reviz

A lightweight engine for reverse engineering data visualizations from the DOM

Parker Ziegler // EPIC Meeting // March 9, 2022
What is this talk about?

A deep dive into a new programming tool for data visualization authors
What is this talk (really) about?

The power of examples to act as intent specification mechanisms for data visualization authors
What is this talk (really) about?

How can reverse engineering “in-the-wild” examples augment existing techniques authoring data visualizations?
What is this talk (really) about?

How can reverse engineering “in-the-wild” examples augment existing techniques for data visualization author?
Roadmap

1. The **promise** of (and **problem** with) examples in data visualization

2. What is **reviz**? How does it address the **problem**?

3. **Questions** and **curiosities** for future work
1. The promise of (and problem with) examples in data visualization
“Examples inspire by showing what’s possible; examples demonstrate specific techniques; and examples are building blocks which help people get started.”

Mike Bostock

“10 Years of Open Source Visualization”
“Developers use different strategies in learning APIs ... Most developers prefer the strategy of looking at examples [12, 21, 28, 33, 36, 38, 40, 41, 45, 51, 52].”

Kyle Thayer, Sarah E. Chasins, Amy J. Ko
“A Theory of Robust API Knowledge”
TOCE '21: ACM Transactions on Computing Education
Examples abound in the data visualization community.

vega-lite has 189 examples on its website.
Examples abound in the data visualization community.

D3 has **363 examples** on its GitHub Wiki.
Examples abound in the data visualization community

Observable has thousands of user-built examples
Why are these examples effective?

“They’re very lightweight, they’re informal, they’re something you can do in 15 minutes. They don’t have the ceremony or the overhead of doing published graphics.”

Mike Bostock

“For Example”, EYE O Festival 2013
Why are these examples effective?

- **Colocation** of source code with the visualization
- **Immediate** interaction between source code and visualization
- Ability to see a visualization applied to your own data
Why are these examples effective?

Ability to see a visualization applied to your own data

“Does this visualization encoding show me something unexpected about my data?”

“Would an alternative encoding better represent what I want to show about my data?”

Explanatory

15
But what if the **code** powering an example *isn’t visible?*
But what if the **code** powering examples isn’t **visible**?

We *could* manually reverse engineer them.
But what if the code powering examples isn’t visible?

Could we automatically reverse engineer them?
1. The promise of (and problem with) examples in data visualization
Roadmap

1. The promise of (and problem with) examples in data visualization

2. What is reviz? How does it address the problem?
A “compiler” from SVG subtrees to data visualizations
A “compiler” from SVG subtrees to data visualizations partial programs in Observable Plot
How does it work?

A user identifies a visualization they want to use for retargeting.

How does it work?

1. They pass its svg subtree to reviz.
How does it work?

1. reviz generates a partial JavaScript program using Obervable’s Plot library.

```javascript
const plot = Plot.plot({
  color: {
    type: "categorical",
    range: ["#C67371", "#ccc", "#7698DE", "#A78903", "#C23734"],
  },
  marks: [
    Plot.dot(data, {
      x: "??",
      y: "??",
      r: 7,
      fill: "??",
      fillOpacity: 0.8,
      stroke: "??",
      strokeOpacity: 1,
      strokeWidth: 1,
    }),
  ],
});
```
How does it work?

1. 

2. 

3. 

The user **fills in the “holes”** in the partial program to get a retargeted visualization.
How does it work?

1. [Image 1]
2. [Image 2]
3. [Image 3]
4. [Image 4]
How does it work?

What **actually happens** in this step?
What actually happens in this step?

It all starts with a **walk**.
What actually happens in this step?

It all starts with a walk.

Visit each node in the svg subtree
What actually happens in this step?

It all starts with a walk.

Visit each node in the svg subtree
What actually happens in this step?

1. **Visit** each node in the **svg** subtree

   Read **geometric** and **presentational** attributes off of the node and its computed styles.
What actually happens in this step?

