Technical Details for Cicero

Anonymous Author(s)

ABSTRACT

This document is supplementary material to Submission 1823 "Cicero: A Declarative Grammar for Responsive Visualization" about technical details. This document first describes an extended version of Vega-Lite that we used as a rendering grammar for our Cicero compiler and outlines further technical details about our Cicero compiler to this extended version of Vega-Lite. Then, this document describes our prototype recommender for responsive visualization transformations.

1 AN EXTENDED VERSION OF VEGA-LITE

Currently, Vega-Lite [6] often makes it complicated or difficult to express common techniques used in many public-facing visualizations where responsive design is critical, limiting us to observe realistic use cases. We extended Vega-Lite [6] primarily to enhance the expressiveness for public-facing visualizations that include additional text elements and informational marks. For example, the latest version of Vega-Lite (5.3) is limited in expressing common strategies for communicative, narrative visualizations in the responsive context, including text wrapping, externalization of elements, complex labeling (e.g., mark labels with multiple data values and varying styles). Furthermore, it is complicated to declare mobile-specific strategies like label-mark serialization and callout lines for annotations to data points using Vega-Lite. Thus, our extended version of Vega-Lite (ExVL) makes it easier to express and render various techniques for communicative visualizations by providing simpler expressions. A ExVL specification is converted to Vega-Lite that handles the primary visualization rendering. ExVL additionally includes separate rendering modules for annotations, informational marks, and title elements that are not fully supported by Vega-Lite. The example cases in our gallery in supplementary material provide various ExVL specifications with descriptions.

An ExVL specification consists of `data`, `transform`, `layout`, `layer`, `nondata`, `interaction`, and `title` objects. A `data` object is an array of JSON-formatted data points, and a `transform` object conveys a set of global data transformations (similar to that in Vega-Lite). A `layout` object includes information about size (`width` and `height`), `composition` (single view, small multiples, and map), map projection details (e.g., translates, scale), axis designs, and `row` and `column` elements. `row` and `column` elements are inspired by Tableau's shelves design [7]. A user can declare at most two `row` and `column` elements each. The first and second `row` elements are converted to `row` and `y` encodings in Vega-Lite, respectively. For small multiples, a user can define a list of filter statements or data values for small multiples using `split` keyword.

A `layer` object is composed of `mark`, `text`, `tooltip`, and `transform` objects. A `mark` object states the mark type of the layer and its visual properties (e.g., color, size, stroke, etc) which are then converted to encoding channels or static mark properties in Vega-Lite. A `text` object contains a list of text elements associated to the current layer. A `text` element can be on one of the mark, axis, and legend. Visibility options for a text element includes externalized (with or without numbering), serialized (to the mark), and callout lines (or ticks). By setting width, a `text` element can be wrapped. Furthermore, we added various expressions for complex labeling (e.g., axis title with summary statistics). The `tooltip` object contains a list of data fields to include in a tooltip and its positioning options (at its triggered position or fixed at the bottom of screen). A layer may have its local data transformations (`transform`).

A `nondata` object contains a list of annotation and emphasis elements that are not associated to data objects (marks, axes, and mark labels). A `nondata` element can be either `text` (annotation) or `mark` (emphasis). They are manually positioned, and an annotation element can be externalized (with or without numbering). An `interaction` object includes definitions of user interactions. The types of supported user interactions include zoom and pan for map, context view, and interactive filtering, while tooltip is separately defined in each layer item).

2 CICERO COMPILER API FOR EXTENDED VEGA-LITE

Our Cicero compiler API provides an architecture for handling, compiling, and rendering the source view and Cicero specifications with a `Cicero` class and `loadCicero` function. Developers for a rendering grammar can attach a Cicero compiler and renderer (i.e., the compiler of the rendering grammar that actually draws the visualization) specific to that grammar. A `Cicero` class instance contains a source specification in any declarative rendering grammar, a Cicero specification, a transformation compiler that compiles the source and Cicero specifications to a target specification, and the compiler function (or equivalent) for the rendering grammar. Our API also provide the `loadCicero` function as a wrapper for creating a `Cicero` class instance, compiling the specifications, and rendering the transformed view. While we developed a Cicero compiler for an extended version of Vega-Lite, other rendering grammars like the default Vega-Lite [6] or ggplot2 [8] can be similarly developed. To fit our Compiler API, a Cicero compiler for a rendering grammar should take source view and Cicero specifications and return the transformed specification (not the rendered view), and a renderer should return a rendered view given a visualization specification and a DOM element (or selector) to draw the visualization in.

