Falx: Synthesis-Powered Visualization Authoring

Chenglong Wang
clwang@cs.washington.edu
University of Washington

Yu Feng
yufeng@cs.ucsb.edu
University of California, Santa Barbara

Rastislav Bodik
bodik@cs.washington.edu
University of Washington

Isil Dillig
isil@cs.utexas.edu
The University of Texas at Austin

Alvin Cheung
akcheung@cs.berkeley.edu
University of California, Berkeley

Amy J. Ko
ajko@uw.edu
University of Washington

ABSTRACT

Modern visualization tools aim to allow data analysts to easily create exploratory visualizations. When the input data layout conforms to the visualization design, users can easily specify visualizations by mapping data columns to visual channels of the design. However, when there is a mismatch between data layout and the design, users need to spend significant effort on data transformation.

We propose Falx, a synthesis-powered visualization tool that allows users to specify visualizations in a similarly simple way but without needing to worry about data layout. In Falx, users specify visualizations using examples of how concrete values in the input are mapped to visual channels, and Falx automatically infers the visualization specification and transforms the data to match the design. In a study with 33 data analysts on four visualization tasks involving data transformation, we found that users can effectively adopt Falx to create visualizations they otherwise cannot implement.

ACM Reference Format:
Chenglong Wang, Yu Feng, Rastislav Bodik, Isil Dillig, Alvin Cheung, and Amy J. Ko. 2021. Falx: Synthesis-Powered Visualization Authoring. In *CHI Conference on Human Factors in Computing Systems (CHI '21), May 8–13, 2021, Yokohama, Japan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3411764.3445249

1 INTRODUCTION

Modern visualization authoring tools, such as declarative visualization grammars like ggplot2 [50], Vega-Lite [37] and interactive visualization tools like Tableau [42] and Voyager [54], are built to reduce data analysts' efforts in authoring visualizations in exploratory data analysis. At the heart of these tools, visualizations are specified using grammars of graphics [52], where every visualization can be succinctly specified using the following three components:

- A graphical mark that defines the geometric objects used to visualize the data (e.g., line, scatter plots, bars),
- A set of visual encodings that map data variables to visual channels (e.g., x, y-positions of points),
- A set of parameters that decide visualization details: coordinate system, scales of axes, legends and titles.

In practice, users only need to specify the mark and the visual encodings in order to create the visualization because many tools use a rule-based engine to automatically fill in parameters for visualization details (often referred to as "smart defaults") unless the user wants further customization. The abstraction of graphical marks, visual encoding channels, and adoption of smart default parameters open an expressive design space for data analysts that allow them to rapidly construct visualizations for exploratory analysis through simple specifications. For example, to visualize the dataset in Figure 1 with three columns Date, Temp (for temperature) and Type as a scatter plot, the user can choose the graphical mark "point" with encodings $\{x \mapsto \text{Date}, y \mapsto \text{Temp}, \textit{color} \mapsto \text{Type}\}$. The visualization tool then creates one point for each row in the input data, by mapping its values in columns Date and Temp to x,y-positions and assigning a color to each point based on its value in column Type. Here, the tool uses the default linear scale for x,y-axis and categorical scale for color, which are default parameters that the user does not need to specify explicitly. The final visualization is rendered in Figure 1 (right).

Date	Temp	Type
09-05	64.4	Low
09-05	87.8	High
09-06	53.6	Low
09-06	80.6	High

Date → x
Temp → y
Type → *color*

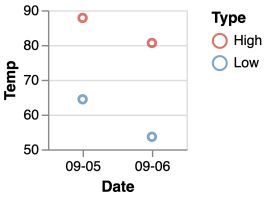


Figure 1: An example dataset and its scatter plot visualization that maps Date **to** x**,** Temp **to** y **and** Type **to** *color*.

In fact, the simplicity of these high-level visualization grammars is grounded in their abstract data model. These grammars expect that the input table is organized in a layout that matches the visualization design [51]: (1) each relation forms a row in the input data and corresponds to exactly one geometric object in the visualization, and (2) each data variable forms a column that can be mapped to a visual channel. In practice, however, the mismatch between the data layout and the visualization design is common due to the following reasons [9, 51]:

- Tables exported from different sources (e.g., database, analysis tool, different team member) may have different layouts and they may not directly match the visualization design.

CHI '21, May 8–13, 2021, Yokohama, Japan

ACM ISBN 978-1-4503-8096-6/21/05…$15.00
https://doi.org/10.1145/3411764.3445249

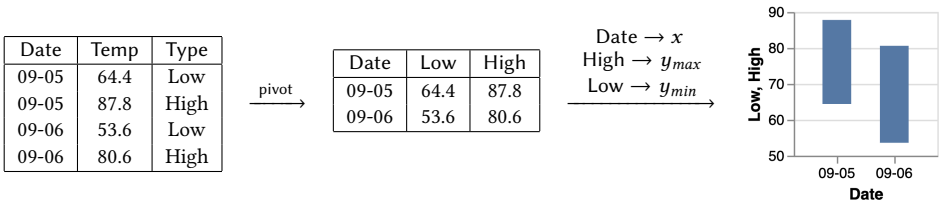


Figure 2: A different visualization design requires transformation of the original input data.

- Different analysis tasks require different visualization designs, and changes in the design can lead to different expected data layout.
- The data may need aggregation (e.g., average, count, culminative sum) or additional computation to derive new values prior to visualization.

In all of these cases, data analysts cannot directly visualize the data with a simple specification. They have to conceptualize the expected data layout and utilize data transformation tools (e.g., tidyverse [51], Trifacta [17]) to transform the data to match the visualization design. These additional tasks create a barrier for data visualizations and greatly increase the effort required for exploratory analysis [7, 9, 18, 53]. For example, if the data analyst decides to change the visualization in Figure 1 to a bar chart with floating bars that show the temperature range during each day (Figure 2 right), the original data layout will no longer match the new design since the new design expects three data columns (date, lowest temperature, highest temperature) that map to x, y_{max} and y_{min}. As a result, the data analyst needs to transpose the table in Figure 1 using a pivot operation (to collect key-values pairs in the Type and Temp columns into new columns) before mapping data columns to visual channels (Figure 2 right).

We propose Falx, a synthesis-based visualization authoring tool to address the challenges outlined above. [1] Falx builds on recent advances in program synthesis: many program synthesis tools (e.g., FlashFill [10], Wrex [6]) have been developed with the promises of automating challenging or repetitive programming tasks for end users by synthesizing programs from user demonstrations. In our design, instead of asking analysts to transform data and specify visualization manually, Falx asks analysts to *demonstrate* the visualization task using examples of mappings from concrete values in the input data (as opposed to table columns) to visual channels. Using these examples, Falx automatically synthesizes the programs to transform and visualize the full data, such that resulting visualizations are consistent with the examples (i.e., all example mappings are contained within the visualization). For example, for the data in Figure 2, the user can create an example bar 

to demonstrate the task and let Falx create the desired visualization for the full dataset (Figure 2 right). Sometimes, the examples can be ambiguous to Falx, and Falx may generate multiple visualizations that match the example but not necessarily the user intent. In such cases, analysts can interact with an exploration panel to inspect the synthesized visualizations and select the desired one. After that, analysts can further fine-tune details of the desired visualization through a post-processing panel.

[1] Demo available at https://falx.cs.washington.edu/

Falx's design has many potential advantages. First, users of Falx specify visualizations by mapping values to visual channels: this approach inherits the simplicity from grammars of graphics but provides more expressiveness since users can use the same examples to specify visualization ideas for inputs with different layouts. Second, Falx offloads the data transformation task to the program synthesizer so that users no longer need to conceptualize the expected data layout or transform the data. Finally, while program synthesizers by design can generate multiple results, users can effectively select and validate the desired visualization from synthesized candidates using the exploration panel in Falx. In general, rather than having to construct a visualization, data analysts demonstrate the task using examples and then select the desired visualization from a candidate pool, which shifts from the challenges of expression to the ease of recognition. With these designs, Falx aims to eliminate users' prerequisites in data transformation and enable data analysts to rapidly author visualizations.

We conducted a user study with 33 participants to test these design hypotheses, studying how users adapt to the new visualization process. Our results show that users of Falx, regardless of previous experience in visualization, can efficiently learn and solve challenging visualizations tasks that cannot be easily solved using the baseline tool ggplot2. However, we also discovered challenges that users face when using the tool and strategies they adopt to solve the problems. We believe these discoveries lead to future opportunities in adopting synthesized-based visualization tools in practice and unveil other potential designs that can further improve the usability of such tools.

2 USAGE SCENARIO

We first go through an example to illustrate the anticipated user experience in Falx (Section 2.2) compared to R (Section 2.1). In this example, a data analyst has the following dataset with New York and San Francisco temperature records from 2011-10-01 to 2012-09-30.

Date	New York	San Francisco
2011-10-01	63.4	62.7
2011-10-05	64.2	58.7
...	...	...
2012-09-25	63.2	53.3
2012-09-30	62.3	55.1

The analyst wants to create a visualization to compare the temperature in the two cities. First, the visualization should contain two lines to show temperature trends in the two cities; these two lines should be distinguished by color. Second, on top of the line chart, a bar chart should be layered on top to show the temperature difference between the two cities for each date. Each bar should

start from the New York temperature and end at the corresponding San Francisco temperature, and the color gradient of the bar should indicate the temperature difference between the two cities on that day. The desired visualization is shown in Figure 3.

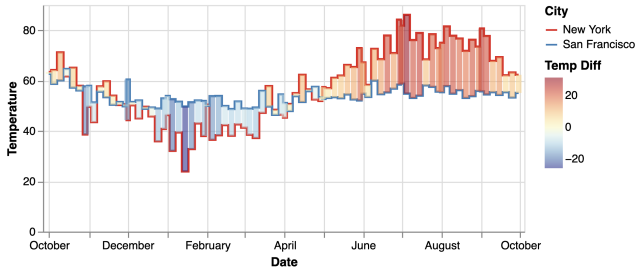


Figure 3: A visualization that compares New York and San Francisco temperatures between 2011-10-01 and 2012-09-30.

