reviz: A lightweight engine for reverse engineering data visualizations from the DOM

PARKER ZIEGLER, University of California, Berkeley, USA
LOGAN CARACO, University of California, Berkeley, USA

1 INTRODUCTION

For many data visualization developers, examples play a foundational role in their design process [6, 8]. Examples help demonstrate challenging techniques, reveal new possibilities for exploring a dataset, and serve as essential building blocks from which to get started. Examples also allow for rapid, iterative visualization sketching, in which users can see their data in many visualization contexts quickly. However, using examples to inform the design of new data visualizations relies on the source code for such examples being accessible. Reproducing closed source visualizations, like those published in journalistic outlets, requires painstaking reverse engineering. In these situations, data visualization developers rely on techniques like manual DOM inspection to reconstruct a target visualization by hand.

We introduce `reviz`, a lightweight engine for automatically reverse engineering data visualizations from the DOM. Given an arbitrary `svg` `Node`, `reviz` produces a partial JavaScript program (a program with holes) that can reproduce an input visualization when applied to a new dataset, a process we call *visualization retargeting*. We show that `reviz` can effectively infer visualization semantics through deductive search over `Sets` of DOM attributes and computed styles found in the `svg` subtree. We also demonstrate that `reviz` excels in automatically reverse-engineering both simple examples from the Plot library's documentation [4] and more complex real-world visualizations published in venues like the New York Times [10] and NPR [12].

2 IMPLEMENTATION

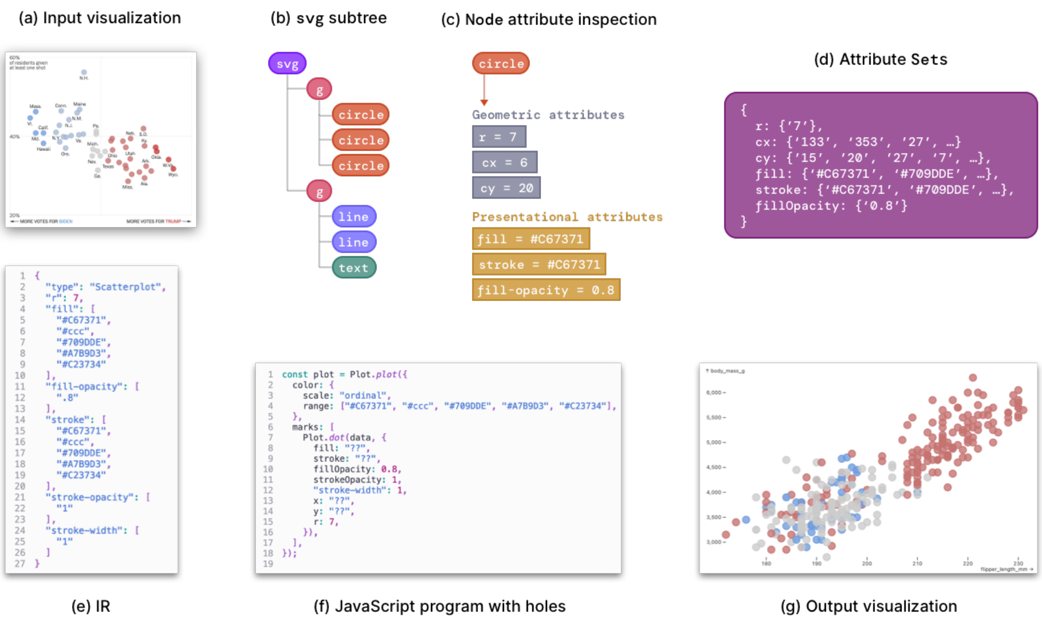
The core implementation of `reviz` can be decomposed into four steps:

(1) Walking the input `svg` subtree and aggregating DOM attributes and computed styles into attribute `Sets`.
(2) Using deductive search over these attribute `Sets` to infer a visualization type.
(3) Constructing an intermediate representation (IR) from the inferred visualization type and attribute `Sets`.
(4) Generating a partial JavaScript program using the `Plot` [5] library that, when applied to a new dataset, produces a visualization similar to the input.

Figure 1 provides a visual overview of these phases. We describe each phase in detail in the subsequent sections.

