L-Vis: Visualizing Language-Level Provenance for Program Comprehension

Joe Wonsil jwonsil@cs.ubc.ca
Francis Nguyen frnguyen@cs.ubc.ca

Introduction

There are many situations where programmers have to understand previously written code. For example, collaborators on projects might review their colleague's analysis scripts, a software engineer might be returning to old code they have personally written long ago, or new researchers inherit legacy code that needs to be maintained. If you haven't personally been through some of these slogs, comprehending these programs can be a difficult task! Luckily, research in program comprehension helps us understand how researchers and programmers approach unfamiliar code.

To aid program comprehension, *provenance*, the history of derivation of an object, is typically captured and often visualized to understand the history of functions and variables throughout the program process. However, the scale of provenance that is collected can overwhelm a person's ability to parse it. In some cases, provenance data of a program can be several magnitudes larger than the length of the program. Visualization is often employed as a powerful tool that leverages human perception to understand these large provenance datasets.

Provenance information is typically represented as network graphs. For example, *VisTrails* visualizes application-level provenance as a tree where a node denotes a separate dataflow that differs from its parent and an edge records the changes [4]. However even with VisTrails' thoughtful design, tools in this space target provenance on the application level rather than the language level and still produce visualizations that break down at scale.

We set out to create a visualization tool, L-Vis, that accurately presents language-level (LL) provenance to easily and effectively facilitate program comprehension through exploration of history. The goal of our tool is to accurately present summaries of R scripts that are not hampered by the potentially large amount of information within provenance. We employ a *top-down* approach to abstracting data for use in the tool where we begin at large scale and create informed subsets of data for more effective exploration.
We frame our task requirement analysis in the framework of Erdös and Sneed's questions for program comprehension [1].

Joe has been working on provenance research in collaboration with Margo Seltzer and has experience with provenance collection in R. Francis has visualization and HCI experience

and has an increasing interest in reproducible science. This project is also a class project for CS 508, Operating Systems.

Data

The provenance of analysis scripts can be increasingly large in relation to the length and complexity of the script. Given the tools available to us, we limit the scope of our collected provenance to R-scripts found on the Harvard Dataverse, an open-source research data repository [15]. We use containR, an end-to-end provenance tracking tool that downloads scripts and collects provenance from Dataverse. ContainR will automatically adjusts the analysis downloaded to fix common bugs found in deposited code. These bugs include working directories hard-coded for a specific user and machine and installing necessary packages.

We scraped >100 analysis scripts written in R from Dataverse. Our main inclusion criteria were a) the scripts are in R, and b) the scripts linked their data sources. Our scraped analysis scripts range in domain as the repository contains analyses from across the natural and social sciences.

Our dataset, therefore, is provenance collected from these scripts. Provenance is collected in PROV-JSON, a serialized format of the PROV data model. This model represents a directed-acyclic graph (DAG). Each DAG represents a single execution of the script. The graph is composed of different types of nodes and edges. Nodes represent everything from executed code to variables declared. What they store varies depending on the node type.

- *Procedure nodes* represent a unit of code the script executed. They contain information such as the code executed, how long it took, and where the code is located in the script.
- *Data nodes* represent information stored in the script, typically variables and files. They store the value (if possible), type of the value, type of the data, and scope.
- *The agent node* stores information about the versions and settings of provenance being collected. There will only ever be one of these.
- *The environment node* holds information about where and when the script was executed. This includes the architecture and operating system of the computer, the path to the file, paths to relevant directories, and how long it took to run. There will only ever be one of these.
- *Library nodes* correspond to a package loaded into the R environment for use in the script and its version.
- *Function nodes* represent a function called in the script that was not defined in the script, therefore was taken from a package.

Edges represent connections between two nodes. Therefore edges will only ever contain

a start and end node.

- *Procedure to Procedure edges* represent control flow. Essentially, in the pair of procedure nodes p1 and p2, this edge means after p1 executed, the next thing to happen in the script was p2.
- *Procedure to Data edges* link data nodes to where they were first initialized or brought into the script.
- *Data to Procedure edges* correspond to where data nodes were used after their creation.
- *Function to Procedure edges* link function nodes to where they were used in the script.
- *Library to Function edges* link the R package a function node came from.

Collecting provenance again after an initial run could result in a different DAG, especially if the input data has changed. For complex scripts, the scale of this data is hundreds of nodes and edges. While we might only visualize a subset of the provenance collected, we purposefully scrape a collection of analysis scripts that vary in complexity and length for future related work.

Tasks

Program comprehension literature primarily focuses on the cognitive models employed by programmers while they try to solve and accomplish real-world tasks such as fixing bugs or adding new features. Most frequently mentioned program comprehension strategies in the literature involve top-down, bottom-up, and interactive approaches. The number of tasks could potentially be very large, so we target the tasks L-Vis will accomplish by prioritizing the **top-down** cognitive strategy. In the top-down strategy, programmers generate assumptions about the structure and end-result of a program and then investigate every subsection of the program with regards to those assumptions. This is similar to visual hypothesis testing in visual analytics. In order to ground the specific tasks, we reviewed both program comprehension literature and informal interviews with peers attempting to understand their own comprehension process [1,2,3,14]. We derived a set of common high-level tasks and requirements from these two sources.