2.1 User Experience in R

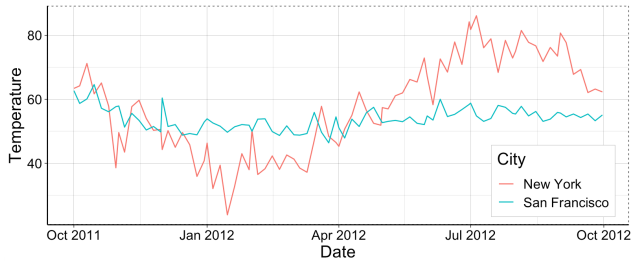
We first illustrate how a data analyst, Eunice, would create this visualization in R using tidyverse [51] and ggplot2 [50], two widely-used libraries for data transformation and data visualization.

After loading the data into a data frame in R, Eunice decides to first create the line chart that shows temperature trends of the two cities. To do so, Eunice chooses the function `geom_line` from the ggplot2 library. In order to create lines with different colors for different categories, Eunice needs to supply four data variables to the `geom_line` function – two variables for specifying x and y positions, one for colors of the line, and the last one for groups of lines (i.e., which points belong to the same line). Since the input data does not have these variables, Eunice needs to use the tidyverse library to transform the input data. To do so, Eunice first conceptualizes the desired data layout: the data should have 3 fields—date (for x-axis), temperature (for y-axis), and city name (for color and group). Eunice recalls a function `pivot_longer` in tidyverse, which supports pivoting the table from a "wide" to a "long" format by collecting column names and values in the column as key-value pairs in the body content. Specifically, Eunice writes the following code to transform the data, which yields the data on the right that matches Eunice's expectation.

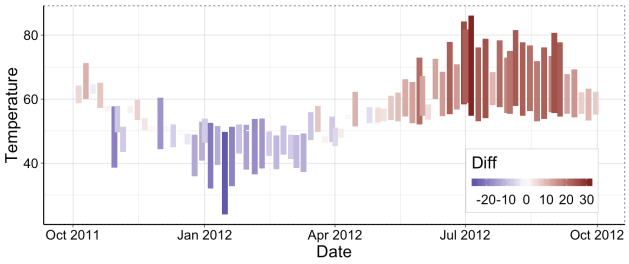
```
df1 <- pivot_longer(data = df,
         cols = ("New York", "San Francisco"),
         names_to = "City", values_to = "Temperature")
```

Date	City	Temperature
2011-10-01	New York	63.4
2011-10-01	San Francisco	62.7
...	...	...
2012-09-30	San Francisco	55.1

After data transformation, Eunice specifies the visualization using the following script. The script maps Date to x-axis , Temperature to y-axis, and City to both color and group. It generates the visualization in Figure 4a.



(a) A line chart that shows temperature trends.



(b) A bar chart that visualizes temperature difference.

Figure 4: Two visualizations created in R that compare New York and San Francisco temperatures.

```
plot1 <- ggplot(data = df1) +
         geom_line(aes(x = `Date`, y = `City`,
            color= `Temperature`, group = `Temperature`))
```

Eunice then proceeds to create bars on top of the first layer to visualize the temperature difference. Eunice first finds the function `geom_rect` from the library that supports floating bars. To visualize temperature difference, Eunice needs to specify positions of bars by mapping Date to x_{min} and x_{max} properties and mapping temperatures of the two cities to y_{min} and y_{max}; she also needs to map the temperature difference between the two cities to *color* to specify bar colors. Since the original data does not contain a column for temperature difference, Eunice uses the `mutate` function from tidyverse to transform the data. Using the following script, Eunice successfully creates the visualization in Figure 4b.

```
df2 <- mutate(df, Diff = `New York` - `San Francisco`)
plot2 <- ggplot(df2) +
         geom_rect(aes(xmin = `Date`, xmax = `Date`,
            ymin = `New York`, ymax = `San Francisco`,
            fill = `Diff`))
```

Finally, Eunice restructures the code to combine the two layers together using a concatenation operator. She also fine-tunes some parameters in ggplot2 to improve visualization aesthetics (e.g., modify titles of the axes and change line chart to a step chart), which generates the visualization that matches her design in Figure 3.

Since Eunice is an experienced data analyst, she manages to go through these data transformation and visualization step and eventually generates the desired visualization. However, a less experienced data analyst, Amelia, finds the visualization task challenging.

- First, Amelia is not familiar with the ggplot2 library, so she struggles in identifying the right functions to use. For example, it is difficult for her to distinguish between `geom_path` and

`geom_line`, and `geom_bar` or `geom_rect`. She is also unfamiliar with how to compose multi-layered visualizations.
- Second, due to her lack of experience with ggplot2, she finds it difficult to conceptualize the expected input layout because different functions and tasks require different data layouts.
- Finally, due to her lack of experience with tidyverse, she needs to spend significantly more time in finding the right operators and implementing the desired transformation.

2.2 User Experience in Falx

Now we show how Amelia, a less experienced data analyst, uses Falx (Figure 5) to create the same visualization.

First, Amelia uploads the input data to Falx's input panel (Figure 5-①) and examines the input data displayed in a tabular view. Amelia decides to first visualize temperature trends of the two cities using a line chart. Amelia goes to the demonstration panel to demonstrate how the first two data points of New York temperatures will be visualized. To do so, Amelia first clicks the "+" icon in the interface and select a line element (Figure 6-①), and Falx pops out an editor panel for Amelia to specify properties of this line element. Amelia clicks on values in the input table and copies the values to specify properties of the line element as follows (Figure 6-②):

- The line segment starts at the point with x_1 = 2011-10-01, $y_1 = 63.4$ (New York temperature on 2011-10-01)
- The line ends at x_2 = 2011-10-05, $y_2 = 64.2$ (New York temperature on 2011-10-05)
- The color of the line is labeled as "New York"

After saving the edits, Falx registers the example and provides a preview that visualizes the example line segment (Figure 6-③) for Amelia to examine. Using this example, Amelia conveys the following visualization idea to Falx: "I want a line chart over the input data that contains the demonstrated line segment". Amelia then presses the "Synthesize" button (in Figure 5-①) to ask Falx to find the desired line chart. Internally, Falx first infers the visualization specification and then runs a data transformation synthesizer to transform the input data to match the visualization specification. After approximately four seconds, Falx finds two visualizations that match the example and displays them in the bottom of the exploration panel (Figure 5-②). Both visualizations contain the example line segment specified by Amelia but they generalize the example differently: the first visualization only visualizes New York temperatures as demonstrated in the example, while the second generalizes the color dimension to other columns in the input data as well, resulting in a visualization that also contains San Francisco temperatures.

After briefly navigating both candidates in the carousel, Amelia finds the second visualization closer to the design in her mind, so she clicks the second visualization to enlarge it in the center view for a detailed check (Figure 5-② top). In the center view, Amelia hovers on the visualization to check details like values of different points in each line. After confirming the visualization matches her design, Amelia moves on to the second layer visualization, which should display temperature differences between the two cities using a series of bars.

Next, Amelia creates an example bar to demonstrate how the temperature difference between the two cities on 2011-01-01 should be visualized (Figure 7 left): the bar is positioned at date 2011-10-01, it starts at 62.7 (San Francisco temperature), ends at 63.4 (New York temperature), and its color shows the temperature difference of 0.7 for that day. Amelia runs the synthesizer to find visualizations that contain both the example line and the example bar. This time, after 9 seconds, Falx finds 8 candidate visualizations that match the examples (Figure 7 middle). To decide which visualization to pick, Amelia can either (1) add a second example bar to demonstrate the temperature difference of the two cities on another date to help Falx resolve the ambiguity, or (2) navigate candidates in the exploration panel to examine them. Amelia decides to use the second approach again. She first rules out some obviously incorrect visualizations (e.g., visualization 2 in Figure 7 middle), then compares similar visualizations, and finally selects the first visualization to check it in detail. After some examination, she decides it matches her design and proceeds to post-process the visualization.

The post processing panel (Figure 5-③) contains a GUI editor that allows Amelia to fine-tune visualization details and a program viewer for viewing and editing the synthesized program. Any changes made during the editing process are directly reflected on the center view panel (Figure 5-②) to provide immediate feedback. Using the post-processing panel, Amelia changes the line mark to step mark and modifies axis titles, which produces the visualization in Figure 7 right. Amelia is happy with this visualization and concludes the task. If Amelia wants to further customize the visualization (e.g., change color scheme, adjust bar spacing), she can directly edit the underlying Vega-Lite program.

In sum, Amelia creates the visualization by iterating through creating examples, exploring synthesized visualizations, and post processing. In this process, she benefits from the following design decisions behind Falx:

- First, while two visualization layers require different data transformations, Amelia does not need to worry about this, as the transformation task is delegated to the underlying synthesizer. In fact, even if the input data comes with a different layout, Amelia can still solve the problem with the same examples.
- Second, Amelia specifies examples by choosing from a small set of visualization marks and specifying mappings from concrete data values to properties. This allows her to create visualizations without programming in the visualization grammar.
- Third, instead of asking Amelia to read synthesized programs to disambiguate synthesis results, Falx provides an exploration interface that allows Amelia to explore and examine results in the visualization space.
- Finally, Falx adopts a scalable synthesis algorithm to explore the exponential number of possible ways to transform and visualize the input data. Each synthesis run takes between 3 and 20 seconds, which makes Amelia conformable at iterating between giving examples and exploring the generated visualizations.

3 SYSTEM ARCHITECTURE

In this section, we first provide a brief review of program synthesis and discuss the design and implementation of Falx, our end-to-end synthesis tool for automating data visualization tasks.