The `Cicero` class is instanced with `name` (the name of a transformed view or Cicero specification), `source` (the source view specification), `metadata` (a Cicero `metadata` object), and `description` (detailed description for the Cicero specification) as shown in line 3–4 of Figure 1. After instancing, users can add a list of Cicero `rule` objects using `addTransformations` method (line 7–8). Users can set a Cicero compiler and renderer for the rendering grammar using a `setCompiler` method (`CiceroToExVL` and `renderExVL` in line 10, respectively). To get the transformed specification in the rendering grammar, users can call a `getTransformed` method (line 12). To render the transformed view, users can use a `getRendered` method with a CSS selector for the HTML element (or DOM) to insert the rendered visualization in (line 14).

The `loadCicero` function makes the above job more pipelined. The `loadCicero` function takes a `ciceroSpec`, `source`, a Cicero compiler and renderer for the rendering grammar (line 18–19). Then, it returns a `Promise` object [1], and the `then` method of the returning `Promise` takes a callback function with a `Cicero` class instance as an argument (line 20–22).

```
1  // create a config object
2  let metadata = { condition: "small" };
3  // create a Cicero instance
4  let cicero = new Cicero("target-view", source,
5    metadata, "This is a Cicero Spec for X");
6  // add Cicero transformation rules
7  let rules = [ ... Cicero transformations ];
8  cicero.addTransforamtions(rules);
9  // set compilers for Cicero and rendering (ExVL)
10 cicero.setCompiler(CiceroToExVL, renderExVL);
11 // get the transformed specification
12 let transformed = cicero.getTransformed();
13 // render the transformed view
14 cicero.getRendered("#dom-element");
15
16 // Using loadCicero function
17 let ciceroSpec = { ... a Cicero specification };
18 loadCicero(ciceroSpec, source, CiceroToExVL,
19   renderExVL) // returns a Promise
20   .then(cicero => {
21     cicero.getRendered("#dom-element");
22   });
```

Figure 1: A Cicero Compiler API use cases. Line 1–14: instancing a `Cicero` class object. Line 16–22: pipelining the job using the `loadCicero` function.

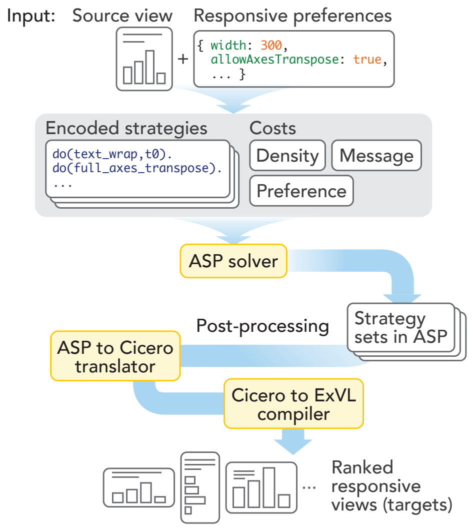
3 RECOMMENDER PROTOTYPE FOR RESPONSIVE VISUALIZATION

Below, we describe the pipeline of our prototype recommender and the strategies that we encoded.

3.1 pipeline

Input: Our prototype recommender (Figure 2A) takes as inputs the ExVL specification of a source view and user preferences for responsive designs. User preferences for our recommender include (1) the intended size of a responsive view, (2) hard constraints for transformation strategies, and (3) a subset of data that can be omitted or added. Users can specify hard constraints including whether to allow for transposing axes, changing encoding channels, modifying mark types, and altering the aspect ratio. For example, if 'allow for modifying mark type' is set as false, then recommendations with changes to mark type are ignored. One can specify a subset of data to omit or add using a filter statement.

Search Space Generation: Given the source view, users' preferences, and transformations strategies, our recommender generates sets of responsive strategies (strategy sets). The source view design



```
{ width: 300,
  allowAxesTranspose: true,
  ... }
```

```
do(text_wrap,t0).
do(full_axes_transpose).
...
```

Figure 2: The pipeline of our prototype recommender.

and preferences are converted to ASP expressions. We encoded a set of transformation strategies in ASP (see the next section). A transformation strategy can be applied or not applied when its condition is satisfied. For example, when the intended chart size is smaller than that of the source view, then our recommender may or may not apply a text-wrapping strategy to text element. We use Clingo [1, 2], a Python library, as our solver over these rules and constraints, similar to past uses of ASP for visualization recommenders, namely Draco [5].

Cost Evaluation: For each strategy set, our recommender assigns three types of costs (*density*, *message*, and *preference*) to the application or omission of each transformation strategy. Then, these costs are normalized to be in the same scale, and then aggregated as a final cost. First, for changes from bigger screen to smaller screen, if a strategy reduces the number of elements or spreads them on screen, it has a density cost of 1, otherwise 0. For instance, a strategy of removing every other axis labels for mobile views has a density cost of 1 while not applying that strategy has a density cost of 0. Similarly, for changes in the opposite direction, if a strategy increases the number of elements or gathers them on screen, it has a density cost of 1, otherwise 0. In this case, adding an axis label between each pair of existing labels has a density cost of 1.

