Visualization By Example

CHENGLONG WANG, University of Washington, USA
YU FENG, University of California, Santa Barbara, USA
RASTISLAV BODIK, University of Washington, USA
ALVIN CHEUNG, University of California, Berkeley, USA
ISIL DILLIG, University of Texas at Austin, USA

While visualizations play a crucial role in gaining insights from data, generating useful visualizations from a complex dataset is far from an easy task. In particular, besides understanding the functionality provided by existing visualization libraries, generating the desired visualization also requires reshaping and aggregating the underlying data as well as composing different visual elements to achieve the intended visual narrative. This paper aims to simplify visualization tasks by automatically synthesizing the required program from simple *visual sketches* provided by the user. Specifically, given an input data set and a visual sketch that demonstrates how to visualize a very small subset of this data, our technique automatically generates a program that can be used to visualize the entire data set.

From a program synthesis perspective, automating visualization tasks poses several challenges that are not addressed by prior techniques. First, because many visualization tasks require data wrangling in addition to generating plots from a given table, we need to decompose the end-to-end synthesis task into two separate sub-problems. Second, because the intermediate specification that results from the decomposition is necessarily imprecise, this makes the data wrangling task particularly challenging in our context. In this paper, we address these problems by developing a new *compositional* visualization-by-example technique that (a) decomposes the end-to-end task into two different synthesis problems over different DSLs and (b) leverages bi-directional program analysis to deal with the complexity that arises from having an imprecise intermediate specification.

We have implemented our visualization-by-example approach in a tool called Viser and evaluate it on 83 visualization tasks collected from on-line forums and tutorials. Viser can solve 84% of these benchmarks within a 600 second time limit, and, for those tasks that can be solved, the desired visualization is among the top-5 generated by Viser in 70% of the cases.

CCS Concepts: • **Theory of computation → Program reasoning**; • **Human-centered computing → Visualization toolkits**.

Additional Key Words and Phrases: Program Synthesis, Data Visualization

ACM Reference Format:
Chenglong Wang, Yu Feng, Rastislav Bodik, Alvin Cheung, and Isil Dillig. 2018. Visualization By Example. *Proc. ACM Program. Lang.* 1, CONF, Article 1 (January 2018), 28 pages.

1 INTRODUCTION

Visualizations play an important role in today's data-centric world for discovering, validating, and communicating insights from data. Due to the prevalence of non-trivial visualization tasks across different application domains, recent years have seen a growing number of libraries that aim to facilitate complex visualization tasks. For instance, there are at least a dozen different visualization libraries for Python and R, and more than ten different visualization libraries for JavaScript have emerged in the past year alone [Andre 2019]. In addition, there has also been a

Authors' addresses: Chenglong Wang, University of Washington, USA, clwang@cs.washington.edu; Yu Feng, University of California, Santa Barbara, USA, yufeng@cs.ucsb.edu; Rastislav Bodik, University of Washington, USA, bodik@cs.washington.edu; Alvin Cheung, University of California, Berkeley, USA, akcheung@cs.berkeley.edu; Isil Dillig, University of Texas at Austin, USA, isil@cs.utexas.edu.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

flurry of research activity around building programming systems like D3 [Bostock et al. 2011] and Vega-Lite [Satyanarayan et al. 2017] to further facilitate real-world visualization tasks.

Despite all these recent efforts, data visualization still remains a challenging task that requires considerable expertise — in fact, so much so that some companies even have job titles like "data visualization expert" or "data visualization specialist." Generally speaking, there are three key reasons that make data visualization a challenging task. First, beyond having a good insight about how the data can be best visualized, one needs to have sufficient knowledge about how to use the relevant visualization libraries. Second, different visualization primitives typically require the data to be in different formats; so, in order to experiment with different types of visualizations, one needs to constantly reshape the data into different formats. Finally, generating the intended visualization typically requires modifications to the original dataset, including aggregating and mutating values and adding new columns to the input tables, and doing so often require deep knowledge in data manipulation.

In this paper, we propose a new technique, coined *visualization-by-example*, for automating data visualization tasks using program synthesis. In our proposed approach, the user starts by providing a so-called *visual sketch*, which is a partial visualization of the input data for just a few input points. Given the original data set T_{in} and a visual sketch S provided by the user, our technique can synthesize one or more visualization scripts whose output is consistent with S for the input data set. These visualization scripts can then be applied to T_{in} to generate several visualizations of the *entire data set*, and the user can choose the desired visualization among the ones that are generated.

Despite these appealing aspects of our approach to end-users, the data visualization problem presents unique challenges from a program synthesis perspective. First, as hinted earlier, data visualization tasks almost always involve two distinct steps, namely (1) data wrangling (reshaping, aggregating, adding new columns etc.) and (2) invoking the appropriate visualization primitives on the transformed data. Asking users to manually decompose the problem into these two individual steps would defeat the point, as the user would have to at least understand which visualization primitives to use and what format they require. Thus, it is imperative to have a compositional technique that can *automatically decompose* the end-to-end task into two separate synthesis problems.

One of the key contributions of this paper is to show how to automatically decompose an end-to-end visualization task into two separate synthesis problems over two different languages. Specifically, given an input data source T_{in} and a visual sketch S, our goal is to learn a *table transformation program* P_{T} and a *visual program* P_{V} such that executing $\mathsf{P}_{\mathsf{V}} \circ \mathsf{P}_{\mathsf{T}}$ on T_{in} yields a visualization that is consistent with the provided visual sketch S. In order to solve this problem in a compositional way, our method infers an intermediate specification ϕ that constrains the output (resp. input) of the target program P_{T} (resp. P_{V}). This intermediate specification is in the form of *table inclusion constraints* $\mathsf{T} \overset{\diamond}{\subseteq} t$ specifying that input t of the visual program must include all tuples in T but it can also contain additional rows and columns. As we demonstrate experimentally, having this intermediate specification is crucial for the scalability of our approach.

A second key contribution of this paper is a new algorithm for synthesizing table transformation programs. While there has been recent work on automating table transformation tasks using programming-by-example [Feng et al. 2017; Wang et al. 2017a], these techniques focus on the case where the specification is a pair of input and output tables. In contrast, the intermediate specification in our setting is a set of table inclusion constraints rather than a concrete output table, and pruning strategies used in prior work are not effective in this setting due to lack of precise information about the output table. To deal with this challenge, we introduce a new table transformation synthesis algorithm that uses lightweight bidirectional program analysis to prune the search space. As we demonstrate experimentally, this new table transformation algorithm results in much faster synthesis compared to prior work [Feng et al. 2017] for automating visualization tasks.

We have implemented the proposed "visualization-by-example" approach in a new tool called VISER and evaluated it on 83 visualization tasks collected from on-line forums and tutorials. Our experiments show that VISER can solve 84% of these benchmarks and, among those benchmarks that can be solved, the desired visualization is among the first 5 outputs generated by VISER in 70% of the cases. Furthermore, given that it takes on average of 11 seconds to generate the top-5 visualizations, we believe that VISER is fast enough to be beneficial to prospective users in practice.

To summarize, this paper makes the following key contributions:

- We introduce the *visualization-by-example* problem and present an algorithm for synthesizing visualization scripts given the original data set and a small visual sketch.
- We show how to decompose the synthesis task into two sub-problems by inferring an intermediate specification in the form of table inclusion constraints.
- We propose a new algorithm for synthesizing table transformations from table inclusion specifications. Our algorithm leverages lightweight bidirectional program analysis to effectively prune the search space.
- We evaluate our approach on over 80 tasks collected from on-line forums and tutorials and show that VISER can solve 84% of the benchmarks, and that the desired visualization is among the top-5 results in 70% of the solved cases.

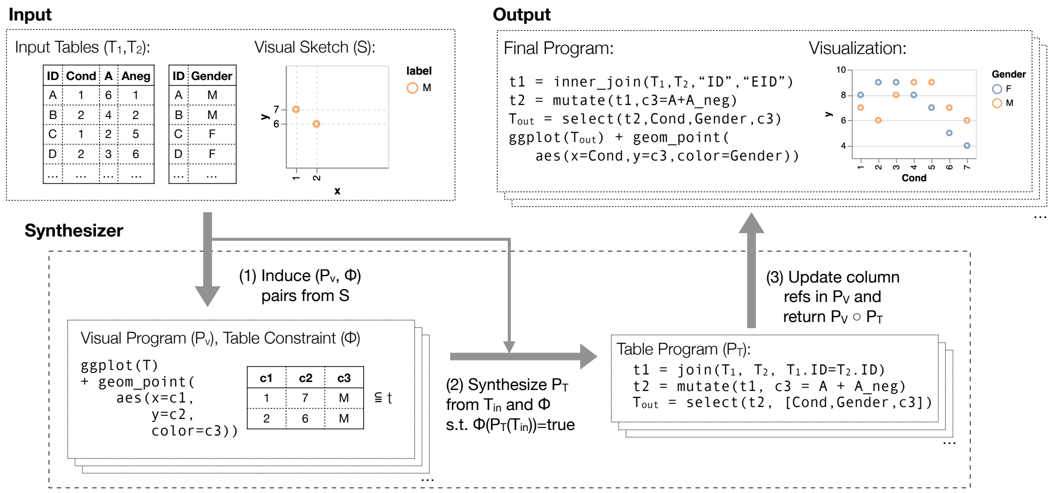
2 OVERVIEW

In this section, we give an overview of our approach with the aid of a simple motivating example depicted in Figure 1. In this example, the user has two tables T_1 and T_2 that record the results of a scientific experiment. Specifically, Table T_1 stores an experiment identifier (ID), a so-called "experiment condition" (Cond), and the experiment result, which consists of an A value as well as an Aneg value. An additional table T_2 stores the gender of the participant in the corresponding study: That is, for each experiment (ID), the Gender column in T_2 indicates whether the participant is male (M) or female (F). The user wants to visualize the result of this experiment by drawing a scatter plot that shows how the sum of A and Aneg changes with respect to Cond for each of the two genders. In particular, the top right part of Figure 1 illustrates the desired visualization. In the remainder of this section, we explain how our approach synthesizes the desired visualization script in R using the `tidyverse` package collection, which includes both visualization primitives (e.g., provided by `ggplot2`) and data wrangling capabilities (e.g., provided by `tidyr` and `dplyr`).

User input. In order to use our visualization tool, VISER, the user needs to provide the data source (i.e., tables T_1 and T_2) as well as a visual sketch S, which is a partial visualization for a tiny subset of the original data source. In this case, since the user wants to draw a scatter plot, the visual sketch is very simple and consists of a few data points, as shown in the "Input" portion of Figure 1. Specifically, the visual sketch contains two points, one at $(1, 7)$ and the other at $(2, 6)$, and both points have label "M". In general, one can think of a visual sketch as a set of *visualization elements* (e.g., point, bar, line, ...) where each visualization element has various attributes, such as coordinates, color, etc. In particular, we can specify the visual sketch shown in Figure 1 as the following set of visual elements:

$$\{\mathsf{point}(v_x = 1, v_y = 7, v_{color} = \mathtt{M}), \mathsf{point}(v_x = 2, v_y = 6, v_{color} = \mathtt{M})\}$$

We refer to this alternative set-based representation of a visualization as a *visual trace.* In general, we can represent both visual sketches as well as complete visualizations in terms of their corresponding


Input
Input Tables (T₁,T₂):
Visual Sketch (S):
label
M
Output
Final Program:
Visualization:
Gender
F
M
Cond
Synthesizer
(1) Induce (Pv, Φ) pairs from S
(2) Synthesize PT from Tin and Φ s.t. Φ(PT(Tin))=true
(3) Update column refs in PV and return PV ∘ PT
Visual Program (Pv), Table Constraint (Φ)
Table Program (PT):
⊆ t

ID	Cond	A	Aneg
A	1	6	1
B	2	4	2
C	1	2	5
D	2	3	6
...	...	...	...

ID	Gender
A	M
B	M
C	F
D	F
...	...

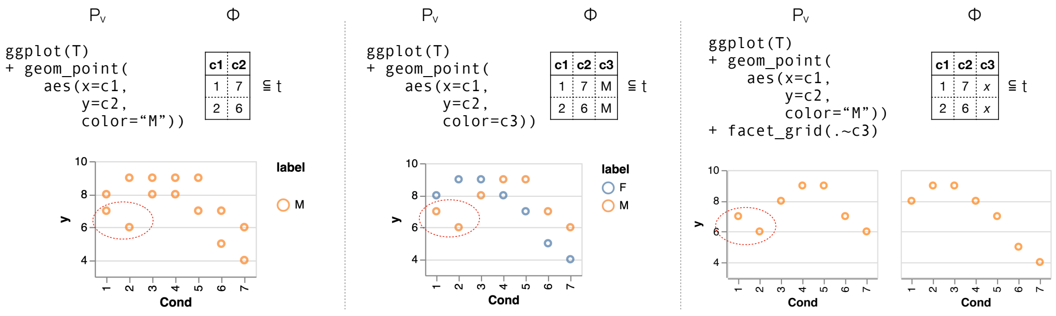
```
t1 = inner_join(T₁,T₂,“ID”,“EID”)
t2 = mutate(t1,c3=A+A_neg)
Tout = select(t2,Cond,Gender,c3)
ggplot(Tout) + geom_point(
    aes(x=Cond,y=c3,color=Gender))
```

```
ggplot(T)
+ geom_point(
    aes(x=c1,
        y=c2,
        color=c3))
```

c1	c2	c3
1	7	M
2	6	M

```
t1 = join(T₁, T₂, T₁.ID=T₂.ID)
t2 = mutate(t1, c3 = A + A_neg)
Tout = select(t2, [Cond,Gender,c3])
```

Fig. 1. Overview of our synthesis algorithm: the system takes as input an input table T_{in} and a visual sketch S, and returns a list of candidate visualizations satisfying the inputs.



```
ggplot(T)
+ geom_point(
    aes(x=c1,
        y=c2,
        color=“M”))
```

c1	c2
1	7
2	6

```
ggplot(T)
+ geom_point(
    aes(x=c1,
        y=c2,
        color=c3))
```

c1	c2	c3
1	7	M
2	6	M

```
ggplot(T)
+ geom_point(
    aes(x=c1,
        y=c2,
        color=“M”))
+ facet_grid(.~c3)
```

c1	c2	c3
1	7	*x*
2	6	*x*

Fig. 2. Sample visual programs and their corresponding intermediate specification for the visual sketch from Figure 1. The bottom part of each figure shows the corresponding visualization if the visual program is adopted; observe that all of these visualizations are consistent with the visual sketch.

visual trace; thus, we use the term "visual trace" interchangeably with both "visualization" and "visual sketch".