1. Reverse engineering of design patterns.
2. Navigate multiple overviews of the system architecture at various levels of abstraction.
3. Investigate specific contexts.
4. Support goal-directed, hypothesis-driven comprehension. For example, the cause of the bug is **x**.
5. View paths or relationships that led to the current focus.
6. Understand syntactic and semantic relationships between variables and functions.

We use the what-why-how framework in order to abstract these sets of tasks to requirements L-Vis should fulfil.

What

Provenance of analysis script as described in **Data**. Once provenance has been successfully collected, the dataset is static.

Why

Scaffold cognitive models of program structure, aiding programmers in the top-down cognitive strategy.

How

Multiple views:

- A high-level context view displaying a summary of all available provenance data. This will be a hierarchical layout inspired by treemaps.
- Examining a specific node's history. Provenance allows users to inspect all code and variables associated with the output of code snippets. Tracing these relationships can facilitate hypothesis-driven facilitation.
- Time-based clustering of nodes based on the algorithm found in InProv.

Proposed Infovis Solution

While our specific visual encoding and idioms are still in process, we define a scenario to guide our design implementation.

Scenario

A final-year PhD student has written scripts to perform analyses on a dataset. Since they have finished their dissertation they upload their data and scripts to containR and the analysis is archived and stored in a container. Next year, a new graduate student joins the lab and is told by their advisor to add a new plot based on a mix of old and new data. Luckily, thanks to containR and provenance, the old scripts still run exactly as they did before because they are running in a container. Additionally, from within the containR website the new student is able to use L-Vis to learn about how the old plots were created and can then cleanly insert new code into the analysis. In particular, the new student uses the detail view of the plot node in order to view all the code-snippets and variables related to generating the plot. To begin building a mental model of the plot generation code, the new student uses the detail view and changes variables in order to see how the plot changes dynamically.

Proposed Implementation Approach

We build our visualization tool on top of containR, an end-to-end provenance tool that allows researchers to upload their analysis scripts and data files and it will collect provenance and then build a Docker container for the analysis. Because containR is a browser application, we will use Javascript and d3.js. These tools allow us to make the solution widely accessible via containR and will allow us to leverage d3.js' submodules on layout algorithms. As containR is primarily deployed with Flask and its built-in templating engine, we may employ other web libraries such as jQuery, React or Vue.js to develop a more complex interface. Francis has prior experience in these web technologies which will help scaffold part of the development effort.

Milestones

Checkpoint	Date	Approx. Hours Required
Data and Task Abstraction	November 6th	5
Proposed Visualization Design	November 8th	6
Data Collection	November 8th	20
Visualization Working Prototype	November 27th	30
"User Study"	November 29th	20
Analysis	December 4th	6
Draft	December 6th	20
Final	December 13th	25

We created our schedule to focus early on abstracting our task and data. We propose our dates in Table 1. Once we identify these abstractions, we can work on finalizing a proposed visualization. We are aiming to have the design around the time of the first status meeting. Given that a visualization tool is the outcome of this project, it is essential to be able to have the design worked on early. With even the fundamentals of the design, we can start prototyping so we can perform a user study.

To ensure we have a variety of data to work with, we are implementing containR, an application designed to be a website, locally so that we can use its features and build on top

of it. The tool has pieces that have fallen out of date and need to be worked on before it can be used fully to collect data.

In the process of identifying our abstractions and design, we can begin writing much of the contextual and related work sections of our paper. Once we finalize the tool's design and start prototyping, we can write about the design during the same period. Our second status meeting will occur in the middle of creating the tool. When the tool is complete we will conduct an informal user study with our peers to observe how effective it is. Once we complete a user study, we will analyze the data and add it to what we already wrote. We will then have a few days to review what we have written. Here we can ensure it follows the story we are trying to tell before submitting the first draft.

After receiving feedback from the first draft, we will make more passes over the paper. During this period, we can complete any final tweaks we may have to make to the tool, such as bug fixes identified during the user study. After we make more reviews of the paper and solidify and adjustments to the tool, we will submit our final project.

Related Work

Visualization of Provenance

DDG Explorer is a visualization tool designed for LL-provenance [6]. DDG Explorer typically uses Rdata-tracker captured provenance to create node-link graph visualizations where each node represents executions, variables or other control structures, and edges represent control and dataflow. VisTrails also utilizes a node-link graph, but is designed for application-level provenance in the domain of visual analytics where the goal is to keep track of data transformations and how datasets have been altered [4]. Here, the aim is to visualize analysis and scientific workflows, allowing the user to keep track of previous steps in their process. For large, complex datasets, the strategies employed by DDG Explorer and VisTrails tends to break down and become increasingly for difficult to interpret.

InProv is a top-down tool designed for whole-system provenance [8]. Rather than the traditional node-link graph, InProv leverages radial layouts of time-based hierarchical groupings to improve accuracy and efficiency of finding nodes in addition to helping users understand high level concepts of the provenance. They find that these layouts improve accuracy and efficiency of finding nodes, but organize system-level provenance instead of LL-provenance. Researchers constructed this tool as an alternative to Orbiter, a node-link graph visualization [9].