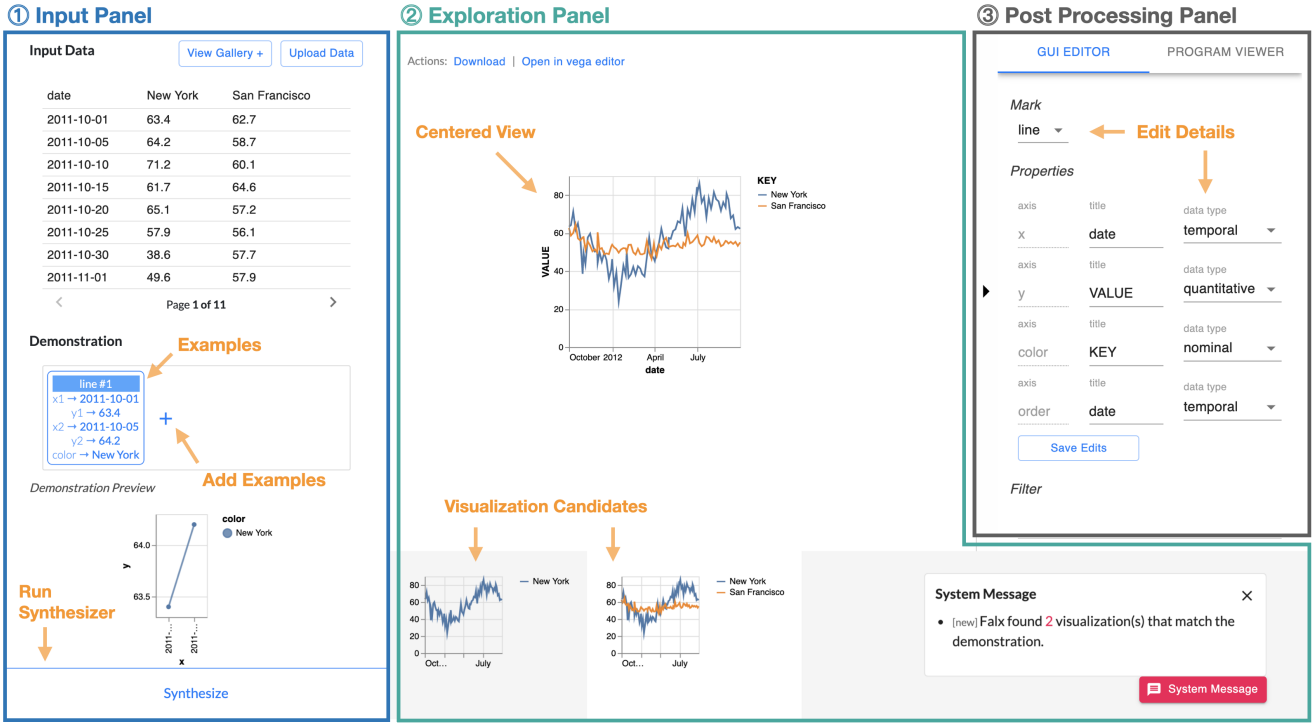


Figure 5: Falx interface has three panels: (1) Data analysts import data and create examples in the input panel. (2) Analysts explore and examine synthesized visualizations in the exploration panel. (3) Analysts edit visualization details in the post processing panel.

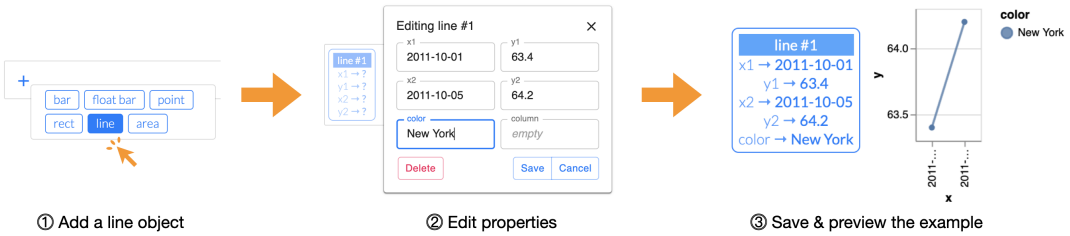


Figure 6: Amelia creates a line segment to demonstrate the visualization task.

3.1 Background: Program Synthesis

In recent years, many program synthesis algorithms have been developed to automate challenging or repetitive tasks for end users by automatically generating programs from high-level specifications (e.g., demonstrations, input-output examples, natural language descriptions). For instance, programming-by-example (PBE) is a branch of program synthesis that aims to synthesize programs that satisfy input-output examples provided by the user, such tools been used for string processing [11, 39], tabular data transformation [7, 46, 55], and program completion [12, 26, 32, 40, 41].

While there are different approaches to synthesize programs, one common method is to perform enumerative search over the space of programs by gradually expanding programs from a context-free grammar of some language [1, 8, 45, 55]. In general, these search techniques traverse the program space according to some cost metric and return the candidate programs that satisfy the user-provided specification. Here, the cost metric can be a model that measures simplicity of programs (e.g., based on number of expressions in the program) [8] or a statistical models that estimate likelihood of the program being correct [2, 32]. To speed up the synthesis process, several recent methods use deduction rules to prune incorrect *partial programs* early in the search process [7, 8]. For instance, Morpheus [7] uses predefined axioms of table operators to detect conflicts before the entire program is generated.

3.2 Falx Synthesizer

The architecture of Falx is shown in Figure 8. To use Falx, a data analyst first provides an input table and creates examples to demonstrate the visualization idea. Once the analyst hits the "synthesize" button, the Falx interface sends the input and examples to the Falx

Figure 7: Amelia's interaction with Falx to create the second layer visualization.

server. Given an input data and an example visualization (in the form of a set of geometric objects), Falx synthesizes pairs of candidate data transformation and visualization programs such that the resulting visualization contains all geometric objects in the visualization example.

To synthesize visualizations consistent with examples from the user, Falx spawns multiple solver threads to solve the synthesis problem in parallel. In each solver thread, Falx first runs a *visualization decompiler* (step 1) to decompile the example visualization into a visualization program and an example table, such that applying the program on the example table yields the example visualization provided by the user. Then, Falx calls the *data transformation synthesizer* (step 2) to infer programs that can transform the input data to a table that contains the example table generated in step 1. Finally, for each candidate data transformation result, Falx generates a *candidate visualization* (step 3) by combining the transformed data with the visualization program synthesized in step 1 and compiling them to Vega-Lite or R scripts for rendering. Synthesized visualizations from all threads are collected and displayed in Falx's exploration panel for the analyst to inspect. In what follows, we elaborate on the details of each step using the same running example in Section 2.

Step1: Visualization Decompilation. Internally, Falx represents visualizations as a simplified visualization grammar similar to ggplot2 and Vega-Lite. In this grammar, a visualization is defined by (1) graphical marks (line, bar, rectangle, point, area), (2) encodings that map data fields to visual channels (x, y, size, color, shape, column, row), and (3) layers, which specify how basic charts are combined into compositional charts. Since Falx only uses this grammar as an intermediate language to capture visualization semantics, visualization details (e.g., scale types) are intentionally omitted. Falx goes through the following three steps to decompile a visualization.

- Falx first infers visualization layers from the user example. In particular, Falx partitions examples provided by the user into groups based on their geometric types and properties, and creates one visualization layer for each group. Each layer corresponds to a simple chart of a particular type (e.g., scatter plot, line chart).
- Then, for each layer, Falx creates one basic visualization and an example table. The example table contains the same number of columns as the number of visual channels in this layer (derived from properties of geometric objects), and the visualization is specified as encodings that map columns in the example table to visual channels.
- Finally, for each example table, Falx fills the table with values from the example geometric objects.

Example 3.1. As shown in Figure 8-①, given the two visual elements provided by the user, Falx infers that the desired visualization should be a multi-layer chart that is composed by a line chart in layer 1 and a bar chart in layer 2 and decompiles the two layers independently. For example, for the second layer, Falx generates a bar chart program $\text{Bar}\{x \mapsto \text{C1}, y \mapsto \text{C2}, y_2 \mapsto \text{C3}, \textit{color} \mapsto \text{C4}\}$ with an example table $T = [(\textit{2011-10-01}, 62.7, 63.4, 0.7)]$ where T represents the desired *output table* that should be the result of the data transformation process. Column names C1, ..., C4 in the bar chart program correspond to names of the four columns in Table T.

Step 2: Data Transformation Synthesis. After decompiling the examples into the visualization program and example tables T, together with the original input table T_{in} provided by the user, Falx reduces the visualization synthesis task into a data transformation synthesis task [7, 46, 47]. For each example table T, the data transformation synthesizer aims to synthesize a transformation program P_t that can transform the input table into a table that contains the example table, i.e., $T \subseteq P_t(T_{\text{in}})$. Falx supports various types of transformation operators commonly used in the `tidyverse` library to handle different layouts of the input from the user (Figure 9).

The data transformation synthesizer uses an efficient algorithm to search for programs that are compositions of operators in Figure 9 satisfying the requirement $T \subseteq P_t(T_{\text{in}})$. Falx starts the search process by constructing sketches of transformation programs (i.e., programs whose arguments are not filled) and then iteratively expands the search tree and fills arguments in these partial programs. To maintain efficiency in this combinatorial search process, Falx uses deduction to prune infeasible partial programs as early as possible (as used in prior work [7, 46, 47]). The deduction engine analyzes properties of partial programs using abstract interpretation [5] and prunes programs whose analysis results are inconsistent with the

