Visualizing the Scripts of Data Wrangling with SOMNUS

Kai Xiong, Siwei Fu, Guoming Ding, Zhongsu Luo, Rong Yu, Wei Chen, Hujun Bao, Yingcai Wu

Abstract—Data workers use various scripting languages for data transformation, such as SAS, R, and Python. However, understanding intricate code pieces requires advanced programming skills, which hinders data workers from grasping the idea of data transformation at ease. Program visualization is beneficial for debugging and education and has the potential to illustrate transformations intuitively and interactively. In this paper, we explore visualization design for demonstrating the semantics of code pieces in the context of data transformation. First, to depict individual data transformations, we structure a design space by two primary dimensions, i.e., key parameters to encode and possible visual channels to be mapped. Then, we derive a collection of 23 glyphs that visualize the semantics of transformations. Next, we design a pipeline, named SOMNUS, that provides an overview of the creation and evolution of data tables using a provenance graph. At the same time, it allows detailed investigation of individual transformations. User feedback on SOMNUS is positive. Our study participants achieved better accuracy with less time using SOMNUS, and preferred it over carefully-crafted textual description. Further, we provide two example applications to demonstrate the utility and versatility of SOMNUS.

Index Terms—Program understanding, data transformation, visualization design.

1 INTRODUCTION

SCRIPTING languages including SAS, R, and Python have been widely accepted by data workers for data transformation. They usually seek to understand the semantics of scripts in various scenarios. For example, validation (or called double-checking in some companies and laboratories) is important for data scientists. A data scientist might seek to understand code pieces written by others, then locate and correct possible mistakes. Understanding the semantics of an intricate script, however, requires advanced programming skills. And sometimes, the process is tedious and error-prone [48], [62], [71].

A number of program visualization techniques have been proposed for debugging and communication. For example, some techniques, such as Whyline [45], Timelapse [14], and FireCrystal [59], utilize visualizations to help programmers identify and fix bugs. Those debugging tools focus on revealing the runtime behavior, such as the values of objects and variables, on allowing programmers to inspect the program state. However, depicting program states benefits little in communicating the semantics of code pieces. Others, such as algorithm visualizations [12], [66] and automatic generation of flowcharts [15], [17], [69], aim to help learners understand the flow of algorithms. However, little attention has been paid to illustrating the process of data transformation.

In this work, we explore visualization design for depicting the semantics of code pieces in the context of data transformation. To present individual data transformations, we first outline a design space consisting of two primary dimensions, i.e., key parameters to encode and potential visual channels that can be mapped. Then, we propose a collection of 23 glyphs that demonstrates the semantics of transformations. Given a code piece containing a series of functions, data tables are created and changed. To illustrate the evolution of tables, we contribute the design and implementation of SOMNUS, a pipeline that accepts a script and data tables as input and results in a graph model where nodes are tables while edges are data transformations. SOMNUS consists of two main components, i.e., Program Adaptor and Visualization Generator. The Program Adaptor parses code pieces, generates input and output tables for each statement, and infer transformations based on rules. On the other hand, the Visualization Generator creates visual representations to illustrate data provenance. We claim that SOMNUS facilitates the understanding of intricate code pieces, including the semantics of individual operations and table dependencies of the entire data wrangling process. To some extent, SOMNUS supports some higher-level tasks such as helping users debug programs of data wrangling and correct errors in the code. Besides, the idea of SOMNUS is general and can be adapted to various scripting languages, including R, Python, etc.

To evaluate the effectiveness of the glyph design and SOMNUS, we conducted a controlled study to compare our visual representations with carefully-crafted textual descriptions. The results show that our participants can understand complex wrangling scripts more accurately in a shorter time and prefer our visualizations in terms of helpfulness and interpretability. In addition, we demonstrate the utility and versatility of SOMNUS with two example applications. The first application shows how SOMNUS can be adapted to Python and facilitates the validation of a piece of wrangling script. As for the second application, SOMNUS is adapted to R and used to support MORPHEUS [25] in interactive data transformation.

To conclude, the contributions in this paper include: 1) a design space consisting of two dimensions that guide the design of a collection of 23 glyphs, 2) a pipeline, called SOMNUS, that visualizes the creation and evolution of data tables across a series

- *K. Xiong, G. Ding, W. Chen, H. Bao, and Y. Wu are with the State Key Lab of CAD&CG, Zhejiang University, Hangzhou, China, and with Zhejiang Lab, Hangzhou, China. E-mails: {kaixiong, dinggm, chenvis}@zju.edu.cn, bao@cad.zju.edu.cn, ycwu@zju.edu.cn.*
- *S. Fu and R. Yu are with the Zhejiang Lab, Hangzhou, China. E-mail: fusiwei339@gmail.com, 1721298964@qq.com.*
- *Z. Luo is with Zhejiang University of Technology, Hangzhou, China, and also with the Zhejiang Lab, Hangzhou, China. E-mail: rickyluozs@gmail.com.*
- *Yingcai Wu and Siwei Fu are the co-corresponding authors.*

Manuscript received xx xxx, 20xx; revised xx xxx, 20xx.

of transformations, 3) a controlled study that evaluates how users perform with visualization and text using comparison tasks, and 4) two example applications that showcase how SOMNUS can benefit different usage scenarios.

2 RELATED WORK

2.1 Program Visualization

Program visualization refers to *"the visualization of actual program code or data structures in either static or dynamic form"* [77]. In program visualization, different audiences vary in analytical tasks, which require tailored visual representations [16]. For example, software engineers and data scientists are dedicated to development activities including programming, debugging, testing, etc. Systems focusing on these activities usually need to visualize the runtime behavior of the program including object states, function calls, etc. Another example is that, data workers [4], [52] and education practitioners expect an effective method for comprehending or learning the semantics of a program. We note that the concerns of different roles are not strictly differentiated. For example, data workers can also develop a wrangling program to find new insights on data. In short, program visualization is usually used for debugging and education tasks [36].

Many debugging tools leverage visualizations to help developers identify and fix bugs. Some of them, such as Hdpv [72], Heapviz [3], and Anteater [24], present task-specific or code-related information about the execution by giving a forest view. Others can reveal the runtime behavior, such as DDD [82], deet [32], ZStep 95 [50], and VisuFlow [57]. Whyline [45] and Theseus [49] introduce visualizations within integrated development environments, while FireCrystal [59] and Timelapse [14] focus on visualizing interactive behaviors on web pages. Hoffswell et al. [35] propose visual debugging techniques to inspect program states for reactive data visualization. A number of works [7], [8], [33], [36], [73] leverage in-situ visualizations to display the program behavior.

SOMNUS can be used for debugging the process of data transformation. However, our technique differs from prior work in two aspects. First, instead of visualizing internal states or variables of programs, SOMNUS shows the semantics of code pieces, which involves input and output tables, the type of data transformation, and parameters of functions. Second, data presented in the aforementioned approaches are generic types such as string and numbers. On the contrary, data, in the context of data transformation, means 2-D data tables consisting of columns and rows. The presentation of 2-D data tables is more challenging than generic data types.

Some program visualization systems are designed for education. They intend to improve students' understanding of particular aspects of programs [77]. Online Python Tutor [29] is a web-based visualization tool that illustrates the runtime state of various data structures, which can be a valuable pedagogical aid for teaching Computer Science courses. Algorithm visualization has been a hot research topic as having a significant impact on students learning behavior [28] and being promising for facilitating education [65]. A variety of algorithm visualizations [12], [18], [31], [66] depict program behavior on every step to facilitate understanding the program. Some tools automatically convert source code to flow charts, including Visustin v7 [60], AutoFlowchart [69], code2flow [17], Flowgen [46], and VizMe [15]. The aforementioned approaches are explicitly designed for some algorithms or applications. Nevertheless, none of them are proposed in the context of data transformation. In this paper, we design and implement SOMNUS that the creation of evolution of data tables across a series of transformations.

2.2 Data Wrangling

Data wrangling is an arduous process of transforming, reformatting, and integrating data to make it more palatable for miscellaneous downstream purposes, including visualization and analysis [42]. Many toolkits written in R (e.g., dplyr [80], tidyr [79]) or Python (e.g., Pandas [64]) have been proposed to support the process. These toolkits provide excellent expressiveness for data workers to wangle data. However, for data workers who are not proficient in R or Python, learning a new programming language or toolkits for wrangling tasks would spend substantial time and effort [67].

To lower the barrier of data wrangling, various interactive systems and prototypes are proposed. Microsoft Excel, Tableau Prep Builder [70], and OpenRefine [37] provide a menu-based GUI for users to iteratively clean, transform, and integrate data. Some systems embed a recommendation engine to suggest possible transformations. Data Wrangler [30], [42] and its commercial successor Trifacta [76] recommend transformations based on users' manipulation. The others, such as Foofah [39] and Wrex [22], borrow ideas from *programming by example* that synthesizes code pieces for data transformation based on a small illustrative example provided by users. Some systems support wrangling for graphs, websites, etc. For example, Ploceus [54], Orion [34], and Origraph [10] support graph editing and construction. On the other hand, Vegemite [51], Dataxformer [2], [55], and WebRelate [38], are designed to transform data from different websites.

The aforementioned approaches assist data workers in conducting data transformations. SOMNUS, on the other hand, targets presenting the process of data transformation. Some tools, such as Tableau Prep Builder [70], OpenRefine [37], Data Wrangler [30], [42], and Trifacta [76], record and present the process of data transformation using textual descriptions. We argue that our visualization design is easier to understand and more effective than textual descriptions, and we report the comparison in Section 6 to justify the argument.

Kasica et al. [43] formed 21 types of operations based on a multi-table framework for data wrangling by two dimensions, i.e., three data types (rows, columns, and tables) and five operations (create, delete, transform, separate, and combine). Furthermore, each type of operation is represented by an intuitive icon. However, these icons are not mapped to data. Inspired by these icons, we design our glyphs by supplementing them with parameters and additional types of visual channels to present the semantics of transformation operations.

2.3 Provenance

Provenance records the history of changes and advances during analysis [63]. Kandel et al. [41] emphasized the significance of capturing provenance from data quality operations and wrangling workflows when data workers share their data and scripts.