Synthesis problem and approach. Given the input data source I and a visual sketch S, our synthesis problem is to infer a *visualization script* P such that P(I) yields a visual trace that is *a superset* of S. As mentioned in section 1, a visualization script consists of a pair of programs P_V (“visual program”) and P_T (“table transformation program”), for plotting and data wrangling respectively. Since P_V and P_T do conceptually different things and are expressed in separate languages, we decompose the overall synthesis task into two sub-tasks, namely that of synthesizing a visual program P_V and separately synthesizing a table transformation program P_T.

Synthesis of visual programs. To achieve the decomposition outlined above, our synthesis algorithm first infers a *set* of visual programs that are *capable* of generating a visualization consistent with S. This inference step is based purely on the visual sketch and does not consider the input data (i.e., tables T_1, T_2). For our running example, there are multiple visual programs (expressible using ggplot2) that can generate the desired result; we show three of these programs in Figure 2. All three programs start with the code "ggplot(T) + geom_point(...)", which indicates that the resulting visualization is a scatter plot drawn from table T. However, the three visual programs differ in the following ways:

- All points generated by the first visual program have the same color (indicated as color="M")
- For the second visual program, the color of the points is determined by the corresponding value of column c3 in table T (indicated as color=c3)
- The visualization generated by the last program contains multiple subplots determined by c_3. That is, the visualization is partitioned into a list of subplots according to different values in column c_3 of the input table.

As indicated in the bottom part of Figure 2, the visualizations generated by all three programs are consistent with the visual sketch in that they contain the two data points specified by the user.

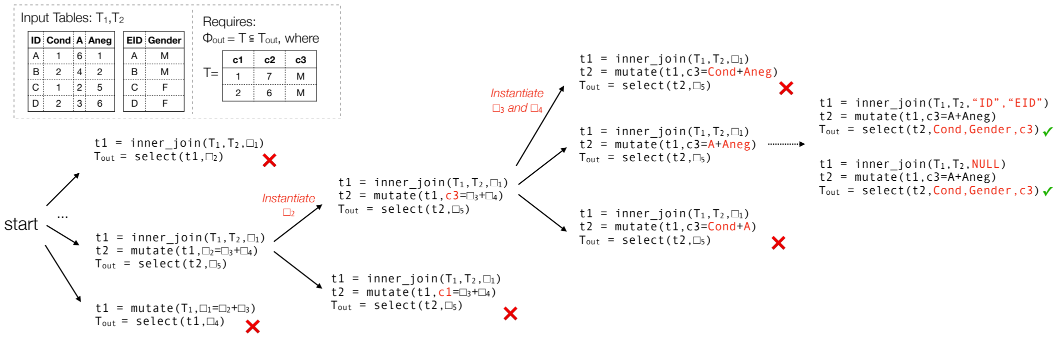
Intermediate specification inference. Next, given the visual sketch S and a candidate visual program $\mathsf{P_V}$, our synthesis algorithm infers an intermediate specification ϕ that constrains the input that $\mathsf{P_V}$ operates on. Furthermore, ϕ has the property that executing $\mathsf{P_V}$ on any concrete table consistent with $\mathsf{T_{in}}$ yields a visual trace that is a superset of S. Going back to our running example, Figure 2 shows the intermediate specifications inferred for each of the three visual programs. These intermediate specifications are of the form $\mathsf{T} \overset{\diamond}{\subseteq} t$ indicating that the input t of the visual program must contain table T but can also include additional rows and columns. Looking at the intermediate specifications from Figure 2, we can make the following observations:

- The inputs of the first visual programs must contain at least two columns (referred to as c1, c2) and these columns should contain the values 1, 2 and 7, 6 respectively. However, the input table can also contain additional attributes and values.
- The specification for the second visual program imposes one additional constraint over the other ones. In particular, the input table must contain an additional third column (referred to as c3), and this column must contain at least two occurrences of value M.
- The specification for the third visual program requires that the table should contain an additional column c3 to specify which subplot each point belongs to. Since the visual sketch contains two points in the same subplot, column c3 contains two duplicate values with the same subplot identifier.

Input for the second synthesis task. As mentioned earlier, the key reason for inferring an intermediate specification is to decompose the problem into two separate synthesis tasks. Thus, given an initial data source $\mathsf{T_{in}}$ and intermediate specification ϕ, the goal of the second synthesis task is to generate a table transformation program $\mathsf{P_T}$ such that applying $\mathsf{P_T}$ to $\mathsf{T_{in}}$ yields a table that is consistent with ϕ. To illustrate how our method synthesizes the desired table transformation program for our running example, let us consider the following data wrangling constructs:

- **Projection:** The construct select$(t, \bar{c})$ computes the projection of table t onto columns $\bar{c}$.
- **Join:** The construct inner_join(t_1, t_2, p) computes the product of tables t_1, t_2 and then filters the result based on predicate p.
- **Mutation:** The construct mutate$(t, c_{target}, c_{arg_1} + c_{arg_2})$ takes as input a table t and returns a table with a new column called c_{target}, where the values in c_{target} are obtained by summing up the two columns c_{arg_1} and c_{arg_2}; [1]

[1]General mutate operator supports arbitrary column-wise computation besides '+', we only consider mutate with '+' in overview for simplicity.


Input Tables: T₁,T₂

ID	Cond	A	Aneg
A	1	6	1
B	2	4	2
C	1	2	5
D	2	3	6

EID	Gender
A	M
B	M
C	F
D	F

Requires:
$\Phi_{out} = T \leqq T_{out}$, where

T=

c1	c2	c3
1	7	M
2	6	M

```
t1 = inner_join(T₁,T₂,□₁)
Tout = select(t1,□₂)
```

```
t1 = inner_join(T₁,T₂,□₁)
t2 = mutate(t1,□₂=□₃+□₄)
Tout = select(t2,□₅)
```

```
t1 = mutate(T₁,□₁=□₂+□₃)
Tout = select(t1,□₄)
```

```
t1 = inner_join(T₁,T₂,□₁)
t2 = mutate(t1,c3=□₃+□₄)
Tout = select(t2,□₅)
```

```
t1 = inner_join(T₁,T₂,□₁)
t2 = mutate(t1,c1=□₃+□₄)
Tout = select(t2,□₅)
```

```
t1 = inner_join(T₁,T₂,□₁)
t2 = mutate(t1,c3=Cond+Aneg)
Tout = select(t2,□₅)
```

```
t1 = inner_join(T₁,T₂,□₁)
t2 = mutate(t1,c3=A+Aneg)
Tout = select(t2,□₅)
```

```
t1 = inner_join(T₁,T₂,□₁)
t2 = mutate(t1,c3=Cond+A)
Tout = select(t2,□₅)
```

```
t1 = inner_join(T₁,T₂,"ID","EID")
t2 = mutate(t1,c3=A+Aneg)
Tout = select(t2,Cond,Gender,c3) ✓
```

```
t1 = inner_join(T₁,T₂,NULL)
t2 = mutate(t1,c3=A+Aneg)
Tout = select(t2,Cond,Gender,c3) ✓
```

Fig. 3. The synthesis process for P_T using T_1, T_2 and ϕ_{out} in Figure 1. At each step, the synthesis algorithm first picks a known variable and expands it (new values expanded at each step are labeled in red), then it evaluates each program sketch using abstract semantics of the table transformation language and prune it if the evaluation process results in conflicts.

We will now illustrate how to synthesize the desired table transformation program P_T for the input tables T_1, T_2 shown in Figure 1 and the intermediate specification ϕ_{out} shown in the second column of Figure 3.

Table transformation synthesis overview. Viser employs an enumerative search algorithm to find a table transformation program that satisfies the specification. Similar to prior program synthesis techniques [Feng et al. 2018, 2017; Wang et al. 2017a], Viser uses lightweight deductive reasoning to prune invalid programs during the search process. However, because the specification does not involve a concrete output table, pruning techniques used in prior work (e.g., [Feng et al. 2017]) are not effective in this context. As mentioned in section 1, our algorithm addresses this issue by leveraging lightweight bidirectional program analysis and an (also lightweight) incomplete inference procedure over table inclusion constraints.

As illustrated schematically in Figure 3, Viser performs enumerative search over *program sketches*, where each program sketch is a sequence of statements of the form `v = op(`$\square_1, \ldots, \square_n$`)` where `op` is one of the data wrangling constructs (e.g., `select`, `inner_join` etc.) and $\square_i$ denotes an unknown argument. Since program sketches contain only table-level operators but not their arguments, the sketch enumeration process is tractable, and the main synthesis burden lies in searching the large number of parameters that each hole $\square_i$ can be instantiated with.

Sketch completion. Given a program sketch, Viser's sketch completion procedure alternates between *hole instantiation* and *pruning* steps until a solution is found or all possible sketch completions are proven *not* to satisfy the specification (see Figure 3). The first step (i.e., hole instantiation) is standard and makes the initial sketch iteratively more concrete by filling each hole with a program variable or constant. The pruning step, on the other hand, is more interesting, and infers table inclusion constraints of the form $e_1 \overset{\diamond}{\subseteq} e_2$ indicating that the table represented by expression e_1 can be obtained from the table represented by expression e_2 by removing rows and/or columns. For the forward (resp. backward) inference, the generated constraints have the shape $t \overset{\diamond}{\subseteq} \mathsf{T}$ (resp. $\mathsf{T} \overset{\diamond}{\subseteq} t$), where t is a program variable and T is a concrete table. Thus, using a combination of forward and backward reasoning, we can obtain "inequality" constraints of the form $\mathsf{T}_1 \overset{\diamond}{\subseteq} t \overset{\diamond}{\subseteq} \mathsf{T}_2$ for each program variable t and use this information to reject a *partially completely sketch* whenever T_1 *cannot* be obtained from T_2 by deleting rows and/or columns.

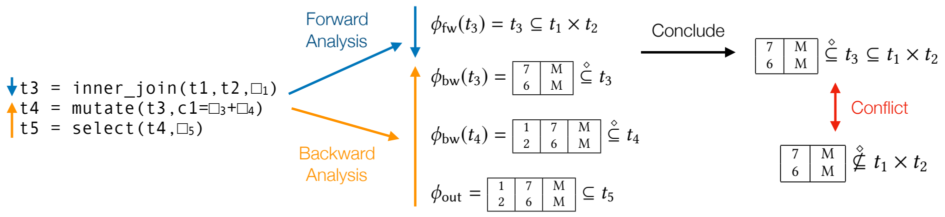


Fig. 4. Demonstration of how VISER prunes invalid abstract programs in Figure 3 using forward and backward analysis. ϕ_{out} is the requirement from the synthesis task, and $\phi_{\text{fw}}(t)$, $\phi_{\text{bw}}(t)$ refers to abstractions of t derived from forward / backward analysis.

Going back to our running example, let us consider the partially completed sketch shown in Figure 4, where t_1, t_2 represent the program's arguments and t_5 is the return variable. By considering the semantics of each construct, we can make the following deductions:

- Since the program's output must conform to $\phi_{\text{out}} = (\mathsf{T} \overset{\diamond}{\subseteq} t)$ (where T is the table from Figure 2) and we have $t = t_5$, we can generate the constraint $\mathsf{T} \overset{\diamond}{\subseteq} t_5$ on variable t_5.
- Together with the above constraint and semantics of `select`, we obtain $\mathsf{T} \overset{\diamond}{\subseteq} t_4$.
- Since we have $\mathsf{T} \overset{\diamond}{\subseteq} t_4$ and `mutate(t3, c1, ? + ?)` generates t_4 by adding column `c1` to t_3, we can deduce all columns of T except for the first one should also be in t_3. Thus, using backwards reasoning, we obtain the constraint $\mathsf{T}' \overset{\diamond}{\subseteq} t_3$ where T' is the table shown on the left-hand side of $\phi_{\text{bw}}(t_3)$.
- Since t_3 is the result of `inner_join(t1, t2, ?)`, we cannot deduce something useful about t_1, t_2 in the backward direction without introducing expensive case splits because we do not whether each value in T' comes from t_1 or t_2. However, going in the *forward* direction, we can generate the constraint $t_3 \overset{\diamond}{\subseteq} T_1 \times T_2$, where T_1, T_2 are the input tables from Figure 1.

Putting together the information obtained from the forwards and backwards analysis, we obtain the constraint $\mathsf{T}' \overset{\diamond}{\subseteq} t_3 \overset{\diamond}{\subseteq} \mathsf{T}_1 \times \mathsf{T}_2$. However, this creates a contradiction with $\mathsf{T}' \overset{\diamond}{\not\subseteq} \mathsf{T}_1 \times \mathsf{T}_2$, allowing us to prune the partially completed sketch from Figure 4.

Synthesis output. Continuing in this manner and alternating between more hole instantiation and pruning steps, VISER finds the sketch completion shown on the bottom right side of Figure 1. The final output of the synthesizer is shown on the top right of the same figure and generates the intended visualization, also shown in the top right of Figure 1.

3 PROBLEM DEFINITION

In this section, we formally define the visualization-by-example problem and then introduce two languages for automating table transformation and plotting tasks.

