Understanding and Enhancing JSON-based DSLs for Visualization

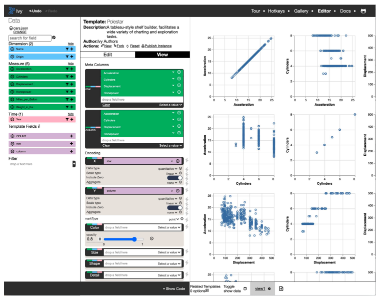
Andrew M. McNutt
PhD Proposal

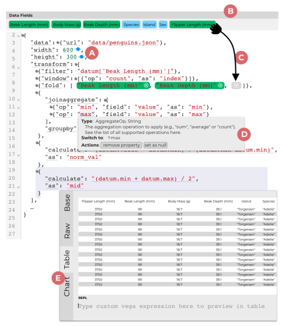
Feburary 10, 2022

Domain Analysis
Domain Motivation
Contextual Motivation

DSL Design
Conceptual Model
Language Composition
Abstraction Mechanisms
Syntax Design

DSL Implementation
Execution Strategy
Interactivity
Relationship with context
Relationship with data



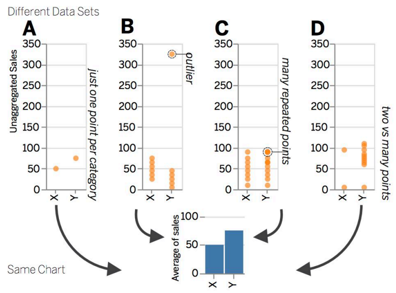


```
{
  "data":{"url": "data/penguins.json"},
  "width": 600,
  "height": 300,
  "transform":[
   {"filter": "datum['Beak Length (mm)']"},
   {"window":[{"op": "count", "as": "index"}]},
   {"fold": [ "Beak Length (mm)", "Beak Depth (mm)", ]},
   {
     "joinaggregate":[
      {"op": "min", "field": "value", "as": "min"},
      {"op": "max", "field": "value", "as": "max"}
     ],
     "groupby"
   },
   {
     "calculate"                                              um.min)",
     "as": "norm_val"
   },
   {
     "calculate": "(datum.min + datum.max) / 2",
     "as": "mid"
   }
  ],
  ...
}
```

Type: AggregateOp, String
The aggregation operation to apply (e.g., "sum", "average" or "count").
See the list of all supported operations here.
Switch to: max
Actions: remove property | set as null

Flipper Length (mm)	Beak Length (mm)	Body Mass (g)	Beak Depth (mm)	Island	Species
3750	181	18.7	39.1	"Torgersen"	"Adelie"
3750	181	18.7	39.1	"Torgersen"	"Adelie"
3750	181	18.7	39.1	"Torgersen"	"Adelie"
3750	181	18.7	39.1	"Torgersen"	"Adelie"
3750	181	18.7	39.1	"Torgersen"	"Adelie"
3750	181	18.7	39.1	"Torgersen"	"Adelie"
3750	181	18.7	39.1	"Torgersen"	"Adelie"
3750	181	18.7	39.1	"Torgersen"	"Adelie"
3750	181	18.7	39.1	"Torgersen"	"Adelie"
3750	181	18.7	39.1	"Torgersen"	"Adelie"
3750	181	18.7	39.1	"Torgersen"	"Adelie"
3750	181	18.7	39.1	"Torgersen"	"Adelie"
3750	181	18.7	39.1	"Torgersen"	"Adelie"
3750	181	18.7	39.1	"Torgersen"	"Adelie"
3750	181	18.7	39.1	"Torgersen"	"Adelie"
3750	181	18.7	39.1	"Torgersen"	"Adelie"

Type custom vega expression here to preview in table



Sec 2: Design
No Grammar
Est April 22

Sec 3: Abstraction
Ivy - CHI21
Completed

Sec 4: Manipulation
JSONG
Est December 22

Sec 5: Validation
Mirages - CHI20
Completed

Abstract— Static declarative domain specific languages represented by JSON are an increasingly common means by which to control a wide variety of types of systems. These range from database queries to application configuration to narrative generation to twitter bots to data visualization and to many other areas. These languages allow human users to concisely specify their intent through logic—and sometimes notation—that is relevant and matched to their task domain, as well as provide a means for computational agents to easily manipulate that form, allowing for powerful recommendation engines and automated analyses.

In this thesis we explore the design space of this form of textual interfaces and describe a suite of tools that improve the end user experience of these powerful tools. In particular, we investigate this space through four projects which variously consider how JSON DSLs are *designed*, how *abstraction* can be integrated into those languages, how interfaces can be designed to specifically facilitate their *manipulation*, as well as how those programs might be automatically *validated*. The through line of these projects is the assertion that treating these textual interfaces as first-class elements of visualization interface design is a valuable choice for end users. We primarily consider languages focused on various data visualization tasks, as there has been substantial work in the visualization research community on this form of interface—although the lessons learned could be applied to any relevant domain. Our initial findings suggest that there is substantial useful work that can be done through this lens and that interventions of the forms described are useful for helping end users learn, use, and re-use programs written in these languages.

1 Introduction

Data visualizations come in an endless array of shapes, sizes, forms, and kinds. Each type attuned to different problems, tasks and contexts. For instance, a Gantt chart can effectively show project planning data, but would do little to support the analysis of stock performance over time. Similarly, the means by which one specifies the attributes and form of a visualization varies based on the task at hand; with some interfaces being better attuned to certain situations than others. To wit, chart choosers (such as those found in Excel) are great for quickly getting a chart put together, but can be bad for data exploration or refining that graphic for presentation.

An increasingly common form of visualization specification is the use of JSON-based domain specific languages (DSLs). These static textual interfaces allow for the declarative specification of both static and interactive graphics in a logically coherent manner that is manipulable by both humans and computational agents. While they exist in a variety of forms and formats they often are inspired by the Grammar of Graphics [132]—a logic for specifying graphics via declaration of the mapping of data elements to graphical elements—which yields a simple and relatively human-readable form which is generally well matched to restricted grammar of JSON —as opposed to ones based on chart types such as scatter plots of bar charts. The most prominent of these tools are those in the Vega family [107, 108], however, as we discuss in Sec. 2, they are far from the only entrants in this paradigm.

The utility of these tools for visualization is closely analogous to that of SQL for declarative interaction with databases. End users can (with training) build powerful analyses through SQL queries, while computational agents can automatically generate queries through ORMs, provide exploration recommendations, and facilitate tasks such as data discovery. Reciprocally, end users can be trained to effectively use tools such as Vega-Lite [107] to explore and create visualizations in real world settings (for instance, Vega-Lite is at the heart of the Kibana analytics system for ElasticSearch [55]), while recommenders and other analysis systems can be used to automatically generate appropriate charts.

The representation of domain-specific programs in a static medium, such as JSON, has a variety of benefits. They are *portable* and can be used across a variety of platforms, allowing visualization created in one environment (such as a graphical editor like Lyra [105, 141]) to be utilized in another context (such as a textual interface like the python-based altair [121]). Because the languages are limited in what they can express they have a greater degree of *security* than might be found in visualizations described in general purpose programming languages. For instance, the MediaWiki Graph extension [131] enables the use of Vega on Wikipedia. This static representation provides all the

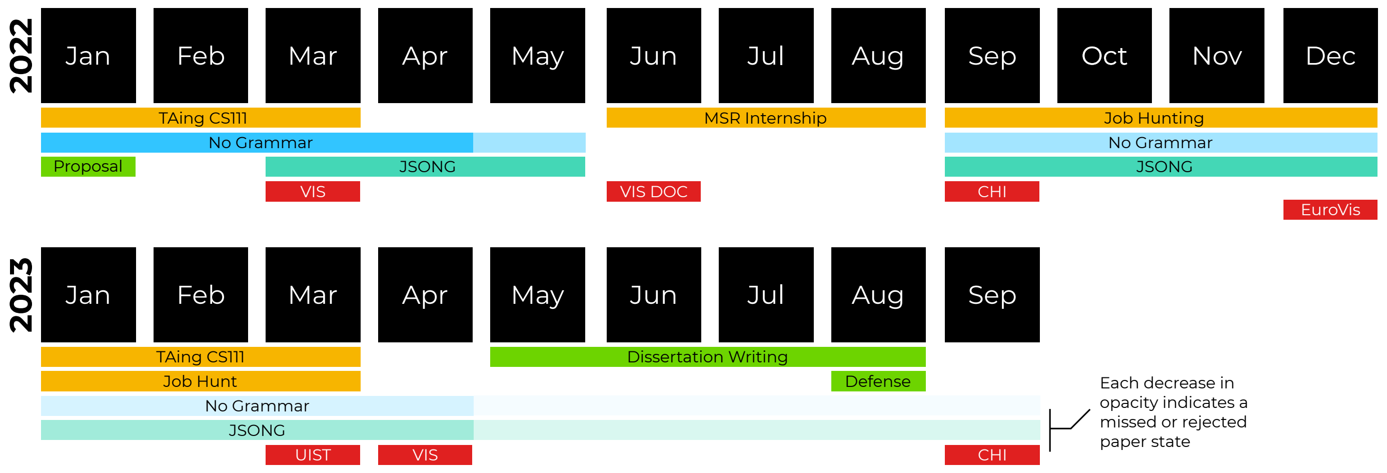


Fig. 1. A proposed timeline for completing the work described in this proposal. Describes dissertation events, project focused periods (JSONG and No Grammar specifically), paper deadlines relevant to those projects, and other major tasks. The deadlines denote conferences, with the exception of `VIS DOC` which refers to the IEEEVIS Doctoral Colloquium.

flexibility of a full visualization language without the issues associated with sending end-user editable executed code to billions of users.