A number of works have been proposed to capture and visualize data provenance. For example, Tableau Prep Builder [70] provides an icon for each operation in a data flow chart. Although these icons are easy to understand, they can not visualize the parameters of operations, such as the specificity of which tables/rows/columns are transformed and how. By contrast, our glyph design can visualize both the type of data transformation and its parameters,

which facilitates the comprehension of semantics because it reveals more details on data changes [53]. TACO [58] is a visual comparison tool for investigating the differences and changes between multiple tabular data over time. However, it focuses on quantitative homogeneous tables and does not support visualizing complex data transformations, including fold and unfold. SOMNUS, on the contrary, focuses on visualizing the semantics of scripts and supports a wide range of data transformations mentioned by Kasica et al. [43]. Some tools leverage animations to visualize data provenance. Data Tweening [44] generates intermediate results for each data transformation in a SQL query session, facilitating the understanding and learning of complex transformations. Datamations [61] explains the transformation steps of a data analysis pipeline by automatically generating a looping animated GIF from code. Animation is useful for communication. However, the exploration of animation is slower as users often replay the animation dozens of times, and they can not control the animation at their own pace [27]. Additionally, those animation tools focus on presenting the data provenance of a single table. In contrast, our work utilizes node-link graphs with glyphs to illustrate data provenance, which can better present the process of data transformations and portray the data provenance of multi-tables simultaneously.

3 DESIGN REQUIREMENT

Our goal is to design a set of visual representations to help data workers understand and communicate a script of data transformation. To this end, we collaborate with two data analysts in a national research lab who have at least three years of expertise in data science. Following Munzner's guidelines [56], we conducted three rounds of interviews to iteratively extract design requirements. Our interviews focused on their working scenarios, such as double-checking, where they are required to understand the semantics of wrangling scripts. One major challenge is that they often need to recall or look up the usage and syntax of various functions. One analyst reported, *"I like Python. But sometimes I need to understand scripts written in R."* He added, *"Cheat sheets are useful (to understand R functions) in many cases. I may also search in Google and RDocumentation*[1] *to understand advanced parameters."* However, there is a comprehension gap between the usage of functions and the semantics of practical code. One analyst complained that he still needs to figure out how a line of code works on data after understanding the R function. In addition, the understanding of individual functions helps little in revealing the entire wrangling process. In light of these complaints and feedbacks collected from interviews, we summarized the following design requirements. Particularly, R1 to R4 target the design of glyphs presenting individual data transformations, while R5 to R7 guide the design of SOMNUS.

R1: Present the Semantics: To help data workers understand a function, our visualization design should precisely present the semantics, including the function name, input, output, and parameters of a function. As the number of functions could be large, designing visualization for each function may burden recognition. Instead, we should present the type of data transformation to which the function belongs. We distinguish between "data transformation" and "function," as the former refers to a manipulation categorized in Kasica et al. [43] while function corresponds to a method in a programming language.

1. https://www.rdocumentation.org/

R2: Link with Data: When writing scripts, data workers usually need to "look at" data tables by printing out a table or temporary results. One analyst usually works with Jupyter [40], and he commented, *"I like to print out results to verify the operations."* Therefore, besides function-specific information, the visualization should reflect detailed information of a table, including content, shape, name, etc.

R3: Depict Necessary Information: Much information is involved in a function, such as function parameters, input and output data tables. We note that not all parameters are essential. Similarly, when a table is large, illustrating all its content is impossible and unnecessary. As a result, we should elicit and encode critical information from a function and representative content in a table.

R4: Keep Encoding Consistent: Glyphs in SOMNUS should have consistent visual encodings. When visualizing a sequence of functions, consistent visual encoding facilitates understanding each data transformation and the entire procedure.

R5: Reveal Table Provenance: Data tables are evolved and correlated through functions. For example, an output table of a data transformation may serve as input for another, and so forth. Data provenance records how data was created and changed, which is significant in tracing data processing changes back to their original sources [11]. As one analyst noted, *"Some operations (such as join) rely on multiple tables. Displaying how tables are correlated is useful in debugging."* Hence, the visualization should provide an overview of the entire data provenance.

R6: Dig into Details: Data workers usually need to validate a series of functions. Hence, they seek to grasp detailed information on each data transformation. Our visualization should allow users to switch between an overview and a detailed view of individual transformations.

R7: Independent of Programming Languages: Data workers may use various toolkits for data transformation, such as dplyr [80] in R and Pandas [64] in Python. To ensure generalizability, our visualization design should be independent of toolkits and programming languages.

4 DESIGN OF GLYPHS

Guided by the aforementioned design requirements, we design a collection of glyphs that presents the semantics of functions. To answer questions like, "which information shall we encode" and "which visual channel can be mapped," we structure the design space by two primary dimensions, i.e., the type of parameters and potential visual channels.

4.1 Parameters Space

The analysis of toolkits helps us identify key information that should be encoded in the design of glyphs (R1, R3). In this paper, we focus on three packages in R and Python, i.e., dplyr, tidyr, and Pandas, because they all target data transformation and are open-source in nature. We have examined 160 functions in total, where 84 are from dplyr, 23 from tidyr, and 53 from Pandas. We read the official documentation of these functions, reveal semantics under different parameters, and map them to the type of transformations. Moreover, we run these functions with different parameters to understand how parameters affect the transformation results. Finally, we categorize six key parameters that should be mapped to visual channels.

Function Name reflects which operation does the function targets. Since the name does not have a one-to-one mapping with data transformations, the information, in some cases, benefits little in communicating the semantics of a function. For example, the *select* function in dplyr can be mapped to three different transformations, i.e., *Delete Columns*, *Rearrange*, and *Transform Columns*, depending on parameters and data tables. We do not emphasize function names in our visualization design.

Data Tables are described as variables in the script and are input and output of a function. We identify a variable string as a table name. Data tables are stored in a well-designed data structure, e.g., data.frame in R or DataFrame in Python, and can vary in dimensions In this paper, we focus on 2-dimension tables, which are collections of rows and columns.

Explicit Columns/Rows are the columns/rows explicitly mentioned as parameters in functions. Explicit columns are usually referred to using column names, while explicit rows are mentioned using row indexes. Taking the statement as an example, *tree2=arrange(trees, Girth)*, the parameter *Girth* is the name of an explicit column in the table *trees*. Another example is that, in the statement *mtcars_temp=slice(mtcars, 1, 5)*, the parameters *1, 5* are two-row indexes of the table *mtcars*. The quantities of explicit columns and rows are usually limited. Since they are key information in a function, they should be illustrated and highlighted in the glyph design.

Implicit Columns/Rows are not listed as parameters in a function. Rather, they are selected in the data transformation based on filtering criteria. For example, when deleting duplicate rows, rows with identical values are compared and filtered. The presentation of implicit columns/rows is beneficial for understanding data transformation. Because the volume of implicit elements is usually large, depicting all these is virtually impossible. As a result, we should select and encode representative ones in the glyph.

Contextual Columns/Rows are not involved in a data transformation. Specifically, they do not meet filtering criteria are not selected during transformation. Similar to explicit and implicit elements, we argue that context is also useful for communicating data transformation. For example, contextual columns keep unchanged when deleting a column to show a contrast to the deleted one [43]. Similar to implicit elements, we should encode a limited number of contexts.

Transformation Parameters Beside the type of data object, a function usually includes a number of parameters to precisely acknowledge function details. They can be inline functions (e.g., sum, min, regular expression, etc.), mathematical operators (e.g., "+," "-," etc.), or transformation-specific identifiers (e.g., separator in *separate* and *unite*). The visualization of these parameters is critical to revealing subtle differences among transformations.

4.2 Design Rationale

Kasica et al. [43] structured a multi-table framework for data wrangling. The framework includes 15 categories of transformations, and each may contain several subtypes. For example, based on whether the operation modifies table schema, *Transform Tables* includes two subtypes, i.e., *Rearrange* and *Reshape*. Further, they designed icons for 21 (sub)types of data transformations. These icons are intuitive and inspiring and provide a good starting point for our glyph design. We distinguish between an icon and a glyph, in which the former is a visual representation only and irrelevant to data; in contrast, the latter, which is widely used in various tasks

Fig. 1. We use *Transform Columns* as an example to showcase different visual channels.

to represent multidimensional data [19], [74], [78], [81], maps data to visual channels such as color, size, etc. By analyzing all icons and the parameter space, we distill the following design guidelines in creating our glyph collection.

Input and Output Tables: Each icon designed by Kasica et al. is composed of three main parts, i.e., an input table, an output table, and an arrow indicating the transformation. We follow this metaphor (Figure 1(a)) in designing our glyph collection because these are necessary for presenting a transformation (R1). In the following description, we use "data" to indicate data tables and use "table" to refer to the table metaphor in a glyph.

Table Shape: The shape of a table is affected by the number of explicit, implicit, and contextual columns/rows (R1, R3). All explicit entities are depicted in the table due to their importance. For implicit entities, we selectively choose one that is representative. In some cases, "one" means "one pair". For example, when depicting "remove duplicate rows", we select two identical rows as implicit entities. Contextual entities are displayed for two reasons. First, it helps to present the semantics of a transformation by posing a contrast to explicit/implicit entities. Second, it retains the table metaphor. For example, in "Create Table", all entities are contextual. We demonstrate 3×3 contextual cells to indicate an empty table. For other transformations, context is limited to one or two columns/rows.

Cell Color: Color encoding is meaningful in Kasica et al. [43]. It is designed for distinguishing cell types (e.g., title cells are white while content cells are colored), indicating the type of data object (e.g., column and row), depicting unchanged columns/rows, and presenting correlated columns/rows (e.g., the icon for *Interpolate*). The color encoding of our glyph is primarily borrowed from Kasica et al. . Further, we extend prior work from two aspects. First, we use white color to represent empty cells. At the same time, title cells are colored dark gray. Second, we use striped cells to depict those with an empty or blank string (R4).

Out-table Text: Some text is displayed outside the table. For example, for transformations targeting specific rows by row index, we present row index aside from the table (R3). Besides, we present table names and the type of transformation that a function belongs to. Following the text in Trifacta [76], we present textual information below the input and output tables to describe the semantics of transformations (Figure 1(c)) (R1).