3.1 Key Concepts and Synthesis Problem

Tables. For the purposes of this paper, a table T with schema $[c_1, \ldots, c_n]$ is an unordered bag (i.e., multi-set) of tuples where each tuple $r = (v_1, \ldots, v_n)$ in T consists of n primitive values (number, string, datetime etc.). Given a tuple $r \in \mathsf{T}$, we use the notation $r[c]$ to denote the value v stored in attribute c of r. We also extend this notation to tables and write $T[\bar{c}]$ to denote the projection of table T on columns $\bar{c}$ and write $T[-\bar{c}]$ to denote the projection of table T on all columns *except* $\bar{c}$. Finally, we write $\mathsf{Mult}(r, \mathsf{T})$ to denote the multiplicity of tuple r in T.

$$
\begin{array}{rcl}
\tau & = & \{e_1, \ldots, e_n\} \\
e & = & \mathsf{bar}(a_x, a_{y_1}, a_{y_2}, a_{color}, a_{subplot}) \\
& | & \mathsf{point}(a_x, a_y, a_{color}, a_{size}, a_{subplot}) \\
& | & \mathsf{line}(a_{x_1}, a_{y_1}, a_{x_2}, a_{y_2}, a_{color}, a_{subplot})
\end{array}
$$

Fig. 5. The visualization trace language $\mathcal{L}_\tau$, where metavariable a refers to constants.

Given a pair of tables $\mathsf{T}_1, \mathsf{T}_2$, we write $\mathsf{T}_1 \subseteq \mathsf{T}_2$ iff $\forall r \in \mathsf{T}_1.\ \mathsf{Mult}(r, \mathsf{T}_1) \leq \mathsf{Mult}(r, \mathsf{T}_2)$. As standard, we define equality to be containment in both directions, i.e., $\mathsf{T}_1 = \mathsf{T}_2$ iff $\mathsf{T}_1 \subseteq \mathsf{T}_2$ and $\mathsf{T}_2 \subseteq \mathsf{T}_1$. We further define a table inclusion constraint $\mathsf{T}_1 \overset{\diamond}{\subseteq} \mathsf{T}_2$ that allows projecting columns in addition to filtering rows. Specifically, we write $\mathsf{T}_1 \overset{\diamond}{\subseteq} \mathsf{T}_2$ iff there exists columns $\bar{c}$ in the schema of T_2 such that $\mathsf{T}_1 \subseteq \mathsf{T}_2[\bar{c}]$.

Visual traces. As stated in Section 2, we define the semantics of visualizations in terms of so-called *visual traces*. A visual trace, denoted τ, is a set of basic visual elements (point, line, bar), together with the attributes of each element (position, size, color, etc.). More concretely, Figure 5 shows a small "language" in which we express visual traces. (The full language supported by our implementation is given in the Appendix.) Here, e denotes a visual element, and a is an *attribute* of that element:

- *Color attribute:* This attribute, denoted a_{color} specifies the color of a visual element.
- *Position attributes:* Position attributes, such as a_x, a_{x_1}, a_{y_2} etc., specify the canvas positions for a visual element. For example, for line, (a_{x_1}, a_{y_1}) specifies the starting point of a line segment, and (a_{x_2}, a_{y_2}) specifies the end point. For the bar visual element, a_{y_1}, a_{y_2} specify the start and end y-coordinates of a (vertical) bar.
- *Size attribute:* The attribute a_{size} specifies the size of a given point element.
- *Subplot attribute:* The attribute $a_{subplot}$ specifies the subplot that a given visual element belongs to. For instance, for the visualization shown in the last column of Figure 2, the points in the first plot have a different $a_{subplot}$ attribute than those in the second one.

In the remainder of this paper, we express both complete visualizations and visual sketches in terms of their corresponding visual trace, and we often use the symbol S to denote traces that correspond to visual sketches. Finally, since visual traces are *sets* of visual elements, the notation $\tau_1 \subseteq \tau_2$ indicates that visualization τ_2 is an *extension* of visualization τ_1.

Problem statement. Given this notion of visual traces, we can now state our *visualization-by-example problem*, which is defined by a pair $(\mathsf{T}_{\mathsf{in}}, \mathsf{S})$. Here, T_{in} is a table [2] and S is a visual trace (i.e., a "program" in the language of Figure 5). Now, let us fix a language $\mathcal{L}_\mathsf{T}$ for expressing table transformation programs and a visualization language $\mathcal{L}_\mathsf{V}$ for generating plots from a given table. Then, our goal is to synthesize a pair of programs $(\mathsf{P}_\mathsf{T}, \mathsf{P}_\mathsf{V})$ such that:

(1) P_T and P_V are programs written in $\mathcal{L}_\mathsf{T}, \mathcal{L}_\mathsf{V}$ respectively,
(2) the output visual trace $\mathsf{P}_\mathsf{T}(\mathsf{P}_\mathsf{V}(\mathsf{T}_{\mathsf{in}}))$ is consistent with S, i.e., $\mathsf{S} \subseteq \mathsf{P}_\mathsf{V}(\mathsf{P}_\mathsf{T}(\mathsf{T}_{\mathsf{in}}))$.

Note that our problem statement strictly generalizes conventional programming-by-example which requires the program output to be equal to the provided output (i.e., $\mathsf{S} = \mathsf{P}_\mathsf{V}(\mathsf{P}_\mathsf{T}(\mathsf{T}_{\mathsf{in}}))$). Thus, the user is still free to provide a small (but complete) input-output example if this is more convenient for the user. However, our generalization has the advantage of freeing the user from the burden of modifying the input data in cases where doing so may be inconvenient.

[2] We note that our implementation can handle multiple tables in the input. However, we consider a single input table in the formal development to simplify presentation and reduce notational overhead.

$$
\begin{array}{rcl@{\qquad}rcl}
\mathsf{P_T}(t) & = & t_1 = e_1; \ldots ; \mathsf{T_{out}} = e_n; & T & = & \mathsf{T_{in}} \mid t \\
e & = & T \mid \mathsf{filter}(T, f) \mid \mathsf{select}(T, \bar{c}) \mid \mathsf{join}(T_1, T_2, f) & f & = & v_1 \; op \; v_2 \mid \mathsf{is\_null}(c) \\
 & \mid & \mathsf{mutate}(T, c_{target}, op, \bar{c}_{arg}) \mid \mathsf{gather}(T, \bar{c}_{id}, \bar{c}_{target}) & v & = & \mathsf{const} \mid c \\
 & \mid & \mathsf{spread}(T, \bar{c}_{id}, c_{key}, c_{val}) & \alpha & = & \mathsf{min} \mid \mathsf{max} \\
 & \mid & \mathsf{summarize}(T, \bar{c}_{key}, \alpha, c_{target}) & & \mid & \mathsf{sum} \mid \mathsf{count} \mid \mathsf{avg}
\end{array}
$$

Fig. 6. The table transformation language $\mathcal{L}_\mathsf{T}$, where $\mathsf{T_{in}}, \mathsf{T_{out}}$ refers to the input/output tables, t refers to table variables, and c refers to column names.

3.2 Table Transformation Language

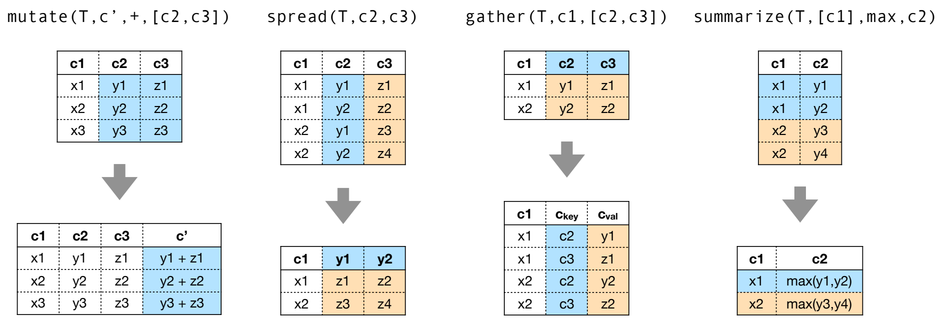
Our table transformation language is shown in Figure 6. This language is inspired by existing data wrangling libraries (e.g., tidyr and dplyr libraries for R), and similar languages have also been used in prior work for automating table transformation tasks using PBE [Feng et al. 2017; Martins et al. 2019]. As shown in Figure 6, a table transformation program $\mathsf{P_T}$ is a sequence of side-effect free statements, where each statement produces a new table by performing some operation on its inputs. As standard in relational algebra, the constructs select and filter are used for selecting columns and rows respectively. As also standard in relational algebra, join is used for taking the cross product of two tables. That is, $\mathsf{join}(T_1, T_2, f)$ is semantically equivalent to $\mathsf{filter}(T_1 \times T_2, f)$, where $\times$ denotes the standard cross product operator in relational algebra.

Besides these standard relational algebra operators, our table transformation language contains four other main constructs, namely spread, mutate, gather, and summarize. Since the semantics of these constructs are somewhat non-trivial, we illustrate their behavior in Figure 7.

(1) The `mutate` construct creates a new column (c_{target}) in the output table by applying an operator *op* on argument columns $\bar{c}_{args}$. For example, in Figure 7, the new column c' is obtained by summing up columns c_2 and c_3.
(2) The `spread` operator pivots a table by changing values to column names. Specifically, `spread` first eliminates the two columns c_{key} and c_{val}, then creates a new column for each value stored in the original column c_{key}, and finally fills new columns using values in the original c_{val} column. The second drawing in Figure 7 illustrates the semantics of `spread`.
(3) The `gather` construct is the inverse of `spread`: It unpivots the input table by moving column names into the table body. Specifically, `gather` first eliminates all columns in $\bar{c}_{target}$, and then creates two new columns c_{key} and c_{val} where c_{key} is filled with column names in $\bar{c}_{target}$ and column c_{val} is filled with values in the eliminated columns. This is illustrated in the third drawing in Figure 7.
(4) The `summarize` construct first partitions its input table into groups based on values in $\bar{c}_{key}$ and then applies the function α to each group to aggregate values in column c_{target}. For example, in the rightmost part of Figure 7, the table is partitioned into two groups based on values in c_1 (labeled with different colors), and column c_2 is populated by taking the maximum of all values in the corresponding partition.

3.3 Visualization Language

Our visualization language $\mathcal{L}_\mathsf{V}$ is shown in Figure 8, which formalizes core constructs in Vega-Lite [Satyanarayan et al. 2017], the `ggplot2` visualization library for R and VizQL [Hanrahan 2006] from Tableau. This formalization enables concise descriptions of visualizations by encoding data as properties of graphical marks. It represents single plots using a set of mappings that map data fields to visual properties, and supports combining single charts into compositional charts through



c1	c2	c3
x1	y1	z1
x2	y2	z2
x3	y3	z3

c1	c2	c3	c'
x1	y1	z1	y1 + z1
x2	y2	z2	y2 + z2
x3	y3	z3	y3 + z3

c1	c2	c3
x1	y1	z1
x1	y2	z2
x2	y1	z3
x2	y2	z4

c1	y1	y2
x1	z1	z2
x2	z3	z4

c1	c2	c3
x1	y1	z1
x2	y2	z2

c1	c_{key}	c_{val}
x1	c2	y1
x1	c3	z1
x2	c2	y2
x2	c3	z2

c1	c2
x1	y1
x1	y2
x2	y3
x2	y4

c1	c2
x1	max(y1,y2)
x2	max(y3,y4)

Fig. 7. Examples of table transformation operators in $\mathcal{L}_\mathrm{T}$ and how they operate on example input tables. Colored cells shows how parts of the output table are computed from the input table.

$$
\begin{array}{rcll}
\mathrm{P_V} & = & \mathsf{MultiPlot}(SP, c_{\mathrm{sub}}) \mid SP & \\
SP & = & \mathsf{MultiLayer}(\bar{L}) \mid L & \\
L & = & \mathsf{Scatter}(c_x, c_y, c_{color}, c_{size}) & \text{(Scatter Plot)} \\
 & \mid & \mathsf{Line}(c_x, c_y, c_{color}) & \text{(Line Chart)} \\
 & \mid & \mathsf{Bar}(c_x, c_y, c_{y_2}, c_{color}) & \text{(Bar Chart)} \\
 & \mid & \mathsf{Stacked}(c_x, c_h, c_{color}) & \text{(Stacked Bar Chart)} \\
c & = & column \mid \epsilon &
\end{array}
$$

Fig. 8. The visualization language $\mathcal{L}_\mathrm{V}$.

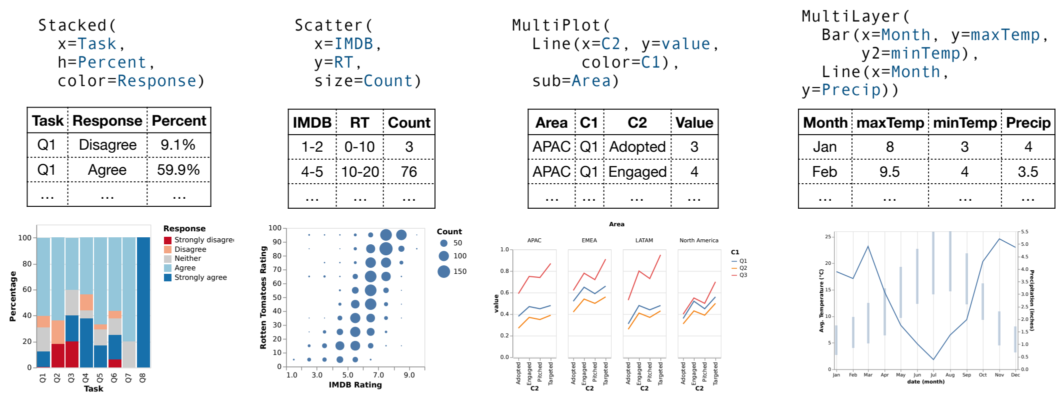
layering and subplotting. A program $\mathrm{P_V}$ in this language takes as input a table T and outputs a visual trace τ. Throughout this paper, we refer to programs in this language as *visual programs.*

As shown in Figure 8, a visual program $\mathrm{P_V}$ either creates a grid of multiple plots using the MultiPlot construct or a single plot *SP*. Each plot can in turn consist of multiple layers (indicated by the MultiLayer construct) or a single layer. Each layer is either a scatter plot (Scatter), a line chart (Line), a bar chart (Bar), or a stacked bar chart (Stacked). The MultiLayer construct in this language is used to compose *different* kinds of charts in the same plot (e.g., a scatter plot and a line chart), but our visualization language is nonetheless rich enough to allow layering the same type of chart within a plot: For example, the Line primitive can be used to render multiple line charts where each individual line chart has a different color.