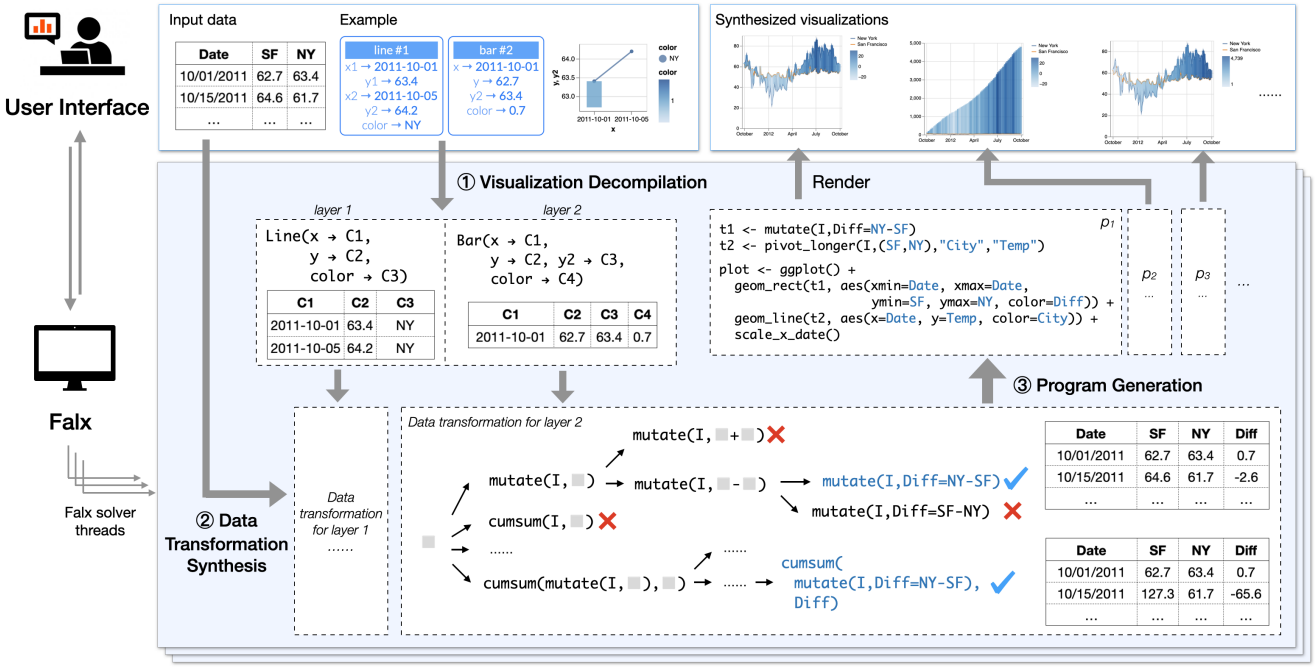


Figure 8: The architecture of the Falx system. Each solver thread synthesizes visualizations that match user examples in three steps: (1) visualization decompilation, (2) data transformation synthesis, and (3) program generation.

Type	Operator	Description
Reshaping	`pivot_longer`	Pivot data from wide to long format
	`pivot_wider`	Pivot data from long to wide format
Filtering	`select`	Project the table on selected columns
	`filter`	Filter table rows with a predicate
Aggregation	`group`	Partition the table into groups based on values in selected columns
	`summarise`	For every group, aggregate values in a column with an aggregator
	`cumsum`	Calculate cumulative sum on a column for each group
Computation	`mutate`	Arithmetic computation on selected columns
	`separate`	String split on a column
	`unite`	Combine two string columns into one with string concatenation

Figure 9: Data transformation operators supported in Falx. For clarity, we omit the parameters of each operator.

example output. Since each partial program corresponds to several dozens of concrete programs, the deduction engine can dramatically prune the search space.

When the search algorithm encounters a concrete program (i.e., with all arguments are filled) that is consistent with the example output, Falx adds the program to the candidate pool. The search procedure terminates either when the designated search space is exhaustively visited or when the given search time budget is reached. All synthesized program candidates are sent to the post-processor to generate visualizations.

Example 3.2. Figure 8-② shows the data transformation synthesis process for the second visualization layer (the bar chart) generated in step ①. Given the original input table I (with three columns Date, SF, and NY) the output table T (with four columns C1, C2, C3, and C4) generated in the last step, Falx aims to transform I into a table that contains the example table T. Starting from an empty program, Falx iteratively expands the unfilled arguments (represented as holes "□") in the partial programs to traverse the search space. When Falx encounters a partial program cumsum(I, □), Falx abstractly analyzes it and concludes that it is infeasible because cumsum cannot transform an input table with three columns into an output table with four columns. Falx the expands the feasible partial programs (e.g., mutate(I, □)) and collects concrete programs that are consistent with the objective (e.g., mutate(I, Diff = NY − SF)).

Optimization. We made several optimizations on top of existing synthesis algorithms [7, 47] to reduce Falx's time to respond. First, the major overhead in synthesis is the cost of analyzing partial programs using abstract interpretation, as it often requires running

expensive operators like aggregation and pivoting on big tables. To reduce this overhead, Falx memoizes abstract interpretation results for partial programs to allow reusing then whenever possible.

Second, instead of aiming to find only on or a few candidate programs that match user inputs like prior algorithms, Falx expects to find as many different programs as possible that satisfy the examples to ensure the correct visualization is included. To ensure diverse outputs, different Falx solver threads start with different initial program sketches to search for different portions of the search space in parallel. To improve responsiveness, Falx sets different timeouts for different threads to allow faster threads to respond to the user while other threads are searching for more complex transformations. In our implementation, we run 2 solver threads in parallel, we set one thread with 5 seconds timeout and another with 20 seconds timeout based on our perception of how long an analyst would be willing to wait as well as the typical time Falx takes to finish traversing different parts of the search space.

Step 3: Processing Synthesized Visualizations. As the final step in visualization synthesis, Falx generates visualizations by combining the visualization program generated in step 1 with table transformation programs generated in step 2.

Concretely, for each data transformation program, Falx applies the table transformation program on the input data to obtain a transformed output and unifies the output table schema with the schema in the visualization program, since the visualization program was filled with placeholder column names C1, C2, ..., etc. Falx then instantiates other visualization details (e.g., scale type, axis domain, etc.) omitted in the visualization grammar and compiles the visualization program into a Vega-Lite (or R) script through syntax-directed translation. For example, in Figure 8-③, Falx generates an R script that both transforms the input and specifies the visualization. Furthermore, Falx notices that the values on the x-axis are dates instead of strings, so it changes the x-axis scale to a temporal scale using the function "`scale_x_date()`".

After compilation, the post-processor removes semantically duplicate visualizations (i.e., visualizations with different specifications but with the same content and detail). Finally, Falx groups and ranks the visualizations based on the complexity of the programs (numbers of expressions). In this way, similar visualizations are grouped together to make comparison easier in the exploration process, and the complexity ranking allows users to explore visualizations constructed from easier transformation programs first before jumping into complex ones. These visualizations are sent to the user interface for rendering to allow user exploration.

4 USER STUDY

To understand Falx's benefits and limitations and to examine how analysts might adopt synthesis-based visualization tools, we conduct a between-subjects evaluation centered on the following questions:

- Does Falx improve user efficiency in creating visualizations compared to a baseline tool?
- How does Falx change the visualization authoring process for different data analysts?
- What strategies do data analysts use to visualize data in Falx?

4.1 Participants

We recruited two groups participants for the study: 16 participants (10 M, 5 F, 1 Unknown, Ages 23-51) for the Falx study, and another 17 participants (12 M, 4 F, Ages 19-60) for the baseline tool study (the R programming language). In the recruiting process, we screened participants by their ability to read a sample visualization. For the baseline group, we additionally required that all participants have experience with R (specifically ggplot2 and tidyverse libraries) for data visualization.

Participants reported their experience in data visualization authoring based on the number of visualizations they created in the past 6 months using any tools. For the Falx study group, there were 6 participants experienced with some visualization tools (created >10 visualizations), 8 with moderate experience with visualization tools (created 1-10 visualizations), and 2 participants with zero experience in creating visualizations in the past. For the baseline group, there were 8 experienced participants (create >10 visualizations) and 9 participants with moderate experience (created 1-10 visualizations).

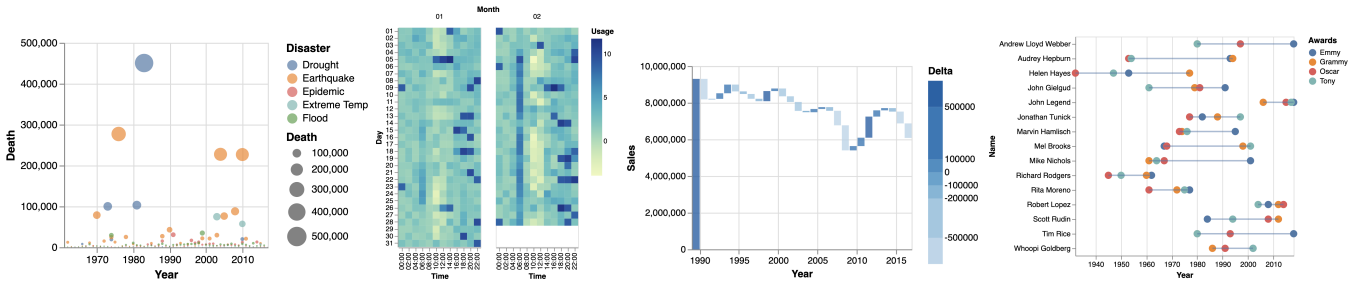
4.2 Procedure

Each participant was asked to complete four visualization tasks, where the Falx study group completed the task using Falx and the baseline group used R to complete the task. We chose R as the baseline tool due to its popularity among data analysts and its ability to support both data transformations and visualizations in the same context, where many other visualization tools requires users to process data and specify visualizations in different contexts.

To better examine the use of Falx, participants in the Falx group first completed a 20-minute tutorial together with a warm-up task with a sample solution (creating a grouped line chart to visualize sea ice level change in the past 20 years). After the tutorial, participants were asked to solve four visualization tasks. For R participants, we also provided the same warm-up task with a sample solution to allow users to get familiar with the environment and the data loading process, so that participants could focus on solving the visualization tasks. During the user study, participants were allowed to refer to any resource on the Internet including documentations and QA forums. We collected screen and audio recordings while participants completed tasks. We then interviewed them after all tasks were completed to reflect on their visualization process and strategies.

To conduct our user study, we developed four different visualization scenarios (Figure 10):

(a) *Disaster Impact*: A scatter plot that visualizes the number of people died from five disasters in the last century.
(b) *Electric Usage*: A faceted heat map for hourly electric usage in each day during the first two months of 2019.
(c) *Car Sales*: A waterfall chart for the number of cars sold in a year. Each bar starts at the sales value in the previous month and ends at the sales values in the month, and its color gradient reflects the increase/decrease compared to the last month.
(d) *Movie Awards*: A layered line/scatter plot for visualizing winners of all four prestigious movie awards. For each celebrity, there are four points showing years these awards were earned



(a) Disaster impact (b) Electric usage (c) Car sales (d) Movie awards winners

Figure 10: Study tasks.

and a line showing the time span for the celebrity to win all four awards.