In-table Text: Presenting data content is critical to assisting data workers to understand a function (R1, R2). Due to limited glyph size, only contents in explicit and implicit columns/rows are depicted in the glyph (Figure 1(b)). Usually, it is not possible to present all values in these elements. Hence, we randomly sample values from data. By encoding in-table text, as well as out-table text, we are able to distinguish data transformations

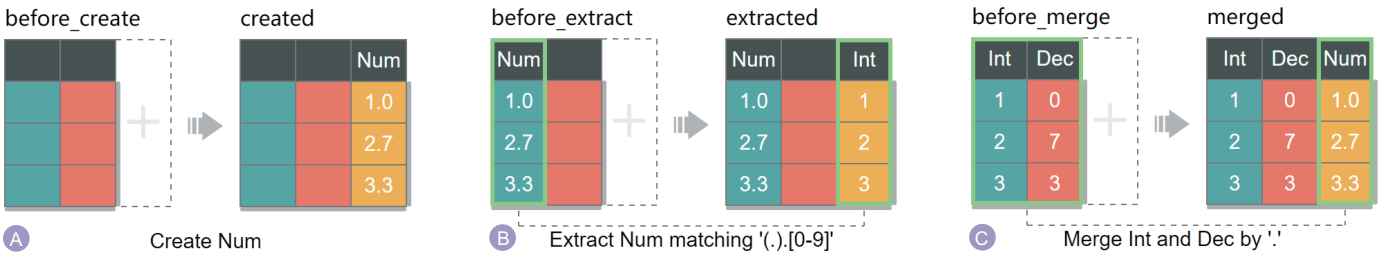

Fig. 2. By depicting in-table and out-table text, a glyph can differentiate different types of transformations. Taking *Create Columns* as an example, (a) shows the creation by filling in values manually, (b) shows extracting values from existing columns, and (c) depicts merging values from existing columns.

with a subtle difference. For example, there are four common subtypes of transformations for *Create Columns* [43], e.g., creating manually (Figure 2(a)), mutating from other columns (example of *Create Columns* in Figure 4), extracting substring of one column (Figure 2(b)), and merging multiple columns (Figure 2(c)). Another type of information shown in a table is special symbols. For example, we use ZA↓ and AZ↓ to illustrate sorting in descending and ascending order, respectively.

Other Considerations: Besides the aforementioned guidelines, we explore visualization techniques that enhance the perception of data tables and transformation (Figure 1(d)). First, to indicate the shape of data, we design both horizontal and vertical scroll bars in glyphs. The size of scroll bars is proportional to the shape of data tables. We acknowledge that this design cannot show a precise number of columns/rows. Instead, it informs that the glyph presents a portion of an entire data table (R2). Second, to emphasize the change of a table (R1), we highlight the correlation between explicit columns in input and output tables.

4.3 Results

Following the aforementioned design space, we derive 21 glyphs for data transformation. Besides, we create glyphs for two more transformations. First, in Kasica et al. [43], *Fold* and *Unfold* share one icon with different arrow directions. We distinct the two operations with two glyphs. Second, we add a glyph for *Rearrange Columns* because it is triggered by a popular function, *select*, in dplyr. To save space, Figure 4 illustrates 15 out of 23 glyphs, and the full glyph collection can be found in the supplemental material.

5 Design of Somnus

In this section, we present the design of Somnus, a pipeline that accepts data tables and a piece of code as input, and results in a visual representation to show the entire procedure of data transformations. Code pieces may contain complex control flow, including a conditional statement, loops, function definition, etc. In this paper, we limit the scope to code consisting of assignment statements only. Though some modules are implemented based on specific programming languages and toolkits, we argue that the design of Somnus can be applied to different programming languages. In the presentation, we use dplyr, a toolkit of R, as examples by default. Figure 3 shows the architecture of Somnus, which consists of two core modules, i.e., Program Adaptor and Visualization Generator.

5.1 Program Adaptor

Program Adaptor aims to generate a series of transformation specifications given data tables and a script. Though we implement

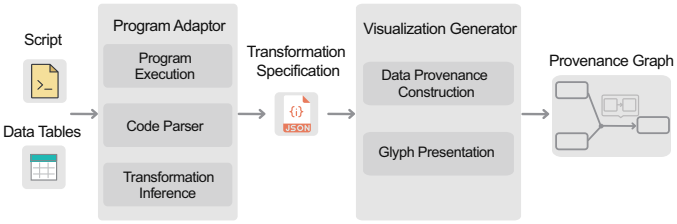

Fig. 3. The architecture of Somnus consists of two major modules: Program Adaptor and Visualization Generator. The Program Adaptor accepts a script and data tables as input and outputs a collection of transformation specifications. The Visualization Generator generates table provenance by utilizing the specifications.

an adaptor for each programming language, all adaptors share three common steps, i.e., Program Execution, Code Parser, and Transformation Inference. The descriptions of this module are independent of programming languages (R7).

5.1.1 Program Execution

After a script and data tables are fed into the Program Adaptor module, the script will be executed using an interpreter based on programming languages. The primary goal of this step is to obtain input and output data for each function, which is beneficial for 1) providing data value when plotting glyphs (R2) and 2) inferring the type of data transformation for each function (R1) (see Transformation Inference for details). The Program Execution steps automatically insert statements for importing necessary libraries to interpret and execute the script correctly.

5.1.2 Code Parser

Code Parser accepts a script as input and parses each line of code to obtain 1) the name of the input and output table, 2) function names, and 3) parameters of functions (R3). In some toolkits, a transformation can be invoked through various approaches. For example, given a data table, named "tbl," containing three columns in order, e.g., "column1", "column2", and "column3". Assume "tbl" is stored as DataFrame in Python, *Rearrange Columns* can be expressed as *pandas.DataFrame(tbl, columns=['column2', 'column1', 'column3'])*. Also, the same transformation can be achieved by *tbl[['column2', 'column1', 'column3']]*. In the current implementation of Somnus, we only support statements that have explicit function names and input and outputs.

Similar to Program Execution, results generated by Code Parser are critical to inferring the type of transformations (R1). Besides, tracing the input and output tables helps to construct the provenance of data.

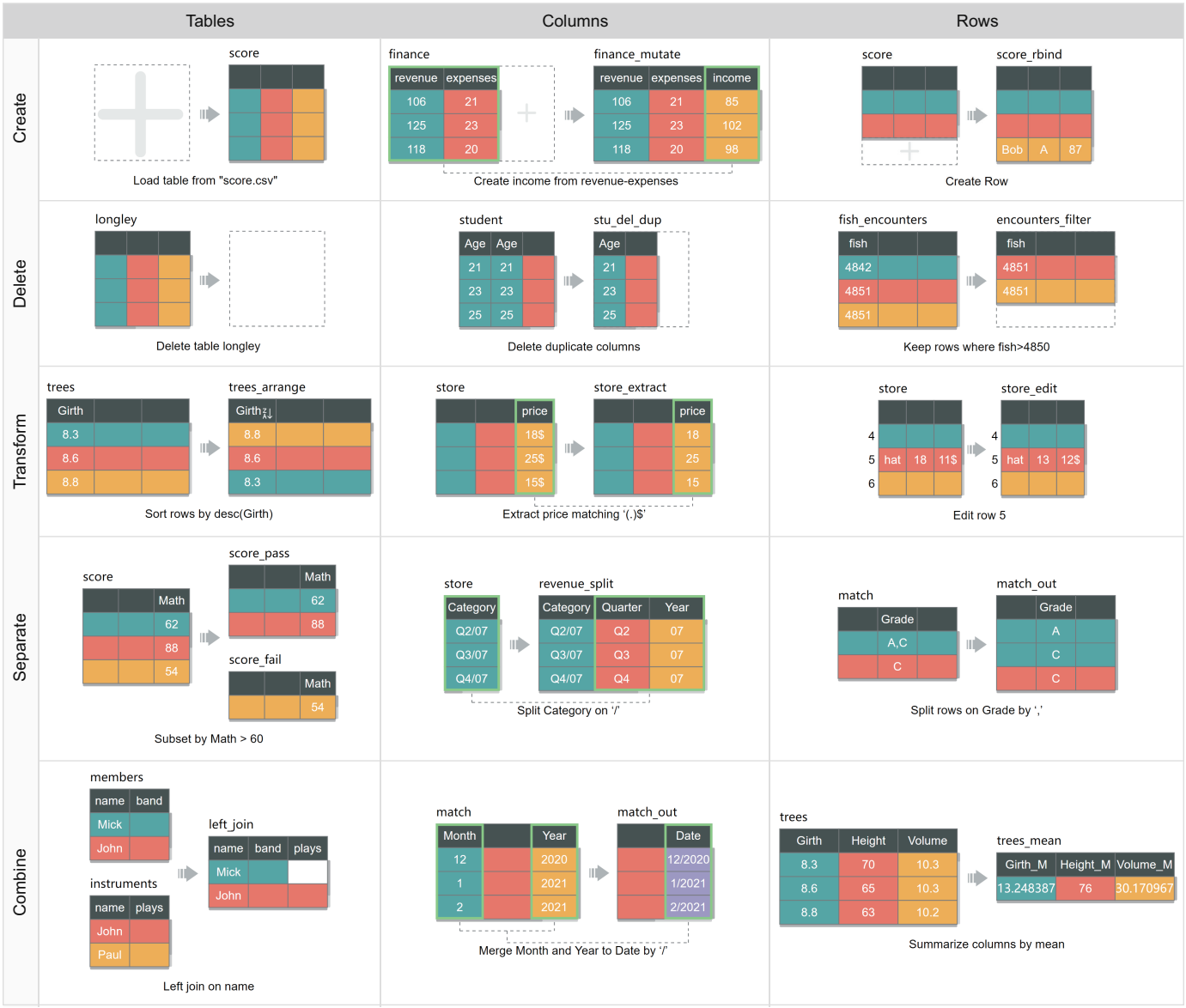

Fig. 4. Following Kasica et al. [43], we display 15 out of 23 transformations by two dimensions, i.e., the type of data object and five operation categories. All glyphs are generated based on real data tables and functions from tidyr [75] and dplyr [21]. The entire glyph collection can be found in the supplemental material, which is available online at https://github.com/xkKevin/Somnus.