While there has been ample work on *using* these visualization JSON-DSLs for various ends, such as recommendation [136] and programming by demonstration [104, 141], there has been little work on improving the human experience of their usage. The primary exception being Hoffswell et al.'s work on debugging Vega charts [46, 47]. In this thesis we will explore the design and usage of these languages both unto themselves and as end-user tools. Using the lessons learned therein we develop a series of systems which utilize and exhibit the unique advantages and disadvantages of this interface form. The essential hypothesis being that consideration of JSON-DSLs as first class interface elements is a valuable design choice for end users.

More formally, we will seek to defend the following thesis:

> Mixed-modality UIs can make JSON-based visualization DSLs more powerful and versatile in their design, as well as easier to use and reuse in end-user systems.

To examine this perspective we will consider four interconnected projects. Each of these will investigate a different aspect of JSON-DSLs. We will begin, in Sec. 2, by trying to understand just what JSON-based DSLs are, by surveying the current state of the art for their design and implementation patterns. Then, in Sec. 3, we will describe an abstraction layer over JSON grammars which can be coherently manipulated by a variety of end users (and hence through a variety of interface modalities) in a visual analytics system called Ivy. While this system demonstrates that the use of mixed-modality interfaces can be useful, we find that editing the textual representation of JSON-DSL to be sometimes error prone or hard to learn. To address these concerns in Sec. 4 we describe a GUI , tentatively called *JSONG* (JSON + GUI), that complements and extends traditional textual editing with direct manipulation, live program visualizations, and structure editing. Finally in Sec. 5 we will describe a method for validation of charts created through the use of these grammars, which we will situate within a larger framework for describing how visualizations can go wrong.

Through these projects we will demonstrate JSON-DSLs can be more effectively used through mixed-modality interfaces (which blend graphical and textual specification) and these abstraction and analysis tools (which allow for reuse and improved usage than without these augmentations).

As suggested throughout this introduction we will primarily focus on languages in the visualization landscape, however there are a multitude of other domains in which JSON-DSLs are utilized. Some of the more familiar applications involve application configuration (such as that of webpack [128]) and NoSQL query languages (such as MongoDB [8] or ElasticSearch [54]), however they have also been used for narrative generation [15, 35], game generation [27], chatbots [62], dance [93], fabrication [118], and a variety of others. We will explicitly touch on some of these areas, however we believe that the general lessons learned from these projects and the philosophy of end-user empowerment through static DSLs is one that can be applied to many domains.

1.1 Plan to completion

Throughout this document we will describe the intended path to completion for each of the sub-projects, but we also give an overview of that plan in Fig. 1. As described above and in Fig. there are four components to this thesis. Two of them (*Ivy* and *Mirages*) are completed. At time of this writing the remaining two projects, *No Grammar* and *JSONG*, are approximately 50% and 10% completed respectively. We intend to submit a preliminary version of this work to the IEEEVIS Doctoral Colloquium. Should there be extra time following the completion of the two remaining projects we will pursue one of several *stretch* goals described in Sec. 6

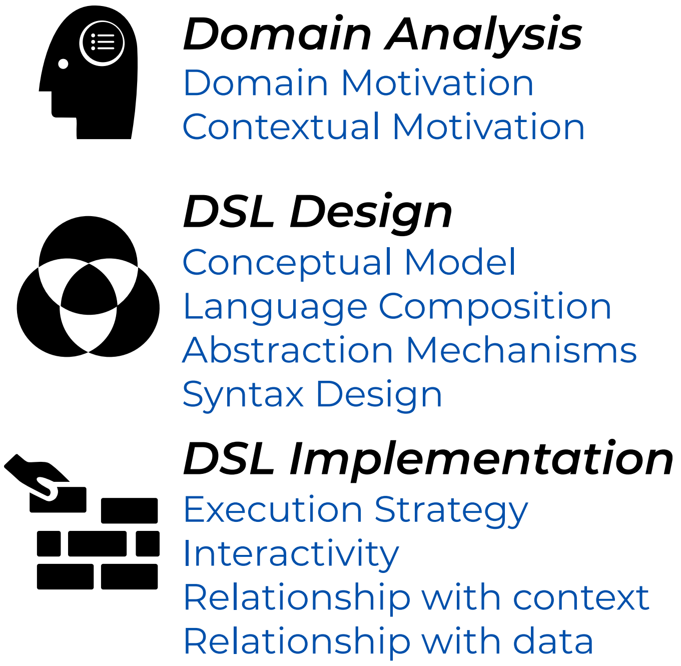


Fig. 2. Following Mernik et al.'s [84] partitioning of the DSL design process, we describe the visualization DSL design through a series of stages, at each stage of which there are design choices to be made. Here we show a *draft* of the patterns categories found in our analysis. The patterns listed here will be covered in explicit detail in the full version of this work.

2 Design: No Grammar to Rule Them All

This section describes work that is ***actively in progress****, and is aimed to be submitted to IEEEVIS22 Conference at the end of March.*

There has been a substantial growth in the use of static JSON-based grammar to create visualizations [97]. Each of these grammars serve various purposes: some focus on particular computational tasks, such as animation, some are focused on certain chart types, such as maps, and some focus on particular data domains. There are a wide variety of approaches to implementing each of the languages, some are interpreted and some compiled. They hold a variety of relationships with one another, some are composed, some compiled, some are restricted versions of another language, and, of course, most are unrelated. For instance, the animation grammar Gemini [58] wraps Vega-Lite [107], which compiles to Vega [108].

Yet, despite the growing popularity of this approach, there has been little study of implementation and design patterns utilized by these various systems. While these languages have allowed for the creation of a variety of systems [106], we suggest that a richer understanding of the design space will allow for richer development of subsequent systems.

In this project, we propose to conduct a survey of this design space This survey will categorize and describe the ways in which each of these JSON languages are designed and implemented across a variety of axes, such as what types of data they use, their host language, the execution strategy, and their relationship to other languages. These categorizations will draw upon previous work on characterizing the design patterns of domain-specific languages, such as Mernik et al.'s [84] characterization of implementation and abstraction patterns, Van Deursen et al.'s [120] survey of 75 DSLs, Fowlers practical consideration of the design and implementation of DSLs [31], as well as Erdweg et al.'s [28] work on typifying the relationships between and composition of languages. Through this analysis we will rarefy a design space, a sketch of which is shown in Fig. 2, that will allow language implementers to identify and execute important decisions for the design of and implementation of their visualization grammars.

Why JSON Grammars? An essential question underpinning this study, as well as this thesis, is why concern ourselves with this particular interface form? Programming language and interface design experts regularly take to social media to opine the prevalence of this approach [63] as declarative JSON grammars lack many of the features that make programming languages usable, such as tools for debugging, autocompletion, and abstraction. According to the *discoverer* of JSON, Douglas Crockford [112] it is a "lightweight data-interchange format" and not a programming or markdown language. While many grammars are expressive, their design always has constraints precluding the specification of that language, such as unusual chart types or elements (such as in the difficulty of specifying Gantt charts in Vega-Lite in a way that does not require numerous round trips to the data). These potential drawbacks notwithstanding, it is the declared context of this proposal that we are interested in trying to better understand how to help end-users operate in this style of language more effectively. These languages are prevalent and so understanding how they are designed is a natural first step in this pursuit. Further, Chasins et al. [11] highlight a growing convergence between HCI and programming language communities, and that programming language backed interfaces is a powerful pattern—a perspective shared by Heer et al. [41] in the design of Trifacta. While simple, JSON languages offer a direct and accessible entry into this mixing point, which may have useful outcomes for more sophisticated languages.

Prior work. This work draws on prior studies of domain-specific languages in both visualization and in general. In describing the design of Encodable—a system for papering over the inconsistent charting interfaces in JS through grammars that are reminiscent of Vega-Lite—Wongsuphasawat [133] sketched a taxonomy of visualization specification languages premised on level of abstraction. He later [134] extended this taxonomy and provided a close reading of a series of visualization grammars, such as examining the way in which a user could form a candlestick plot. Pu et al. [97] organized a special interest group at CHI21 on visualization grammars, which highlighted the pressing need for more formal study of these entities. We expand upon these works through a more in-depth survey, which locates the designed task, implementation strategy, and embedding style—among other features.

There are a wide range of domain-specific languages for visualization that *do not* utilize static grammar. Rautek et al.'s [99] ViSlang provides a system for making and coordinating small DSLS. A number of DSLs focus on scientific visualization [13, 26, 60], a space which Shen et al. [113] consider as part of their survey of visual computing DSLs. While these languages are interesting, their use of non-standardized syntax requires more substantial tools to manipulate computationally than JSON static grammars.

Our approach to this study also draws upon a number of previous works that describe design patterns of other aspects of visualization and data analysis systems. Heer and Manash [40] described design patterns for visualization systems with a particular focus on data management concerns. Lau et al. [68] surveyed the design patterns exhibited in computational notebooks. Our work also seeks to typify the relationship between visualization specification and data management, however our focus is more related to the language by which the visualization is specified. Gathani et al. [32] survey the space of SQL debugging tools, which is naturally related to our concerns as the ecosystem around a language is an essential as the core of the language itself.