In terms of its semantics, a visual program $\mathrm{P_V}$ specifies how each tuple in the input table corresponds to a visual element in the output trace; thus, all constructs in $\mathcal{L}_\mathrm{V}$ refer to column names in the input table. For instance, for the MultiPlot construct, the column name c_{sub} specifies that tuples sharing the same value of c_{sub} are to be visualized in the same subplot, whereas tuples with different values of c_{sub} belong to two different subplots. Similarly, for the Line construct, tuples that agree on the value of c_{color} are rendered as part of the same line chart, whereas tuples that disagree on the c_{color} value correspond to different layers.

In what follows, we explain the semantics of our visualization language with the aid of the examples shown in Figure 9.

Example 1. *The first example in Figure 9 shows a visual program for rendering a stacked bar chart visualization of a study report. This program specifies using* `x=Task` *that each different (stacked) bar on the x-axis should correspond to a different task (Q1, Q2 etc.) from the input table. The second argument,* `y=Percent`*, specifies that the height of each bar (within a stack) is determined by the* `Percent` *column*



```
Stacked(
  x=Task,
  h=Percent,
  color=Response)
```

Task	Response	Percent
Q1	Disagree	9.1%
Q1	Agree	59.9%
...	...	...

```
Scatter(
  x=IMDB,
  y=RT,
  size=Count)
```

IMDB	RT	Count
1-2	0-10	3
4-5	10-20	76
...	...	...

```
MultiPlot(
  Line(x=C2, y=value,
       color=C1),
  sub=Area)
```

Area	C1	C2	Value
APAC	Q1	Adopted	3
APAC	Q1	Engaged	4
...	...	...	...

```
MultiLayer(
  Bar(x=Month, y=maxTemp,
      y2=minTemp),
  Line(x=Month,
y=Precip))
```

Month	maxTemp	minTemp	Precip
Jan	8	3	4
Feb	9.5	4	3.5
...	...	...	...

Fig. 9. Examples of visualization operators in $\mathcal{L}_V$ and their corresponding visualizations.

in the input table. Finally, the third argument, `color=Response`*, specifies that the color of each bar (within the stack) is determined by the value stored in the* `Response` *column of the input table.*

Example 2. *The second visual program in Figure 9 renders a scatter plot that visualizes the correlation between IMDB and Rotten Tomato reviews. Specifically, the* `Scatter` *construct draws a point for each row in the input table. In our example, the first (resp. second) argument specifies that the x (resp. y) coordinate is determined by the value in the* `IMDB` *(resp.* `RT`*) column of the input table. Finally, the third argument* `size=Count` *specifies that the size of the point is determined by the corresponding value stored in the* `Count` *column.*

Example 3. *The third program from Figure 9 renders multiple subplots, as specified by the* `MultiPlot` *construct. The second argument* `subplot=Area` *specifies that each subplot corresponds to a separate value in the* `Area` *column of the input table (APAC, North America etc.). The first argument, on the other hand, specifies that each subplot is a line chart. Furthermore, since the third argument of the* `Line` *construct is* `color=C1`*, each subplot consists of multiple line charts of different colors, determined by the value of the* `C1` *column in the input table. Finally, the (x, y) coordinates of the points within each line chart are determined by the values in* `C2` *and* `Value` *columns respectively.*

Example 4. *The last program in Figure 9 draws a layered chart consisting of a bar chart and a line chart using the MultiLayer construct. Here, bars show the temperature range for each month because the* `x` *value corresponds to the* `Month` *field in the input table, and* `y` *and* `y2` *correspond to the* `maxTemp` *and* `minTemp` *values for that month. On the other hand, the line chart shows the precipitation for each month; this is again specified using* `x=Month` *and* `y=Precip`.

4 SYNTHESIS ALGORITHM

In this section, we first give an overview of our top-level synthesis algorithm (Section 4.1) and then present techniques for learning visual programs (Section 4.2) and table transformation programs (Section 4.3) respectively.

4.1 Overview

Algorithm 1 describes our top-level visualization-by-example algorithm, which takes as input a table T_{in} and a visual sketch S (expressed as a visual trace in the notation of Figure 5) and returns a pair of programs (P_T, P_V) such that $S \subseteq P_V(P_T(T_{in}))$ (or $\perp$ to indicate failure). As mentioned

Algorithm 1 Top-level Synthesis Algorithm

```
1: procedure SYNTHESIZE($\mathsf{T_{in}}$, S)
2:     input: Input table $\mathsf{T_{in}}$, visual sketch S
3:     output: A table transformation program $\mathsf{P_T}$ and visual program $\mathsf{P_V}$, or $\bot$ if failure
4:     $\Omega \leftarrow$ LEARNVISUALPROGS(S)
5:     for all $(\mathsf{P_V}, \psi) \in \Omega$ do
6:         $r \leftarrow$ LEARNTABLETRANSFORM($\mathsf{T_{in}}, \psi$)
7:         match $r$
8:             case $\bot$: continue
9:             case $(\mathsf{P_T}, \sigma)$: return $(\mathsf{P_T}, \mathsf{P_V}[\sigma])$
10:    return $\bot$
```

previously, our synthesis algorithm is compositional in the sense that it uses an intermediate specification to guide the search for table transformation programs.

Internally, the SYNTHESIZE procedure first uses the input visual sketch S to infer a *set* Ω of intermediate synthesis results. Each intermediate result $r \in \Omega$ is a pair $(\mathsf{P_V}, \psi)$, where $\mathsf{P_V}$ is a visual program that is consistent with the provided visual sketch and ψ is a constraint that imposes certain requirements on the input to $\mathsf{P_V}$. In other words, for a given visual program $\mathsf{P_V}$, ψ serves as an intermediate specification that constrains the space of possible table transformation programs. However, since we do not know the column names used in this intermediate table, both $\mathsf{P_V}$ and ψ refer to "made-up" column names to be resolved in the next phase.

In the second phase of the algorithm (lines 5-9), we try to find a table transformation program that satisfies the intermediate specification. Specifically, for each intermediate synthesis result $(\mathsf{P_V}, \psi)$, the LEARNTABLETRANSFORM procedure is used to synthesize a table transformation program $\mathsf{P_T}$ and a column mapping σ such that $\mathsf{P_T}(\mathsf{T_{in}})$ satisfies the constraint $\psi[\sigma]$. Thus, if LEARNTABLETRANSFORM does not return $\bot$ to indicate failure, the program $\mathsf{P_V}[\sigma] \circ \mathsf{P_T}$ is guaranteed to satisfy the end-to-end specification defined by $(\mathsf{T_{in}}, \mathsf{S})$.

4.2 Synthesis of Visual Programs

In this section, we describe the LEARNVISUALPROGS procedure used in Algorithm 1. This procedure is described using inference rules of the form $\tau \Uparrow (\mathsf{P_V}, \psi)$ where τ is a visual trace, $\mathsf{P_V}$ is a visual program, and ψ is a constraint. The meaning of this judgment is that, if the input table T satisfies constraint ψ, then $\mathsf{P_V}(\mathsf{T})$ yields a visualization that is consistent with τ (i.e., $\tau \subseteq \mathsf{P_V}(\mathsf{T})$). Observe that, for a given visual trace τ, there may be multiple programs that are consistent with it — i.e., we can have $\tau \Uparrow (\mathsf{P}^i_{\mathsf{V}}, \psi^i)$ for multiple values of i. This is the reason why the LEARNVISUALPROGS procedure used in Algorithm 1 returns a set rather than a singleton. In what follows, we explain each of the inference rules from Figure 10 in more detail.

Multiple plots. The first rule, labeled Multi-Plot, is used to synthesize programs for generating multiple subplots. Since each element in a visual trace has an attribute that identifies which subplot it belongs to, we first partition the visual elements according to the value of this attribute. This allows us to obtain n different visual traces $\tau_1, \ldots, \tau_n$, and we recursively synthesize a visual program p_i and a constraint ψ_i for each visual trace τ_i. However, since the MultiPlot construct takes a *single* program as argument, this means that all subplots must be generated using the *same* visual program; thus, the premise of this rule stipulates that all p_i's must be the same program p. On the other hand, each subplot can impose different restrictions on the input table; thus, the input

$$\frac{\tau_i \in \mathsf{partitionBySubplots}(\tau) \qquad \mathsf{c_{sub}}\text{ fresh} \qquad \tau_i \Uparrow (p, \psi_i) \qquad i \in [1, n]}{\tau \Uparrow (\mathsf{MultiPlot}(p, \mathsf{c_{sub}}), \bigwedge_i \psi_i)} \ \text{(Multi-Plot)}$$

$$\frac{\tau_i \in \mathsf{partitionByType}(\tau) \qquad \tau_i \Uparrow (l_i, \psi_i) \qquad i \in [1, n]}{\tau \Uparrow \left(\mathsf{MultiLayer}(\bar{l}_i), \bigwedge_{i=1}^{n} \psi_i\right)} \ \text{(Multi-Layer)}$$

$$\frac{\mathsf{T} = \bigcup_{i=1}^{n}(a_x^i, a_y^i, a_c^i, a_s^i, a_{sub}) \qquad \mathsf{c_x}, \mathsf{c_y}, \mathsf{c_{color}}, \mathsf{c_{size}}\ \text{fresh}}{\bigcup_{i=1}^{n} \left\{\mathsf{point}(a_x^i, a_y^i, a_c^i, a_s^i, a_{sub})\right\} \Uparrow \left(\mathsf{Scatter}(\mathsf{c_x}, \mathsf{c_y}, \mathsf{c_{color}}, \mathsf{c_{size}})\right), \mathsf{T} \overset{\diamond}{\subseteq} t} \ \text{(Scatter)}$$

$$\frac{\mathsf{T} = \bigcup_{i=1}^{n}(a_x^i, a_y^i, a_{y_2}^i, a_c^i, a_{sub}) \qquad \mathsf{c_x}, \mathsf{c_y}, \mathsf{c_{y_2}}, \mathsf{c_{color}}\ \text{fresh}}{\bigcup_{i=1}^{n} \left\{\mathsf{bar}(a_x^i, a_y^i, a_{y_2}^i, a_c^i, a_{sub})\right\} \Uparrow \left(\mathsf{Bar}(\mathsf{c_x}, \mathsf{c_y}, \mathsf{c_{y_2}}, \mathsf{c_{color}})\right), \mathsf{T} \overset{\diamond}{\subseteq} t} \ \text{(Simple Bar)}$$

$$\frac{\begin{array}{c}\mathsf{T} = \bigcup_{i=1}^{n}(a_x^i, a_{y_2}^i - a_y^i, a_c^i, a_{sub}) \qquad \mathsf{c_x}, \mathsf{c_h}, \mathsf{c_{color}}\ \text{fresh} \\ \psi_0 = \forall i \in [1, n].\ \sum_{r \in \{r \in \mathsf{T_{in}} \mid r[\mathsf{c_x}] = a_x^i \wedge r[\mathsf{c_{sub}}] = a_{sub} \wedge r[\mathsf{c_{color}}] < a_c^i\}} r[\mathsf{c_h}] = a_y^i\end{array}}{\bigcup_{i=1}^{n} \left\{\mathsf{bar}(a_x^i, a_y^i, a_{y_2}^i, a_c^i, a_{sub})\right\} \Uparrow \left(\mathsf{Stacked}(\mathsf{c_x}, \mathsf{c_h}, \mathsf{c_{color}}), \psi_0 \wedge \mathsf{T} \overset{\diamond}{\subseteq} t\right)} \ \text{(Stacked Bar)}$$

$$\frac{\begin{array}{c}\mathsf{T_1} = \bigcup_{i=1}^{n}(a_{x_1}^i, a_{y_1}^i, a_c^i, a_p) \quad \mathsf{T_2} = \bigcup_{i=1}^{n}(a_{x_2}^i, a_{y_2}^i, a_c^i, a_p) \quad \mathsf{c_x}, \mathsf{c_y}, \mathsf{c_{color}}\ \text{fresh} \\ \psi_0 = \forall i \in [1, n]. \nexists r \in \mathsf{T_{in}}.\ (r[\mathsf{c_{color}}] = a_c^i \wedge r[\mathsf{c_{sub}}] = a_p) \rightarrow a_{x_1}^i \leq r[\mathsf{c_x}] \leq a_{x_2}^i\end{array}}{\bigcup_{i=1}^{n} \left\{\mathsf{line}(a_{x_1}^i, a_{y_1}^i, a_{x_2}^i, a_{y_2}^i, a_c^i, a_p)\right\} \Uparrow \left(\mathsf{Line}(\mathsf{c_x}, \mathsf{c_y}, \mathsf{c_{color}}), \psi_0 \wedge \mathsf{T_1} \overset{\diamond}{\subseteq} \mathsf{T_{in}} \wedge \mathsf{T_2} \overset{\diamond}{\subseteq} t\right)} \ \text{(Line)}$$

Fig. 10. Inference rules describing synthesis of visual programs

has to satisfy all of these constraints (i.e., $\bigwedge_{i=1}^{n} \psi_i$). Finally, since we do not know which column of the input table is used to generate different subplots, we make up a fresh column name called $\mathsf{c_{sub}}$ and return the synthesized program $\mathsf{MultiPlot}(p, \mathsf{c_{sub}})$ as the solution.

Multiple layers. The second rule, Multi-Layer, is similar to the Multi-Plot rule and is used to generate programs that compose different types of charts. Similar to the previous rule, we again partition elements in the visual trace according to their type (i.e., point, bar etc.) to obtain n different traces $\tau_1, \ldots, \tau_n$ and recursively synthesize a visual program l_i and a constraint ψ_i for each τ_i. Then, the synthesized program $\mathsf{MultiLayer}(\bar{l})$ will generate a visualization consistent with the visual sketch as long as the input table satisfies $\bigwedge_{i=1}^{n} \psi_i$.