5.1.3 *Transformation Inference*

After parsing individual functions and their input and output tables, we build a mapping between functions and the type of transformations (R1). In most cases, one function is mapped to one data transformation. We create rules for mapping the name and parameters of a function to one type of transformation. For example, we map *filter* to *Delete Rows*, *separate* to *Separate Columns*, and *count* to *Summarize*. However, function information is not enough in some cases. For example, given the data table ("tbl") mentioned above, the statement *select(tbl, "column3", "column1", "column2")* equals to *Rearrange Columns*. On the other hand, if the input table has four columns in order, e.g., "column3", "column1", "column2", and "column4", the same statement results *Delete Columns* as "column4" is omitted in the output table. In this case, we derive the type of transformations by comparing the input table with the output.

We note that some functions involve a sequence of transformations. Taking the data table ("tbl") as an example, the function *select(tbl, column1, column4 = column2)* first performs *Delete Columns* by deleting "column3". Then, it *Transform Columns* by renaming "column2" to "column4". In these cases, we identify multiple transformations for a function. One challenge is to obtain input and output tables for each transformation, which are critical to glyph generation. The current prototype establishes rules and replaces a function with multiple ones, where each corresponds to a transformation. Then, the entire script is executed again to derive the input and output data tables.

To save screen space, some functions can be grouped and merged. For example, the two functions, e.g., *rename(tbl, column4 = column1)* and *rename(tbl, column5 = column2)*, can be combined into one *rename(tbl, column4 = column1, column5 = column2)*. In such cases, we depict the two functions using one data transformation. The current prototype supports the combination of consecutive functions in three cases, i.e., *Rename Columns*, *Delete Columns*, and *Delete Rows*. We plan to investigate more rules for combining the semantics of functions in future research.

5.1.4 Failure Modes

To improve reliability, SOMNUS is able to deal with five types of failure modes. First, if the script contains operations that are unsupported, SOMNUS can compile and run the script properly. However, there are no glyphs for these operations. Second, if the function is not supported in the Program Adaptor, such as *drop_na* in tidyr, SOMNUS shows the function name only. Third, for operations that are not function-based, e.g., "*df = df[df.col1 >0]*" in Pandas, SOMNUS displays nothing on the edges. Fourth, SOMNUS decomposes operations involving many columns and rows into multiple ones and visualizes them with a sequence of glyphs, as described in Section 5.1.3. Finally, SOMNUS does not fully support the parsing of non-assignment statements, such as loops or conditional statements. We acknowledge that this policy may hinder data workers from understanding the logic of the entire script. For example, given a conditional statement like *"if CONDITION then OPERATION1 else OPERATION2"*, SOMNUS displays only one glyph representing either *OPERATION1* or *OPERATION2* based on the results of Program Execution.

5.2 Visualization Generator

The result of Program Adaptor is a set of transformation specifications accompanied by input and output data tables. This module aims to generate visual representations depicting both an overview and detailed information for transformations.

5.2.1 Constructing Data Provenance

Data provenance can be formed as a graph, where nodes are data tables and edges are data transformations (R5). The provenance graph is layered, and we leverage the Eclipse Layout Kernel [23] for positing graph nodes. As shown in Figure 5, each node is rectangular, and we depict useful table information in each node, including the line index where the table is created, the table name, and the size of the table.

Edges connecting nodes are data transformations. According to the number of input and output tables, we categorize the edges into three types. In most cases (Figure 8(d)), a transformation accepts a table as input and outputs a transformed one. We depict the edge as a directed line. Second, some transformations merge multiple data tables into one, such as *Extend*, *Supplement*, and *Match*. These transformations are shown as convergence edges (Figure 7(c)). Similarly, transformations that result in multiple output tables from one input are depicted as divergence edges, as shown in Figure 5.

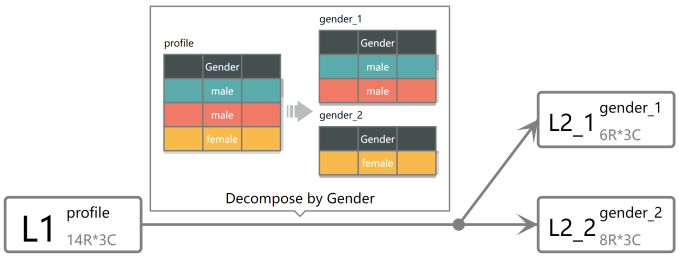

Fig. 5. The transformation, *Decompose*, results in a divergence edge, which is generated by two statements, *profile = read.csv("profile.csv")* and *gender = group_split(profile, Gender)*.

5.2.2 Presenting Glyphs

To present the details of data transformations, we depict glyphs aside from each edge in the provenance graph (R6). Since glyphs may contain in-table and out-table text, they are placed horizontally without rotation for better readability. For functions that are not supported by our glyph collection, no glyph is displayed.

5.3 Implementation

SOMNUS is implemented as a web-based client-server system. The backend is implemented using flask, while the frontend is built in Vue.js and D3.js [13]. The web interface consists of four panels, i.e., a Data Panel, a Script Panel, a Table Panel, and a Graph Panel. The Data Panel (Figure 8(a)) allows users to upload their input tables as needed. Users need to select a programming language and copy-paste a piece of data wrangling code to the Script Panel (Figure 8(b)). Then the backend runs based on the input tables and the script provided by the user. The Graph Panel (Figure 8(d)) displays table provenance (R5), while the Table Panel (Figure 8(c)) is used to show the intermediate tables generated in the process of data wrangling (R2). To assist the investigation of lengthy table provenance, the Graph Panel supports zooming and panning (R6).

Interactions across panels are integrated to facilitate the exploration among script, data tables, and data provenance. First, when a user clicks on a node (i.e., data table) in the provenance graph, the Table Panel displays the detailed table. Similarly, when clicking on edge (i.e., transformation) in the graph, its function in the Script Panel is located and highlighted, and vice versa.

We focus on data wrangling from two popular programming languages, i.e., R and Python. Specifically, the current prototype supports 25 commonly used functions from tidyr [75] (e.g., *separate*, *gather*, *spread*, etc.) and dplyr [21] (e.g., *filter*, *select*, *mutate*, etc.), and ten functions from Pandas [64] (e.g., *pandas.unique*, *pandas.merge*, *pandas.concat*, etc).

6 USER STUDY

To assess the effectiveness of the visualization design, we conducted a controlled study centered on two high-level questions: 1) does the glyph design improve user efficiency in comprehending the semantics of data wrangling? and 2) does the provenance graph facilitate the understanding of data dependencies? We ran the evaluation using real-world data tables and scripts written in the R programming language. All documentation, including scripts, data tables, questions, etc., are provided in the supplemental material.

6.1 Participants and Apparatus

We recruited 20 volunteers (4 females and 16 males) aged 22 to 35 ($\mu = 25.45$, $\sigma = 3.33$). The majority of participants (15/20) were postgraduate students majoring in Statistics or Computer Science, while the others worked as data analysts or algorithm engineers in a national research lab. They were all proficient in programming using Python, JAVA, or Javascript and had experience in data transformation. In addition, to ensure that all participants had difficulties understanding scripts, we only recruited those who had not written a line of code in R. Participants completed the study using a desktop computer (3.20GHz 8-Core Intel Core i7, 32 GB memory) with a 27-inch monitor (3840×2160 resolution) and an external mouse and keyboard, and the study was distributed through Google Chrome on a Windows 10 machine.

6.2 Techniques

To our knowledge, no prior work targets visualizing the semantics of data transformation. Hence, we compared the visualization design with textual description derived from a commercial data wrangling system, Trifacta [76]. Some description was not directly supported by Trifacta, such as mapping the values from one column into another (a subtype of *Transform Columns*). In these cases, we generated text by combining the descriptions of two transformations, e.g., create *a new column* from *original columns* and delete *the original ones* (italic text will be replaced by column names). To align with SOMNUS in describing a sequence of transformations, we included additional information in the textual description, including line index, the shape of data tables, and the output table name of a function.

The design of our glyph collection contained textual information describing the type of transformation in the glyph. We noted that the comparison between pure text and visualization with text would be unfair. To evaluate the effectiveness of visual representation, we removed the textual description of glyphs in the study. We envisioned that our glyph design and SOMNUS would be more effective by including text.

6.3 Tasks and Design

We performed a within-subject design with two experimental techniques and ten experimental tasks. To address any memory learning effects, we created two different sets of ten tasks. The orders of the two techniques and the task sets were counterbalanced using a Latin square. Within each technique, participants completed ten tasks which were shown in a fixed order. Thus, the whole study contained 2 techniques $\times$ 2 sets $\times$ 10 tasks $=$ 40 trials. Each task trial included a piece of code, a visual or textual explanation for the code, and a multiple-choice question where each question had one or more correct answers. In addition, the study system provided data tables and documentations of functions for reference. We chose the multiple-choice test for evaluation because it could increase participants' confidence in completing tasks and be more convenient for statistical analysis of test results over the constructed-response test [47], [68]. In our study, all questions and choices are carefully designed from varying perspectives of table changes and dependencies to answer the above two high-level questions. However, we do not guarantee they can genuinely measure participants' understanding as comprehension is abstract and hard to access directly.

The ten study tasks consisted of five function understanding tasks (*Func*) followed by five script understanding tasks (*Script*). Moreover, each question was required to select all correct choices. *Func* focused on the semantics of individual functions, including the output tables and operations. Example choices were statements such as, *"The output table has a different number of rows with the input table"* and *"This function renames the column A to B"*. *Script* focused on the understanding of data provenance through a sequence of functions. Example questions were, *"Which data tables contribute to the creation of table A?"* and *"How many data transformations are performed from table A to B?"* We carefully designed the task questions and choices to avoid ambiguous answers and maintain the same difficulty across task sets. To assist the exploration of data provenance, the study system supports zooming and panning. A participant answered a question correctly if and only if all correct options were selected.

6.4 Data

To evaluate how our visual design performed in real-world scenarios, we selected candidate code pieces and data tables provided by Kasica et al. [43], collected from Github [9], [20]. We focused on functions belonging to two toolkits, i.e., dplyr [21] and tidyr [75]. We randomly chose a set of statements for *Func* and consecutive code pieces for *Script*. Due to the limitation of the programming adaptor, some statements, or code pieces, were inadequate for our studies, such as those without explicit function names and input tables. Hence, we replaced these statements with their functional alternatives. In addition, we removed comments to avoid misleading.

