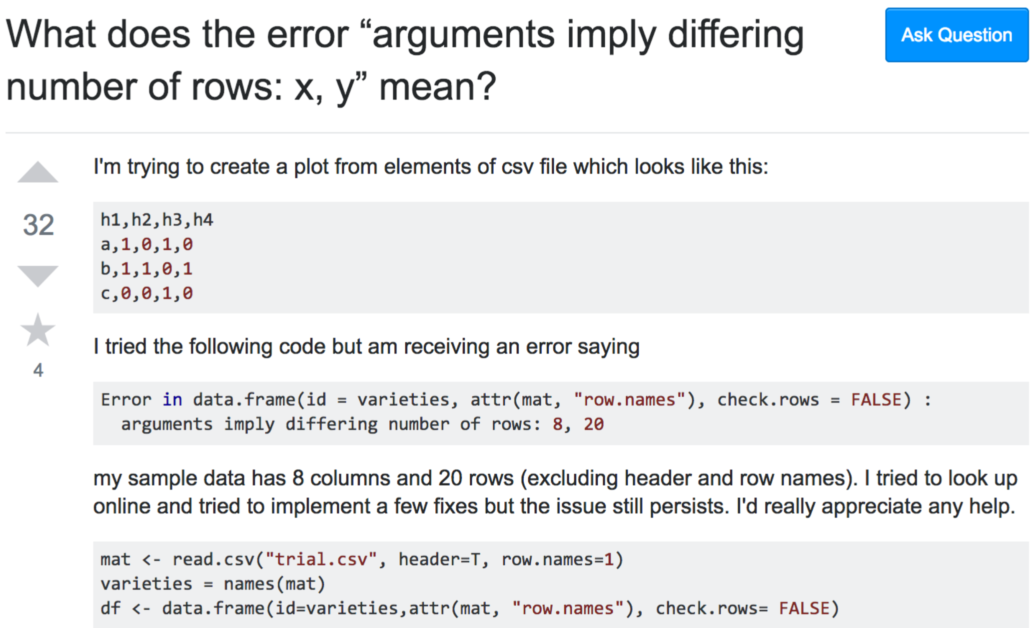

Figure 3: Stack Overflow Page to Resolve an Error

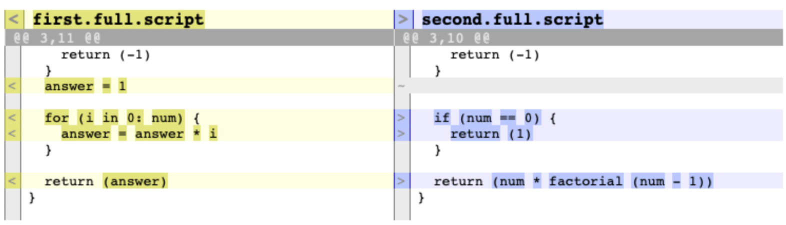

Figure 4: Comparing scripts using provExplainR.