2.1 Survey

Our survey encompasses systems from both academia, industry, and other open source contexts. We searched Google Scholar, ACM Digital Library, IEEE Xplore, and Github. Given the influence of the works on Vega on this style of paper we also reviewed all papers that cite any of the papers defining Vega and Vega-lite [107–109]. We have so far used the following keywords: "JSON", "XML", "YAML", "visualization", "map", "chart", "grammar", "language", "DSL', and "domain-specific language". We include the closely related YAML and XML-based languages, as understanding other visualization grammars built on top of comparable static languages are similarly useful and, given the now waning ubiquity of XML, provide evidence of the longer term

Axis	Values
Domain or task	The self described purpose of that language, such as animation, XR, genomics, etc
Language Source	Where the language arises from, e.g. Academic, Industry, Open Source
Extensible	Whether and how the language can be extended, e.g. by end users, through an API, etc
Abstraction Mechanisms	Which simple abstraction mechanisms the language features, e.g. control flow or variables
Embedded language	Whether the language contains another language, these include SQL, JS Snippets
Execution strategy	How the language is implemented, e.g. compiled, interpreted, composed
Output Type	What the execution of the language produces, e.g. vector, raster, text, interactive websites
Data Type	The type of input data transformed by the visualization, such as CSVs or domain-specific file formats
Juxtaposition strategy	The way in which multiple graphics are combined, e.g. via operators
Data manipulation strategy	The way in which language users can manipulate the input data, e.g. filters
Visualization/domain model	The logical model underlying the grammar, such as Grammar of Graphics, series-based, or the reference model
Encodable taxonomy	Where the grammar fits within Wongsuphasawat's [133] abstraction based taxonomy
When and How taxonomy	Where the grammar fits within Mernik et al.'s DSL taxonomy [84]
Annotated bib taxonomy	Where the grammar fits within Van Deursen et al.'s DSL taxonomy [120]

Fig. 3. Current axes of analysis in the survey. In addition, we also consider a number of boolean axes, including: *Produces Interactives*, *Has an Algebra*, *Has Alternate API*, *References Grammar of Graphics*, *Open Source*, *Formal Definition Available*, and *Has GUI by Default*.

prevalence of this type of design pattern. In contrast: while SVG, HTML, and other high-level static markup languages qualify under this definition, we exclude them as they are capable of producing far more than just visualization and their intent is not focus on the visualization or other associated tasks. In addition to these straightforward methods we also utilized snowball sampling, as some grammars would reference other similar systems.

Our criterion for inclusion identifies any system that explicitly uses JSON or describes itself as using JSON (or another standard static language such as XML or YAML) as a way to control its functionality and contains a (broadly defined) visualization. In particular, we follow our own exceptionally broad definition of a visualization [77] as being a transformation of data into a visual form meant to be interpreted by a human. So far we have identified **56 languages** which qualify under our survey domain. This search method suggests a bias towards grammars occurring in published works (44 of our current sample are drawn from academia), however we believe that the patterns exhibited by this sample are evocative of the general case.

Our criterion excludes a number of systems. For instance, some systems merely subset Vega-Lite in order to demonstrate a separate functionality, for instance GraphScape [59] operates over a non-interactive and non-composable subset of Vega-Lite in order to explore sequence recommendation. We exclude these systems from our analysis as they merely *use* a visualization grammar rather than constructing one for their own purposes. Further, while a number of visual builder systems possess systematically described languages (such as Visception [66]) they do not utilize a static carrier language so they are excluded as well. These systems are of interest and their approach to representation should be studied in future work.

Plan for analysis. After data gathering we will code each system based on a variety of axes, which are sketched out in Fig. 3. We will then organize and theme this sample to form a design space. Mernik et al. [84] describe the process of creating a DSL as consisting of a sequence of stages: decision, analysis, design, and implementation. We organize our discussion around this partitioning, with the minor adjust of merging the first two categories. Further, following prior work [88, 134], we intend to focus some of our analysis on close readings of examples in order to better understand syntactic patterns exhibited in the grammars.

As part of our survey we will also harvest all of the provided examples of each language (which can range from a handful to hundreds of examples). In order to facilitate reader exploration and further our own analysis we will prepare this survey as an explorable website, an early version of which is available at `https://festive-blackwell-a343de.netlify.app/`. This will allow for easy exploration of the space of languages. While this component is a minor contribution, given that assembling the examples of each of these languages has proved to be a surprisingly non-trivial process, we believe that surfacing this database of examples will provide a valuable point of comparison for subsequent grammar designers.

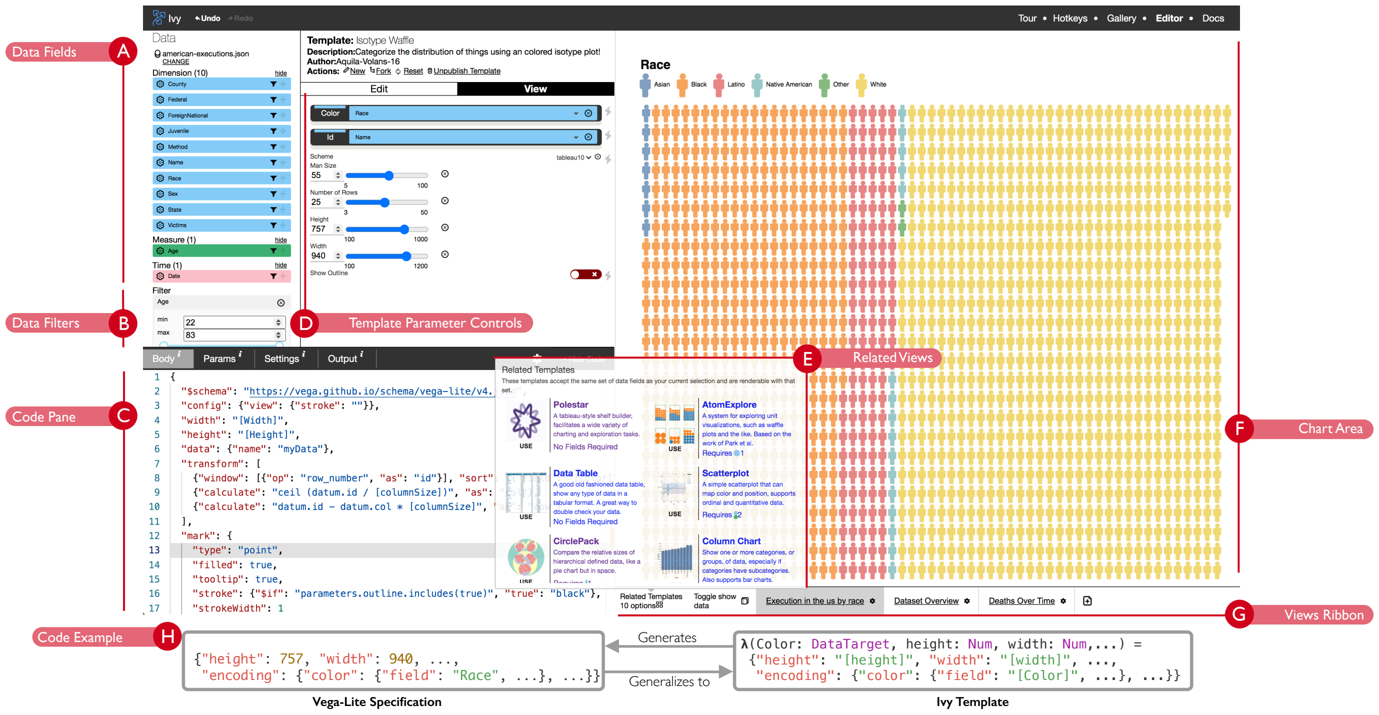
Initial results. Our initial reading of the survey suggests a number of intriguing findings.

Language design tends to emphasize one of several considerations: task (such as animation [58]), data domain (such as genomics data [75]), conceptual model (such as S-expressions [92] or the Grammar of Graphics [107]). Some languages utilize a more expressive algebra while others are more limited, which appears to be related to the domain, with the latter sometimes being more closely related to what is typically thought of as API design than grammar design.

Language implementation tends to follow one of several patterns including compilation and interpretation across a variety of embedding forms including language chaining, unification, wrapping, extension, and restriction. Many grammars "reimplement" the wheel, painstakingly recreating aspects of other languages, such as data transformations. There appear to be distinct families of syntax, with Vega-Lite guiding the majority of systems. To this end, most grammars tend to be focused on a level of abstraction similar to that of the Grammar of Graphics.

2.2 Plan for completion and intended contributions

The central contribution of this work will be a description of the design space of static visualization languages, with a minor contribution documenting and collating examples of all the languages usage. Our aim is to submit a paper describing this work to VIS22, tentatively titled "No Grammar to Rule Them All: A Survey of JSON-Based Visualization DSLs". If we miss this deadline or the paper is rejected, our fall back plan is to submit the work to CHI22 or EuroVIS23. The survey began in early fall 2021 and continued on through January 2022. The survey has begun to reach a stable state and analysis has begun.



```
{
  "$schema": "https://vega.github.io/schema/vega-lite/v4.
  "config": {"view": {"stroke": ""}},
  "width": "[Width]",
  "height": "[Height]",
  "data": {"name": "myData"},
  "transform": [
    {"window": [{"op": "row_number", "as": "id"}], "sort"
    {"calculate": "ceil (datum.id / [columnSize])", "as":
    {"calculate": "datum.id - datum.col * [columnSize]",
  ],
  "mark": {
    "type": "point",
    "filled": true,
    "tooltip": true,
    "stroke": {"$if": "parameters.outline.includes(true)", "true": "black"},
    "strokeWidth": 1
```

```
{"height": 757, "width": 940, ...,
 "encoding": {"color": {"field": "Race", ...}, ...}}
```

```
λ(Color: DataTarget, height: Num, width: Num,...) =
 {"height": "[height]", "width": "[width]", ...,
  "encoding": {"color": {"field": "[Color]", ...}, ...}}
```

Fig. 4. The Ivy visualization editor combines textual specification with GUI-based shelf building and chart choosing. Here, an "Isotype Waffle" *template* is used visualize the people executed in the USA under the death penalty by race since 1977 [10]. Templates are functions with typed parameters that abstract JSON specifications in declarative visualization grammars.