1. Visit each node in the `svg` subtree

   - **Read** geometric and **presentational** attributes off of the node and its computed styles.
What actually happens in this step?

1. **Visit** each node in the svg subtree

2. **Read** geometric and presentational attributes off of the node and its computed styles

---

**Store** attributes of all nodes in attribute sets

```json
{
  width: {'20'},
  height: {'10', '13', '27', '3', ...},
  x: {'133', '353', '27', ...},
  fill: {'#7b16ff'},
  stroke: {'none'},
  strokeOpacity: {'1'}
}
```
What actually happens in this step?

1. **Visit** each node in the **svg** subtree

2. **Read geometric** and **presentational** attributes off of the node and its computed styles

3. **Store** attributes of all nodes in **attribute sets**

Apply a set of **predicate functions** associated with a **visualization type**

```json
{
    type: "BarChart"
}
```

- hasMarkType('rect')
- hasConsistentGeomAttr('width')
- hasXScaleType('discrete')
What actually happens in this step?

4. Apply a set of predicate functions associated with a visualization type

Compute the “true rate” associated with a visualization type

```json
{ type: "BarChart" }
```

- hasMarkType('rect') ✓
- hasConsistentGeomAttr('width') ✓
- hasXScaleType('discrete') ✗

numPredicates = 3
truePredicates = 2
trueRate = 0.666
What actually happens in this step?

4. Apply a set of predicate functions associated with a visualization type

```
{ type: "BarChart" }
```

- hasMarkType('rect') ✓
- hasConsistentGeomAttr('width') ✓
- hasXScaleType('discrete') ✗

numPredicates = 3
truePredicates = 2
trueRate = 0.666

5. Compute the “true rate” associated with a visualization type

```
{ type: "Scatterplot" }
```

- hasMarkType('circle') ✗
- hasConsistentGeomAttr('r') ✗

numPredicates = 2
truePredicates = 0
trueRate = 0

```
{ type: "Histogram" }
```

- hasMarkType('rect') ✓
- hasConsistentGeomAttr('width') ✓
- hasXScaleType('continuous') ✓

numPredicates = 3
truePredicates = 3
trueRate = 1
What actually happens in this step?

4. Apply a set of predicate functions associated with a visualization type

```
{ type: "Histogram" }
```

- `hasMarkType('rect')` ✔️
- `hasConsistentGeomAttr('width')` ✔️
- `hasXScaleType('continuous')` ✔️

Select the visualization type with the highest "true rate"

5. Compute the "true rate" associated with a visualization type

```
{ type: "BarChart" }
```

- `hasMarkType('rect')` ✔️
- `hasConsistentGeomAttr('width')` ✔️
- `hasXScaleType('discrete')` ✗

```
{ type: "Scatterplot" }
```

- `hasMarkType('circle')` ✗
- `hasConsistentGeomAttr('r')` ✗
What actually happens in this step?

4. Apply a set of **predicate functions** associated with a **visualization type**

5. Compute the **true rate** associated with a **visualization type**

6. Select the **visualization type** with the highest **true rate**

Combine the **visualization type** and **attribute sets** into an IR

**Visualization type**
```json
{ type: "Histogram" }
```

**Attribute sets**
```json
{ 
  width: ['20'],
  height: ['10', '13', '27', '3', ...],
  x: ['133', '353', '27', ...],
  fill: ['#716ff'],
  stroke: ['none'],
  strokeOpacity: ['1']
}
```

**Intermediate Representation**
```json
{
  "type": "Histogram",
  "width": 20,
  "fill": [
    "#716ff"
  ],
  "fill-opacity": [1],
  "stroke": [
    "none"
  ],
  "stroke-opacity": [1],
  "stroke-width": [1]
}
```
Where are we now?