2.1 Walking the input svg subtree

`reviz` begins its analysis by walking the input `svg` subtree using the browser's native `document.createTreeWalker` API [1]. `document.createTreeWalker` returns a `TreeWalker` instance that provides methods for advancing through `Nodes` in the subtree. As `reviz` visits each `Node` it runs a series of callback functions that read a collection of pre-defined attributes off of the `Node` and its computed styles. Inspecting the `Node`'s computed styles ensures that attributes that may be inherited from a parent `Node` or defined in a separate CSS stylesheet, such as `fill` and `stroke`, are still collected. In addition, `reviz` maintains an array of `nodeNames` encountered in the sub-tree (e.g. `rect`, `text`, `g`, etc.).



```
{
  r: {'7'},
  cx: {'133', '353', '27', …}
  cy: {'15', '20', '27', '7', …},
  fill: {'#C67371', '#709DDE', …},
  stroke: {'#C67371', '#709DDE', …},
  fillOpacity: {'0.8'}
}
```

```
{
  "type": "Scatterplot",
  "r": 7,
  "fill": [
    "#C67371",
    "#ccc",
    "#709DDE",
    "#A7B9D3",
    "#C23734"
  ],
  "fill-opacity": [
    ".8"
  ],
  "stroke": [
    "#C67371",
    "#ccc",
    "#709DDE",
    "#A7B9D3",
    "#C23734"
  ],
  "stroke-opacity": [
    "1"
  ],
  "stroke-width": [
    "1"
  ]
}
```

```
const plot = Plot.plot({
  color: {
    scale: "ordinal",
    range: ["#C67371", "#ccc", "#709DDE", "#A7B9D3", "#C23734"],
  },
  marks: [
    Plot.dot(data, {
      fill: "??",
      stroke: "??",
      fillOpacity: 0.8,
      strokeOpacity: 1,
      "stroke-width": 1,
      x: "??",
      y: "??",
      r: 7,
    }),
  ],
});
```

Fig. 1. **A high-level overview of how reviz reverse-engineers and retargets data visualizations. The reviz architecture is somewhat akin to a traditional compiler, transforming svg subtrees into partial visualization programs. reviz uses deductive search over attribute Sets to infer a visualization type and produce its IR, which is then used to generate the partial visualization program. A user fills holes in this program (denoted by "??") to generate an output visualization (g).**

The callback functions push each `Node`'s attribute values into `Set`s. Using a `Set` data structure ensures that duplicate attribute values encountered on different `Node`s in the subtree are only encoded once in the intermediate representation (IR). The result of walking the entire subtree is an object mapping attribute names to the `Set` of unique values for that attribute encountered on anywhere in the subtree (see Step (d) in Figure 1).

2.2 Inferring the visualization type from attribute Sets

Once `reviz` has finished walking the `svg` subtree and generated attribute `Set`s for all attributes of interest, it begins running inference procedures on these `Set`s. These procedures also take into account the mode element type of the subtree, derived by finding the mode of the `nodeName` array maintained during the walk.

2.2.1 Restricting visualization types. To make implementation more tractable, we restrict the set of visualization types that `reviz` supports; similar restrictions are found in related work [13, 15]. Our supported visualization types are defined by the following set T:

$$T = \{\text{“}Scatterplot\text{”}, \text{“}BubbleChart\text{”}, \text{“}StripPlot\text{”}, \text{“}BarChart\text{”}, \text{“}StackedBarChart\text{”}, \text{“}Histogram\text{”}\}$$

2.2.2 Predicate functions over attribute `Set`*s.* For each visualization type $t \in T$, we define a *unique* set of predicate functions P that are associated with that visualization type. All predicate functions take in a collection of attribute `Set`s as input and return a Boolean value as output. If all predicate functions associated with a visualization type return `true` when applied to a collection of attribute `Set`s, we can infer that the input visualization is of that type. We define this formally as follows:

$$\begin{array}{rlrl}
\textbf{circle-based charts } c ::= & \text{“}Scatterplot\text{”} & \textbf{rect-based charts } r ::= & \text{“}BarChart\text{”} \\
\mid & \text{“}BubbleChart\text{”} & \mid & \text{“}StackedBarChart\text{”} \\
\mid & \text{“}StripPlot\text{”} & \mid & \text{“}Histogram\text{”}
\end{array}$$

$$\begin{array}{rll}
\textbf{Presentational attribute } p ::= & fill : [s_0, ..., s_n] & s \textit{ is valid CSS color} \\
\mid & fill - opacity : [n_0, ..., n_z] & n \geq 0 \wedge n \leq 1 \\
\mid & stroke : [s_0, ..., s_n] & s \textit{ is valid CSS color} \\
\mid & stroke - opacity : [n_0, ..., n_z] & n \geq 0 \wedge n \leq 1 \\
\mid & stroke - width : [s_0, ..., s_n] & s \textit{ is valid CSS unit}
\end{array}$$

$$\begin{array}{rlrl}
\textbf{Presentational attribute list } pl ::= & p & \textbf{Start } S ::= & \{type : c, r : n, pl\} \\
\mid & pl, p & \mid & \{type : r, width : n, pl\}
\end{array}$$