We created two datasets, and each corresponded to one task set. To maintain the same difficulty level across the two datasets, we established three rules in dataset creation. First, both datasets contained the same number of *Combine Tables* and *Separate Tables*. Because these transformations involved more data tables compared to the rest, they might pose challenges in understanding table provenance. Second, each dataset included the same number of functions for *Func* and *Script*. In our study, five functions were applied to *Func*, and nine functions were used for *Script*. To keep the datasets distinct, functions in one dataset could not be reused in the other. However, exceptions existed for two functions, i.e., *read.csv()* and *group_by()*, to ensure that code pieces can be correctly interpreted. Specifically, *read.csv()* loaded data at the beginning of code pieces while *group_by()* served as a prerequisite for other functions, such as *summarise()* and *mutate()*, to achieve some transformations.

6.5 Procedure

The study began with a brief introduction to data transformation. Then, we collected demographic information of each participant, including the experience of programming, age, occupation, etc. Prior to the main experiment for each technique, participants performed ten training tasks with a separate dataset. During training sessions, participants were instructed to think aloud, and the experimenter helped answer questions and overcome difficulties. We reminded participants that they could always skip a task when they were not confident about the answer. In the main experiment, participants were asked to complete ten tasks with each technique. The data tables and documentations of functions provided by the study system were folded by default. Participants could click to expand this information for reference. Our system recorded the clicks for further analysis. In addition, the system recorded task completion times and participants' answers. After the main experiment for each technique, participants were asked to rate the usefulness and intuitiveness of the technique using a seven-point Likert scale. After the study, a semi-structured interview was conducted to collect their feedbacks. We took notes during the whole session. Each participant took approximately one hour to finish the study and received 10 dollars as compensation.

6.6 Quantitative Results

Accuracy: The individual answer-level results of each question across the two techniques are provided in the supplemental material. We found that the accuracies of Q1 in *Func* Set1 and Q4 in *Func* Set2 were very low in both two techniques. There were two possible reasons for the results. First, the semantics of some *Combine* operations like *Summarize* and *Supplement* were hard to describe, which involved the rule of combination, and the number of rows

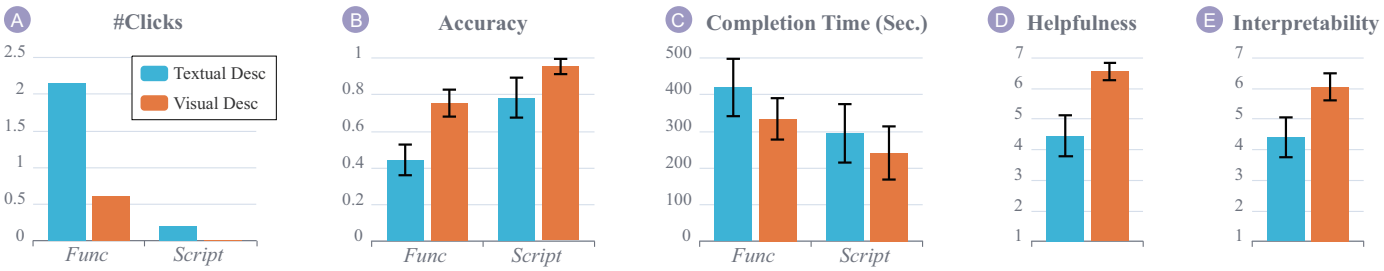

Fig. 6. Among participants, (a) shows the average number of clicks in terms of two task sets and two techniques, (b) and (c) are the average accuracy and completion time. (d) and (e) display user ratings in terms of helpfulness and interpretability. The error bars indicate the 95% confidence interval.

and columns. Second, participants seldom chose the NOTA option (i.e., *"None of the above"*), which was the correct answer for the two questions. As P16 explained, *"When I am not sure about the answer, I tend to choose the one that seems correct (instead of NOTA)."* Besides, visual descriptions' accuracy was significantly higher than that of textual descriptions (see Q5 in *Func* Set2 and Q4 in *Func* Set1), which indicates that our visual design is superior to text in describing complex operations, including *fold* and *left join*. Figure 6(b) depicts the results with a 95% confidence interval. On average, participants achieved a much higher accuracy with visual description ($\mu = 0.85$, $\sigma = 0.16$) than with textual description ($\mu = 0.61$, $\sigma = 0.27$) for all tasks. Especially for *Func*, participants got an average accuracy of 44% ($\sigma = 0.18$) with the baseline technique. And they achieved an accuracy of 75% ($\sigma = 0.16$) using our approach. In our study, textual description described what the transformation was. However, it helped little in communicating how the transformation performed. As P9 commented, *"Though the text told me that the function performs left join, I do not know exactly how left join works."* On the contrary, visual description helped participants understand the semantics of transformations that they were unfamiliar with.

Completion time: We performed an independent-samples t-test with a null hypothesis that the participants took the same amount of time finishing tasks with each technique. We found a marginally positive effect of our approach with which participants completed *Func* faster than the baseline technique ($p < 0.1$). We also ran a paired two-sample Wilcoxon signed-rank test to identify whether the presentation order of two techniques affected the task completion time. The results indicated no significant effect of the order on the completion time for *Func* ($p = 0.1536$) while a notable significant effect for *Script* ($p = 0.0083$). That is, participants, performed faster using the later technique in *Script*.

Number of clicks: In *Func*, participants expanded function documentation and data tables 2.15 times using textual descriptions. On the contrary, they clicked 0.6 times using our approach. Compared to the baseline approach, participants sought less information in completing the tasks. This result indicated that our approach helped participants understand the semantics of transformations with necessary visual encoding. The number of documentation and table clicks was much fewer in *Script*, which might be because this information helped little in script understanding tasks.

Preference: For comparing the helpfulness and interpretability of the two techniques across ten tasks, we ran Mann-Whitney's U tests to evaluate the difference in the responses of our seven-point Likert Scales. We found our technique ($\mu = 6.55$, $\sigma = 0.61$) was significantly more helpful in assisting participants to understand transformation than textual descriptions ($\mu = 4.45$, $\sigma = 1.43$): $U = 34.5$, $p < 0.01$. In terms of interpretability, our technique ($\mu = 6.05$, $\sigma = 0.95$) was easier to understand than textual descriptions ($\mu = 4.40$, $\sigma = 1.39$): $U = 71$, $p < 0.01$.

6.7 Qualitative Feedback

All participants showed great interest in the visualization design. Some participants pointed out that data visualization is a universal language that simplifies learning and communicating. For the *Func* tasks, participants appreciate the design of glyphs. As P6 mentioned that *"it (the glyph visualization) is intuitive and informative."* P14 added, *"I do not need to look up the documentation of functions because glyphs explain all."* For the *Script* tasks, the node-link diagram presents table provenance using a sequence of glyphs, which is efficient for navigation. As P3 noted that, *"It is laborious to extract the table dependencies from textual descriptions compared to the visualization."* Besides, participants also provided valuable suggestions for our design.

Comments on the glyphs: The glyph design can be improved from the following aspects. First, the color encoding of different glyphs may be confusing. P8 noted, *"I would think they (columns with the same color in different glyphs) are the same columns."* She further suggested, *"Different columns should be depicted using a different color (in the provenance graph)."* Second, some participants (P10, P14) pointed out that text was superior to visualization in some cases. For example, the *filter* function in R deletes rows due to some conditions. When multiple conditions are passed as parameters, our glyph design cannot distinguish whether BOTH conditions are applied, or EITHER condition is used. On the contrary, a textual description can articulate it clearly. P18 and P7 recommended integrating textual description and visualization to utilize the strength of the two techniques.

Comments on the provenance graph: The design of the provenance graph may suffer from two issues. First, the provenance graph would be too long when the number of transformations increases. As a result, participants continuously zoomed and panned the graph in finishing tasks. P2 suggested that the pipeline could be presented vertically so that he could explore it using a scroll bar. In addition, P10 commented, *"The pipeline (provenance graph) should be folded by default, and can be expanded on demand."* Second, some participants (P3, P5) suggested supporting programs with complex control flow. P6 commented, *"The statements in the study are too simple. I wonder how the visualization performs in programs with IF-ELSE (conditional statement) or FOR (loop statement) statements."*

7 Example Applications

To demonstrate how SOMNUS can be applied to different usage scenarios, we design and implement two prototypes based on

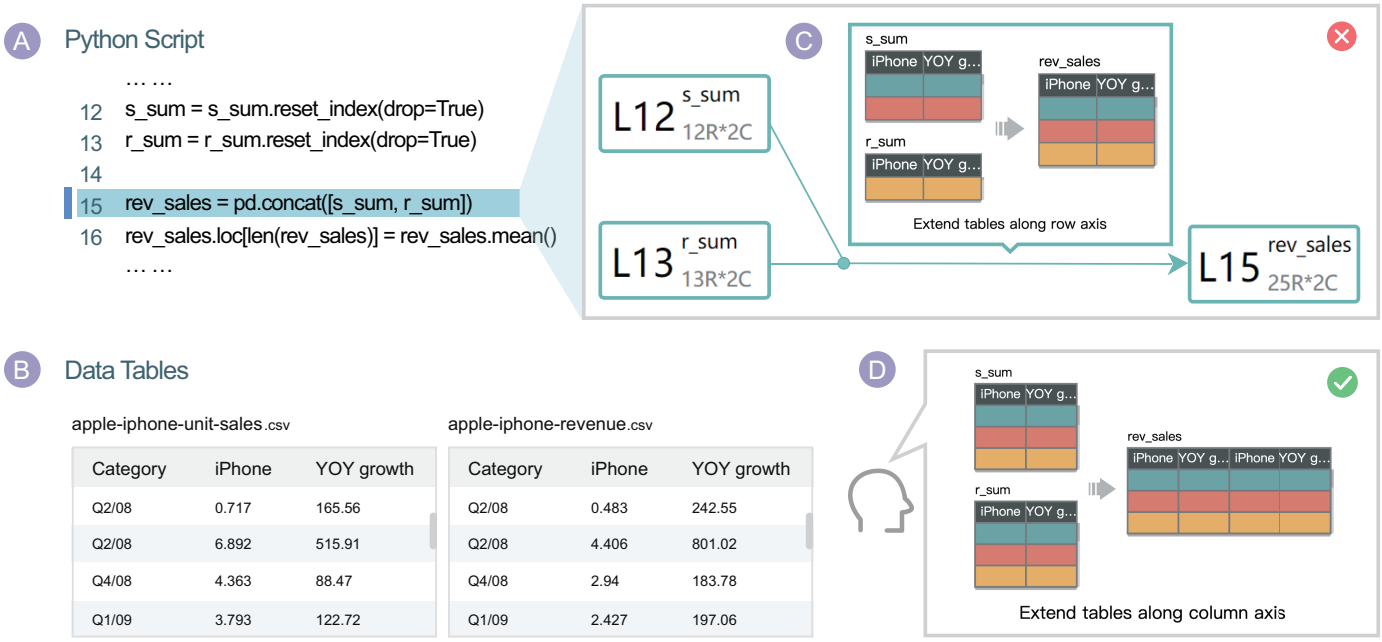

Fig. 7. The application of SOMNUS in validating the process of data transformation written in Python. SOMNUS takes a piece of code (a) and data tables (b) as input, and outputs a visualization showing table provenance across data transformations. (c) shows a snippet of the visualization. By exploring the table provenance, a data worker can identify errors in the transformation with ease. (d) depicts the correct transformation beared in mind by data workers.