Scatter plot. The next rule is used to synthesize a visual program that renders a scatter plot. Since all elements in a scatter plot must be points, the precondition of this rule requires that the visual trace is a set of points with the same subplot attribute. Furthermore, for each point p with attributes $\bar{a}^i$ in the visual sketch, there must be a corresponding row in the input table that contains exactly the values $\bar{a}^i$. To express this requirement on the input table, we construct a table T that contains rows $\bar{a}^i$ and generate the constraint $\mathsf{T} \overset{\diamond}{\subseteq} t$ where t refers to the input table for the synthesized visual program. Finally, since we do not know the names of the columns in the input table, we introduce placeholder column names $\bar{c}$ and return the program $\mathsf{Scatter}(\bar{c})$.

Bar charts. The next two rules, labeled Simple Bar and Stacked Bar, both generate bar charts and are very similar to the previous Scatter rule. For Stacked Bar, $\mathsf{c_h}$ represents the height of the bar rather than the absolute y-position; thus, we compute entries in column $\mathsf{c_h}$ as $a_{y_2}^i - a_y^i$ for the i'th row in the table sketch. Also, in addition to the constraint $\mathsf{T} \overset{\diamond}{\subseteq} t$, the Stacked Bar rule imposes an additional constraint on the input table. In particular, since the bars in a Stacked Bar chart must be stacked directly on top of each other, constraint ψ_0 essentially stipulates that the starting y position of one bar is precisely the end y-position of the previous stack below it. In practice, when computing constraint ψ_0, we compute the end y position of the stack below by summing the heights of all the individual bars below the current one.

Line chart. The final rule is used to synthesize a Line program in our visualization language. Recall that a line visual element is defined by its two end points (a_{x_1}, a_{y_1}) and (a_{x_2}, a_{y_2}), and these end points must correspond to two different rows in the input table. Thus, we generate two different constraints $\mathsf{T}_1 \overset{\diamond}{\subseteq} t$ and $\mathsf{T}_2 \overset{\diamond}{\subseteq} t$ that describe requirements imposed by the left end and right end of each line segment respectively. Finally, the constraint ψ_0 in the second line of the premise imposes the following additional restriction: If the visual sketch contains a line segment with (a_x, a_y) and (a'_x, a'_y) as its end points, there should not be another entry in the input table that belongs to the same line chart (i.e., same color and subplot) but where the x value is in the range (a_x, a'_x). Without this additional constraint ψ_0, the generated visualization would not be guaranteed to satisfy the provided visual sketch.

Properties. Our visual program inference procedure enjoys the following properties that are important for the soundness and completeness for the overall approach.

PROPERTY 1 (DECOMPOSITION). *Suppose that $\tau \Uparrow (\mathsf{P_V}, \psi)$ and T is a table that satisfies constraint ψ (i.e., $T \models \psi$). Then, we have $\tau \subseteq \mathsf{P_V}(T)$.*

The above property shows the soundness of the overall the synthesis algorithm. In particular, let $\mathsf{P_V}$ be a visual program synthesized in the first phase. Based on the above property, as long as we can find a table transformation program $\mathsf{P_T}$ that satisfies the specification $(\mathsf{T_{in}}, \psi)$, then we are guaranteed that the composition $\mathsf{P_V} \circ \mathsf{P_T}$ will satisfy the specification of the overall synthesis task.

PROPERTY 2 (COMPLETENESS). *Let τ be a visual sketch, and suppose that there exists a table T and a visual program $\mathsf{P_V}$ in $\mathcal{L}_V$ such that $\tau \subseteq \mathsf{P_V}(T)$. Then, we have $\tau \Uparrow (\mathsf{P_V}, \psi)$ such that $T \models \psi$.*

This second property shows the completeness of the overall synthesis algorithm. In particular, it states that, if there exists a table T and visual program $\mathsf{P_V}$ such that $\mathsf{P_V}(\mathsf{T})$ is consistent with the given visual sketch, then our inference procedure will (a) return $\mathsf{P_V}$ as one of the solutions, and (b) T will satisfy the constraint ψ associated with $\mathsf{P_V}$.

4.3 Synthesizing Table Transformations via Bidirectional Reasoning

In this section, we describe the LEARNTABLETRANSFORM function used in Algorithm 1. This procedure is given in Algorithm 2 and takes as input the original input table $\mathsf{T_{in}}$ and the intermediate specification ψ_{out} generated during the first phase. LEARNTABLETRANSFORM either returns a program $\mathsf{P_T}$ such that $\mathsf{P_T}(\mathsf{T_{in}})$ is consistent with the specification ψ_{out} or yields $\bot$ to indicate failure. If synthesis is successful, LEARNTABLETRANSFORM additionally returns a mapping σ from the made-up column names used in ψ_{out} to the actual column names used in $\mathsf{P_T}(\mathsf{T_{in}})$.

From a high level, the outer loop of Algorithm 2 lazily enumerates program sketches based on the language from Section 3.2. In this context, a program sketch $\mathbb{P}$ is a sequence of instructions of the form $t = \mathtt{op}(\square_1, \ldots, \square_n)$ where t is a program variable, op is a construct in the table transformation language (e.g., `mutate`, `join`), and each $\square_i$ is a *hole* representing an unknown argument. To obtain a program that is a completion of $\mathbb{P}$, we need to fill each of the holes in the sketch with previously defined program variables or column names from the input table.

Algorithm 2 Table transformation synthesis algorithm.

```
1: procedure LearnTableTransform($\mathsf{T}_{\text{in}}, \psi_{\text{out}}$)
2:     $\psi_{\text{in}} \leftarrow (t_0 \subseteq \mathsf{T}_{\text{in}} \wedge \mathsf{T}_{\text{in}} \subseteq t_0)$
3:     while existsNextSketch() do
4:         $\mathbb{P}_0 \leftarrow$ getNextSketch();
5:         $W \leftarrow \{(\mathbb{P}_0, \sigma) \mid \sigma \in$ Mappings(Cols($\psi_{\text{out}}$), Cols($\mathbb{P}_0$))};
6:         while $\neg W$.isEmpty() do
7:             $(\mathbb{P}, \sigma) \leftarrow W$.next()
8:             if IsComplete($\mathbb{P}$) then
9:                 if $\mathbb{P}(\mathsf{T}_{\text{in}}) \models \psi_{\text{out}}[\sigma]$ then return $(\mathbb{P}, \sigma)$
10:                else continue;
11:            $\phi \leftarrow$ Analyze$^+(\psi_{\text{in}}, \mathbb{P}) \wedge$ Analyze$^-(\psi_{\text{out}}[\sigma], \mathbb{P})$;
12:            if UNSAT($\phi$) then continue
13:            $\Box_k \leftarrow$ chooseHole($\mathbb{P}$)
14:            $W \leftarrow W \cup \left\{(\mathbb{P}[\Box_k \mapsto v], \sigma') \,\middle|\, v \in \text{dom}(\Box_k), \text{Mappings}(\text{Cols}(\psi_{\text{out}}), \text{Cols}(\mathbb{P}) \cup \{v\}\right\}$
       return $\perp$
```

In more detail, the algorithm maintains a worklist W of elements $(\mathbb{P}, \sigma)$ where $\mathbb{P}$ is a (partially completed) program sketch and σ is a possible mapping from the made-up column names in ψ_{out} to actual column names in the output table. In particular, σ maps each column name used in ψ_{out} to an element in Cols($\mathbb{P}$), where Cols($\mathbb{P}$) includes both the columns used in T_{in} as well as any additional columns mentioned in $\mathbb{P}$. In each iteration of the inner while loop, the algorithm dequeues (at line 8) a pair $(\mathbb{P}, \sigma)$ and checks whether $\mathbb{P}$ is a complete program (i.e., no holes). If this is the case and $\mathbb{P}$ satisfies the specification under mapping σ (line 9), we then return $(\mathbb{P}, \sigma)$ as a solution. On the other hand, if $\mathbb{P}$ contains any remaining holes, we perform bidirectional program analysis (line 11) to check if there is any completion of $\mathbb{P}$ that *can* satisfy the (intermediate) specification ψ_{out}. In particular, line 11 of Algorithm 2 generates a constraint ϕ that is a conjunction of atomic predicates of the form $e_1 \subseteq^* e_2$ where $\subseteq^*$ represents the table inclusion relations defined earlier ($\subseteq$, $\overset{\diamond}{\subseteq}$), and each e_i is either a program variable or a concrete table. If these generated constraints result in a contradiction, there is *no* sketch completion that is consistent with ψ_{out}; thus, the algorithm moves on to the next element in the worklist (line 12). On the other hand, if we cannot prove the infeasibility of $\mathbb{P}$, we pick one of the holes $\Box_k$ used in the sketch and add a new set of partial programs to the worklist by instantiating that hole with some element in its domain (line 13). The domain of the hole is determined by its type, columns in the input schema for the given statement, and previously defined variables in the partial program. Since hole $\Box_k$ may have been filled with a new column name $v \notin$ Cols($\mathbb{P}$), we therefore also update the worklist to consider any new mappings σ' that we have not previously considered.

As is evident from the above discussion, a key part of our table transformation synthesis algorithm is the Analyze$^+$ and Analyze$^-$ procedures for performing forward and backward inference to generate table inclusion constraints. These procedures are described in Figure 11 and Figure 12 using inference rules of the form $\phi \downarrow s : \phi'$ (for the forward analysis) and $\phi \uparrow s : \phi'$ (for the backward analysis). The meaning of the judgment $\phi \downarrow s : \phi'$ is that, assuming ϕ holds *before* executing statement s, then ϕ' must hold *after* executing s. Similarly, $\phi \uparrow s : \phi'$ means that, if ϕ holds *after* executing s, then ϕ' must hold *before* s (i.e., ϕ' is a *necessary* precondition for ϕ but may not be *sufficient* to guarantee it). Since the inference rules shown in Figure 11 and Figure 12 follow from the semantics of the table transformation language, we do not explain them in detail. However,

$$\frac{\phi \Rightarrow t \subseteq^* \mathsf{T}}{\phi \downarrow t' = \mathsf{filter}(t, \_) : \phi \land (t' \subseteq^* \mathsf{T})} \qquad \frac{\phi \Rightarrow t \subseteq^* \mathsf{T}}{\phi \downarrow t' = \mathsf{select}(t, \_) : \phi \land (t' \subseteq^* \mathsf{T})}$$

$$\frac{\phi \Rightarrow t \subseteq^* \mathsf{T} \quad \mathsf{T}' = [\![\mathsf{mutate}(\mathsf{T}, c_t, op, \bar{c})]\!]}{\phi \downarrow t' = \mathsf{mutate}(t, c_t, op, \bar{c}) : \phi \land (t' \subseteq^* \mathsf{T}')} \qquad \frac{\phi \Rightarrow (t_1 \subseteq^* \mathsf{T}_1 \land t_2 \subseteq^* \mathsf{T}_2)}{\phi \downarrow t' = \mathsf{join}(t_1, t_2, \_) : \phi \land (t' \subseteq^* \mathsf{T}_1 \times \mathsf{T}_2)}$$

$$\frac{\phi \Rightarrow t \subseteq^* \mathsf{T} \quad \mathsf{T}' = [\![\mathsf{gather}(\mathsf{T}, \bar{c}_{id}, \bar{c}_{target})]\!]}{\phi \downarrow t' = \mathsf{gather}(t, \bar{c}_{id}, \bar{c}_{target}) : \phi \land (t' \subseteq^* \mathsf{T}')} \qquad \frac{\phi \Rightarrow t \subseteq^* \mathsf{T} \quad \mathsf{T}' = [\![\mathsf{spread}(\mathsf{T}, \bar{c}_{id}, c_{key}, c_{val})]\!]}{\phi \downarrow t' = \mathsf{spread}(t, \bar{c}_{id}, c_{key}, c_{val}) : \phi \land (t' \overset{\diamond}{\subseteq} \mathsf{T}')}$$

$$\frac{\forall i \in [1, n].\ \phi_{i-1} \downarrow t_i = e_i : \phi_i}{\phi_0 \downarrow \{t_1 = e_1; \ldots; t_n = e_n\} : \phi_n} \text{(Chain)}$$

Fig. 11. Forward inference. Operator "$\subseteq^*$" refers to either $\subseteq$ or $\overset{\diamond}{\subseteq}$. In cases where the premise of no rule matches, we have an implicit judgment $\phi \downarrow s : \phi$ to propagate the input constraint.

$$\frac{\phi \Rightarrow \mathsf{T} \overset{\diamond}{\subseteq} t'}{\phi \uparrow t' = \mathsf{filter}(t, \_) : \phi \land (\mathsf{T} \overset{\diamond}{\subseteq} t)} \qquad \frac{\phi \Rightarrow \mathsf{T} \overset{\diamond}{\subseteq} t' \quad \mathsf{T}' = \mathsf{T}[-c_{target}]}{\phi \uparrow t' = \mathsf{mutate}(t, c_{target}, \_, \_) : \phi \land (\mathsf{T}' \overset{\diamond}{\subseteq} t)}$$

$$\frac{\phi \Rightarrow \mathsf{T} \overset{\diamond}{\subseteq} t'}{\phi \uparrow t' = \mathsf{select}(t, \_) : \phi \land (\mathsf{T} \overset{\diamond}{\subseteq} t)} \qquad \frac{\phi \Rightarrow \mathsf{T} \overset{\diamond}{\subseteq} t' \quad \mathsf{T}' = [\![\mathsf{gather}(\mathsf{T}, \bar{c}_{id}, \mathsf{schema}(\mathsf{T}) - \{\bar{c}_{id}\})]\!]}{\phi \uparrow t' = \mathsf{spread}(t, \bar{c}_{id}, \_, \_) : \phi \land (\mathsf{T}' \overset{\diamond}{\subseteq} t)}$$

$$\frac{\phi \Rightarrow \mathsf{T} \overset{\diamond}{\subseteq} t' \quad \mathsf{T}' = \mathit{RemoveDuplicates}(\mathsf{T}[\bar{c}_{id}])}{\phi \uparrow t' = \mathsf{gather}(t, \bar{c}_{id}, \_) : \phi \land (\mathsf{T}' \overset{\diamond}{\subseteq} t)} \qquad \frac{\phi \Rightarrow \mathsf{T} \overset{\diamond}{\subseteq} t' \quad \mathsf{T}' = \mathsf{T}[-c_{target}]}{\phi \uparrow t' = \mathsf{summarize}(T, \_, \_, c_{target}) : \phi \land (\mathsf{T}' \overset{\diamond}{\subseteq} t)}$$

$$\frac{\forall i \in [1, n].\ \phi_i \uparrow t_i = e_i : \phi_{i-1}}{\phi_n \uparrow \{t_1 = e_1; \ldots; t_n = e_n\} : \phi_0} \text{(Chain)}$$

Fig. 12. Backward inference. Operator "$\subseteq^*$" refers to either $\subseteq$ or $\overset{\diamond}{\subseteq}$, and *RemoveDuplicates* removes duplicate tuples from the input table. As in the forward analysis, we assume an implicit rule $\phi \uparrow s : \phi$ that applies if none of the other premises are met.

a key design decision is that our analysis *on purpose* does not compute strongest post-conditions (for the forward analysis) or strongest necessary preconditions (for the backward analysis) in order to ensure that the cost of deductive reasoning does not overshadow its benefits. For example, in the reasoning rule for `summarize` in Figure 12, we over-approximate all aggregation functions as uninterpreted functions; thus the inferred pre-condition only requires that input table t include content from non-aggregated columns (T') in the output table t'. While a more precise analysis rule could consider the underlying semantics of different aggregation operators, this kind of reasoning would be prohibitively expensive [Wang et al. 2018a] and outweigh the benefits obtained from better pruning.