Fig. 2. **The grammar describing the encoding of the `reviz` IR. Metavariables s and n refer to string constants and numeric constants (any value encoded by the JavaScript `Number` type), respectively.**

> Let $P = \{p_1, ..., p_n\}$ represent the set of predicate functions associated with a visualization type $t \in T$. Let $A = \{a_r, a_{cx}, ..., a_{fill}, a_{stroke}, ...\}$ be the set of attribute `Set`s for an input visualization v. We then say that v is of type t iff $\forall p \in P \forall a \in A.p(a)$.

In practice, not all predicate functions are concerned with all attribute `Set`s; often, they are only concerned with one. For example, the predicate function `hasConsistentAttr` checks to see if the cardinality of an attribute `Set` a is 1; if it returns `true`, this suggests that all values for that attribute in the input visualization are the same. An example for which this predicate returns `true` is the attribute `Set` a_r for a scatterplot; all `circle` `Node`s in a scatterplot should, by definition, have the same value for their `r` attribute.

2.3 Constructing the intermediate representation (IR)

Once `reviz` has inferred the visualization type t for an input visualization v, it constructs an intermediate representation (IR) that encodes the minimum essential information to recover a semantically-consistent output visualization. This includes the *chart type*, *geometric attributes* that define geometries of marks, and *presentational attributes* that define the presentation of marks. We formalize the construction of our IR using the grammar given in Figure 2.

The IR is encoded using a JavaScript object. One of the key benefits of using a JavaScript object rather than an S-expression style syntax is that we obviate the need for an additional parser. `reviz` runs in the browser, so it's beneficial to use a data structure the browser can natively parse. Another benefit is that the IR can be statically typed by our implementation language, TypeScript [3].

2.4 Code generation

With the IR in hand, the final step in the `reviz` lifecycle involves generation of the output JavaScript program. This program produced by `reviz` uses the `Plot` library to handle low-level visualization details like defining scales, constructing data marks, and drawing axes. Plot exposes a *Grammar of Graphics*[16]-like API in JavaScript, making it an ideal library to target in code generation. We chose Plot rather than a JSON-based alternative like Vega-Lite [2] to take advantage of Plot's automatic scale inference. While Vega-Lite requires end users to specify the scale types of their

$$
\begin{array}{llll}
r ::= & \{type : \text{"}Scatterplot\text{"}, a\} & H ::= & \bullet \\
| & \{type : \text{"}BubbleChart\text{"}, a\} & | & \{H\} \\
| & \{type : \text{"}StripPlot\text{"}, a\} & | & Plot.dot(\text{"??"}, \{H\}) \\
| & \{type : \text{"}BarChart\text{"}, a\} & | & Plot.dotX(\text{"??"}, \{H\}) \\
| & \{type : \text{"}StackedBarChart\text{"}, a\} & | & Plot.barY(\text{"??"}, \{H\}) \\
| & \{type : \text{"}Histogram\text{"}, a\} & | & Plot.rectY(\text{"??"}, \{...Plot.binX(\{y : \text{"}count\text{"}\}), \{H\}\})
\end{array}
$$

Fig. 3. **The redexes (r) and contexts (H) of the contextual semantics defining how the `reviz` IR is evaluated to an incomplete JavaScript program. Non-reducible expressions include geometric and presentational attributes encoded in the IR, which fill the options context {H} but are not themselves reduced. Metavariable a refers to the rest of the IR object not under evaluation.**

$$
\begin{array}{l}
< \{type : \text{"}Scatterplot\text{"}\}, \sigma > \rightarrow < Plot.dot(\text{"??"}, \{\bullet\}), \sigma > \\
< \{type : \text{"}BubbleChart\text{"}\}, \sigma > \rightarrow < Plot.dot(\text{"??"}, \{\bullet\}), \sigma > \\
< \{type : \text{"}StripPlot\text{"}\}, \sigma > \rightarrow < Plot.dotX(\text{"??"}, \{\bullet\}), \sigma > \\
< \{type : \text{"}BarChart\text{"}\}, \sigma > \rightarrow < Plot.barY(\text{"??"}, \{\bullet\}), \sigma > \\
< \{type : \text{"}StackedBarChart\text{"}\}, \sigma > \rightarrow < Plot.barY(\text{"??"}, \{\bullet\}), \sigma > \\
< \{type : \text{"}Histogram\text{"}\}, \sigma > \rightarrow < Plot.rectY(\text{"??"}, \{...Plot.binX(\{y : \text{"}count\text{"}\}), \{\bullet\}\}), \sigma >
\end{array}
$$