SOMNUS. The first prototype helps data scientists validate the procedure of data transformation written in Python, while the second reveals intermediate data transformations given source and target tables.

7.1 Double-checking

In this study, we collaborate with two data scientists in a national research lab. They usually work together to finish an analytical report. One critical task in their work is validation, or called double-checking in practice. Specifically, when a data scientist finishes a workflow, the other needs to check and validate the entire workflow by scrutinizing the code and independently reproducing the workflow. However, identifying errors in the code is not an easy task, which requires a deep understanding of a large number of functions and data models. Inspired by the real-world use case, the first application illustrates how SOMNUS aids data scientists in validating and debugging a script of the wrangling process.

Assume Lucy and Jane are two data scientists working in a national research lab. After Jane finishes a data transformation procedure written in Python, Lucy is invited to validate the piece of code to ensure accuracy. The goal of the code is to combine two tables [26] and compute average iPhone unit scales and revenue across years. Lucy first uploads the two input tables (Figure 7(b)) in the Data Panel and copy-pastes a code piece to the Script Panel. After clicking "Upload and Run", the provenance graph is displayed in the Graph Panel. To examine each step, Lucy explores individual transformations in the provenance graph by zooming and panning. A glyph showing the combination of two tables catches her eye. As shown in Figure 7(c), the two tables are combined along the row axis. However, Lucy makes sure that the two tables should be concatenated by column (Figure 7(d)). To reason the result, she clicks the glyph to locate and highlight the 15th line of code (Figure 7(a)). She notices that the parameter *axis* is not explicitly mentioned in the *concat* function. By default, however, the concatenation is performed along the row axis with implicit *axis=0*. Hence, Lucy corrects the statement to *rev_sales=pd.concat([s_sum,r_sum], axis=1)*, and finally obtains the correct results.

7.2 MORPHEUS Revisited

MORPHEUS [25] is a program synthesis algorithm that generates a script for data processing. The algorithm accepts multiple source tables and a target table as input and automatically outputs lines of R code to reflect the process of transformation. MORPHEUS is useful in a number of scenarios. For example, the output script can automate the process of data transformation and can be reused and revised for future applications. The output of MORPHEUS, however, is hard to understand due to obscure function usage and parameters. In the second application, we apply SOMNUS to explain the scripts generated by MORPHEUS. The adapted system shown in Figure 8 is almost identical to the SOMNUS system. The difference is threefold. First, the Data Panel accepts multiple source tables and a target table, which are passed to MORPHEUS on the server side. Second, based on code pieces returned by MORPHEUS and data tables, SOMNUS runs and yields a series of input and output tables for each function and a table provenance and passes them to the client side. Third, the Script Panel shows the script that is not editable.

The application is motivated by a real-world case from Stack-Overflow [1]. Assume Devin has two original tables (*input1.csv, input2.csv*) and one target table (*output1.csv*) at hand. He first uploads those tables to the system to understand the correct approach to transforming the original tables to the target table. After clicking the "Upload and Run" button, a piece of code is shown in the Script Panel, and a provenance graph is displayed in the Graph Panel. From the provenance graph, Devin sees seven edges, indicating that the entire process takes seven data transformations.

Devin has no prior knowledge about R and dplyr. To get an idea of individual functions, such as *mutate*, he clicks the 8th line of

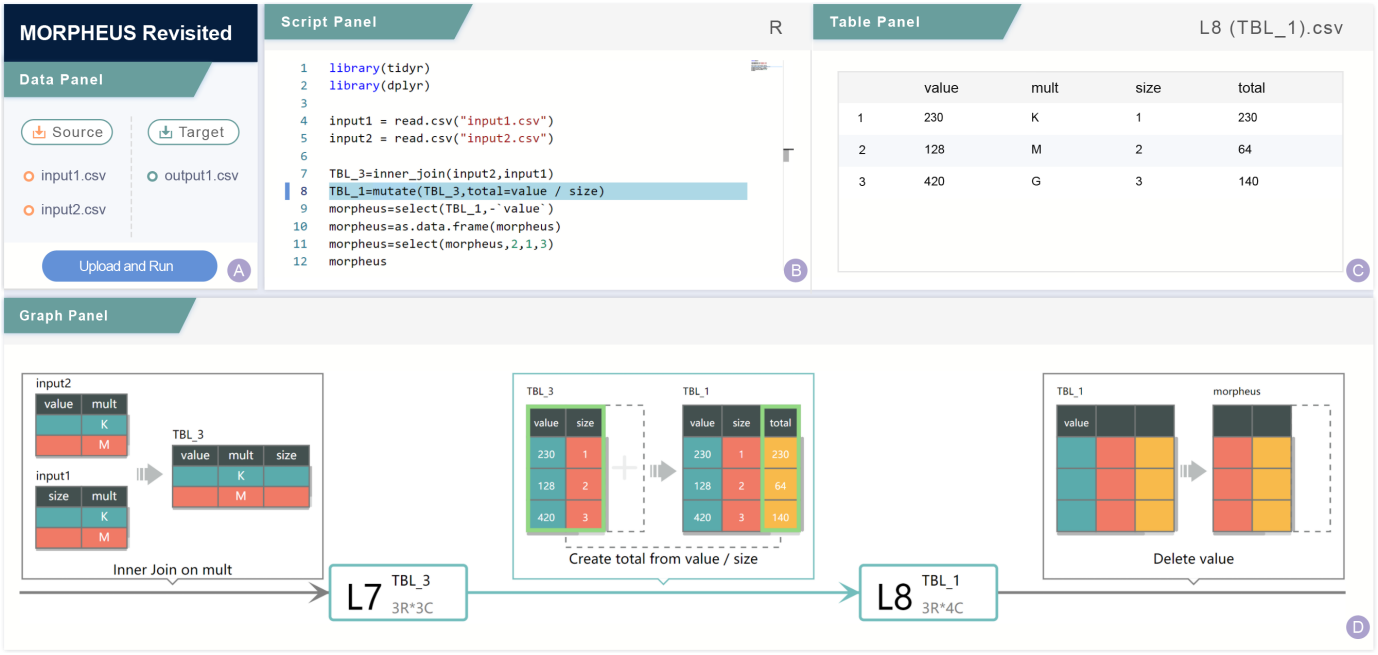

Fig. 8. By combining MORPHEUS, our system generates and illustrates a series of data transformations given source data tables and a target table. The system comprises four panels, i.e., a Data Panel allowing users to upload data tables, a Script Panel showing code pieces in R, a Table Panel shows intermediate data tables, and a Graph Panel that depicts table provenance.

code in the Script Panel (Figure 8(b)). Then the transformation and its input and output tables are located and highlighted in the Graph Panel (Figure 8(d)). He figures out that the function creates a new column called *"total"* from *"value"* divided by *"size"*. From the Script Panel, Devin observes two *select* functions. He clicks the two lines of code and finds they perform different transformations, i.e., one removes a column *"value"* while the other rearrange columns.

8 DISCUSSION

The evaluation shows that our glyph collection and SOMNUS are effective in presenting data transformations, and SOMNUS can be generalized to various programming languages and example applications. Besides feedbacks and suggestions listed in Section 6, we identify some limitations in the design and implementation of SOMNUS.

First, the scalability of SOMNUS is limited in terms of the number of functions and parameter combinations. The Code Parser and Transformation Inference modules of SOMNUS are customized for each function and parameter. The current prototype supports 25 functions from tidyr and dplyr in R and ten functions from Pandas in Python with a small set of parameters. Extending our work to a number of functions and parameters is possible. However, it would be tedious. To align with a large number of functions that are typically used for data transformation, a promising direction is to explore learning-based algorithms that can map a function and its parameters to a type of transformation at scale. This work can act as a starting point for generating training data for such algorithms. In addition, if a lengthy script contains numerous operations, the provenance graph would be too long to navigate. We acknowledge that the basic layout of transformation workflows will result in node-link diagrams with a suboptimal aspect ratio that require frequent panning/zooming. We notice that a number of interaction techniques are designed to navigate lengthy content. For example, focus+context screens [5] can facilitate the exploration of multiscale documents. In addition, the collapse-to-zoom navigation [6] is proposed to explore lengthy web pages. In future iterations, we plan to integrate advanced interaction techniques to alleviate the issue.

Second, the generalizability of the glyph space has yet been explored. By depicting in-table and out-table text in the glyph, the glyph collection can be generalized to a larger number of transformations, as shown in Figure 2. On the other hand, the glyph design lacks support for some commonly used data transformations, such as *Transpose*. To what extent does the glyph space adapts to transformations is unknown. In future research, we plan to explore the mapping between the glyph space and transformation space to understand the scope.

Third, the presentation of in-table text may result in inconsistencies in some data tables. For example, in Figure 4, the *Combine Rows* shows the results of the *colMeans* function in R, which derives the mean value for each column. However, the mean value of the input table does match the results in the output table. That is because all in-table text is derived from the original data. In the current stage, we combine textual description and visualization in the glyph design to alleviate the weakness caused by inconsistencies. In addition, if the text in glyphs, including column names, cell contents, and summary descriptions, is long, it can not be fully displayed by default. This can be difficult for users to spot their difference, especially when the text has the same prefix. Currently, the text elision issue is resolved through interaction. That is, when a user hovers over the omitted text in the glyph, the whole text will be displayed.