For the same reason, our procedure for checking satisfiability of table inclusion constraints (described in Figure 13) is also incomplete and intentionally over-approximates satisfiability. Thus, while the unsatisfiability of the generated constraints ensures the infeasibility of a given partially completed sketch, the converse is not true – that is, our deductive reasoning technique may fail to prove infeasibility of a sketch even though no valid completion exists.

$$\frac{\phi = e_1 \subseteq^* e_2 \land \phi_0}{\phi \Rightarrow e_1 \subseteq^* e_2} \qquad \frac{\phi \Rightarrow e_1 \subseteq^* e_2 \quad \phi \Rightarrow e_2 \subseteq^* e_3}{\phi \Rightarrow e_1 \subseteq^* e_3} \qquad \frac{\phi \Rightarrow e_1 \subseteq e_2}{\phi \Rightarrow e_1 \overset{\diamond}{\subseteq} e_2}$$

$$\frac{\phi \Rightarrow T_1 \subseteq^* t \quad \phi \Rightarrow t \subseteq^* T_2 \quad T_1 \not\subseteq^* T_2}{\phi \Rightarrow \bot}$$

Fig. 13. Inference rules for checking satisfiability of table inclusion constraints. Operator "$\subseteq^*$" refers to either $\subseteq$ or $\overset{\diamond}{\subseteq}$, and metavariable e refers to either a program variable t in ϕ or a concrete table T.

Properties. We end this section by describing some salient properties of Algorithm 2 that are important for the soundness and completeness of the end-to-end synthesis approach.

Property 3 (Forward Analysis). *Let $\mathbb{P}$ be a partially completed sketch with argument t and return parameter t'. Then, if $t = T_{in} \downarrow \mathbb{P} : \phi$ and $\phi \Rightarrow (t' \overset{\diamond}{\subseteq} T')$, then* any *completion* P *of* $\mathbb{P}$ *satisfies* $\mathsf{P}(T_{in}) \overset{\diamond}{\subseteq} T'$.

By design, our forward analysis rules exploits the fact that many table transformation operators are monotonic over the input. This property essentially captures the correctness of forward inference. In particular, it says that, if we deduce that the output of $\mathbb{P}$ on T_{in} is a sub-table of T', then this is true for every completion of $\mathbb{P}$. The following property states something similar for the backward analysis:

Property 4 (Backward Analysis). *Let ϕ be a constraint and $\mathbb{P}$ be a partially completed sketch with input parameter t. Then, if $\phi \uparrow \mathbb{P} : \phi'$ and $\phi' \Rightarrow (\mathsf{T}' \overset{\diamond}{\subseteq} t)$, then for any completion* P *of* $\mathbb{P}$ *and any input table* T *such that* $\mathsf{P}(\mathsf{T}) \models \phi$, *we have* $\mathsf{T}' \overset{\diamond}{\subseteq} \mathsf{T}$.

Similarly, our backward analysis rules by design conservatively propagate known values from the output to inputs. This property states that any conclusions reached by backward inference apply to all completions of $\mathbb{P}$. Finally, we can state the following property about the correctness of our pruning strategy:

Property 5 (Pruning Soundness). *Given a partially completed sketch $\mathbb{P}$, suppose we have $\psi \uparrow \mathbb{P} : \phi^-$ and $(t = \mathsf{T}_{in}) \downarrow \mathbb{P} : \phi^+$. Let P be a completion of $\mathbb{P}$ such that $P(T_{in}) \models \psi$, and let σ be the resulting valuation after executing P on T_{in}. Then, we have $\sigma \models \phi^+ \land \phi^-$.*

In other words, our pruning technique never rules out completions of $\mathbb{P}$ that actually satisfy the given specification $(\mathsf{T}_{\mathsf{in}}, \psi)$.

5 IMPLEMENTATION

We have implemented the proposed technique in a tool called Viser, which is written in Python. Viser takes two inputs, namely the original data source (which can consist of one or more tables) as well as a visual sketch. Currently, Viser requires the visual sketch to be expressed as a visual trace; however, with some additional engineering effort, it would be possible to integrate Viser with visual demonstration interfaces such as Lyra [Carr et al. 2014] or VisExemplar [Saket et al. 2017a] to automatically generate visual traces from the demonstration.

Extension to visualization Language. Viser can generate visual programs in Vega-Lite [Satyanarayan et al. 2017], ggplot2, and a subset of `Matplotlib`. To handle all of these libraries, our implementation supports a richer visualization DSL than the one given in Figure 8. In particular, Viser supports two additional visualization constructs, `AreaChart` and `StackedAreaChart`, which provide another mechanism for visualizing quantities that change over time. To support this richer visualization language, we also extend our visual trace language from Figure 5 with an element called `area`. Besides adding new constructors, we also extend existing constructors to take additional attributes as input. For example, the attribute a_{shape} allows specifying the shape associated with each point in a scatter plot. Another attribute, a_{order}, for line charts allows specifying a custom order instead of using the default x-axis value.

Extension to table transformation Language. The table transformation language used in our implementation extends Figure 6 with a few additional constructs inspired by commonly used operators in the `tidyverse` *R* package. For example, the table transnformation DSL in our implementation allows another construct called `separate` that is commonly used for table reshaping as well as a construct called `cumsum` for computing cumulative sum for a given column. Our implementation also allows a more expressive version of the `mutate` construct that supports a broader set of binary computations including arithmetic operations and string concatenation.

Multi-layered visualizations. To simplify presentation in the technical section, we assumed that a visual program takes a single table as input. However, in many visualization libraries (e.g., `ggplot2`), the semantics of the `MultiLayer`$(l_1, \ldots, l_n)$ construct is that each different nested visual program l_i operates on the i'th input table. To support these richer semantics, our implementation synthesizes multiple different table transformation programs for each layer. To achieve this goal, the inference procedure for visual programs generates n different intermediate specifications, one for each layer in the visual program, and we use the same table transformation procedure to synthesize n different programs.

Ranking. Following the Occam's razor principle, Viser explores programs in increasing order of program size up to a fixed bound K. In practice, to leverage the inherent parallelism of our algorithm, Viser uses multiple threads to search for solutions of different sizes and ranks programs according to their size.

6 EVALUATION

In this section, we evaluate the effectiveness of our approach on 83 real world visualization tasks collected from online forums and tutorials for advanced users. The goal of our evaluation is to examine the following research questions:

(1) Can Viser solve real-world visualization tasks based on small visual sketches?
(2) Does the decomposition of the synthesis task into two sub-problem improve synthesis efficiency?
(3) Are there any advantages to using our proposed table transformation algorithm compared to re-using an existing state-of-the-art technique?

6.1 Benchmarks

We evaluate Viser on 83 visualization benchmarks [3], 63 of which are collected from highly-reputed visualization tutorials for Excel [Duggirala 2019; E90E50 2019; Peltier 2019] and Vega-Lite [Vega-Lite 2019], and 20 of which are collected from the `ggplot2` sub-forum on StackOverflow. To collect these benchmarks, we went through a few hundred visualization examples and retained all tasks that conform to the following criteria:

(1) The target visualization is expressible in our language. (We note that more than 80% of these visualizations are expressible in our language and discuss the remaining ones in Section 6.6.)
(2) The example contains the actual input table.

[3] Available at https://chenglongwang.org/falx-project

(3) The task requires some form of table transformation to generate the intended visualization.
(4) There is a way to produce the target visualization based on information in the example.

We have these criteria because (1) tasks that cannot be achieved using our visualization language are out of scope for this work, (2) we need the raw data as an input to our tool, (3) we do not want to evaluate on trivial benchmarks, and (4) we need the target visualization to determine if our tool can produce the correct program.

Among our 83 benchmarks, 40 of them contain subplots or multi-layered charts. Furthermore, for most benchmarks, the original data source consists of a single table whose size ranges from 4×3 to 3686×9, with average size 100×10.

6.2 Key Results

To evaluate Viser on these benchmarks, we programmatically generated small visual sketches consisting of 4 randomly sampled visual elements per layer. Specifically, given a target visualization expressed in our visual trace language, we sampled 4 elements from the corresponding visual trace and used this as our visual sketch. While the number 4 is somewhat arbitrary, we believe that a visual sketch with four elements is small enough that it would not be too onerous for users to construct such a sketch.

Given these randomly generated visual sketches, we evaluated Viser using the following methodology. We fixed a time budget t, and we let Viser explore multiple visualization scripts consistent with the provided sketch within this time budget. Then, for a given value of t, we consider the benchmark to be solved if any of the programs explored by Viser within that time budget generates the intended visualization.

Time budget	# solved
1s	26
10s	49
60s	62
600s	70

Table 1. Summary of experimental results.

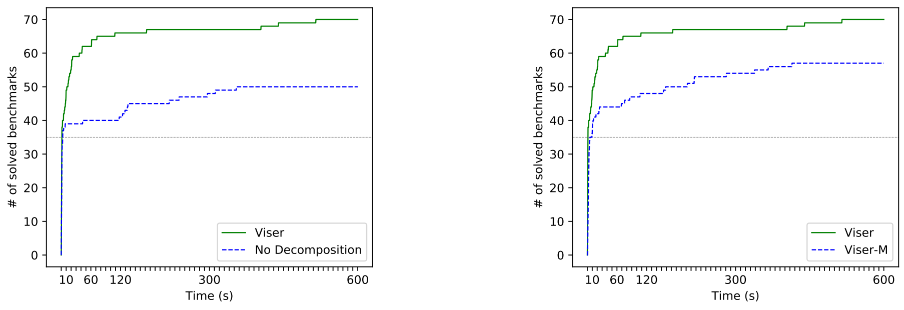
# samples	top-1	top-5	top-10	>10
1	14	29	36	66
2	20	37	43	68
3	22	45	53	69
4	26	49	57	70
6	31	52	57	70
8	30	58	63	70

Table 2. Impact of size of visual sketch on the ranking.

The results of this experiment are summarized in Table 1. For a time budget of 600 seconds, Viser is able to solve 70 out of 83 benchmarks (84%). If we reduce the time budget to a minute, then Viser can solve 75% of the benchmarks. Furthermore, 59% of the benchmarks can be solved within 10 seconds, and 31% can be solved within one second.

Table 2 explores the same experimental data from a different perspective. Specifically, given a value k, let us consider a benchmark to be "solved" if the desired visualization is one of the first-k visualizations returned by the tool. As shown in the first row of Table 2, among the 70 benchmarks that can be solved within the 600 second time limit, 26 (37%) of them are ranked as the top-1 solution, and 49 (70%) and 57 (81%) are ranked as top-5 and top-10 respectively. Given that a user can quickly look through 10 visualization results and decide if any of them is the desired visualization, we believe these results affirmatively answer our first research question.

In Table 2, we also explore the impact of sketch size on synthesis results. Specifically, recall that we generate the visual sketches by randomly sampling n elements from the target visualization, and, so far, our discussion focused on the results for $n = 4$. Table 2 shows the ranking of the desired



(a) Comparison against baseline with no decomposition (b) Comparison against Viser-M

Fig. 14. Comparison of Viser against different baselines

visualization as we increase n to 6 and 8 and decrease to 1, 2, 3 respectively. For cases with more visual trace samples, since the visual sketch contains more information as we increase n, Viser synthesizes fewer spurious programs and the ranking of the target visualization improves as a result. [4] This finding indicates that users can incrementally add more visual elements to the output example and gradually refine the synthesis results when the initial top-ranked solutions fail to meet the user's expectation.

6.3 Evaluating Impact of Decomposition

As mentioned throughout the paper, a key design choice underlying our technique is to decompose the visualization task by inferring an intermediate specification for each possible visual program. In this section, we aim evaluate the empirical significance of this decomposition.

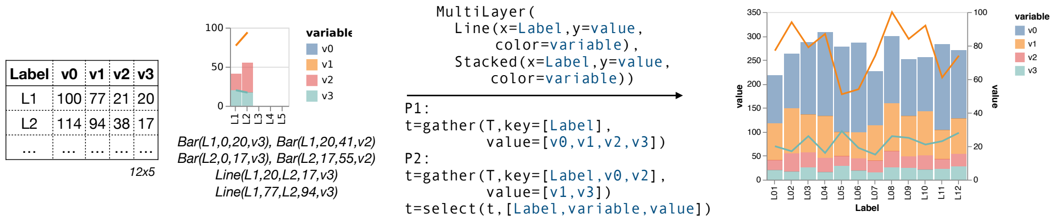
To perform this study, we implement a baseline using the following methodology: Similar to Viser, the baseline first infers a visual program $\mathsf{P_V}$ consistent with the sketch as discussed in Section 4.2; however, the baseline approach does not generate an intermediate specification. Then, during table transformation synthesis, for every enumerated program $\mathsf{P_T}$, the baseline checks whether $\mathsf{P_V}(\mathsf{P_T}(\mathsf{T_{in}}))$ is consistent with the visual sketch. In other words, without the intermediate specification, table transformation synthesis in the baseline approach degenerates into enumerative search.