The **frontend** of the reviz compiler

```json
{
  "type": "Histogram",
  "width": 20,
  "fill": [
    "#7b18ff"
  ],
  "fill-opacity": ["1"],
  "stroke": [{"none"},
    "stroke-opacity": ["1"],
    "stroke-width": ["1px"]
}
```
What actually happens in this step?

Intermediate Representation

```json
{
    "type": "Histogram",
    "width": 20,
    "fill": [ 
        "#7b16ff"
    ],
    "fill-opacity": [ 
        "1"
    ],
    "stroke": [ 
        "none"
    ],
    "stroke-opacity": [ 
        "1"
    ],
    "stroke-width": [ 
        "1px"
    ]
}
```

Partial Observable Plot Program

The **backend** of the reviz compiler
Where are we now?

Intermediate Representation

Apply **contextual semantics** to rewrite terms in the IR to the output program

```
{  
  "type": "Histogram",  
  "width": 20,  
  "fill": [  
    "#7b18ff"  
  ],  
  "fill-opacity": [  
    "1"  
  ],  
  "stroke": [  
    "none"  
  ],  
  "stroke-opacity": [  
    "1"  
  ],  
  "stroke-width": [  
    "1px"  
  ]
}
```

```
'Plot.barY(data,  
  Plot.binX(  
    {  
      y: 'count',  
    },  
    {  
      x: '${PROGRAM_HOLE}',  
      ${EVAL_HOLE}  
    }  
  )
  
'${camelCase(attrName)}: ${val}:${isLastAttr ? '', '${EVAL_HOLE}' : '1}'
```
Where are we now?

Intermediate Representation

```json
{
    "type": "Histogram",
    "width": 20,
    "fill": [
        "#7b1fff"
    ],
    "fill-opacity": [
        "1"
    ],
    "stroke": [
        "none"
    ],
    "stroke-opacity": [
        "1"
    ],
    "stroke-width": [
        "1px"
    ]
}
```

Program

```
'${\text{EVAL_HOLE}}'
```

Start with an empty program, represented by a “hole”.

43
Where are we now?

Intermediate Representation

```
{
    "type": "Histogram",
    "width": 20,
    "fill": [
        "#7b18ff"
    ],
    "fill-opacity": [1],
    "stroke": [
        "none"
    ],
    "stroke-opacity": [1],
    "stroke-width": [1px]
}
```

Program

```
'Plot.barY(data,
    Plot.binX(
        
        y: 'count',
        
        x: '${PROGRAM_HOLE}',
        ${EVAL_HOLE}
    
    )
)
```

Apply the rewrite for the first key in the IR.

Leave a hole signaling where evaluation should continue.
Where are we now?

Intermediate Representation

```
{
  "type": "Histogram",
  "width": 20,
  "fill": [
    "#7b16ff"
  ],
  "fill-opacity": [
    "1"
  ],
  "stroke": [
    "none"
  ],
  "stroke-opacity": [
    "1"
  ],
  "stroke-width": [
    "1px"
  ]
}
```

Program

```
'Plot.barY(data,
  Plot.binX({
    y: 'count',
  },
  x: '${PROGRAM_HOLE}',
  fill: '#7b16ff',
  ${EVAL_HOLE}
})
```

- Keep applying rewrite rules for keys in the IR.
- Keeping leaving holes signaling where evaluation should continue.

45
Where are we now?

Intermediate Representation

```
{
    "type": "Histogram",
    "width": 20,
    "fill": [
        "#7b16ff"
    ],
    "fill-opacity": [
        "1"
    ],
    "stroke": [
        "none"
    ],
    "stroke-opacity": [
        "1"
    ],
    "stroke-width": [
        "1px"
    ]
}
```

Program

```
'Plot.barY(data, Plot.binX(
    {
        y: 'count',
    },
    {
        x: '${PROGRAM_HOLE}',
        fill: '#7b16ff',
        fillOpacity: 1,
        ${EVAL_HOLE}
    }
)}
```

Keep applying rewrite rules for keys in the IR.