For each visualization task, we provided as input a table that can be directly imported into the tools. We also explicitly described visualization designs to the participants in text so that participants could focus on implementation. Finally, we asked participants that they do not need to optimize the design — a task was considered correctly solved as long as the semantics of the visualization created by the participant matched the example solution regardless of the process and details. In this study, we did not restrict the time participants could spend on each task, but we provided users the option of quitting a task after spending more than 20 minutes without success. Thus, participants could complete each task with one of three outcomes: (1) submit a correct solution, (2) submit a wrong solution, or (3) give up after trying for at least 20 minutes.

We interviewed each participant after they finished all four tasks. For both Falx and baseline groups, we interviewed participants about (1) challenges they encountered while solving the tasks and their solutions, (2) common errors they made and how they fixed them, (3) their confidence about the solutions they submitted and what checks they performed to ensure correctness, and (4) what additional resources they used during the study and how they helped. We additionally asked participants in the Falx group to reflect on their visualization authoring process and interviewed them about (1) strategies adopted when creating examples to demonstrate the visualization task, (2) strategies adopted to explore the synthesized visualizations, and (3) their prior visualization experience and how Falx could potentially fit in their routine work.

The total session was less than 2 hours for all participants. To address learning effects or other carryover effects, we counterbalanced the tasks using a Latin square. We performed our analysis using mixed effect models, treating participants as a random effect and modeling tool, tasks, and experience level as fixed effects.

4.3 Task Completion

Figure 11 shows the percentage of participants that correctly finished each task. Falx participants generally had higher completion rates in all tasks. We observed a statistically significant difference in the completion rate in the car sales visualization ($p < 0.05$); others were not significant. Among nine failed tasks by Falx users, seven were due to incorrect solutions and, in two cases, participants quit the task after 20 minutes. Among 20 failed cases in the R study group, there were 9 incorrect solutions ans 11 cases where participants quit after 20 minutes.

Task	R ($N = 17$)		Falx ($N = 16$)	
	n	%	n	%
Disaster Impact	16	94.1%	14	87.5%
Electric Usage	13	75.6%	14	87.5%
Car Sales	5	29.4%	11	68.8%
Movie Awards	14	82.4%	16	100%

Figure 11: The number and percentage of participants correctly finished each study task.

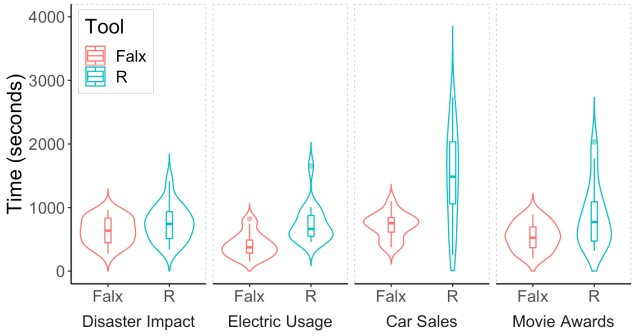


Figure 12: Violin plot showing the amount of time participants spent on each task for both Falx and R study groups.

Figure 12 shows task completion time in Falx. Using Wilcoxon rank sum test with Holm's sequential Bonferroni procedure for p value correction, we observed a significant improvement in user efficiency for car sales visualization ($t_{\text{Falx}} = 715 \pm 202s$, $t_{\text{R}} = 1473 \pm 743s$, $\mu_{\text{R}} - \mu_{\text{Falx}} = 758s$, $p < 0.01$) [2] and electric usage visualization ($t_{\text{Falx}} = 411 \pm 192s$, $t_{\text{R}} = 740 \pm 297s$, $\mu_{\text{R}} - \mu_{\text{Falx}} = 329s$, $p < 0.001$). While Falx participants were also generally faster in the other two tasks, there was no significant difference for the movie industry celebrity visualization ($t_{\text{Falx}} = 544 \pm 215s$, $t_{\text{R}} = 861 \pm 490s$, $\mu_{\text{R}} - \mu_{\text{Falx}} = 323s$, $p = 0.07$) or the disaster impact visualization ($t_{\text{Falx}} =$

[2] We use t_{Falx} and t_{R} to show the mean and standard deviation of time participants in Falx and R groups spent on each task. We use $\mu_{\text{R}} - \mu_{\text{Falx}}$ to represent the difference of the mean time between the two groups.

$638 \pm 209s$, $M_R = 754 \pm 279s$, $\mu_R - \mu_{Falx} = 116s$, $p = 0.23$). Participants from the R study group noted that the key reasons for failing on the car sales visualization task was the difficulty of finding the correct API (for waterfall chart) together with the complex transformation behind it (which required calculating a cumulative sum). Falx users also noted they found the car sales visualization difficult due to unfamiliarity with the visualization type. On the other hand, R users reported that the movie awards visualization and the disasters impact visualization were relatively easier since they expected the same pivot operator to transform the input, which is commonly encountered by R users, and the visualization types were relatively standard (line chart and scatter plot).

We found no significant interaction between user experience level (defined in Section 4.1) and task completion time ($p = 1$ for all tasks in both study groups using Wilcoxon rank sum test with Holm's sequential Bonferroni correction).

4.4 Task Experience

In this section, we describe qualitative feedback from participants in both groups about general (non-Falx related) visualization challenges both during the study and in their daily work, and how Falx can help with solving some of these challenges. We leave discussions of Falx-specific visualization challenges to Section 4.5.

As described in Section 4.2, we conducted a semi-structured interview for participants of both groups about visualization challenges they encountered both in the study and in their daily work, and how some of these challenges are typically overcome. To analyze this data, two of the researchers collaboratively conducted a qualitative inductive content analysis on the interviewer's notes, with a sensitizing concept of *visualization challenges and solutions*. In this process, two researchers independently labeled interview notes and then collaboratively discussed and compared high level labels to resolve disagreements in the initial codes.

4.4.1 Finding the right visualization function. The first challenge frequently mentioned by participants was discovering or recalling the correct visualization function. In the R study group, 14 out of 17 participants described this challenge, especially for the car sales task that most participants failed on. Some participants noted that the difficulty came from both finding the right term to search and distinguishing similar candidate functions. For example, participant R14 [3] noted that *"I wasn't aware that geom_rect() would be more helpful than geom_bar(). One thing that made it more challenging was the fact that this kind of bar chart has no proper name. I tried searching 'non-contiguous bar charts in R', but I didn't get many useful results."*. These challenges are also common in compositional charts: R10 noted *"creating the line with the dots is something I never did before so didn't know how to achieve it"*. To address these challenges, participants noted that online example galleries and forums are *"essential to their work"* (R1). Besides, two participants had *"an internal file – R code dictionary"* (R7) and *"a collection of some own code snippets"* (R1) to reduce search effort.

Falx group participants also described that they faced similar challenges of finding right functions in their daily work and Falx could help address them. For example, F1 mentioned: *"Falx can generate something that you cannot easily do. For example, the multi-layered visualization for the movie dataset would be very difficult to do in Excel or Google doc, you may need to specify some formula to specify relationship between two layers."* Participant F11 mentioned that Falx helped with complex tasks because *"It allows you to start by creating a relatively simple visualization in the beginning, which is good, then it allows you to build more complex stuff on top of it which is also helpful."*

4.4.2 Data transformation. Data transformation was another frequently mentioned challenge, including both conceptualizing the expected data layout and implementing the transformation. For example, R17 mentioned *"it [the car sales task] also seems to require some extra aggregation to get the starting and ending value for each rectangle to be drawn, which makes it even more difficult."* About implementation, R9 said that *"the vocabulary of the tidyverse is critical for trying to do what you want to do, otherwise it is all impossible to achieve."*, and R14 mentioned that *"I had an idea of what I needed to do, but I wasn't able to search the right things on Google to arrive at a useful code snippet for it."*

Participants from the Falx group mentioned similar issues in their work routine. For example, *"Tableau won't do data preparation and you need to manually put them together"* (F7), *"pivoting table is already something at an intermediate level in Tableau and many people cannot use it"* (F2). Due to lack of skill of preparing data programmatically, some participants would do it manually. For example, *"if I need to pivot data, I do it manually – e.g., just copy the data to a blank area [in Excel] and pivot it"* (F8). Participants appreciated that Falx automatically handled data transformations. Participant F5 mentioned *"I like the fact that it [Falx] solves the data transformation and visual encoding. I'm pretty familiar with visual encoding so it is fine when the data is in the right shape. But I find transforming data annoying."* Participant F15 mentioned *"I didn't think about data format at all in the process"*. F7 mentioned *"Tableau won't do data preparation because you need to manually put them together and drag drop them for you. Falx is pretty automated on this."*

4.4.3 Learning to create expressive visualizations. Due to the inherent challenge in visualization and data transformation in these tools, participants mentioned many of existing tools had a learning barrier for new users. For example, F4 mentioned that *"the learning curve is pretty steep (Tableau), and we spent a lot of time learning these tools"*.

On the other hand, while Falx was a new visualization tool, most users found it easy to learn, despite some users requiring some time in the beginning to get used to *"the paradigm shift from my normal understanding"* (F6). For example, participant F4 mentioned that *"the ramp up time [for Falx] is pretty short and it's pretty easy to use."*, and F6 mentioned that *"anyone with basic Excel knowledge should be able to use Falx"*.

4.5 Visualization Strategies in Falx

Since Falx is a new tool for data visualization, besides understanding its ability to address existing visualization challenges, we also investigated how participants used Falx to solve visualization tasks. We conducted an inductive content analysis on the interviewer's

[3] We use R1-R17 to denote participants from the R study group and F1-F16 to denote participants from the Falx group.

notes about Falx experience similar to that in Section 4.4. In this section, we discuss observations about participants' visualization process in Falx and their indications for future synthesizer-based visualization tool design.