Figure 14a compares the performance of Viser against this baseline without decomposition. Here, the x-axis shows the time budget per benchmark, and the y-axis shows the percentage of benchmarks that can be solved within the given budget. Furthermore, the solid green line corresponds to Viser, and the dashed blue line corresponds to the baseline without decomposition. As we can see from this figure, having an intermediate specification greatly benefits our synthesis algorithm. In particular, without decomposition, the percentage of benchmarks solved within a 600s (resp. 120s) time-limit drops from 84% (resp. 80%) to 60% (resp. 49%).

6.4 Evaluating Table Transformation Algorithm

In this section, we evaluate the impact of using our new table transformation algorithm over an existing technique that addresses the same problem. To perform this evaluation, we use a variant of Viser that we refer to ViserM that uses Morpheus [Feng et al. 2017] as its table transformation

[4]The reader may notice that $n = 6$ does better compared to $n = 8$ for the top-1 result; this is caused by the random sampling of visual elements.



Label	v0	v1	v2	v3
L1	100	77	21	20
L2	114	94	38	17
…	…	…	…	…

12x5

Bar(L1,0,20,v3), Bar(L1,20,41,v2)
Bar(L2,0,17,v3), Bar(L2,17,55,v2)
Line(L1,20,L2,17,v3)
Line(L1,77,L2,94,v3)

```
MultiLayer(
   Line(x=Label,y=value,
        color=variable),
   Stacked(x=Label,y=value,
        color=variable))
```

```
P1:
t=gather(T,key=[Label],
         value=[v0,v1,v2,v3])
P2:
t=gather(T,key=[Label,v0,v2],
         value=[v1,v3])
t=select(t,[Label,variable,value])
```

Fig. 15. Illustration of visualization task #1.

back-end. However, since the original Morpheus tool is written in C++, we instead use a newer implementation of Morpheus written in Python [Martins et al. 2019] (by the original Morpheus authors). Furthermore, since Morpheus does not support our table inclusion constraints, we "translate" the generated intermediate specification to Morpheus' constraint language. In particular, given an intermediate specification ϕ, our "translation" infers the strongest formula expressible in Morpheus' language, which consists of equality and inequality constraints on the number of rows or columns of the output table.

The results of this comparison are presented in Figure 14b, which plots the number of benchmarks that can be solved within a given time budget for both Viser and ViserM. As we can see from this figure, the table transformation synthesizer proposed in this paper yields much better results compared to Morpheus. In particular, within a 600s (resp. 120s) time-limit, ViserM can solve 69% (resp. 58%) of the benchmarks compared to 84% (resp. 80%) for Viser.

6.5 Example Tasks

To give the reader some intuition about the class of tasks that can be automated using Viser, we highlight three representative visualization tasks from our benchmark set.

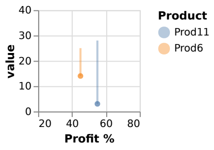
Task #1. Figure 15 shows a visualization task involving multiple layers, consisting of a stacked bar chart and a (multi-layered) line chart. The left-hand side of the figure shows the original data source, and right next to it, we show the visual sketch (and its corresponding visual trace) that we use to automate this visualization task. The synthesized visualization script is indicated on both sides of the arrow (visual program on top; table transformation at the bottom). Finally, the right-most part of the figure shows the resulting visualization that is obtained by applying the synthesized script to the input table.

Observe that the synthesized visual program refers to columns such as `variable` and `value` that do not exist in the original table and that are introduced by the table transformation program. Further, as discussed in section 5, Viser synthesizes as many table transformation programs as there are layers; thus, we have two separate table transformation programs for this example. The visualization shown on the right is one of the top-2 visualizations produced by Viser for this example.

Task #2. Figure 16 shows a scenario in which a user has data on corporate profits and wants to generate a so-called "cherry chart". Since most visualization libraries do not have a "cherry chart" primitive, generating this plot requires layering a line chart with a scatter plot. The left half of Figure 16 shows the input to Viser, and right half shows the synthesized programs and the corresponding visualization of the entire dataset. As in the previous example, we have multiple table transformation programs, one for each layer, and both the visual program and the table transformation programs refer to a column called `value` that does not exist in the original table and

Prod	Base	Inc	Profit
Prod1	61	70	132
Prod2	128	22	47
...	...	...	...

13x4



Line(Prod6,45,11,45,25)
Line(Prod11,55,25,55,28)
point(Prod6,45,11)
point(Prod11,55,25)

```
 MultiLayer(
    Line(x=Profit,y=value,
          color=Product),
    Scatter(x=Profit,y=value,
          color=Product))
```

```
P1:
t=mutate(T,Top,Base+Inc)
t=gather(t,key=[Prod,Profit],
          value=[Base,Top])
t=select(t,[Profit,value,Product])
P2:
t=gather(T,[Prod,Profit],[Base])
t=select(t,[Profit,value,Product])
```

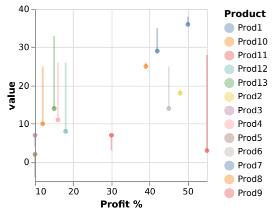
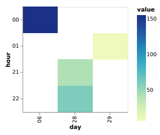


Fig. 16. Illustration of task #2.

DayHour	Site	Type	Value
06_00	e90e50	PC	22
06_00	e90e50fx	PC	8
06_01	e90e50	Mobile	8
...	...	...	...

1986x4



point(06,00,155)
point(28,21,42)
point(28,22,60)
point(29,01,8)

```
Scatter[rect](
    x=Day,y=Hour,
    color=sumValue)
```

```
t=group_by(T,[DayHour])
t=summarize(t,sum,Value)
t=separate(t,DayHour)
t=select(t,[Day,Hour,sumValue])
```

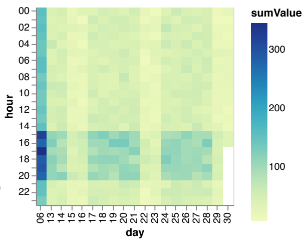


Fig. 17. Illustration of task #3.

that is introduced by the gather operation. In this case, the intended visualization shown on the right is top result returned by VISER.

Task #3. Figure 17 shows a visualization task that requires drawing a so-called "heat map" that visualizes the number of visits to various websites at each hour on different days. In this case, the input data is very large and contains close to 2000 rows. Furthermore, the corresponding data transformation program is quite complex and requires both computation as well as reshaping. Specifically, the table transformation program computes the total number of website visits for each one-hour time period, which is stored in a column called sumValue introduced by summarize. The visual program refers to this new sumValue column to generate the desired heat map. Given the visual trace shown in Figure 17, the intended visualization (on the right) is ranked within the top 20 visualizations, but if we provide a visual sketch with 8 elements instead of 4, then the intended visualization is ranked number one.

6.6 Discussion of Limitations

As reported in Section 6.2, VISER cannot synthesize 13 of the 83 benchmarks within a time limit of 10 minutes. To understand the limitations of VISER in practice, we manually inspected these benchmarks and explain the insights we gained from our examination. Specifically, we highlight two reasons that are responsible for VISER not finding the desired visualization within the given time budget.

- *Size of the input table.* The size of the input table can affect the performance of VISER in two ways. First, the search space covered by the table transformation language grows exponentially as we increase the size of the table. This is because constructs in the table transformation language use names of columns as arguments, and, furthermore, rows can become columns during the reshaping process. Second, the table transformation synthesis

engine tracks table inclusion constraints where one size of the inclusion is a concrete table. Thus, the larger the initial input table, the more overhead associated with program analysis.
- *Complex table transformations.* Some tasks in our benchmark suite require very sophisticated table transformations that Viser is unable to explore within the given time budget. For instance, some benchmarks that Viser cannot solve within 10 minutes require a combination of relational operations, string manipulation, column-wise arithmetic computations, and table pivoting.

In addition, recall from Section 6.1 that approximately 20% of the visualization tasks we inspected are not expressible in our visualization language. These benchmarks fall into roughly three classes: (1) visualizations involving continuous functions, (2) visualizations that require custom shapes provided by the user (e.g., emoji icons), and (3) visualizations that cannot be placed in a standard coordinate system, (e.g., tree-maps and parallel coordinates).

7 RELATED WORK

In this section, we survey closely related work on data visualization and program synthesis.

Automation for visualization. There has been significant recent interest in (semi-)automating various types of visualization tasks. These efforts include both visualization recommendation systems as well as visualization exploration tools. Among these, visualization recommendation systems like Draco [Moritz et al. 2019], CompassQL [Wongsuphasawat et al. 2016a], and ShowMe [Mackinlay et al. 2007] recommend top completions of an incomplete visualization program. On the other hand, visualization exploration tools, such as VisExamplar [Saket et al. 2017b], Visualization-by-Sketching [Schroeder and Keefe 2016], Polaris [Stolte et al. 2008], and Voyager [Wongsuphasawat et al. 2016b, 2017], aim to generate diverse visualizations based on user demonstrations, which can include graphical sketches, manipulation trajectories, and constraints. All of these existing systems focus on creating visualizations for a fixed dataset and require the user to prepare the data for a specific visualization API. In contrast, our approach also handles the data preparation and wrangling aspect of data visualization and can be viewed as being more user-friendly in this respect. However, it is worth noting that many of these systems are complementary to the approach proposed in this paper. For example, our approach can be used in conjunction with existing systems to rank synthesis results that are consistent with the demonstration. Furthermore, our approach can work with existing visualization demonstration interfaces [Saket et al. 2017b; Satyanarayan and Heer 2014] to reduce user effort in creating a visual sketch.

Automating table transformations. Our technique for synthesizing table transformations is related to several recent techniques for automating data wrangling [Feng et al. 2018, 2017; Harris and Gulwani 2011; Tran et al. 2009; Wang et al. 2017a; Zhang and Sun 2013]. Among these, Scythe generates SQL queries from input-output examples and prunes the search space by grouping partial queries into equivalence classes [Wang et al. 2017a]. The Morpheus system automates table transformation tasks that arise in R programming and leverages logical specifications of R library functions to prune the search space using SMT-based reasoning [Feng et al. 2017]. Morpheus's successor, Neo, generalizes this technique to other domains and further uses logical specifications to learn from failed synthesis attempts [Feng et al. 2018]. A unifying theme among all these prior efforts is that the specification is a pair of concrete input and output tables. In contrast, our specification does not involve a concrete output table but rather a set of table inclusion constraints; furthermore, our approach works with the original (potentially very large) dataset and does not require the user to craft a small representative input table. The large input table assumption is likely to be problematic for systems like Scythe and Morpheus that require evaluating the partial program on the input table. Furthermore, since the output specification is much weaker in our context compared to existing systems, forward reasoning alone is not sufficient to meaningfully prune the search space, as demonstrated in our evaluation.

Program analysis for program synthesis. Given the large search space that must be explored by program synthesizers, a common trick is to perform lightweight program analysis to prune the search space [Feser et al. 2015; Polikarpova et al. 2016; Wang et al. 2017b, 2018b]. The particular flavor of program analysis varies between different synthesizers and ranges from domain-specific deduction [Feser et al. 2015; Wang et al. 2017b] to abstract interpretation [Wang et al. 2018b] to SMT-based reasoning [Feng et al. 2018; Polikarpova et al. 2016]. Furthermore, some of these techniques leverage program analysis to construct a compact version space [Polozov and Gulwani 2015; Wang et al. 2018b] while others use it to prune partial programs in enumerative search [Feng et al. 2018; Wang et al. 2017b]. Similar to these efforts, we also use program analysis to prove infeasibility of partial programs but with two key differences: First, our analysis is tailored to table transformation programs and infers inclusion constraints between tables. Second, since neither forward nor backward reasoning is sufficient to meaningfully prune the search space on their own, we use bi-directional analysis to improve pruning power without having to resort to heavy-weight semantics motivated by prior work in program analysis [Chandra et al. 2009; Dhurjati et al. 2006; Reps et al. 1995] and verification [Wang et al. 2018a]. In this respect, our synthesis method is similar to Synquid [Polikarpova et al. 2016] which uses a form of bidirectional refinement type checking to prune its search space. However, unlike Synquid which requires precise refinement type specifications of components, our method uses lightweight semantics that are tailored specifically for our table transformation DSL. Furthermore, in contrast to Synquid which leverages an SMT solver, our method uses a custom, and deliberatively incomplete, solver for checking satisfiability at low cost.

Compositional program synthesis. As mentioned throughout the paper, our technique decomposes the synthesis task into two separate sub-problems. In this respect, our method is similar to prior efforts on compositional program synthesis [Feser et al. 2015; Maina et al. 2018; Miltner et al. 2018; Phothilimthana et al. 2016; Polikarpova et al. 2016; Polozov and Gulwani 2015; Raza et al. 2015]. Among these techniques, λ^2 uses domain knowledge about the DSL constructs to infer input-output examples for sub-expressions whenever feasible [Feser et al. 2015], FlashMeta (and its variants) use inverse semantics of DSL constructs to propagate examples backwards [Polozov and Gulwani 2015], and Optician [Maina et al. 2018; Miltner et al. 2018] decomposes the synthesis process using DNF regular expression outlines. Synquid also tries to decompose the overall specification into sub-goals using a technique referred to as "round-trip type checking" [Polikarpova et al. 2016]. On a slightly different note, the technique of Raza et al. [Raza et al. 2015] also performs synthesis in a compositional way, but it leverages natural language to identify sub-problems and asks the user to provide input-output examples for each auxiliary task. In contrast to all of these techniques, our method decomposes the visualization synthesis task into two sub-problems over *different* DSLs and uses the inverse semantics of the visualization DSL to infer precise constraints on the input table. The inferred specification is precise in the sense that any table transformation program that satisfies this specification is guaranteed to be a valid solution.