3 Abstraction: Ivy

This section describes work that appeared as a full paper at CHI21 [82]. The system is available at `https://ivy-vis.netlify.app/`

Every user interface design involves compromise. Which tasks should be made easy at the expense of making other tasks cumbersome or even impossible?

There are several common user interface modalities for creating visualizations, each with distinct trade-offs [36], as in Fig. 5 ***Chart choosers*** (as in Excel) allow users to rapidly construct familiar visualizations at the expense of flexibility. ***Shelf builders*** (as in Tableau) facilitate dynamic exploration but can obstruct the construction of specific chart forms or the addition of visual nuances. The declarative ***textual programming*** languages discussed so far in this document are highly expressive but can impede precise configuration.

Ideally, interfaces of varying complexity could be integrated such that both novice users (for whom chart choosers are often best suited [37]) and experts (whose most profitable interface will vary) obtain the benefits of each modality as their tasks require. Unfortunately this territory remains under-explored, as visualization systems tend to prefer one-size-fits-all designs. The static JSON-based declarative visualization grammars that we have looked at so far in this proposal are an enticing starting point as they provide significant flexibility for specifying visualizations as text. However, they lack the abstraction mechanisms found in full-featured programming languages. This project considers the question:

Can we extend declarative grammars with abstraction mechanisms for reuse, in a way that facilitates explorability as in shelf builders and ease of use as in chart choosers?

We answer this question by introducing a novel abstraction mechanism called ***parameterized declarative templates***. These templates abstract "raw" declarative specifications with parameters that specify data fields for a visual encoding (e.g. `Color`) and design parameters (e.g. `height` and `width`), making them more easily reused. To test applicability of this notion we systematically apply this idea in a prototype visualization editor, called Ivy, in which templates are created and instantiated through text- and GUI-based manipulation.

3.1 Template Language Design

Templates provide a simple set of abstractions over JSON-based grammars. Put simply, a template is a function specified in a superset of JSON, which includes variables and simple control flow operators, that when applied to arguments produces a chart in a particular visualization grammar. Templates are grammar-agnostic as they abstract arbitrary JSON specifications. Each *template* consists of two components: a body and parameters. Template bodies consist of JSON literals, as well as variables and conditional expressions. The former are value references to template parameters. The latter are equivalent to if statements in other languages, and similarly depend on the result of *predicate*, which is a "raw" JavaScript code string that evaluates to a boolean value. Template parameters are abstract over data fields and stylistic choices in the definition of a visualization and define GUI elements that allow users to specify argument values for these parameters. These are then evaluated by traversing the JSON during which conditionals are evaluated and variables substituted. The complete template syntax and semantics is described in the full version of the paper.

3.2 Interface Design

Equipped with the notion of templates, we next describe the Ivy UI. As shown in Fig. 4, the application consists of two panes, one for chart editing and another for chart viewing. The chart editing pane contains a data column filled with Tableau-style "pills" representing data columns, and an encoding column with "shelves" for those pills to be placed upon. This encoding column can be used to instantiate (i.e. provide arguments for) the parameters of the template, or to edit the GUI of the current template. The editing pane also includes a code editor which can manipulate the current template and UI state textually. Here, we describe how Ivy supports the *creation*, *selection*, and *application* of templates to produce charts.

Template Selection as Chart Choosing. The root of Ivy is a template gallery, which is populated with a library of system-provided and Ivy user created templates. Simpler templates allow users to jump quickly to familiar visual forms (such as line or bar charts), while more sophisticated templates privilege thinking with their data [101].

Each template is accompanied by a set of user-defined examples, namely, settings chosen by users to instantiate the template, with data bindings and output renderings with respect to a collection of predefined datasets. These examples serve as "crowd-sourced documentation" for how individual templates operate. Furthermore, this adds an element of opportunistic programming: to create templates, users can borrow small snippets—such as a well-formatted list of color schemes—and use them in their own creations.

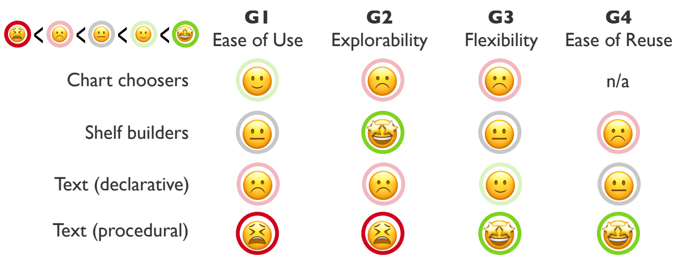
Template Application via Shelf Building. After selecting a template and uploading a dataset, the user is presented with a shelf builder-style GUI for setting the template parameters and specifying basic data filters. We choose this GUI design seeking to exploit the same affordances that drive the explorability of shelf builders. Specifically, our design closely follows that of the Polestar shelf builder system, which Wongsuphasawat et al. [135, 136] constructed as a simulacra of Tableau to serve as a baseline comparison in the development of their recommendation-based exploration systems. While emulating Polestar is a relatively small threshold to overcome in the context of VA systems, it demonstrates the promise of our template-based approach. Systems such as Tableau or PowerBI possess features that—although larger and more complex—are not substantially *different* from those in Polestar.

To select data parameters of interest, users drag-and-drop from a list of data fields, color-coded according to their *data roles*, onto encoding shelves, as in Fig. 4a, d. Following prior work [1, 115, 135] roles include `Measure`● (quantitative fields), `Dimension`● (nominal or ordinal fields), and `Time`● (temporal fields). When a dataset is loaded, we make heuristic guesses about the role for each column, which the user can later modify. We use roles in Ivy to construct a naive automatic *Add to Shelf* feature (akin to Tableau's Add to Sheet [76]), except ours is simply based on order and data role. If a template has three `DataTargets`, the first of which allows only a `Measure`● while the latter two allow anything, clicking *Add to Shelf* on a `Dimension`● will add it to the second parameter. In addition to the visual aesthetics of Polestar (and hence that of Tableau), we also emulate the *functionality* of its shelf-building interface through a "default" library template called IvyPolestar. The only features not replicated are the Automatic Mark Type—implementation of which, though possible in Ivy, was beyond the scope of the paper—and the chart bookmarks—which we replaced with a notion of view tabs.

Template Creation and Text Editing. Templates can be created or modified in two ways, either by modifying the textual representation or through GUI interactions. The textual representation facilitates both small tweaks, as well as creating new templates. For instance, users may copy code snippets found online—such as in language documentation or Stack Overflow—and templatize them to suit their task. Templates can also be created by "freezing" and refining the GUI state when interacting with an existing template. For example, a user might apply a full-featured template, such as IvyPolestar, to construct something resembling their desired chart, fork the text output as a new template (as in Fig. 4d), and then provide fine textual grained updates. As ever, we believe that exposing textual representation to the end user furthers flexibility and reusability.

To ease the construction of templates, Ivy uses domain-specific pattern matching and rewrite rules to suggest potential transformations to users. For instance, if a user were to find a chart in the Vega documentation that they wanted to copy, they would simply start a new template and paste the code into Ivy. The code pane then suggests ways to transform the code. For example, if a value in a Vega-Lite spec is used where a data reference is expected (e.g. `"field": "age"`), then Ivy suggests swapping `"age"` with a reference to a new parameter. Rules are defined by Ivy developers, rather than Ivy users.

Template-Based View Search. The systematic formulation and application of templates allows us to emulate recommendation and exploration features found in a variety of existing charting systems as a consequence of our design. Here we highlight two such features that follow naturally from the use of templates: one arises by fixing the arguments and varying the template, and the other by fixing the choice of template and varying the arguments. The gallery in Ivy is equipped with *catalog search*, which allows users to search across the set of available templates based on compatibility with a set of specified columns of interest—specifically, by using a simple type-compatibility algorithm that compares template parameter types with the data roles of selected columns. Comparisons in visual analytics (VA) are often made temporally, requiring the analyst to hold mental reference to each of the values under consideration. To reduce this cognitive burden, Ivy users can *fan out* a template by applying multiple settings and rendering their output simultaneously. These mechanistic approaches successfully emulate *smart* recommendation systems [69, 135, 136] without succumbing to the uninterpretable black box nature of these systems, which, as we've noted [78], can be opaque, inflexible, brittle, and domineering.



😫 < ☹️ < 😐 < 🙂 < 🤩	G1 Ease of Use	G2 Explorability	G3 Flexibility	G4 Ease of Reuse
Chart choosers	🙂	☹️	☹️	n/a
Shelf builders	😐	🤩	😐	☹️
Text (declarative)	☹️	☹️	🙂	😐
Text (procedural)	😫	😫	🤩	🤩

Fig. 5. Chart making modalities have distinct strengths and weaknesses. Using *parameterized declarative templates*, Ivy strives to combine the strengths of several common modalities in this design space.

3.3 Evaluation

We assessed our template-based approach by considering two questions: ***Do templates facilitate organization and reuse of existing visualizations?*** and ***Is Ivy's multimodal UI approachable by real users?***

We address the first of these questions by reproducing all of the examples in both the Vega-Lite example gallery [123] (consisting of 166 examples) and the chart chooser found in Google Sheets [33] (consisting of 32 examples) as templates, looking for opportunities to factor related visualizations into templates. This yielded 3.5x and 1.8x *compression ratios*, respectively, where compression is the number of examples constructible by a given template. This demonstrates that our simple template abstraction mechanisms enhance the flexibility of existing declarative grammars while improving their reusability by serving a variety of use cases.

We consider the latter question through a small approachability study (n=5). This study sought to understand whether this form of multimodality was usable. It consisted of a series of structured tutorial tasks followed by a series of open-ended data exploration and template construction tasks. Participants were professional data analysts who had recently graduated from MSCAPP. All participants were able to complete all tasks and found the system to be generally usable, yielding a mean system usability score of $\mu = 68.0$—describable as being between "OK" and "Good" [2]. Participants were enthusiastic about mixing code and graphical specification. P5 commented that the combination *"feels more useful than just coding"*. More critical than users' perception of the usability, which may have been positively biased, is the demonstration that they were able to navigate the system and use the interlocking modalities to achieve various tasks.