4.5.1 Strategies for creating examples. Data analysts initiate interactions with Falx by creating examples. As a synthesis-powered visualization tool, poorly constructed examples can be highly ambiguous and lead to long running time and a large number of visualization candidates. Also, while users can carefully create multiple examples to increase Falx's performance, it requires more effort. Falx users identified the following strategies to create examples effectively:

- *Sketching visualizations before demonstration:* Three participants mentioned that sketching the visualization design on paper helped them understand geometry of the visualization, and it helped them creating better examples. For example, participant F13 mentioned *"I sketch out first to get a general understanding of what the visualization would look like, and then use that to drop points."*.
- *Selecting representative data points to demonstrate:* Seven participants mentioned that they considered using *"representative points"* (F7) when creating demonstrations in order to reduce ambiguity to Falx. For example, participant F1 mentioned that *"[In the disaster impact task], I chose a cause that contains non-zero value in that year, because it's a unique value that can avoid confusion of the tool"*.
- *Start from a few examples, add more later if necessary:* Eight participants mentioned that they *"tried to shoot for minimum input"* (F6) for simplicity. In this way, they can *"run the tool to see what it returns"* (F1) before spending more effort on examples, and they would "add more to help narrow it down if there are many visualizations pop up" (F9). Additionally, participant F11 noted that *"It's easy to add multiple elements to mess up with the demonstration. A small number of elements make it easier to go back and fix"*.
- *Start with multiple examples to minimize interaction iterations:* Instead of starting from minimal inputs, 6 participants preferred to create more examples in the beginning to *"avoid ambiguity"* (F2). They remarked that *"it doesn't take that much time to add data points"* (P8) and multiple examples can *"avoid having to wait and choosing from multiple solutions"* (F8).

During the process of creating and revising examples, seven participants found the demo preview panel useful since it allowed them to *" understand more about how a certain layout would look like"* (F11) and it *"helps put me on the right track of solving the task."* (F13). However, nine participants said they did not find it helpful because they *"don't know if it tells enough to help understand anything [about synthesis results]"* (F7); they preferred to *"just click synthesis to get the result since synthesis is pretty fast"* (F14).

Some challenges participants encountered in creating examples included (1) unfamiliarity with terms in Falx (e.g., F4 mentioned "*'size' is a term that I'm not familiar with.*") and (2) not getting used to demonstrate visualization ideas using values (e.g., F6 mentioned *''I was struggling with the paradigm shift about when to use values and when to use table headers"*). In general, the fast response time of Falx enabled participants to get over these challenges through trial and error (e.g., F1 mentioned *"If there is anything wrong, I'll go back and do edits on the points."*), and they *"get faster in later tasks once understand the difference"* (F6). In future, Falx could adopt a mixed-initiative interface [17] to improve experience for new users. In addition, we observed that many participants felt like they were interacting with an intelligent tool (e.g., F13 mentioned *"the tool is quite good at learning from what I demonstrated"*) and they were willing to provide more informative inputs (e.g., F16 *"tried to write the expression because I don't know how Falx would do computation"*). In future, Falx could take advantage of this to support more complex visualization tasks by synthesizing programs from users more informative inputs besides examples (e.g., formulas that describe how certain values in the examples are derived from the input).

4.5.2 Strategies for exploring synthesis results. After creating examples to demonstrate the visualization task, users interact with Falx to explore the synthesized visualizations and identify the desired solution. Prior work [22, 27] has shown that a main barrier for adoption of synthesis-based programming tools is that users have difficulty understanding and trusting synthesized solutions, especially when there are many solutions consistent with the user demonstration.

We discovered from the interview that many participants shared the following similar 4-step process to select the desired visualization from synthesized visualizations by investigating visualization from coarse to fine:

- *Step 1: Check against the high-level picture.* First, participants noted that it was easy to quickly exclude many visualizations that are obviously far from the desired visualization. For example, *"having too many options is a bit overwhelming, but just keeping in mind what the result you look like can help narrow down the solution"* (F11).
- *Step 2: Check axes and invariants.* After excluding the obviously wrong solutions, participants often investigate domains and ranges of each axis to further refine synthesis results. For example, *"I first looked at color labels, I noticed they tend to be wrong in wrong visualizations – e.g., some charts only contain 2 labels instead of 4"* (F16).
- *Step 3: Compare similar visualizations.* Then, participants investigated similar visualizations to find their difference. For example, *"In the electric case, there is one mistake [in a candidate visualization] with 2019 showing up on y axis, it's small and not obvious. But then, I was able to tell the difference by comparing the two visualizations directly, and notice that year showed up in the 'hour' field"* (F2).
- *Step 4: Inspect visualization detail.* Finally, participants *"check carefully about the values to make sure they are correct"* (F5). An example of such detailed checking is to check values in the chart against known values in the input data: *"if there is a specific value that I know is correct – for example, in the last example (disasters), I knew the total death for 1961 was, then I hover over the output to check if the value is correct "* (F6).

After these steps, participants were confident about the result. In fact, while participants mentioned that their confidence about solutions could be negatively affected by unfamiliarity of visualization

types (e.g., F9 mentioned *"I don't do much heatmap so I'm less confident"*), they mentioned that the checking process can raise their confidence about the chosen solution. For example, participants got more confident after *"comparing them [candidates] with my sketch"* (F6), *"looking at solutions and finding their difference"* (F14), or *"checking details"* (F2). They further noted that in many cases, *"it's almost impossible for Falx to get it wrong because these values are all pretty unique"* (F14). In general, participants found the exploration panel *"quite useful"* because it *"allows to choose the best visualization out of that"* (F7).

In sum, Falx's exploration panel allowed users to directly inspect solutions in the visualization space following a coarse-to-fine process, which helped them to disambiguate solutions and trust the chosen results. In the future, Falx's interface could be improved to augment users' exploring strategies. For example, Falx could directly summarize the differences among the synthesized visualizations to allow users to make comparisons easier. Also, Falx's center view panel could support displaying traces that show how properties of each geometric object are derived from the input, which could make the synthesis process more transparent and make checking details easier.

4.6 Workflow Implications

Finally, participants reflected on how Falx might fit into their workflow. For example, F13 mentioned *"I'll absolutely use this if this is a product. Even as it is now I'll use it"*. Participants found several scenarios that Falx can be helpful.

- Create visualizations for discussions and presentations. For example, F1 noted that *"visualizations generated by Falx can meet standards of presentation slides"* and *"Falx can generate something that you cannot easily do in Excel"*.
- Prototyping complex analysis. For example, F16 mentioned *"Falx is very useful in the prototyping stage because it's very fast to use."* F7 further noted that they can *"take a sample to visualize and then extend to the full visualization"* using Falx for analyzing big datasets.
- Benefit non-experienced users. Six participants mentioned that Falx can be *"more beneficial to new users that cannot create charts"* (F2). Also, Falx can be *"a good teaching tool to help people understand data"* (F7).
- Reduce team collaboration effort. Participant F11 described that visualization readers were often different from visualization creators in their team, and modifying visualization required team efforts. F11 mentioned that Falx could help with it: *"a person presents me with a visualization, but I want to view something differently. Instead of getting back to the person to re-do it, I can probably just use Falx to do it, which would be more efficient"*.

However, several participants also mentioned Falx may not fit well to their current workflow when they need *"very high standard visualizations"* (F1) that requires extensive customization. Another limitation of the current version of Falx is the lack of *"deep integration with other tools"* (F1), e.g., database for handling big datasets and data cleaning tools for *"handling null / dirty data"* (F4). But in general, participants thought that Falx would be helpful when used in the right scenarios and *"would be pretty interesting to try Falx in some of these tasks"* (F5).

5 RELATED WORK

Falx builds on top of prior research on grammar based visualization tools, data transformation tools, program synthesis algorithms and automated visualization design systems.

Grammar-based Visualization. Following the initial publication of the Grammar of Graphics [52], high level grammars [37, 42, 50] for data visualizations have grown increasingly popular as a way of succinctly specifying visualization designs. In contrast to low level visualization languages like Protovis [4], D3 [13], and Vega [38] that are designed for creating highly-customizable explanatory visualizations, these high level grammars aim to enable analysts to rapidly construct expressive graphics in exploratory analysis. For example, ggplot2 [49, 50] and Vega-Lite [37] are two visualization grammars that allow users to specify visualizations using visual encodings. In both tools, low level visualization details are handled by default parameters unless users want customization. Tableau [42] adopts a graphical interface approach to enable users to rapidly create views to explore multidimensional database. In Tableau, users drag-and-drop data variables onto visual encoding "shelves", which are later translated into a high-level grammar similar to ggplot2. These tools expect the input data layout to match the design such that (1) each row corresponds to a graphical object, and (2) each column can be mapped to a visual channel. In practice, the mismatch between the design and the input data layout is common, which raises a barrier for creating visualizations [9, 53].

Falx formalizes visualizations in the same way, and synthesized programs are compiled to ggplot2 or Vega-Lite for rendering. Falx's user interface also inherits the expressiveness and simplicity of Grammar of Graphics design, by allowing users to create examples of visual encodings to demonstrate visualization ideas. The main difference is that Falx relaxes the constraints on input data layout and allows users to use layout-independent examples to demonstrate visualization ideas. Falx then automatically infers the visualization spec and synthesizes data transformations to match the data with the design from the examples, which saves users' construction efforts.

Data Transformation Tools. The need to prepare data for statistical analysis and visualization has led to the development of many tools for data transformation [6, 17, 31, 51]. Since different analysis objective requires different layout, users need to frequently transform data throughout the analysis process [16, 51, 53]. Potter's Wheel [31] is a graphical interface that allows users to interactively choose transformation operators and inspect transformation outputs. Wrangler [17] is a mixed initiative data transformation tool which can suggest transformations based on the input data. Tidyverse [51] is a data transformation library in R, which allows users to interleave data transformation code, analysis code and visualization code in the same environment to reduce the effort of context switch. Several synthesis-powered data transformation tools [3, 6, 7, 30, 46] have been proposed to help automate data transformation. For example, Prose [30] includes several programming-by-example tools that automatically synthesize programs for data

cleaning and transformation from input-output examples. Morpheus [7] and Scythe [46] are two specialized data transformation synthesizer with better scalability and expressiveness.