Fig. 4. **Reduction rules defining redex evaluation. Visualization types in the `reviz` IR act almost as template generators for Plot API calls. Redex evaluation in `reviz` leads to the creation of a hole {•}, which becomes the next context of evaluation after the visualization type.**

dataset (e.g. `"quantitative"`, `"nominal"`, `"temporal"`, etc.), Plot infers them automatically. This greatly reduces the number of holes we leave unfilled in the generated program.

The Plot program generated by `reviz` is *incomplete*, with holes denoted by `"??"` in the output. Holes in a `reviz` program are placeholders that correspond to encoding channels; they represent a case where the value of a particular visual attribute (e.g. the `x` position of a `circle` in a scatterplot) is derived from a column in the input dataset rather than determined statically. Thus, holes need to be manually filled by a user with a column name from the dataset they wish to visualize. Future work could experiment with requesting a user's dataset ahead of time, allowing us to preemptively fill holes and generate recommended visualizations rather than incomplete programs.

2.4.1 A contextual semantics for `reviz`*'s code generation.* The code generation process for `reviz` involves iterative evaluation of key-value pairs in the IR to new portions of the output partial JavaScript program, which invokes APIs from the `Plot` library. We use a contextual semantics to formalize this evaluation. The reducible expressions (redexes) of the contextual semantics consist of the visualization types encoded in the IR grammar. Intuitively, evaluation from the IR to the output program starts with the `type` key and the empty context, •. One round of reduction and evaluation yields a program that invokes a top-level `Plot` API like `dot` or `barY`. From here, evaluation consists entirely of copying geometric and presentational attributes from the IR into the options object represented by the hole {•}. Figures 3 and 4 define the full semantics.

3 EVALUATION

We evaluated `reviz` against 6 example visualizations from the `Plot` documentation [4] and 4 example visualizations from journalistic outlets like the New York Times[10] and NPR [12]. Our evaluation sought to answer two questions:

(1) Given an input visualization, can `reviz` correctly infer its chart type?
(2) Given an input visualization and a new dataset, can `reviz` produce a visually similar output visualization?

While the first question is independently verifiable, the second entails some qualitative ambiguity. By "visually similar", we mean that the output visualization matches visual characteristics like color, opacity, and linework of the input visualization. Some differences in the input and output visualizations will be the product of differences in the datasets themselves. For example, our sample visualization from the New York Times uses an ordinal color scale with five different colors for `fill` and `stroke`. There is no guarantee that all five of these colors will be used by a new dataset applied to the program `reviz` generates; for example, the dataset may map `fill` and `stroke` to a column with more than five (or fewer than five) discrete values.

Acknowledging this ambiguity, we found that `reviz` correctly inferred the chart type for **nine out of ten** visualizations and produced visually similar output visualizations for **all ten** examples. The one case in which `reviz` failed to correctly infer the visualization type was the Dot Plot example [7] developed by Mike Bostock on Observable. In this instance, `reviz` inferred the chart type to be `"Scatterplot"` although it should have inferred `"StripPlot."`

4 RELATED WORK

Our work on `reviz` is most closely related to that of Harper and Agrawala[9], who designed a tool for interactively restyling D3 visualizations. `reviz` improves on their work by avoiding reliance on implementation details of D3 (e.g. the attachment of the `__data__` property to DOM nodes) to recover chart semantics. This allows `reviz` to run on any `svg`-based data visualization, not just those produced by D3. Additionally, Harper and Agrawala's work only allows the end user to *restyle* existing visualizations. Because `reviz` generates partial `programs`, end users can apply the program to their own data (*visualization retargeting*), run the program in their own codebase, or even extend the program with additional visualization functionality.

There has been significant work on reverse engineering data visualizations from bitmap images [11, 13, 14]. This work tends to employ deep learning approaches to recover chart semantics. For example, Poco and Heer rely on a convolutional neural network (CNN) to infer chart types, using this information to produce Vega-Lite specifications from input visualizations. Accuracy in chart type recognition is quite high in this domain, with both Rev [13] and ChartSense [11] reporting correct identification on >90% of input visualizations. However, these approaches come with a significant drawback—they don't run in the browser, where end users are most likely to encounter the types of visualizations they want to directly reverse-engineer. Conversely, `reviz` is designed to run directly in the browser, taking advantage of native DOM APIs to run efficiently. In fact, `reviz` can synthesize correct partial programs in <10ms from visualizations with up to 200 data elements. This is a promising result that shows `reviz`'s viability as a tool for rapid visualization sketching.