3.4 Key Findings and Contributions

This investigation demonstrated that this form of multimodality is an approachable system design for users with modest VA experience. This connection between text and GUI appears to help users learn and comprehend JSON-based charting grammars, which may be unfamiliar or difficult to understand. The repeatable customization found in templates might also, for example, enable practitioners to explore designs in a structured manner. While this architecture does have some limitations (such as the sometimes clumsy textual editing and that the grammar agnosticism prevents the system from providing grammar specific affordances) the benefits appear to outweigh the limitations. Our central contribution is demonstrating that template-style abstraction can be usefully applied to create approachable multimodal interfaces.

4 Manipulation: JSONG

This section describes work that is in the ***early stages of development****, and is aimed to be submitted to EuroVIS23 or UIST23.*

JSON's restricted textual form provides a number of useful benefits for the domains addressed by our languages of concern. For instance, in terms of the cognitive dimensions of notation framework [38], it decreases the *viscosity* (as syntax is typically quite terse requiring few changes to make big alterations), their small domain-focused form helps to maintain *consistency*, and their naturally abstracted form can help improve the *abstraction*. However, it also imposes a variety of expressiveness limitations. The strict syntax can be difficult to manipulate, as even small errors will cause programs to be unreadable by many parsers, yielding a disproportionate error response. Further, the terse syntax leaves little wiggle room and can make it difficult to learn the language. Depending on the grammar it can be difficult to evaluate incomplete specifications, yielding difficulties for *progressive evaluation*. Just as in other declarative languages it can be difficult to debug [32] requiring the programmer to hold a potentially unfamiliar execution model held by the DSL in their mind to address possible subtle errors (providing *visibility*, *hard mental operations* difficulties)

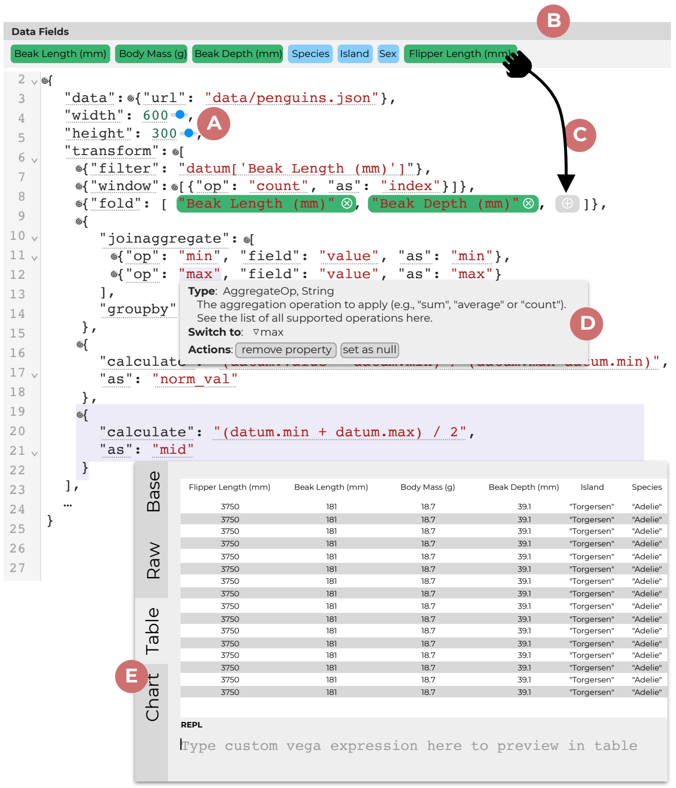
Previous approaches to these problems have generally involved **GUI Facades** or **Live Programming Annotations**.

GUI facades is a design pattern in which the languages is completely obscured from the user by placing them behind a GUI, such as in Lyra [105, 141] or Voyager [135, 136]. Heer et al. [41] describe this in terms of the GUI lifting over the grammar, and is described as an useful pattern and worthwhile of study by Chasins et al. [11]. This pattern is advantageous in that it allows for complex interactions with the underlying grammar, however it loses the ability to precisely tune or modify those programs and becomes limited by the design of the GUI. The mixed-GUI and textual interface we described with Ivy weaves together these approaches, however it succumbed to the shortcomings and difficulties of manual text editing, and the interplay between writing JSON and writing JS snippets sometimes proved challenging.

Live Programming Annotations, in contrast, embed contextual or evaluation information into a text editor itself. This practice is common in live-coding contexts which tend to blur evaluation and editing. For instance Omnicode [52] visualizes all program values all the time. Hoffswell et al. [47] aid the debugging of Vega programs via contextual inline-annotations such as sparklines and other word sized graphics. Their system shows the state of Vega's signal variables at the heart through these in-line annotations, which they find to be helpful for debugging, thereby reducing some of the required *hard mental operations*. Similarly, Lieber et al. [72] describe a method for surfacing run-time call counts to JS programmers in inline.

In this project we will address these concerns through a strategy akin to the latter of these design patterns, with an eye towards being able to make an Ivy-style integration into a GUI. We focus on this combination of modalities because we believe that exposing programmatic representations of GUIs to end users is useful, and seek to explore the utility of a system that takes JSON-DSLs as a first class design component. In particular we propose to develop a graphical interface, tentatively titled JSONG (JSON + GUI) that combines structure and projection-based editing in order to provide relevant information directly in context to the tasks being addressed. Both approaches have been used in other contexts, however they have not been generally made to support JSON grammars specifically. This could be because JSON grammars are still relatively uncommon compared to other programming languages or application interfaces, or, as discussed in Sec. 2 because JSON grammars are often looked down upon by experts. Further, this continues our expressed philosophy of meeting the users where they are: JSON grammars are prominent across a variety of contexts and a lack of tooling specifically for those languages. This creates an execution gap in users ability to effectively and concisely express their intents.

Here we lift our focus from visualization to encompass JSON grammars from other domains, including the narrative generation language Tracery and MongoDB's aggregation language. While the lessons are applicable to visualization, we believe exploring other domains will prompt greater utility in the design of this system.


B
Data Fields
Beak Length (mm)
Body Mass (g)
Beak Depth (mm)
Species
Island
Sex
Flipper Length (mm)
A
C
D
Type: AggregateOp, String
The aggregation operation to apply (e.g., "sum", "average" or "count").
See the list of all supported operations here.
Switch to: ▿max
Actions: remove property set as null
E
Base
Raw
Table
Chart
REPL
Type custom vega expression here to preview in table

```
{
  "data":{"url": "data/penguins.json"},
  "width": 600,
  "height": 300,
  "transform":[
   {"filter": "datum['Beak Length (mm)']"},
   {"window":[{"op": "count", "as": "index"}]},
   {"fold": [ "Beak Length (mm)", "Beak Depth (mm)",  ]},
   {
     "joinaggregate":[
      {"op": "min", "field": "value", "as": "min"},
      {"op": "max", "field": "value", "as": "max"}
     ],
     "groupby"
   },
   {
     "calculate                                    um.min)",
     "as": "norm_val"
   },
   {
     "calculate": "(datum.min + datum.max) / 2",
     "as": "mid"
   }
  ],
  ...
}
```

Flipper Length (mm)	Beak Length (mm)	Body Mass (g)	Beak Depth (mm)	Island	Species
3750	181	18.7	39.1	"Torgersen"	"Adelie"
3750	181	18.7	39.1	"Torgersen"	"Adelie"
3750	181	18.7	39.1	"Torgersen"	"Adelie"
3750	181	18.7	39.1	"Torgersen"	"Adelie"
3750	181	18.7	39.1	"Torgersen"	"Adelie"
3750	181	18.7	39.1	"Torgersen"	"Adelie"
3750	181	18.7	39.1	"Torgersen"	"Adelie"
3750	181	18.7	39.1	"Torgersen"	"Adelie"
3750	181	18.7	39.1	"Torgersen"	"Adelie"
3750	181	18.7	39.1	"Torgersen"	"Adelie"
3750	181	18.7	39.1	"Torgersen"	"Adelie"
3750	181	18.7	39.1	"Torgersen"	"Adelie"
3750	181	18.7	39.1	"Torgersen"	"Adelie"
3750	181	18.7	39.1	"Torgersen"	"Adelie"
3750	181	18.7	39.1	"Torgersen"	"Adelie"
3750	181	18.7	39.1	"Torgersen"	"Adelie"

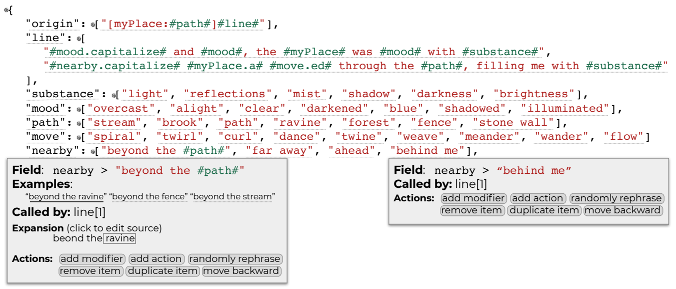
Fig. 6. A mockup of JSONG configured to be part of an Ivy-style application. (A) shows standard contextual type based enhancements (here a number dragger). (B) shows a list of data fields held outside of the editor which can be dragged directly onto a drag target (C) within the program specified as a projection. (D) shows the result of clicking on any value underlined field: a popover featuring JSON-Schema and AST generated type information which can be used to structurally manipulate the value. (E) a custom projection defined for each element in the transform pipeline that allows for inspection of the data transform at that stage, accessed by clicking on the control element at the start of the object.