Falx inherits the transformation language design in tidyverse [51], and Falx is a realization of prior program synthesis algorithms [7, 47] as an interactive system for visualization authoring. Falx's main difference from automated data transformation tools is the unification of the visualization task and transformation tasks. In this way, Falx users do not need to conceptualize expected data layout or frequently switch between visualization and data transformation tools. The unification also enables Falx users to easily explore synthesis results in the visualization space as opposed to program space, which is considered challenging [27]. Besides data layout transformation, many data preparation tools also support data cleaning (e.g., handling missing data or invalid data) [48], data normalization (collecting non-relational data into relation format) [3], and string formatting [6, 11, 56]. Falx currently does not support directly visualizing dirty or non-relational data. In the future, Falx could work with these tools to further automate visualization process.

Visualization Automation. Automated visualization tools [15, 29, 35] have been proposed to help data analysts to explore the visualization design space. Draco [29] and Dziban [24] use constraint logic approaches to model design knowledge, and they can recommend visualization designs from partial specifications. VizNet [15] uses a deep neural network trained from visualization corpus to suggest designs. Voyager [54] combines recommendation and exploration for mixed-initiative design exploration. VisExemplar [35] allows users to demonstrate changes in the visualization layout to explore alternative visualizations designs. Falx is complementary to these design automation tools. Falx allows users to implement visualization designs they have in mind without data layout constraints, while design automation tools helps users to explore visualization designs from a fixed data layout. A combination of the two approaches could potentially help users to explore a larger visualizations design space without data layout constraints.

User Interaction with Program Synthesizers. In general, program synthesizers can be categorized into exploration tools and implementation tools. Synthesis-based exploration tools aim to generate a large number of solutions from users' weak constraints to aid users to explore the search space [29, 43]. For example, Scout [43] is a synthesis-based exploration tool to discover mobile layout ideas. In these tools, users interact with an exploration interface to navigate and save interesting solutions. Implementation tools [6, 7, 11, 30, 46, 56], instead, aim to synthesize programs to help solve a concrete task (e.g., implement a design that a user already have in mind). In these tools, the main interaction objective is to help users to disambiguate spurious programs that happen to be consistent with the user specification but are incorrect for the full task [27]. To solve this challenge, Wrex [6] generates readable programs for users to inspect and edit; Regae [56] and FlashProg [27] interactively ask users disambiguating questions to refine synthesis results; PUMICE [23] lets users collaborate with the agent to recursively resolve any ambiguities or vagueness through conversations and demonstrations.

Falx is an implementation tool for data visualization. Falx's contribution to the user interaction model is that Falx brings the exploration design (from exploration tools) to address the disambiguation and trust challenges in implementation tools. Allowing users to explore and examine synthesized programs in the visualization space reduces the barrier for user interaction (e.g., users do not need to be familiar with underlying programs to disambiguate [27]) and increases users' confidence about solutions.

Expressive Visualization Design Tools. Besides tools for standard visualization authoring, many visualization tools have been proposed to let designers create more expressive visualizations. Examples of these tools are Data illustrator [25], Lyra [36], Charticular [33], Data-driven Guides [20], and StructGraphics [44]. Besides high-level design layout (e.g., x,y ,column) and standard mark properties (e.g., color, shape), these tools let users customize marks to create more expressive glyphs (e.g., compound marks, parametric marks). These tools expect users to prepare data into a tidy format to start with, but they support rich visualization designs. Falx, in comparison, supports standard visualization designs but automates data transformation.

Several design reconstruction tools (e.g., VbD [35], Liger [34], iVolVER [28]) are proposed to let designers create expressive visualization by destructing and reconstructing existing visualization designs. Using these tools, users can transform existing visualizations to new ones by demonstrating desired design changes. Functionally, these tools are design exploration tools that take as input a visualization design and produce a new visualization design. They differ from Falx because Falx takes data as input and maps it to a visualization design for initial design authoring.

There are opportunities to combine Falx with these tools for better visualization authoring. Falx can work with expressive designs tools to support authoring complex visualizations from non-tidy data: users can first design customized marks using example data values, and the tool would automatically synthesize binding between data and these fine-grained mark properties from these examples. Falx can also work with design reconstruction tools to allow users to first use Falx to create initial design from data, and then subsequently interactively explore new designs by transforming the initial design.

6 DISCUSSION

We have presented Falx, a novel synthesis-powered visualization authoring tool that allows users to demonstrate a visualization design using examples of visual encodings and then receive suggestions for visualization designs. Our goal was to create a system that does not require users to manually specify the visualization or worry about data transformations, thereby improving user efficiency and reducing the learning burden on novice analysts. Our study found that Falx often achieved these goals: Falx users were able to effectively adopt Falx to solve visualization tasks that they could otherwise cannot solve, and in some cases, they do so more quickly. We next discuss some implications of this work in guiding future research.

Data Layout-Flexible Visualization Exploration. Besides visualization authoring, combining Falx with data exploration tools like

GraphScape [21] Voyager [54], GraphScape [21], VbD [35] and Dziban [24] might enable new design exploration tools that allow users to discover both new relations from the dataset and new designs to visualize the them. Using existing design exploration tools, users can explore diverse visualization designs from an input data; but since existing tools generates designs that are specific to the input data layout, the design space that can be explored is limited. Integrating Falx with these design exploration tools could enable novel design exploration tools that can assist users to explore design space without being constrained by data layouts. For example, in an anchored design exploration scenario [21, 24, 35], users can demonstrate data layout changes alongside design changes using this new tool to incrementally discover data insights from a larger design space. Similarly, Falx might also work with visualization recommendation engines [15, 29] to find better designs for the dataset based on initial visualizations created by users using examples to suggest data layout independent designs.

Visualization Learning. As we discovered from our study, users often describe existing programming tools as "flexible, powerful" but "having a steep learning curve." Falx can fill in this gap by helping data analysts to learn to create visualizations. Since Falx does not require its users to have programming expertise, new users can learn visualization and data transformation concepts using Falx by first creating visualization using demonstrations and then inspecting synthesized programs. For example, Falx could generate readable code like Wrex [6] for users to learn to use visualization APIs, enabling them to access the flexibility and power of code.

Bootstrapping Complex Data Analysis. Falx currently focuses on inexperienced data analysts, but it could also potentially benefit experienced data analysts by bootstrapping complex data analysis tasks. For example, data analysts could first create visualizations in Falx and then build complex analyses by iteratively editing synthesized programs. To achieve this goal, Falx needs more transparency and better integration with programming environments. For example, Falx could expose synthesized programs during the synthesis process and allow users to steer the synthesis process to better disambiguate results. Falx could also be integrated into programming environments like mage [19], Wrex [6] or Sketch-n-Sketch [14] to make program editing easier.

All of these possibilities, as well as prior work applying program synthesis to design (e.g., [29, 43]), suggest a promising future for augmenting design work with synthesis-based techniques. We hope Falx provides one exemplar for how to adapt core techniques in synthesis into powerful interactive tools that empower human creativity.

7 ACKNOWLEDGEMENT

This work has been supported in part by the NSF Grants ACI OAC–1535191, FMitF CCF-1918027, OIA-1936731, IIS-1546083, IIS-1955488, IIS-2027575, CCF-1723352, the Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA NSF CCF-1723352), Department of Energy award DE-SC0016260, the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA CMU 1042741-394324 AM01, a grant from DARPA, FA8750–16–2–0032, as well as gifts from Adobe, Facebook, Google, Intel, VMWare and Qualcomm. We would also like to thank anonymous reviewers for their insightful feedback on paper revision.

REFERENCES

[1] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 934–950. https://doi.org/10.1007/978-3-642-39799-8_67

[2] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=ByldLrqlx

[3] Daniel W Barowy, Sumit Gulwani, Ted Hart, and Benjamin Zorn. 2015. FlashRelate: extracting relational data from semi-structured spreadsheets using examples. *ACM SIGPLAN Notices* 50, 6 (2015), 218–228.

[4] Michael Bostock and Jeffrey Heer. 2009. Protovis: A graphical toolkit for visualization. *IEEE transactions on visualization and computer graphics* 15, 6 (2009), 1121–1128.

[5] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. https://doi.org/10.1145/512950.512973

[6] Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. 2020. Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists. In *CHI '20: CHI Conference on Human Factors in Computing Systems, Honolulu, HI, USA, April 25-30, 2020*, Regina Bernhaupt, Florian 'Floyd' Mueller, David Verweij, Josh Andres, Joanna McGrenere, Andy Cockburn, Ignacio Avellino, Alix Goguey, Pernille Bjøn, Shengdong Zhao, Briane Paul Samson, and Rafal Kocielnik (Eds.). ACM, 1–12. https://doi.org/10.1145/3313831.3376442

[7] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proc. Conference on Programming Language Design and Implementation*. ACM, 422–436.

[8] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-output Examples. In *Proc. Conference on Programming Language Design and Implementation*. ACM, 229–239.

[9] Malu AC Gatto. 2015. Making research useful: Current challenges and good practices in data visualisation. (2015).

[10] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 317–330. https://doi.org/10.1145/1926385.1926423

[11] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proc. Symposium on Principles of Programming Languages*. ACM, 317–330.

[12] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete completion using types and weights. In *Proc. Conference on Programming Language Design and Implementation*. ACM, 27–38.

[13] Jeffrey Heer and Michael Bostock. 2010. Declarative Language Design for Interactive Visualization. *IEEE Trans. Vis. Comput. Graph.* 16, 6 (2010), 1149–1156. https://doi.org/10.1109/TVCG.2010.144

[14] Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology, UIST 2019, New Orleans, LA, USA, October 20-23, 2019*, François Guimbretière, Michael Bernstein, and Katharina Reinecke (Eds.). ACM, 281–292. https://doi.org/10.1145/3332165.3347925

[15] Kevin Zeng Hu, Snehalkumar (Neil) S. Gaikwad, Madelon Hulsebos, Michiel A. Bakker, Emanuel Zgraggen, César A. Hidalgo, Tim Kraska, Guoliang Li, Arvind Satyanarayan, and Çagatay Demiralp. 2019. VizNet: Towards A Large-Scale Visualization Learning and Benchmarking Repository. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, CHI 2019, Glasgow, Scotland, UK, May 04-09, 2019*, Stephen A. Brewster, Geraldine Fitzpatrick, Anna L. Cox, and Vassilis Kostakos (Eds.). ACM, 662. https://doi.org/10.1145/3290605.3300892

[16] Sean Kandel, Jeffrey Heer, Catherine Plaisant, Jessie Kennedy, Frank Van Ham, Nathalie Henry Riche, Chris Weaver, Bongshin Lee, Dominique Brodbeck, and Paolo Buono. 2011. Research directions in data wrangling: Visualizations and transformations for usable and credible data. *Information Visualization* 10, 4 (2011), 271–288.