Fourth, the shortcoming of individual glyphs has yet been explored. Though the controlled study reveals the overall effectiveness of visual description compared to textual description (*Func* in Figure 6), it is far from enough to exploit the shortcoming of individual glyphs, which requires enumerating possible functions and their parameters. For example, the glyph designed for *Delete*

Rows may be ineffective when a data table does not contain counterexamples. Because the semantics of filtering conditions can hardly be visualized without counterexamples. To obtain a comprehensive understanding of the performance of the glyph collection, we plan to validate individual glyphs using various functions and parameter combinations and conduct a large-scale user study for evaluation.

Fifth, the color encoding of glyphs may cause confusion in the provenance graph. As one participant in the user study pointed out, columns with the same color in different glyphs may be mistakenly regarded as identical columns. We intend to combine visual designs with interactions to resolve this ambiguity in our future work.

9 Conclusion and Future Work

In this paper, we develop visualization techniques to illustrate the semantics of code pieces in the context of data transformation. To present individual transformations, we explore design space consisting of two primary dimensions, i.e., key parameters to be encoded and possible visual channels that can be mapped. Based on the design space, we derive a collection of glyphs targeting 23 types of transformations. We argue that the glyph collection can adapt to a broader range of transformations by depicting in-table and out-table text. To illustrate a sequence of statements, we design and develop SOMNUS, a pipeline that accepts code pieces and data tables as input and generates a graph showing table provenance across a series of data transformations. The results of a controlled study have demonstrated the effectiveness and intuitiveness of the visualization design for our study participants. Through example applications, we show how SOMNUS can be adapted to different programming languages and usage scenarios.

In the future, we plan to enhance SOMNUS by supporting a large number of functions and parameters in dplyr (R), tidyr (R), and Pandas (Python). However, the manual enhancement could be laborious and tedious. We plan to investigate algorithms that automatically map statements to data transformations to facilitate the adaption of functions and parameters. Next, since complex control flow is commonly used in data transformation, we would like to explore how to visualize conditional statements and loops in the provenance graph.

Acknowledgments

This work was supported by NSFC (62072400, 62002331) and the Collaborative Innovation Center of Artificial Intelligence by MOE and Zhejiang Provincial Government (ZJU). The work was also partially funded by the Zhejiang Lab (2021KE0AC02, 2020KE0AA02). We are grateful to our study participants and anonymous reviewers for their insightful feedback.

References

[1] recursive error in dplyr mutate. https://stackoverflow.com/questions/30374143/recursive-error-in-dplyr-mutate, 2015.

[2] Z. Abedjan, J. Morcos, I. F. Ilyas, M. Ouzzani, P. Papotti, and M. Stonebraker. DataXFormer: A robust transformation discovery system. In *Proceedings of IEEE International Conference on Data Engineering*, pp. 1134–1145, 2016.

[3] E. E. Aftandilian, S. Kelley, C. Gramazio, N. Ricci, S. L. Su, and S. Z. Guyer. Heapviz: interactive heap visualization for program understanding and debugging. In *Proceedings of International Symposium on Software Visualization*, pp. 53–62, 2010.

[4] L. Bartram, M. Correll, and M. Tory. Untidy data: The unreasonable effectiveness of tables. *IEEE Transactions on Visualization and Computer Graphics*, 28(01):686–696, 2022.

[5] P. Baudisch, N. Good, V. Bellotti, and P. Schraedley. Keeping things in context: a comparative evaluation of focus plus context screens, overviews, and zooming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 259–266, 2002.

[6] P. Baudisch, X. Xie, C. Wang, and W.-Y. Ma. Collapse-to-zoom: viewing web pages on small screen devices by interactively removing irrelevant content. In *Proceedings of Annual ACM Symposium on User Interface Software and Technology*, pp. 91–94, 2004.

[7] F. Beck, F. Hollerich, S. Diehl, and D. Weiskopf. Visual monitoring of numeric variables embedded in source code. In *Proceedings of First IEEE Working Conference on Software Visualization*, pp. 1–4, 2013.

[8] F. Beck, O. Moseler, S. Diehl, and G. D. Rey. In situ understanding of performance bottlenecks through visually augmented code. In *Proceedings of International Conference on Program Comprehension*, pp. 63–72, 2013.

[9] beecycles. Power of irma. https://github.com/beecycles/Power_of_Irma, 2018.

[10] A. Bigelow, C. Nobre, M. Meyer, and A. Lex. Origraph: Interactive network wrangling. In *Proceedings of IEEE Conference on Visual Analytics Science and Technology*, pp. 81–92, 2019.

[11] C. Bors, T. Gschwandtner, and S. Miksch. Capturing and visualizing provenance from data wrangling. *IEEE Computer Graphics and Applications*, 39(6):61–75, 2019.

[12] M. Bostock. Visualizing algorithms. https://bost.ocks.org/mike/algorithms/, 2014.

[13] M. Bostock, V. Ogievetsky, and J. Heer. D^3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, 2011.

[14] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst. Interactive record/replay for web application debugging. In *Proceedings of Annual ACM Symposium on User Interface Software and Technology*, pp. 473–484, 2013.

[15] J. Cheon, D. Kang, and G. Woo. VizMe: An annotation-based program visualization system generating a compact visualization. In *Proceedings of the International Conference on Data Engineering*, pp. 433–441, 2019.

[16] N. Chotisarn, L. Merino, X. Zheng, S. Lonapalawong, T. Zhang, M. Xu, and W. Chen. A systematic literature review of modern software visualization. *Journal of Visualization*, 23(4):539–558, 2020.

[17] code2flow. online interactive code to flowchart converter. https://app.code2flow.com/. Accessed: Jan 29, 2022.

[18] C. Demetrescu, I. Finocchi, and J. T. Stasko. Specifying algorithm visualizations: Interesting events or state mapping? In *Proceedings of Software Visualization*, pp. 16–30, 2002.

[19] Z. Deng, D. Weng, X. Xie, J. Bao, Y. Zheng, M. Xu, W. Chen, and Y. Wu. Compass: Towards better causal analysis of urban time series. *IEEE Transactions on Visualization and Computer Graphics*, 28(1):1051–1061, 2022.

[20] B. S. D. Desk. Baltimore police overtime in fiscal years 2018 and 2019. https://github.com/baltimore-sun-data/baltimore-police-overtime, 2020.

[21] dplyr. R package: dplyr v0.7.8. https://www.rdocumentation.org/packages/dplyr/versions/0.7.8. Accessed: Jan 29, 2022.

[22] I. Drosos, T. Barik, P. J. Guo, R. DeLine, and S. Gulwani. Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists. In *Proceedings of CHI Conference on Human Factors in Computing Systems*, pp. 1–12, 2020.

[23] Eclipse. Eclipse Layout Kernel (ELK). https://www.eclipse.org/elk/. Accessed: Jan 29, 2022.

[24] R. Faust, K. Isaacs, W. Z. Bernstein, M. Sharp, and C. Scheidegger. Anteater: Interactive visualization of program execution values in context, 2020.

[25] Y. Feng, R. Martins, J. Van Geffen, I. Dillig, and S. Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. *ACM SIGPLAN Notices*, 52(6):422–436, 2017.

[26] B. Figures. Apple iphone unit sales and revenue. https://barefigur.es/companies/apple/iphone/, 2021.

[27] D. Fisher. Animation for visualization: Opportunities and drawbacks. In *Beautiful Visualization*. O'Reilly Media, 2010.

[28] S. Grissom, M. F. McNally, and T. Naps. Algorithm visualization in cs education: comparing levels of student engagement. In *Proceedings of ACM Symposium on Software Visualization*, pp. 87–94, 2003.

[29] P. J. Guo. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of ACM Technical Symposium on Computer Science Education*, pp. 579–584, 2013.

[30] P. J. Guo, S. Kandel, J. M. Hellerstein, and J. Heer. Proactive wrangling: Mixed-initiative end-user programming of data transformation scripts. In *Proceedings of Annual ACM Symposium on User Interface Software and Technology*, pp. 65–74, 2011.

[31] S. Hansen, N. H. Narayanan, and M. Hegarty. Designing educationally effective algorithm visualizations. *Journal of Visual Languages & Computing*, 13(3):291–317, 2002.

[32] D. R. Hanson and J. L. Korn. A simple and extensible graphical debugger. In *Proceedings of the USENIX Annual Technical Conference*, pp. 183–174, 1997.

[33] M. Harward, W. Irwin, and N. Churcher. In situ software visualisation. In *Proceedings of Australian Software Engineering Conference*, pp. 171–180, 2010.

[34] J. Heer and A. Perer. Orion: A system for modeling, transformation and visualization of multidimensional heterogeneous networks. *Information Visualization*, 13(2):111–133, 2014.

[35] J. Hoffswell, A. Satyanarayan, and J. Heer. Visual debugging techniques for reactive data visualization. *Computer Graphics Forum*, 35(3):271–280, 2016.

[36] J. Hoffswell, A. Satyanarayan, and J. Heer. Augmenting code with in situ visualizations to aid program understanding. In *Proceedings of CHI Conference on Human Factors in Computing Systems*, pp. 1–12, 2018.

[37] D. Huynh. Openrefine. https://openrefine.org. Accessed: Jan 29, 2022.

[38] J. P. Inala and R. Singh. WebRelate: integrating web data with spreadsheets using examples. *Proceedings of ACM on Programming Languages*, 2(POPL):1–28, 2017.

[39] Z. Jin, M. R. Anderson, M. Cafarella, and H. Jagadish. Foofah: Transforming data by example. In *Proceedings of ACM International Conference on Management of Data*, pp. 683–698, 2017.

[40] Jupyter. Jupyter notebook. https://jupyter.org. Accessed: Jan 29, 2022.

[41] S. Kandel, J. Heer, C. Plaisant, J. Kennedy, F. Van Ham, N. H. Riche, C. Weaver, B. Lee, D. Brodbeck, and P. Buono. Research directions in data wrangling: Visualizations and transformations for usable and credible data. *Information Visualization*, 10(4):271–288, 2011.

[42] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 3363–3372, 2011.

[43] S. Kasica, C. Berret, and T. Munzner. Table Scraps: An actionable framework for multi-table data wrangling from an artifact study of computational journalism. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):957–966, 2021.