4.1 System Design

Our proposed solution involves three major forms of augmentation to the traditional text editor: direct manipulation, projectional augmentations, and type-based structure editing. We selected these families of enhancements because they are richly expressive while still privileging the textual interface without substantial obfuscation.

Direct Manipulation Enhancements. Direct manipulation [114] in user interfaces is often highlighted as an essential way to close the gulf of execution [89] found in many computer systems. Textual programming languages have a failure in *closeness of mapping* famously highlighted by Victor's various direct manipulation experiments [124].

There have been a variety of systems that explore various ways to cross this gulf, causing many enhancements to be now standard (or easily accessible via extension) in modern code editors, such as sliders for numerical values, inline color pickers for colors, toggles for boolean values. We will employ a variety of these enhancements in our design as appropriate. Beyond these now everyday offerings a number of works have explored more exotic interactions. For instance, Lee et al. [70] describe a system for drag-and-drop refactoring in Eclipse. Hempel and Chugh explore this gap by proposing a variety of mechanisms to modify text and graphics bidirectionally [44] providing in-situ widgets for output manipulation. Our approach differs in that we work over a more restricted language and so are able to offer more expressive manipulation controls. For instance, we propose to add controls that allow for manipulating structurally similar elements simultaneously; such as changing the content of every object in an array.



```
{
  "origin": ["[myPlace:#path#]#line#"],
  "line": [
    "#mood.capitalize# and #mood#, the #myPlace# was #mood# with #substance#",
    "#nearby.capitalize# #myPlace.a# #move.ed# through the #path#, filling me with #substance#"
  ],
  "substance": ["light", "reflections", "mist", "shadow", "darkness", "brightness"],
  "mood": ["overcast", "alight", "clear", "darkened", "blue", "shadowed", "illuminated"],
  "path": ["stream", "brook", "path", "ravine", "forest", "fence", "stone wall"],
  "move": ["spiral", "twirl", "curl", "dance", "twine", "weave", "meander", "wander", "flow"]
  "nearby": ["beyond the #path#", "far away", "ahead", "behind me"],
```

Field: nearby > "beyond the #path#"
Examples:
"beyond the ravine" "beyond the fence" "beyond the stream"
Called by: line[1]
Expansion (click to edit source)
beond the ravine

Actions: add modifier | add action | randomly rephrase | remove item | duplicate item | move backward

Field: nearby > "behind me"
Called by: line[1]
Actions: add modifier | add action | randomly rephrase | remove item | duplicate item | move backward

Fig. 7. Each domain and grammar necessarily presents unique challenges unique. For instance, users of the narrative generation grammar Tracery are likely concerned with partial evaluation of the grammar and the way in which the various pieces are combined.

Projectional Augmentations. Traditional editors only allow the programmer to edit text. While this is deeply expressive, it is far from the only representation available. Projectional editing allows the user to specify edits [6] to the AST representation of the program without modifying the text, typically through an abstracted representation of the program. This ensures that the program is always in a valid state (a concern shared by the structure editing). These alternative representations are often presented alongside the text as live-programming annotations and can be designed in such a way as to match the domain the user is working in. For instance, Lerner's projection boxes [71] provide live graphical visualizations of the dynamic execution of python programs as a means to allow interactive program synthesis [29, 94]. This approach is related to livelits [90] which allow for programmatic specification through live direct manipulation GUIs in a functional programming language, which is itself an extension of previous work on GUI-based manipulation widgets in Java [91].

We propose to follow a similar tack by enabling the creation of grammar specific projections that are able to show both static and dynamically evaluated information about the program. These *projections* will be optionally manipulable so that changes can be made structurally to the program. For example, in Fig. 6 we show a sketch of a version of the editor configured to be part of a visual analytics application. Rather than abstracting the relationship between the data fields and their place in spec as Ivy does, this approach instead allows users to drop pills directly onto relevant parts of spec itself. The location of projections can be specified through two query languages: one based on the key path and one based on the JSON-Schema.

Type-based structure editing. Despite being occasionally maligned for their lack of types [20], JSON supports a variety of type information, both from its limited grammar, as well as from external specifications such as JSON Schema [96]. We intend to use both of these families of tools in order to allow the user to manipulate JSON programs in such a way that they are always in a valid state. This will include basic language-level manipulations (such as inserting, deleting, and reordering elements), domain-based type information (such as selecting from a set of allowed values or adding a new object or inlining a remote data source), as well as novel interactions (such as, again, manipulating structurally similar elements simultaneously).

While some structure editors restrict editing exclusively to graphical representations (such as Scratch [100]) others allow for mixed text and structure editing. Deuce [44] enables users to edit both the textual version of code as well as through an opt-in structure editing mode in a FPL focused on SVG manipulation. IdyllStudio [16] provides a structure editor specialized to interactive data-journalism based on a custom DSL. Ko and Myers described systems for constructing structure editors abstractly [64, 65]. Our approach contrasts with these in that we address a more restricted language of JSON rather than a custom FPL, DSL, or on ASTs in general. Our approach also resembles a number of community created tools, such as jet [95] and jsoneditor [19]. We expand on these in several ways: our support for JSON-Schema is substantially more robust than other offerings, in that we are able to process large schemas, such as those of Vega and Vega-Lite. We do so by forking the JSON parser from VSCode's JSON LSP [86], which already featured strong support schema inferences (which is likely a component of the motivation for the use of the Monaco editor in the Vega Editor [122]).

4.2 Need Finding Survey

Beyond our described families of modifications to the traditional textual interface, we intend to guide the construction of our design based on a need finding survey of users of JSON-DSLs. In particular we seek to better understand the usage and pain points they have in learning, writing, and using programs written in this style. While we are confident that the problems described by our cognitive dimensions discussion are prominent, both from our own experience using these languages and from prior works seeking to alleviate these specific problems [46, 47, 110], we believe that it will be useful to consult with users of these textual interfaces about the problems that arise for them in their local context. We will target a variety of communities including those likely to be using Vega (visualization), Tracery (games), and MongoDB (databases). That said, we will phrase the survey in such a way that it is accessible to any one with experience using any JSON language.

4.3 Evaluation

We will evaluate how our GUI-enhanced approach by considering two questions: ***Can this approach replicate the prior work?*** and ***How does this design compare to familiar forms of editing?***

Expressiveness. In order to demonstrate the utility of this modular approach we will construct examples of our system applied to a variety of JSON-based DSLs, both in the context of visualization and otherwise. We will replicate the inline annotations described by Hoffswell et al. [47]. We will create editors for Tracery (such as the one shown in Fig. 7) and MongoDB.

Usability study. While the ability to create these enhancements to a standard textual JSON interface is useful, we also seek to understand how well these approaches enhance users ability to accomplish actual tasks. We will conduct a between-subjects study of users accomplishing tasks using JSON-DSLs in versions of our editor both with the various augmentations activated and with them deactivated. We will evaluate them based on the correctness of their solutions, how long they took to complete it, and will collect qualitative feedback on the experience using this style of manipulation.

4.4 Plan for completion and intended contributions

The central contribution to this work is the demonstration that this style of augmentation is practical and enhances the usability of JSON-DSLs. Our aim is to submit a paper describing this work in the late Fall of 2022, tentatively titled "Yours for a JSONG: Modular contextual editor augmentations for JSON DSLs almost for free", to a HCI or visualization venue (such as EuroVIS23, ICSE23, VIS23, or UIST23). A proof-of-concept prototype was developed in Fall of 2021 in order to validate the feasibility of the described system augmentations. The need finding design study will occur during Spring 2022 and will occur concurrently with further prototyping and implementation. Evaluation and writing will occur during Fall 2022.

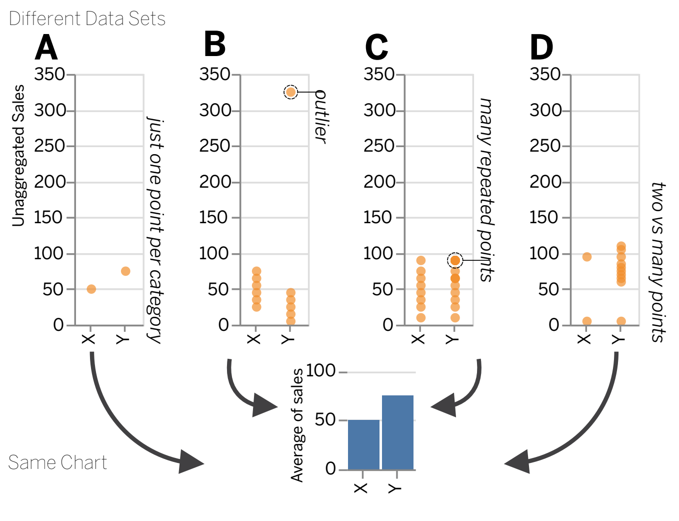


Fig. 8. Visualizations that aggregate data can mask their underlying data distributions, which can create mirages, as in this bar chart over synthetic data. Not all distributions equally support the message that sales in Y are reliably higher than X. For instance, this difference could be driven by a single outlier (B), a possibly erroneously repeated value (C), or be an artifact of high variability caused by low sample size (D).

5 Validation: Mirages

This section describes work that appeared as a full paper at CHI 2020 [80], which won an honorable mention for best paper.

Visualizations, like all forms of communication, can mislead or misrepresent information. Visualizations often hide important details, downplay or fail to represent uncertainty, or interplay with complexities in the human perceptual system. Viewers often encounter charts produced by analytical pipelines that may not be robust to dirty data or statistical malpractice. It is straightforward to generate charts that, through chance, callousness [83], or carelessness, appear to show something of interest in a dataset, but do not in fact reliably communicate anything significant or replicable. We refer to the charts that superficially convey a particular message that is undermined by further scrutiny as visualization *mirages*. We define a visualization mirage as follows:

> A **visualization mirage** is any visualization wherein a cursory reading would appear to support a particular message arising from the data, but a closer re-examination of the visualization, backing data, or analytical process would invalidate or cast significant doubt on this support.