8 CONCLUSION

In this paper, we introduced *visualization-by-example*, a new program synthesis technique for generating visualizations from visual sketches. Given the original raw data and a visual sketch consisting of a few visual elements, our technique can automatically synthesize visualization scripts that yield a visualization consistent with the user's visual sketch. Our technique decomposes the synthesis problem into two sub-tasks by inferring an intermediate specification in the form of table inclusion constraints. This intermediate specification is then used to guide the synthesis of table transformation programs using a combination of bi-directional program analysis and lightweight inference over table inclusion constraints.

We have implemented the proposed method as a new tool called VISER that allows users to explore different visualizations for the entire data set based on a small visual sketch. Notably, and unlike any other visualization tool, VISER can perform any necessary data wrangling tasks, including reshaping and aggregation. We have evaluated VISER on a benchmark suite consisting of 83 visualization tasks obtained from on-line forums and tutorials. Given a visual sketch consisting of four visual elements and a time budget of 600 seconds, VISER can solve 84% of these tasks. Furthermore, among the 70 tasks that can be solved within the time budget, the desired visualization is ranked within top 5 in 76% of the cases. Beyond showing that VISER can help automate real-world data visualization tasks, our evaluation also confirms the importance of decomposing the synthesis task as well as the necessity of our proposed table transformation synthesizer.

In the near future, we are interested in integrating VISER with visual demonstration interfaces proposed in the visualization literature. Such interfaces can make VISER more user-friendly by providing a graphical user interface that allows users to draw visual sketches rather than write visual traces in a semi-formal language. We also plan to improve the search heuristics underlying VISER so that visualization scripts that generate the intended visualization are more likely to be explored first.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation through grants ACI OAC–1535191, IIS-1546083, IIS-1651489 and IISOAC-1739419; DARPA awards FA8750–14–C–0011, FA8750–16–2–0032 and FA8750-16-2-0032; the Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA NSF CCF-1723352); the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA CMU 1042741-394324 AM01; DOE award DE-SC0016260; and gifts from Adobe, Mozilla, Nokia, Qualcomm, Google, Huawei, and NVIDIA. We would also like to thank anonymous reviewers for their valuable comments on paper revising.

REFERENCES

Louie Andre. 2019. 20 Best Data Visualization Software Solutions of 2019. https://financesonline.com/data-visualization/. (2019).

Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D^3 Data-Driven Documents. *IEEE Trans. Vis. Comput. Graph.* 17, 12 (2011), 2301–2309. https://doi.org/10.1109/TVCG.2011.185

H Carr, P Rheingans, H Schumann, Arvind Satyanarayan, and Jeffrey Heer. 2014. Lyra: An Interactive Visualization Design Environment. In *Eurographics Conference on Visualization*, Vol. 33. 10.

Satish Chandra, Stephen J. Fink, and Manu Sridharan. 2009. Snugglebug: a powerful approach to weakest preconditions. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*. 363–374. https://doi.org/10.1145/1542476.1542517

Dinakar Dhurjati, Manuvir Das, and Yue Yang. 2006. Path-Sensitive Dataflow Analysis with Iterative Refinement. In *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*. 425–442. https://doi.org/10.1007/11823230_27

Purna Duggirala. 2019. *Chandoo.org Website*. https://chandoo.org/wp/category/visualization/

E90E50. 2019. *E90E50 Website*. https://sites.google.com/site/e90e50/

Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. 420–435.

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.

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 229–239. https://doi.org/10.1145/2737924.2737977

Pat Hanrahan. 2006. VizQL: a language for query, analysis and visualization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*. 721. https://doi.org/10.1145/1142473.1142560

William R Harris and Sumit Gulwani. 2011. Spreadsheet table transformations from examples. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 317–328.

Jock D. Mackinlay, Pat Hanrahan, and Chris Stolte. 2007. Show Me: Automatic Presentation for Visual Analysis. *IEEE Trans. Vis. Comput. Graph.* 13, 6 (2007), 1137–1144. https://doi.org/10.1109/TVCG.2007.70594

Solomon Maina, Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2018. Synthesizing quotient lenses. *PACMPL* 2, ICFP (2018), 80:1–80:29. https://doi.org/10.1145/3236775

Ruben Martins, Jia Chen, Yanju Chen, Yu Feng, and Isil Dillig. 2019. Trinity: An Extensible Synthesis Framework for Data Science. https://github.com/fredfeng/Trinity/. (2019).

Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2018. Synthesizing bijective lenses. *PACMPL* 2, POPL (2018), 1:1–1:30. https://doi.org/10.1145/3158089

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

Jon Peltier. 2019. *Peltiertech Website*. https://peltiertech.com/

Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodík, and Dinakar Dhurjati. 2016. Scaling up Superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*. 297–310. https://doi.org/10.1145/2872362.2872387

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. *Proc. Conference on Programming Language Design and Implementation* (2016), 522–538.

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*. 107–126. https://doi.org/10.1145/2814270.2814310

Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. 2015. Compositional Program Synthesis from Natural Language and Examples. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*. 792–800. http://ijcai.org/Abstract/15/117

Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*. 49–61. https://doi.org/10.1145/199448.199462

Bahador Saket, Hannah Kim, Eli T. Brown, and Alex Endert. 2017a. Visualization by Demonstration: An Interaction Paradigm for Visual Data Exploration. *IEEE Trans. Vis. Comput. Graph.* 23, 1 (2017), 331–340. https://doi.org/10.1109/TVCG.2016.2598839

Bahador Saket, Hannah Kim, Eli T. Brown, and Alex Endert. 2017b. Visualization by Demonstration: An Interaction Paradigm for Visual Data Exploration. *IEEE Trans. Vis. Comput. Graph.* 23, 1 (2017), 331–340. https://doi.org/10.1109/TVCG.2016.2598839

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

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

David Schroeder and Daniel F. Keefe. 2016. Visualization-by-Sketching: An Artist's Interface for Creating Multivariate Time-Varying Data Visualizations. *IEEE Trans. Vis. Comput. Graph.* 22, 1 (2016), 877–885. https://doi.org/10.1109/TVCG.2015.2467153

Chris Stolte, Diane Tang, and Pat Hanrahan. 2008. Polaris: a system for query, analysis, and visualization of multidimensional databases. *Commun. ACM* 51, 11 (2008), 75–84. https://doi.org/10.1145/1400214.1400234

Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. 2009. Query by output. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*. 535–548. https://doi.org/10.1145/1559845.1559902

Vega-Lite. 2019. *Vega-Lite Examples*. https://vega.github.io/vega-lite/examples/

Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017a. Synthesizing highly expressive SQL queries from input-output examples. In *Proc. Conference on Programming Language Design and Implementation*. ACM, 452–466.

Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2018a. Speeding up symbolic reasoning for relational queries. *PACMPL* 2, OOPSLA (2018), 157:1–157:25. https://doi.org/10.1145/3276527

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017b. Synthesis of data completion scripts using finite tree automata. *PACMPL* 1, OOPSLA (2017), 62:1–62:26. https://doi.org/10.1145/3133886

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2018b. Program Synthesis using Abstraction Refinement. In *Proc. Symposium on Principles of Programming Languages*. ACM, 63:1–63:30.

Kanit Wongsuphasawat, Dominik Moritz, Anushka Anand, Jock D. Mackinlay, Bill Howe, and Jeffrey Heer. 2016a. Towards a general-purpose query language for visualization recommendation. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. 4. https://doi.org/10.1145/2939502.2939506

Kanit Wongsuphasawat, Dominik Moritz, Anushka Anand, Jock D. Mackinlay, Bill Howe, and Jeffrey Heer. 2016b. Voyager: Exploratory Analysis via Faceted Browsing of Visualization Recommendations. *IEEE Trans. Vis. Comput. Graph.* 22, 1 (2016), 649–658. https://doi.org/10.1109/TVCG.2015.2467191

Kanit Wongsuphasawat, Zening Qu, Dominik Moritz, Riley Chang, Felix Ouk, Anushka Anand, Jock D. Mackinlay, Bill Howe, and Jeffrey Heer. 2017. Voyager 2: Augmenting Visual Analysis with Partial View Specifications. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems, Denver, CO, USA, May 06-11, 2017*. 2648–2659. https://doi.org/10.1145/3025453.3025768

Sai Zhang and Yuyin Sun. 2013. Automatically synthesizing sql queries from input-output examples. In *Proc. International Conference on Automated Software Engineering*. IEEE, 224–234.

A APPENDIX

A.1 Full Visualization Language

In this section, we present definitions of the full visualization language $\mathcal{L}_{\mathsf{V}}$ and the full trace language $\mathcal{L}_{\tau}$ we use in practice for the visualization by example task.

$$
\begin{array}{rcll}
\mathsf{P_V} & = & \mathsf{MultiPlot}(SP, c_{\mathsf{row}}, c_{\mathsf{col}}) \mid SP & \text{(Faceted Chart)} \\
SP & = & \mathsf{MultiLayer}(\bar{L}) \mid L & \text{(Layered Chart)} \\
L & = & \mathsf{Scatter}[mark](c_x, c_y, c_{shape}, c_{color}, c_{size}, c_{text}) & \text{(Scatter Plot)} \\
 & \mid & \mathsf{Line}(c_x, c_y, c_{width}, c_{order}, c_{color}, c_{detail}) & \text{(Line Chart)} \\
 & \mid & \mathsf{Bar}(c_x, c_{x_2}, c_y, c_{y_2}, c_{color}, c_{width}) & \text{(Bar Chart)} \\
 & \mid & \mathsf{StackedBar}[orient](c_x, c_h, c_{color}, c_{width}) & \text{(Stacked Bar Chart)} \\
 & \mid & \mathsf{Area}(c_x, c_{x_2}, c_y, c_{y_2}, c_{color}) & \text{(Area Chart)} \\
 & \mid & \mathsf{StackedArea}[orient](c_x, c_h, c_{color}) & \text{(Stacked Area Chart)} \\
c & = & column \mid \epsilon & \\
mark & = & \mathsf{point} \mid \mathsf{circle} \mid \mathsf{text} \mid \mathsf{rect} \mid \mathsf{tick} & \\
orient & = & \mathsf{horizontal} \mid \mathsf{vertical} &
\end{array}
$$

Fig. 18. The full visualization language $\mathcal{L}_{\mathsf{V}}$.

Figure 18 defines our full visualization language $\mathcal{L}_{\mathsf{V}}$. A visual program $\mathsf{P_V}$ either creates a grid of multiple plots using the MultiPlot construct or a single plot SP (where grid index for each subplot is determined by its value in column c_{col} and c_{row}). Each plot can in turn consist of multiple layers (indicated by the MultiLayer construct) or a single layer. Each layer is either a scatter plot (Scatter[*mark*], where the paramter *mark* decides the shape of scatter points), a line chart (Line), a bar chart (Bar), a stacked bar chart (StackedBar), an area chart (Area) or a stacked area chart (StackedArea). The MultiLayer construct in this language is used to compose *different* kinds of charts in the same plot (e.g., a scatter plot and a line chart), as our visualization language is already rich enough to allow layering the same type of chart within a plot.

Visual traces encode semantics of visualizations. A visual trace, denoted τ, is a set of basic visual elements, i.e., point, line, barH (horizontal bar), barV (vertical bar), or area, together with

$$
\begin{array}{rcll}
\tau & = & \{e_1, \ldots, e_n\} & \\
e & = & \mathsf{barV}(a_x, a_{y_1}, a_{y_2}, a_{width}, a_{color}, a_{col}, a_{row}) & \text{(Vertical Bar)} \\
 & | & \mathsf{barH}(a_y, a_{x_1}, a_{x_2}, a_{width}, a_{color}, a_{col}, a_{row}) & \text{(Horizontal Bar)} \\
 & | & \mathsf{point}(a_x, a_y, a_{shape}, a_{color}, a_{size}, a_{col}, a_{row}) & \\
 & | & \mathsf{line}(a_{x_1}, a_{y_1}, a_{x_2}, a_{y_2}, a_{width}, a_{color}, a_{col}, a_{row}) & \\
 & | & \mathsf{area}(a_{x_{tl}}, a_{y_{tl}}, a_{x_{bl}}, a_{y_{bl}}, a_{x_{tr}}, a_{y_{tr}}, a_{x_{br}}, a_{y_{br}}, a_{color}, a_{col}, a_{row}) & \\
\mathit{mark} & = & \mathsf{point} \mid \mathsf{circle} \mid \mathsf{text} \mid \mathsf{rect} \mid \mathsf{tick} &
\end{array}
$$

Fig. 19. The full trace language $\mathcal{L}_\tau$, where metavariable a refers to constants.

the attributes of each element (position, size, color, etc.). Figure 19 shows the language in which we express visual traces. Here, e denotes a visual element, and a is an *attribute* of that element:

- *Color attribute:* This attribute, denoted a_{color} specifies the color of a visual element.
- *Position attributes:* Position attributes, such as a_x, a_{x_1}, a_{y_2} etc., specify the canvas positions for a visual element. For line, (a_{x_1}, a_{y_1}) and (a_{x_2}, a_{y_2}) specifies the starting and the end points of a line segment. For the bar visual element, a_{y_1}, a_{y_2} specify the start and end y-coordinates of a (vertical) bar. Similarly, attributes $a_{x_{tl}}, a_{y_{tl}}, a_{x_{bl}}, a_{y_{bl}}, a_{x_{tr}}, a_{y_{tr}}, a_{x_{br}}, a_{y_{br}}$ specity x, y positions of top left / bottom left / top right / bottom right corners of an area element.
- *Size / Shape attributes:* Attributes a_{size} and a_{shape} specify the size and the shape variation of a given point element in a scatter plot.
- *Width attribute:* The attribute a_{width} specifies the width of a given barH / barV / Line element.
- *Subplot attribute:* Attributes a_{col}, a_{row} specify the subplot that a given visual element belongs to. For instance, a point with $a_{col} = 1$ and $a_{row} = 2$ belongs to the subplot located the first column and second row of visualization containing multiple plots.