[44] M. Khan, L. Xu, A. Nandi, and J. M. Hellerstein. Data tweening: incremental visualization of data transforms. *Proceedings of the VLDB Endowment*, 10(6):661–672, 2017.

[45] A. J. Ko and B. A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 151–158, 2004.

[46] D. A. Kosower, J. J. Lopez-Villarejo, and S. Roubtsov. Flowgen: Flowchart-based documentation framework for c++. In *Proceedings of IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 59–64, 2014.

[47] W. L. Kuechler and M. G. Simkin. How well do multiple choice tests evaluate student understanding in computer programming classes? *Journal of Information Systems Education*, 14(4):389, 2003.

[48] C. Lewis and G. Olson. Can principles of cognition lower the barriers to programming? In *Proceedings of Empirical studies of programmers: second workshop*, pp. 248–263, 1987.

[49] T. Lieber, J. R. Brandt, and R. C. Miller. Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 2481–2490, 2014.

[50] H. Lieberman and C. Fry. ZStep 95: A reversible, animated source code stepper. In *Software Visualization: Programming as a Multimedia Experience*, pp. 277–292, 1997.

[51] J. Lin, J. Wong, J. Nichols, A. Cypher, and T. A. Lau. End-user programming of mashups with vegemite. In *Proceedings of International Conference on Intelligent User Interfaces*, pp. 97–106, 2009.

[52] J. Liu, N. Boukhelifa, and J. R. Eagan. Understanding the role of alternatives in data analysis practices. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):66–76, 2020.

[53] S. Liu, G. Andrienko, Y. Wu, N. Cao, L. Jiang, C. Shi, Y.-S. Wang, and S. Hong. Steering data quality with visual analytics: The complexity challenge. *Visual Informatics*, 2(4):191–197, 2018.

[54] Z. Liu, S. B. Navathe, and J. T. Stasko. Network-based visual analysis of tabular data. In *Proceedings of IEEE Conference on Visual Analytics Science and Technology*, pp. 41–50, 2011.

[55] J. Morcos, Z. Abedjan, I. F. Ilyas, M. Ouzzani, P. Papotti, and M. Stonebraker. DataXFormer: An interactive data transformation tool. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 883–888, 2015.

[56] T. Munzner. A nested model for visualization design and validation. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):921–928, 2009.

[57] L. Nguyen, S. Krüger, P. Hill, K. Ali, and E. Bodden. VisuFlow: a debugging environment for static analyses. In *Proceedings of IEEE/ACM International Conference on Software Engineering: Companion*, pp. 89–92, 2018.

[58] C. Niederer, H. Stitz, R. Hourieh, F. Grassinger, W. Aigner, and M. Streit. TACO: visualizing changes in tables over time. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):677–686, 2018.

[59] S. Oney and B. Myers. FireCrystal: Understanding interactive behaviors in dynamic web pages. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 105–108, 2009.

[60] A. Oy. Visustin v7 flow chart generator. https://www.aivosto.com/visustin.html, 2013.

[61] X. Pu, S. Kross, J. M. Hofman, and D. G. Goldstein. Datamations: Animated explanations of data analysis pipelines. In *Proceedings of CHI Conference on Human Factors in Computing Systems*, pp. 1–14, 2021.

[62] Y. Qian and J. Lehman. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education*, 18(1):1–24, 2017.

[63] E. D. Ragan, A. Endert, J. Sanyal, and J. Chen. Characterizing provenance in visualization and data analysis: an organizational framework of provenance types and purposes. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):31–40, 2016.

[64] J. Reback, jbrockmendel, W. McKinney, J. V. den Bossche, T. Augspurger, P. Cloud, S. Hawkins, M. Roeschke, gfyoung, Sinhrks, A. Klein, P. Hoefler, T. Petersen, J. Tratner, C. She, W. Ayd, S. Naveh, M. Garcia, J. Darbyshire, J. Schendel, R. Shadrach, A. Hayden, D. Saxton, M. E. Gorelli, F. Li, M. Zeitlin, V. Jancauskas, A. McMaster, P. Battiston, and S. Seabold. pandas-dev/pandas: Pandas 1.4.0, Jan 2022.

[65] C. A. Shaffer, M. Akbar, A. J. D. Alon, M. Stewart, and S. H. Edwards. Getting algorithm visualizations into the classroom. In *Proceedings of ACM Technical Symposium on Computer Science Education*, pp. 129–134, 2011.

[66] C. A. Shaffer, M. Cooper, and S. H. Edwards. Algorithm visualization: a report on the state of the field. In *Proceedings of SIGCSE Technical Symposium on Computer Science Education*, pp. 150–154, 2007.

[67] N. Shrestha, C. Botta, T. Barik, and C. Parnin. Here we go again: why is it difficult for developers to learn another programming language? In *Proceedings of IEEE/ACM International Conference on Software Engineering*, pp. 691–701, 2020.

[68] M. G. Simkin and W. L. Kuechler. Multiple-choice tests and student understanding: What is the connection? *Decision Sciences Journal of Innovative Education*, 3(1):73–98, 2005.

[69] W. Soft. AutoFlowchart. https://autoflowchart.informer.com/. Accessed: Jan 29, 2022.

[70] T. Software. Tableau prep builder. https://www.tableau.com/products/prep. Accessed: Jan 29, 2022.

[71] J. Sorva, V. Karavirta, and L. Malmi. A review of generic program visualization systems for introductory programming education. *ACM Transactions on Computing Education*, 13(4):1–64, 2013.

[72] J. Sundararaman and G. Back. HDPV: interactive, faithful, in-vivo runtime state visualization for c/c++ and java. In *Proceedings of ACM Symposium on Software Visualization*, pp. 47–56, 2008.

[73] B. Swift, A. Sorensen, H. Gardner, and J. Hosking. Visual code annotations for cyberphysical programming. In *Proceedings of International Workshop on Live Programming*, pp. 27–30, 2013.

[74] T. Tang, Y. Wu, L. Yu, Y. Li, and Y. Wu. VideoModerator: A risk-aware framework for multimodal video moderation in E-commerce. *IEEE Transactions on Visualization and Computer Graphics*, 28(1):846–856, 2022.

[75] tidyr. R package: tidyr v1.1.3. https://www.rdocumentation.org/packages/tidyr/versions/1.1.3. Accessed: Jan 29, 2022.

[76] Trifacta. Trifacta wrangler. https://www.trifacta.com/products/wrangler-editions/#wrangler. Accessed: Jan 29, 2022.

[77] J. Urquiza-Fuentes and J. A. Velázquez-Iturbide. A survey of program visualizations for the functional paradigm. In *Proceedings of Program Visualization Workshop*, pp. 2–9, 2004.

[78] J. Wang, J. Wu, A. Cao, Z. Zhou, H. Zhang, and Y. Wu. Tac-Miner: Visual tactic mining for multiple table tennis matches. *IEEE Transactions on Visualization and Computer Graphics*, 27(6):2770–2782, 2021.

[79] H. Wickham. *tidyr: Tidy Messy Data*, 2020. R package version 1.1.2.

[80] H. Wickham, R. François, L. Henry, and K. Müller. *dplyr: A Grammar of Data Manipulation*, 2021. R package version 1.0.4.

[81] L. Ying, T. Tang, Y. Luo, L. Shen, X. Xie, L. Yu, and Y. Wu. GlyphCreator: Towards example-based automatic generation of circular glyphs. *IEEE Transactions on Visualization and Computer Graphics*, 28(1):400–410, 2022.
[82] A. Zeller and D. Lütkehaus. DDD—a free graphical front-end for unix debuggers. *ACM Sigplan Notices*, 31(1):22–27, 1996.

Kai Xiong is a Ph.D. student at the State Key Laboratory of CAD&CG, Zhejiang University, and works under the supervision of Prof. Yingcai Wu. He holds a bachelor's degree in Computer Science from Xidian University. His main research interests center on visual analytics and data wrangling. He is also interested in how to apply artificial intelligence to data visualization.

Siwei Fu is an associate research scientist in Zhejiang Lab. His main research interests include: visual analytics, intelligent user interface, and natual language interface. He received his Ph.D. degree in Computer Science and Engineering from the Hong Kong University of Science and Technology. For more information, please visit https://fusiwei339.bitbucket.io/

Guoming Ding received his B.S. degree in Mechanical Engineering from Xi'an Jiaotong University in 2020. He is currently pursuing the master's degree with the State Key Lab of CAD&CG, Zhejiang University. His research interests mainly include the visualization and causal analysis.

Zhongsu Luo received his B.S. degree in Software Engineering from Zhejiang University of Technology in 2020. He is currently pursuing the master's degree in Zhejiang University of Technology. His research interests include the visualization, and visual analysis.

Rong Yu received the master's degree in visual communication design from Hangzhou Normal University, China, in 2015. She is currently a senior research engineer in Zhejiang Lab, China. She has eight years of expertise in graphic design and UI/UX design. For more information, please visit https://dribbble.com/yurongt

Wei Chen is a professor at the State Key Lab of CAD&CG, Zhejiang University. His research interests are visualization and visual analysis. He has published more than 30 IEEE/ACM Transactions and IEEE VIS papers. He actively served as a guest or associate editor of IEEE Transactions on Visualization and Computer Graphics, IEEE Transactions on Intelligent-Transportation Systems, and Journal of Visualization. For more information, please refer to http://www.cad.zju.edu.cn/home/chenwei/

Hujun Bao is a professor with the State Key Laboratory of CAD&CG and the College of Computer Science and Technology, Zhejiang University Zhejiang, China. He leads the 3D graphics computing group in the lab, which mainly makes researches on geometry computing, 3D visual computing, real-time rendering, and their applications. His research goal is to investigate the fundamental theories and algorithms to achieve good visual perception for interactive digital environments, and develop related systems.

Yingcai Wu is a professor at the State Key Lab of CAD&CG, Zhejiang University, China. His primary research interests lie in information visualization and visual analytics, with focuses on sports science and urban computing. He received his Ph.D. degree in Computer Science from the Hong Kong University of Science and Technology. Prior to his current position, Dr. Wu was a postdoctoral researcher at the University of California, Davis from 2010 to 2012, and a researcher in Microsoft Research Asia from 2012 to 2015. For more information, please visit http://www.ycwu.org