In this project, we consider a conceptual model of these visualization mirages and show how users' choices can cause errors in all stages of the visual analytics (VA) process that can lead to untrue or unwarranted conclusions from data. Using our model we observe a gap in automatic techniques for validating visualizations, specifically in the relationship between data and chart specification. We address this gap by developing a theory of *metamorphic testing for visualization* which synthesizes prior work on metamorphic testing [111] and algebraic visualization errors [61]. Through this combination we seek to alert viewers to situations where minor changes to the visualization design or backing data have large (but illusory) effects on the resulting visualization, or where potentially important or disqualifying changes have no visual impact on the resulting visualization. We develop a proof of concept system that demonstrates the validity of this approach, and call for further study in mixed-initiative visualization verification. While this technique is generally applicable to any context in which the types of a visualization program are known, this approach is particularly applicable to JSON-DSLs as they can be manipulated computationally in a direct and concise manner.

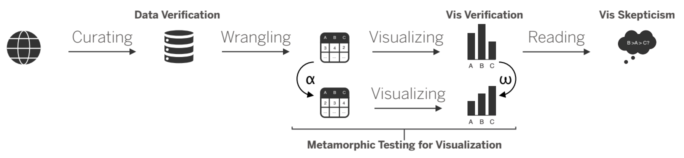


Fig. 9. A pipeline model of visualization indicating the places in which error can arise throughout the process.

A Mirage Model. A long sequence of contexts and decisions, from the initial data curation and collection, to the eventual reader's literacies and assumptions, determine the message that a visualization delivers. Mistakes, errors, or intentionally deceptive choices anywhere along this process can create visualization mirages, from dirty data [57] to cognitive biases [21]. Failures can occur at an early stage, but not result in a mirage until a later stage. For instance, missing data as a result of an error in data collection may be visible in a particular visualization, such as univariate data in a dot plot, and so be unlikely to lead to an error in judgment. Yet this data flaw may be invisible in a less robust visualization-design such as a histogram [18]. Whether the missing data results in a mirage is contingent on the choice of eventual visualization design. We suggest how choices can create errors and highlight the way that those errors can propagate to become mirages in Fig. 9. In the full version of this work we locate 83 types of known error that can be classified as mirages and locate them at each point in this process when the designer can exert agency.

5.1 Metamorphic Testing for Visualization

There have been a variety of approaches to avoiding mirages in visual analytics. These include data verification (such as in the Vizier system [9], the machine-learning-focused data linter [51], or Barowy et al.'s [3, 4] systems for debugging data in spreadsheets) and visualization verification (such as in line-up protocols et al.'s [48, 130]), as well as through more familiar means such as recommendation [69], smart defaults [30], and personal reflection [137]. There has been comparatively less consideration towards *detecting* errors that occur in the relationship between data and chart. Prior work principally focuses on static validation [12, 50, 67, 79, 98] rather than dynamically validating existing charts. To address this gap we combine a concept from the software engineering community, *metamorphic testing*, which focuses on detecting errors in contexts that lack a truth oracle, with work from Algebraic Visualization Design (AVD) [61], to form a notion of metamorphic testing for visualization. The most simplistic specification of AVD asserts that *a good visualization will change significantly if that it's data is changed significantly*, which we can represent graphically (figure from [81]):

A change to the input should be

$$\underbrace{\overbrace{v \circ r_2 \circ \alpha}^{\text{change to the input}} = \omega \circ v \circ r_1}_{}$$

reflected by a change in the image

$$\begin{tikzcd} \text{Data} & \text{Representation} & \text{Output Image} \\ D_1 \arrow[r, "r_1"] \arrow[d, "\alpha"'] & R_1 \arrow[r, "v"] & V_1 \arrow[d, "\omega"] \\ D_2 \arrow[r, "r_2"] & R_2 \arrow[r, "v"] & V_2 \end{tikzcd} \qquad (1)$$

Change in representation   Visualization process

Failures of these assertions can result in "hallucinators" (visualizations that look dramatically different despite being backed by similar or identical data) and "confusers" (visualizations that look identical despite being backed by dramatically different data). In the worst case, visualizations can be completely non-responsive to their backing data, functioning as mere number decorations and creating what Correll & Heer [17] refer to as visualization "non-sequiturs." These AVD failures directly tie to our notion of mirages (as they can result in visualizations that are fragile, non-robust, or non-responsive), but, by providing a language of manipulations of data and visualization specification, lend themselves to mixed-initiative or automatic testing. AVD provides a useful framework for designing tests that detect failures that require little domain knowledge. We can simply induce trivial or non-trivial data change, and check for corresponding changes in the resulting visualization.

Metamorphic Testing. In complex software systems it can be difficult or prohibitively expensive to verify whether or not the software is producing correct results. In the field of software testing distinguishing between correct and incorrect behavior is known as the "test oracle problem" [5]. The metamorphic testing (MT) ideology attempts to address this challenge by verifying properties of system outputs across input changes [111]. Rather than checking that particular inputs give correct outputs, MT asserts that properties called *metamorphic relations* should remain invariant across all appropriate metamorphoses of a particular data set. MT has been successfully applied to a wide variety of domains [23, 111, 140]. Donaldson et al. [23], whose concerns were focused on shaders, formalize this technique, asserting that the following equation should be invariant:

$$\forall x : p(f_I(x)) = f_O(p(x))$$

where x is a given shader program, p a shader compiler, f_I perturbations to the input, and f_O changes to the output (usually the identity under their framework). The definition of equality in MT plays a significant role in the effectiveness of its analysis.

Applying Metamorphic Testing. We now introduce the idea of using metamorphic testing as a mechanism to verify individual visualizations. Tang et al. [116] describe visualization as the function $vis(Data, Spec)$. This suggests two key aspects across which we can execute metamorphic manipulations: alterations to the data and alterations to the design specification. This perspective has the advantage that we can test a wide variety of types of visualization without knowing much about the chart being rendered. We observe that the representation of MT above is isomorphic to AVD's commutativity relation. MT is a concrete way to test the invariants of systems in general, whereas AVD describes the types of invariance-failures that occur with visualizations specifically. Observing this overlap we define a **Metamorphic Test for Visualization (MTV)** as a function parameterized by an equality measure (Eq), an input perturbation (α), a visual perturbation (ω), which evaluates a tuple of data and chart specification (denoted as a pair as x), and returns a Boolean:

$$MTV :: (Eq, \alpha, \omega) \Rightarrow (spec, data) \Rightarrow Boolean$$
$$MTV(Eq, \alpha, \omega)x = Eq(v(\alpha(x)), \omega(v(x)))$$

We leave v, the visualization system itself, out of the parameterization because we are interested in testing for problems in the relationship between data and chart specification, as opposed to validating the system mapping chart specification to data space (which we assume to be error free). This formulation clearly describes the relationship between expectation and permutation in a manner that we believe allows for concise and unambiguous descriptions of invariance tests.

To our knowledge MT has not previously been used in visualization contexts, though there has been prior work that uses implicitly related techniques. Most notably are Guo et al.'s [39] metamorphic-like strategy to detect instances of Simpsons's paradox in VA systems, Gotz et al.'s [34] "Inline Replication" analysis of the visual impact of "alternative" analyses and tests for the reliability of a given chart, or Dragicevic et al.'s [24] "Multiverse Analysis."

5.2 Proof Of Concept

We implemented a proof of concept system for inducing morphisms on Vega-Lite [107] specs and their backing data in order to identify potential mirages or unreliable signals in charts. Our primary goal in this system is to demonstrate the validity of our metamorphic testing. While there are a wide variety of types of test this approach might generate, we describe here four metamorphic tests for visualization (MTVs) Each test should have predictable impacts on the resulting image. Failing to adhere to a prediction (and hence violating an MT relation) can indicate an error in the backing data or visual specification of the chart, pointing to a potential mirage. ***MTV: Shuffle*** asserts that changes to the order of the input data should not change the rendered image. ***MTV: Bootstrap*** asserts that the apparent patterns in visualizations should be robust: that is, a particular relationship should continue to be present across minor changes [18, 74]. ***MTV: Contract Records*** asserts that findings should remain consistent when the number of records has been contracted (emulating dirty data). ***MTV: Randomize*** asserts that randomizing the relationship between two variables should result in non-significant findings.

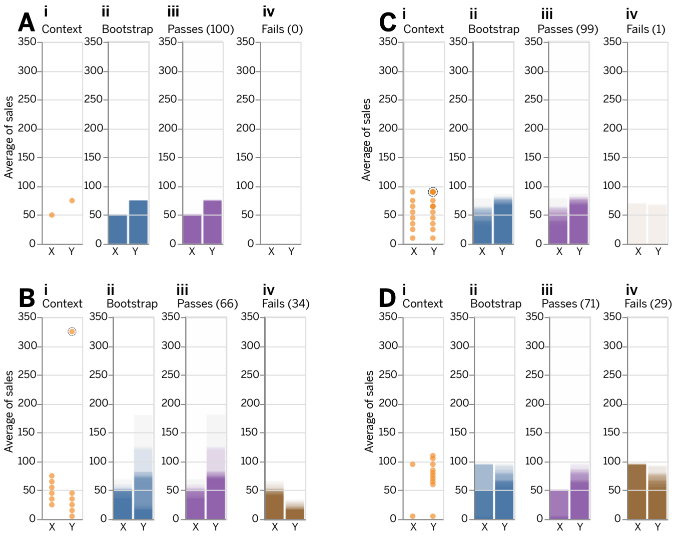


Fig. 10. Our ***MTV: Bootstrap*** test applied to the quartet of distributions shown in Fig. 8, shown here in (i). In each case we execute our test $N = 100$ times. We create 100 new potential bar charts by resampling from all the data that are aggregated into each bar. The results of this test are overlaid (ii-iv) for each low opacity "potential" bar chart on top of each other. Large "fuzzy" bands indicate that the specific values in the chart are not robust to resampling. We sort our bar charts into two categories: "passes" (iii) where the original insight that AVG $X > Y$ is preserved, and "fails" where this insight is not preserved (iv). A high number of passes indicate a robust insight.