Second, Kim et al. [4] propose five forms of changes to implied messages in visualization under responsive transformations: omitting or adding information and interaction, changes to the amount of concurrent information within a single scroll height, the discoverability of information (i.e., whether it is toggled), and changes to graphical perception (e.g., aspect ratio changes). We assign message cost of 1 for each transformation that cause such changes to visualization messages. For example, transposing an overly wide view to a longer visualization has a message cost of 1 because it reduces the amount of information that are concurrently visible within a single scroll height.

Third, to reflect preferences in applying responsive transformation, our prototype includes a preference cost according to their

[1] A type of JavaScript object that better enables asynchronous operation of functions. See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise for details.

popularity or frequency in use cases [3, 4]. Given a strategy, if it is commonly applied (more than 50% of the cases), applying it has preference cost of 2, otherwise 0. If it is less commonly applied (more than 10% of the cases), then it has preference cost of 1, otherwise 1, which makes it more random. If it is rarely used (less than 10% of the cases), then it has preference cost of 0, otherwise 2. For instance, disproportionate rescaling is highly common, so it has preference cost of 2, while it has message cost of 1 (as it causes changes to graphical perception). Users may change this preferences cost based on their own preferences like style guidelines of an organization. To prevent too many transformations, we assign a preference cost of $20 - \text{count}(strategies)$.

Post-processing: The ASP to Cicero translator converts each of the stable strategy sets generated by ASP solver into a Cicero specification. Then, our Cicero compiler for ExV produces ExVL specifications for target views.

3.2 Encoded strategies

We encode a diverse set of automatable strategies that we observed in our case studies with 13 realistic use cases and prior design pattern analyses [3, 4]. Strategies denoted by `M` are for desktop-to-mobile transformations, and those denoted by `D` are for mobile-to-desktop transformations. The non-prefixed strategies can be applied to both directions of transformation.

- **Changes to layout**
  - Transposing axes
  - Partial axes transpose (see the Aid Budget case in the paper)
  - Resizing the chart (proportionately or disproportionately)
- **Changes to data**
  - `M`-Omitting a specified subset of data
  - `D`-Filtering in a specified subset of data
- **Changes to mark properties and encoding channels**
  - Rescaling the size channel.
  - `M`-Removing detail encodings like image, color, and size
  - `M`-Changing the mark type (from bar, line, scatterplot to heatmap)
  - `M`-Changing small multiples to a heatmap
- **Changes to text elements**
  - `M`-Externalizing non-data/data annotations
  - `M`-Numbering externalized data annotations
  - `D`-Internalizing non-data/data annotations
  - `M`-Wrapping text elements
- **Changes to references**
  - Repositioning legends
  - `M`-Serializing axis labels
  - `D`-Parallelizing axis labels
  - `M`-Converting axis labels to legends
  - `M`-Removing every other axis labels
  - `D`-Adding every other axis labels
  - Adding ticks for mark labels
- **Changes to interaction**
  - `M`-Fixing the tooltip position
  - `D`-Unfixing the tooltip position
  - `M`-Removing tooltip
  - `D`-Adding tooltip
  - Removing a context view
  - Adding a context view (for time-serial visualizations)
  - Removing zoom
  - Adding zoom (for map visualizations)

REFERENCES

[1] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. 2014. Clingo = ASP + Control: Preliminary Report. (2014). arXiv:1405.3694 https://arxiv.org/abs/1405.3694.

[2] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. 2011. Potassco: The Potsdam Answer Set Solving Collection. *AI Commun.* (2011), 18 pages. https://doi.org/10.3233/AIC-2011-0491

[3] Jane Hoffswell, Wilmot Li, and Zhicheng Liu. 2020. Techniques for Flexible Responsive Visualization Design. In *ACM Human Factors in Computing Systems (CHI)*. https://doi.org/10.1145/3313831.3376777

[4] Hyeok Kim, Dominik Mortiz, and Jessica Hullman. 2021. Design Patterns and Trade-Offs in Authoring Communication-Oriented Responsive Visualization. *Computer Graphics Forum (Proc. EuroVis)* 40 (2021), 00–00. Issue 3. https://doi.org/10.1111/cgf.14321

[5] Dominik Moritz, Chenglong Wang, Gregory 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. Visualization & Comp. Graphics (Proc. InfoVis)* (2019). https://doi.org/10.1109/TVCG.2018.2865240

[6] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2017). https://doi.org/10.1109/TVCG.2016.2599030

[7] Tableau Software. 2003. Tableu. https://www.tableau.com/ Last accessed Aug 15, 2021.

[8] Hadley Wickham. 2010. A Layered Grammar of Graphics. *Journal of Computational and Graphical Statistics* 19, 1 (2010), 3–28. https://doi.org/10.1198/jcgs.2009.07098