5 CONCLUSION

We presented `reviz`, a new tool for synthesizing partial visualization programs from `svg` subtrees. Our work on `reviz` suggests that the DOM can be mined as a rich source of information for inferring visualization semantics. However, we argue that a tool like `reviz` is more appropriate for rapid, exploratory visualization sketching than developing production-facing data visualizations. `reviz` does not currently handle many visualization types, nor does it support multilayered visualizations. We see these as important future research directions to make automated reverse engineering usable in everyday data visualization development. `reviz` is open source and available at: **https://github.com/parkerziegler/reviz**.

REFERENCES

[1] [n.d.]. Document.createTreeWalker() - Web APIs | MDN. https://developer.mozilla.org/en-US/docs/Web/API/Document/createTreeWalker

[2] [n.d.]. A High-Level Grammar of Interactive Graphics. https://vega.github.io/vega-lite/

[3] [n.d.]. JavaScript With Syntax For Types. https://www.typescriptlang.org/

[4] [n.d.]. Observable Plot / Observable / Observable. https://observablehq.com/@observablehq/plot

[5] 2021. Observable Plot. https://github.com/observablehq/plot original-date: 2020-10-29T22:30:31Z.

[6] Mike Bostock. [n.d.]. 10 Years of Open-Source Visualization / Mike Bostock / Observable. https://observablehq.com/@mbostock/10-years-of-open-source-visualization

[7] Mike Bostock. [n.d.]. Dot Plot / D3 / Observable. https://observablehq.com/@d3/dot-plot

[8] Mike Bostock. 2013. Eyeo 2013 - Mike Bostock. https://vimeo.com/69448223

[9] Jonathan Harper and Maneesh Agrawala. 2014. Deconstructing and restyling D3 visualizations. In *Proceedings of the 27th annual ACM symposium on User interface software and technology (UIST '14)*. Association for Computing Machinery, New York, NY, USA, 253–262. https://doi.org/10.1145/2642918.2647411

[10] Danielle Ivory, Lauren Leatherby, and Robert Gebeloff. 2021. Least Vaccinated U.S. Counties Have Something in Common: Trump Voters. *The New York Times* (April 2021). https://www.nytimes.com/interactive/2021/04/17/us/vaccine-hesitancy-politics.html

[11] Daekyoung Jung, Wonjae Kim, Hyunjoo Song, Jeong-in Hwang, Bongshin Lee, Bohyoung Kim, and Jinwook Seo. 2017. ChartSense: Interactive Data Extraction from Chart Images. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. Association for Computing Machinery, New York, NY, USA, 6706–6717. https://doi.org/10.1145/3025453.3025957

[12] Sean McMinn, Ruth Talbot, and Jess Eng. 2020. America's 200,000 COVID-19 Deaths: Small Cities And Towns Bear A Growing Share. *NPR* (Sept. 2020). https://www.npr.org/sections/health-shots/2020/09/22/914578634/americas-200-000-covid-19-deaths-small-cities-and-towns-bear-a-growing-share

[13] Jorge Poco and Jeffrey Heer. 2017. Reverse-Engineering Visualizations: Recovering Visual Encodings from Chart Images. *Computer Graphics Forum* 36, 3 (June 2017), 353–363. https://doi.org/10.1111/cgf.13193

[14] Manolis Savva, Nicholas Kong, Arti Chhajta, Li Fei-Fei, Maneesh Agrawala, and Jeffrey Heer. 2011. ReVision: automated classification, analysis and redesign of chart images. In *Proceedings of the 24th annual ACM symposium on User interface software and technology (UIST '11)*. Association for Computing Machinery, New York, NY, USA, 393–402. https://doi.org/10.1145/2047196.2047247

[15] Chenglong Wang, Yu Feng, Rastislav Bodik, Alvin Cheung, and Isil Dillig. 2019. Visualization by example. *Proceedings of the ACM on Programming Languages* 4, POPL (Dec. 2019), 49:1–49:28. https://doi.org/10.1145/3371117

[16] Leland Wilkinson. 2005. *The Grammar of Graphics (Statistics and Computing)*. Springer-Verlag, Berlin, Heidelberg.