[17] Sean Kandel, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. 2011. Wrangler: interactive visual specification of data transformation scripts. In *Proceedings of the International Conference on Human Factors in Computing Systems, CHI 2011, Vancouver, BC, Canada, May 7-12, 2011*, Desney S. Tan, Saleema Amershi, Bo Begole, Wendy A. Kellogg, and Manas Tungare (Eds.). ACM, 3363–3372. https://doi.org/10.1145/1978942.1979444

[18] Sean Kandel, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. 2012. Enterprise Data Analysis and Visualization: An Interview Study. *IEEE Trans. Vis. Comput. Graph.* 18, 12, 2917–2926. https://doi.org/10.1109/TVCG.2012.219

[19] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. 2020. mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks. In *UIST '20: The 33rd Annual ACM Symposium on User Interface Software and Technology, Virtual Event, USA, October 20-23, 2020*, Shamsi T. Iqbal, Karon E. MacLean, Fanny Chevalier, and Stefanie Mueller (Eds.). ACM, 140–151. https://doi.org/10.1145/3379337.3415842

[20] Nam Wook Kim, Eston Schweickart, Zhicheng Liu, Mira Dontcheva, Wilmot Li, Jovan Popovic, and Hanspeter Pfister. 2017. Data-Driven Guides: Supporting Expressive Design for Information Graphics. *IEEE Trans. Vis. Comput. Graph.* 23, 1 (2017), 491–500. https://doi.org/10.1109/TVCG.2016.2598620

[21] Younghoon Kim, Kanit Wongsuphasawat, Jessica Hullman, and Jeffrey Heer. 2017. GraphScape: A Model for Automated Reasoning about Visualization Similarity and Sequencing. In *ACM Human Factors in Computing Systems (CHI)*. http://idl.cs.washington.edu/papers/graphscape

[22] Tessa Lau. 2009. Why Programming-By-Demonstration Systems Fail: Lessons Learned for Usable AI. *AI Mag.* 30, 4 (2009), 65–67. https://doi.org/10.1609/aimag.v30i4.2262

[23] Toby Jia-Jun Li, Marissa Radensky, Justin Jia, Kirielle Singarajah, Tom M. Mitchell, and Brad A. Myers. 2019. PUMICE: A Multi-Modal Agent that Learns Concepts and Conditionals from Natural Language and Demonstrations. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology, UIST 2019, New Orleans, LA, USA, October 20-23, 2019*, François Guimbretière, Michael Bernstein, and Katharina Reinecke (Eds.). ACM, 577–589. https://doi.org/10.1145/3332165.3347899

[24] Halden Lin, Dominik Moritz, and Jeffrey Heer. 2020. Dziban: Balancing Agency & Automation in Visualization Design via Anchored Recommendations. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–12.

[25] Zhicheng Liu, John Thompson, Alan Wilson, Mira Dontcheva, James Delorey, Sam Grigg, Bernard Kerr, and John T. Stasko. 2018. Data Illustrator: Augmenting Vector Design Tools with Lazy Data Binding for Expressive Visualization Authoring. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, CHI 2018, Montreal, QC, Canada, April 21-26, 2018*, Regan L. Mandryk, Mark Hancock, Mark Perry, and Anna L. Cox (Eds.). ACM, 123. https://doi.org/10.1145/3173574.3173697

[26] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid mining: helping to navigate the API jungle. In *Proc. Conference on Programming Language Design and Implementation*. ACM, 48–61.

[27] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin G. Zorn, and Sumit Gulwani. 2015. User Interaction Models for Disambiguation in Programming by Example. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology, UIST 2015, Charlotte, NC, USA, November 8-11, 2015*, Celine Latulipe, Bjoern Hartmann, and Tovi Grossman (Eds.). ACM, 291–301. https://doi.org/10.1145/2807442.2807459

[28] Gonzalo Gabriel Méndez, Miguel A. Nacenta, and Sebastien Vandenheste. 2016. iVoLVER: Interactive Visual Language for Visualization Extraction and Reconstruction. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems, San Jose, CA, USA, May 7-12, 2016*, Jofish Kaye, Allison Druin, Cliff Lampe, Dan Morris, and Juan Pablo Hourcade (Eds.). ACM, 4073–4085. https://doi.org/10.1145/2858036.2858435

[29] Dominik Moritz, Chenglong Wang, Greg L. Nelson, Halden Lin, Adam M. Smith, Bill Howe, and Jeffrey Heer. 2019. Formalizing Visualization Design Knowledge as Constraints: Actionable and Extensible Models in Draco. *IEEE Trans. Vis. Comput. Graph.* 25, 1 (2019), 438–448. https://doi.org/10.1109/TVCG.2018.2865240

[30] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: a framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 107–126. https://doi.org/10.1145/2814270.2814310

[31] Vijayshankar Raman and Joseph M Hellerstein. 2001. Potter's wheel: An interactive data cleaning system. In *VLDB*, Vol. 1. 381–390.

[32] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proc. Conference on Programming Language Design and Implementation*. ACM, 419–428.

[33] Donghao Ren, Bongshin Lee, and Matthew Brehmer. 2019. Charticulator: Interactive Construction of Bespoke Chart Layouts. *IEEE Trans. Vis. Comput. Graph.* 25, 1 (2019), 789–799. https://doi.org/10.1109/TVCG.2018.2865158

[34] Bahador Saket, Lei Jiang, Charles Perin, and Alex Endert. 2019. Liger: Combining Interaction Paradigms for Visual Analysis. *CoRR* abs/1907.08345 (2019). arXiv:1907.08345 http://arxiv.org/abs/1907.08345

[35] Bahador Saket, Hannah Kim, Eli T Brown, and Alex Endert. 2016. Visualization by demonstration: An interaction paradigm for visual data exploration. *IEEE transactions on visualization and computer graphics* 23, 1 (2016), 331–340.

[36] Arvind Satyanarayan and Jeffrey Heer. 2014. Lyra: An Interactive Visualization Design Environment. *Comput. Graph. Forum* 33, 3 (2014), 351–360. https://doi.org/10.1111/cgf.12391

[37] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Trans. Vis. Comput. Graph.* 23, 1 (2017), 341–350. https://doi.org/10.1109/TVCG.2016.2599030

[38] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. 2016. Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization. *IEEE Trans. Vis. Comput. Graph.* 22, 1 (2016), 659–668. https://doi.org/10.1109/TVCG.2015.2467091

[39] Rishabh Singh and Sumit Gulwani. 2016. Transforming spreadsheet data types using examples. In *Proc. Symposium on Principles of Programming Languages*. ACM, 343–356.

[40] Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. 2005. Programming by sketching for bit-streaming programs. In *Proc. Conference on Programming Language Design and Implementation*. ACM, 281–294.

[41] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 404–415.

[42] Chris Stolte, Diane Tang, and Pat Hanrahan. 2002. Query, analysis, and visualization of hierarchically structured data using Polaris. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, July 23-26, 2002, Edmonton, Alberta, Canada*. ACM, 112–122. https://doi.org/10.1145/775047.775064

[43] Amanda Swearngin, Chenglong Wang, Alannah Oleson, James Fogarty, and Amy J. Ko. 2020. Scout: Rapid Exploration of Interface Layout Alternatives through High-Level Design Constraints. In *CHI '20: CHI Conference on Human Factors in Computing Systems, Honolulu, HI, USA, April 25-30, 2020*, Regina Bernhaupt, Florian 'Floyd' Mueller, David Verweij, Josh Andres, Joanna McGrenere, Andy Cockburn, Ignacio Avellino, Alix Goguey, Pernille Bjøn, Shengdong Zhao, Briane Paul Samson, and Rafal Kocielnik (Eds.). ACM, 1–13. https://doi.org/10.1145/3313831.3376593

[44] Theophanis Tsandilas. 2020. StructGraphics: Flexible Visualization Design through Data-Agnostic and Reusable Graphical Structures. *IEEE Transactions on Visualization and Computer Graphics* (2020).

[45] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. 2013. TRANSIT: specifying protocols with concolic snippets. (2013), 287–296. https://doi.org/10.1145/2491956.2462174

[46] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 452–466. https://doi.org/10.1145/3062341.3062365

[47] Chenglong Wang, Yu Feng, Rastislav Bodik, Alvin Cheung, and Isil Dillig. 2019. Visualization by example. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–28.

[48] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Synthesis of data completion scripts using finite tree automata. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 62:1–62:26. https://doi.org/10.1145/3133886

[49] Hadley Wickham. 2010. A layered grammar of graphics. *Journal of Computational and Graphical Statistics* 19, 1 (2010), 3–28.

[50] Hadley Wickham. 2011. ggplot2. *Wiley Interdisciplinary Reviews: Computational Statistics* 3, 2 (2011), 180–185.

[51] Hadley Wickham et al. 2014. Tidy data. *Journal of Statistical Software* 59, 10 (2014), 1–23.

[52] Leland Wilkinson. 2012. The grammar of graphics. In *Handbook of Computational Statistics*. Springer, 375–414.

[53] Kanit Wongsuphasawat, Yang Liu, and Jeffrey Heer. 2019. Goals, Process, and Challenges of Exploratory Data Analysis: An Interview Study. *CoRR* abs/1911.00568 (2019). arXiv:1911.00568 http://arxiv.org/abs/1911.00568

[54] Kanit Wongsuphasawat, Dominik Moritz, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. 2015. Voyager: Exploratory analysis via faceted browsing of visualization recommendations. *IEEE transactions on visualization and computer graphics* 22, 1 (2015), 649–658.

[55] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. 2016. Synthesizing transformations on hierarchically structured data. In *Proc. Conference on Programming Language Design and Implementation*. ACM, 508–521.

[56] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L. Glassman. 2020. Interactive Program Synthesis by Augmented Examples. (2020), 627–648. https://doi.org/10.1145/3379337.3415900