Workflow and Trace Visualization

We also draw on the design space of workflow visualization and trace visualization for inspiration. In the domain of visualization and visual analytics, provenance is often captured and visualized for analysis and scientific workflows.

ProvThreads captures provenance data and displays user interaction logs to show topic modeled data [12]. ProvThreads represents user interactions using colored lines that represent different data topics. While ProvThreads provides encodings that are not typical provenance visualization, the tasks that they target are not related to program comprehension; rather the tool is built to analyze user workflows and decision points in scientific analysis.

ExtraVis is a trace visualization tool that provides two interactive views of large execution traces [13]. One view displays a massive sequence view of the trace and the second produces a "circular bundle" view that hierarchically projects the program's structural entities on a circle. ExtraVis is informed by Pacione et al's framework of nine principal activities, which we hope to leverage for L-Vis [14]. ExtraVis visualizes system-level traces as opposed to LL-provenance. While the tool does answer some questions on granular program comprehension for debugging, it fails to encapsulate high-level program comprehension.

We are currently looking for more papers in this domain to define the large design space of visualizations from which we are drawing from (GraphTrace, JavaVis, TraceGraph, Oasis).

Program Comprehension

Erdös and Sneed proposed seven questions that would need to be answered for a programmer to be able to understand a program [1].

1. Where is a particular subroutine/procedure invoked?
2. What are the arguments and results of a function?
3. How does control flow reach a particular location?
4. Where is a particular variable set, used or queried?
5. Where is a particular variable declared?
6. Where is a particular data object accessed?
7. What are the inputs and outputs of a module?

Provenance contains a direct answer to most of these questions. Information captured for provenance includes intermediate data values, packages and their functions used, program inputs and outputs, and connections and dependencies between variables.

We use these questions to establish our system requirements of our provenance visualization tool: an effective visualization generated should be able to answer all of these questions in order to improve program comprehension.

References

[1] K. Erdos and H. M. Sneed, "Partial comprehension of complex programs (enough to perform maintenance)," in *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No.98TB100242)*, pp. 98–105, June 1998.

[2] M.A. Storey, "Theories, methods and tools in program comprehension: past, present and future," in *13th International Workshop on Program Comprehension (IWPC'05)*, pp. 181–191, May 2005.

[3] J. Sillito and G. C. Murphy, "Questions programmers ask during software evolution tasks," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 23–34, 2006.

[4] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo, "Vistrails: Visualization meets data management," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, (New York, NY, USA), pp. 745–747,ACM, 2006.

[5] J. a. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "noworkflow: A tool for collecting, analyzing, and managing provenance from python scripts," *Proc. VLDB Endow.*, vol. 10, pp. 1841–1844, Aug. 2017.

[6] B. Lerner and E. Boose, "Rdatatracker and ddgexplorer," in *Ludascher B., Plale B. (eds) Provenance and Annotation of Data and Processes* (IPAW 2014), vol. 8628, pp. 288–290, 2015.

[7] T. D. Huynh, M. O. Jewell, A. S. Keshavarz,D. T. Michaelides, H. Yang, and L. Moreau, "The PROV-JSON serialization," 2013

[8] M. A. Borkin, C. S. Yeh, M. Boyd, P. Macko, K. Z. Gajos, M. Seltzer, and H. Pfister, "Evaluation of file system provenance visualization tools," *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, pp. 2476–2485, Dec2013

[9] P. Macko and M. Seltzer, “Provenance map orbiter: Interactive exploration of large provenance graphs,” in *International Workshop on Theory and Practice of Provenance* (TaPP2019), 2011.

[10] K. Cheung and J. Hunter, “Provenance explorer– customized provenance views using semantic inferencing,” in *The Semantic Web* - ISWC 2006 (I. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, M. Uschold, and L. M.Aroyo, eds.), (Berlin, Heidelberg), pp. 215–227, Springer Berlin Heidelberg, 2006.

[11] J. Zhao, C. Wroe, C. Goble, R. Stevens,D. Quan, and M. Greenwood, “Using semantic web technologies for representing escience provenance,” in *The Semantic Web* – ISWC2004 (S. A. McIlraith, D. Plexousakis, andF. van Harmelen, eds.), (Berlin, Heidelberg),pp. 92–106, Springer Berlin Heidelberg, 2004.

[12] S. Mohseni, A. M. Pena, and E. D. Ragan, “Provthreads: Analytic provenance visualization and segmentation,” ArXiv,vol. abs/1801.05469, 2018.

[13] B. Cornelissen, A. Zaidman, A. van Deursen,and B. V. Rompaey, “Trace visualization for program comprehension: A controlled experiment,”2009 IEEE 17th International Conference on Program Comprehension, pp. 100–109,2009.

[14] M. J. Pacione, M. Roper, and M. Wood, “A novel software visualisation model to support software comprehension,”11th Working Conference on Reverse Engineering, pp. 70–79, 2004.

[15] “The Dataverse Project.”