Keeping leaving holes signaling where evaluation should continue.
Where are we now?

Intermediate Representation

```javascript
{
  "type": "Histogram",
  "width": 20,
  "fill": [
    "#7b16ff"
  ],
  "fill-opacity": [
    "1"
  ],
  "stroke": [
    "none"
  ],
  "stroke-opacity": [
    "1"
  ],
  "stroke-width": [
    "1px"
  ]
}
```

Program

```javascript
'Split.barY(data,
  Plot.binX(
    y: 'count',
    x: '${PROGRAM_HOLE}',
    fill: '#7b16ff',
    fillOpacity: 1,
    stroke: 'none',
    strokeOpacity: 1,
    strokeWidth: 1
  )
)
```
Where are we now?

The **frontend** of the reviz compiler

The **backend** of the reviz compiler

```json
{
    "type": "Histogram",
    "width": 20,
    "fill": [
        "#7b1fff"
    ],
    "fill-opacity": [1],
    "stroke": ["none"],
    "stroke-opacity": [1],
    "stroke-width": [1]
}
```

```
'Plot.barY(data,
    Plot.binX(
        {y: 'count'},
        {
            x: '${PROGRAM_HOLE}',
            fill: '#7b1fff',
            fillOpacity: 1,
            stroke: 'none',
            strokeOpacity: 1,
            strokeWidth: 1
        }
    )
)
```
How do we decide where to leave holes?

If there is variance of any kind in a visual variable, *assume it is because that variable is mapped to a column in the data.*
How do we decide where to leave holes?

If there is variance of any kind in a visual variable, assume it is because that variable is mapped to a column in the data.
How do we decide where to leave holes?

The harder case (in my opinion) is actually about making semantic judgements from limited information!
How do we decide where to leave holes?

The harder case (in my opinion) is actually about **making semantic judgements from limited information**!

---

```
stroke: [
  'rgb(249, 200, 175)',
  'rgb(250, 213, 193)',
  ...,  
  'rgb(249, 232, 222)'  
]
```

How do we **infer** that this is a **diverging** color scheme?

```
color: {
  type: "diverging",
  scheme: "BuRd",
  legend: true
},
stroke: "??"
```
Roadmap

1. The promise of (and problem with) examples in data visualization

2. What is reviz? How does it address the problem?

3. Questions and curiosities for future work
Roadmap

3. Questions and curiosities for future work
What is the right interaction model?

How do we make extracting the svg subtree as simple as possible?
What is the right interaction model?

How do we make **filling in holes** as simple as possible?
How can we scale support for more visualization types?

For each new visualization type, we need to define new predicates.

```json
{ type: "BarChart" }
hasMarkType('rect')
hasConsistentGeomAttr('width')
hasXScaleType('discrete')
```
How can we scale support for more visualization types?

For each **new visualization type**, we need to define **new predicates**.

```json
{ type: "BarChart" }

hasMarkType('rect')

hasConsistentGeomAttr('width')

hasXScaleType('discrete')
```

Could we instead **learn these predicates** by training a model on **svg subtrees**?

```json
{ type: "BarChart" }
```

![SVG subtree diagram]
Can we reverse engineer interactivity?

Can we detect interactive behavior like tooltip rendering on hover?

If so, how might we encode this in our IR?
Can we reverse engineer animation?

Can we detect animation like draw-on-render effects?

If so, how might we encode this in our IR?
Thank You!

Examples
- reviz.vercel.app
- observablehq.com/@parkerziegler/hello-reviz
- github.com/parkerziegler/reviz

Notebook

Source

reviz is a new, open-source library for automatically reverse engineering data visualizations from the DOM. You can think of it as a compiler that transforms raw data into visualizations. It's based on Observable.js, which itself is a powerful tool for creating web applications. When you use reviz, you provide a set of visualization options, and reviz generates the HTML, CSS, and JavaScript code for the resulting visualization. This makes it easy to integrate visualization features into your web projects.