To test this approach, we conducted a series of simulation studies in which synthetic data was evaluated by each of these tests. Fig. 10 shows the results of one family of these tests. In general, the impact of our morphisms became larger as the severity of our data manipulations increased: the fragility of the values in a given bar chart increases as the means become closer together, the sample size shrinks, outliers are added, or the variability increases. While we recognize that our simulation does not fully capture the utility of our proposed metamorphic tests, we present these initial results as evidence that our tests can be used as measures for the robustness of signals in visualizations.

5.3 Key Findings and Contributions

We believe that MT offers a useful complement to directly testing data or chart specifications, as it requires a smaller set of assumptions and parameters than statistical tests, and is portable across visualization toolkits and is especially well-suited to visualization grammars as they can be manipulated programmatically with great clarity. We see the types of visualization tests described here as being analogous to testing methods from software engineering. Direct tests, like unit tests, verify isolated properties (for instance, that quantitative axes begin at '0' in bar charts); while metamorphic tests, like integration tests, look to see that the whole image is working as desired. We believe that, in tandem, these validation approaches offer an effective way to target a wide variety of charting errors arising in the Wrangling and Visualization steps of visual analytics. We believe our model and testing approach provide ample starting ground for future work on automated detection of subtle errors in visualization, as well as validating the design of visualizations based on the relationship between their data and design.

6 Conclusion

JSON-based grammars are an increasingly popular way to specify intent in a variety of complex domains that is expressive, concise, portable, and frequently secure. In this proposal we have described a series of systems and approaches that enrich the aspects of those languages that make them strong while reducing the difficulty of using them in a variety of situations. In particular we have argued that the development of purpose specific interfaces can enhance JSON-DSLs as a direct interface modality, and therein that mixed graphical and textual interfaces are a particularly intriguing way to approach this domain.

6.1 Stretch Goals

If there is sufficient time left after the completion the work described here, we will consider some of the following projects:

Let Ivy grow over other modalities. We believe that mixed GUI and textual template editing is a useful approach to constructing visualizations based on grammars. So far we have only evaluated this approach in the context of a self-contained visual analytics system, however these approaches may also have useful application in a variety of contexts. In particular we propose to port the core of the Ivy template manipulations to spreadsheets, notebooks, and visual builders (such as Illustrator).

Several prior works have explored combinations of user interface modalities for creating visualizations. Liger [103] mixes together shelf-based chart specification and *visualization by demonstration*. Hanpuku [7], Data-Driven Guides [56], GraphPad [139], StructGraphics [119], and Data Illustrator [73] combine visual editor-style manipulation with visual chart specification or textual programming. Victor [124] explored a prototype that combined a spreadsheet with direct manipulation and manual view specification. Systems including mage [53], Wrex [25], and B2 [138] expand on these ideas by intermingling text and graphical specification in computational notebooks. ReVize [49] seeks to support multiple modalities by chaining together analysis tools through a Vega-Lite-based API.

This exploration would allow us to compare and contrast the affordances of each of the mediums against a constant background of Ivy's template-based affordances.

JSON Grammar Case Studies. While we found quite a number of insightful grammars in Sec. 2, we believe that we could usefully explore the space further through a series of case studies.

Domain Specific Dialects: A Gantt Grammar. Some grammars focus on particular chart forms (such as graph and map grammars) others on embedding particular computations into a language (such as SetCola's [45] high-level constraints). In this case study we explore a domain-first language design by creating a grammar for focused Gantt charts—a form of project planning diagram—show both temporal and logical dependencies across aspects of a project. This visual language is of a relatively limited form that receives extremely high use from practitioners in the context of tools like Microsoft Project. While they can be produced through a wide variety of visualization grammars, they are often shoehorned-in in a manner that is unnatural to the data domain. For instance, in Tableau and Vega-Lite one can describe them, but they typically require external calculation in order to construct an appropriate layout. These calculations can be non-trivial, for instance, calculating the critical path (which is the essential blocking tasks of a project) requires a topological sort. f The externalization of an important part of the exploration process can slow down and impede the construction of these charts injecting numerous potentially time-consuming round trips to the data as adjustments are made. This grammar explores notations that are aligned with data domain rather than the mark domain.

Task Specific Dialects: An Annotation Grammar. A common task in the construction of visualization is the addition of annotations. These textual or graphical additions typically add clarifications or call outs to the graphic in a way that increases the readability or visual intrigue. Yet, annotations tend to fall far outside of the language in which visualizations are phrased. The strategy found in extant JSON-languages tends to hard code these decorations (as in this Vega-Lite chart), making it difficult to design them in a way that is maintainable or natural to the annotation task. We will describe a grammar for annotation, describe the design space associated with this task, and provide a proof-of-concept implementation based around Vega-Lite.

Beyond problems related to their conceptual specification, annotations are often graphically nuanced requiring minor adjustments causing textual specification to be tedious. We will instrument a wrapping layer over our annotation grammar so that direct manipulation can be used to specify the annotations in such a way that those changes are propagated back to the textual representation. This form of bidirectional manipulation is naturally connected with similar techniques seen in the context of mixing drawing with traditional programming languages [14, 22, 43], as well as in visualization to a certain extent [102, 103, 105, 141].

Compile towers: 3D Unit Visualizations for Cheap. While handcrafting interpreters for these domain-specific languages can yield powerful benefits (such as the big-data zooming found in Kyrix-S [117]), doing so can be labor intensive. This can mean that the focus of development is on the grammar and semantics of a language, rather than on usability features such as expressive color spaces or tooltips. One such example is found in the implementation artifact of the Atom unit visualization language [92]. This L-System inspired grammar seeks to describe the design space of so-called "unit visualizations", which are forms in which each mark in the output represents a single data entity. For example, the waffle plot shown in Fig. 4 is a unit visualization. The artifact produced as part of that work is limited: it has implementation errors and missing cases, and does not feature any of a variety of features that might contribute to its usability. In this case study we seek to ensure that this grammar is more generally sustainable by transferring the ad hoc d3-based rendering system into a compiler. We will target the Vega compiler for its ubiquity, tooltips, and color palette support. Further, we can reuse a recently developed deck.gl [127] renderer for Vega (built for a recent version of SandDance [85]) to explore unit visualizations in 3D almost for free.

Bidirectional Editing of JSON Grammars. While JSONG's structure and projectional approach to enhancing the usability narrows the gap in usability of these JSON-DSLs, an even more direct strategy would allow users to directly manipulate the output of these programs and have those modifications be propagated back to the program themselves. This would be analogous to the work that Sketch-n-Sketch [42, 43] does for SVG programming as well as the sketch-based visualization generation via program synthesis demonstrated by Wang et al. [125, 126].

6.2 Future work

Beyond these extensions there are a number of lines of research which would be natural follow ups to the work described in this proposal, but we believe are out of scope for the completion of this thesis.

The creation of JSON-DSLs is a time consuming process. It involves developing a careful understanding of the domain, modeling it in a manner that both fits into JSON and maintains a connection with the original context, and then producing a mechanism for evaluating programs in the language. In future work we would like to develop abstract tools to help bridge the latter two of these steps through the development of visualization JSON-DSL workbench. In such a system users define the syntax and semantics of their language, which would then produce execution libraries and other support tools (such as a JSON Schema and or a visualization linter [12, 50, 79]) as appropriate. Pu et al. [97] highlight that debugging and testing visualization grammars is a hard problem and having a formal version of the semantics of a visualization language would aid in the verification and construction of novel grammars.

More focused on the specific domain of visualization, there is a wealth of potential in the space of automated visualization analysis. As described in Sec. 5 most visualization analysis is either rudimentary lints or recommendation (which in the language of Draco [87] would be autocompletes). Subsequent work should work to investigate the spectrum between these mediated by agency, in such a way that automated guidance can be applied in a polite [129] and context specific fashion.

References

[1] M. Agarwal, A. Srinivasan, and J. T. Stasko. VisWall: Visual Data Exploration Using Direct Combination on Large Touch Displays. In *30th IEEE Visualization Conference, IEEE VIS 2019 - Short Papers, Vancouver, BC, Canada, October 20-25, 2019*, pp. 26–30. IEEE, 2019. doi: 10.1109/VISUAL.2019.8933673

[2] A. Bangor, P. Kortum, and J. Miller. Determining what individual sus scores mean: Adding an adjective rating scale. *Journal of usability studies*, 4(3):114–123, 2009.

[3] D. W. Barowy, E. D. Berger, and B. Zorn. Excelint: Automatically finding spreadsheet formula errors. *Proceedings of ACM Programming Languages*, 2(OOPSLA):148:1–148:26, Oct. 2018. doi: 10.1145/3276518

[4] D. W. Barowy, D. Gochev, and E. D. Berger. Checkcell: Data debugging for spreadsheets. *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications OOPSLA*, 49(10):507–523, 2014. doi: 10.1145/2660193.2660207

[5] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2014. doi: 10.1109/TSE.2014.2372785

[6] T. Berger, M. Völter, H. P. Jensen, T. Dangprasert, and J. Siegmund. Efficiency of projectional editing: A controlled experiment. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 763–774, 2016.

[7] A. Bigelow, S. Drucker, D. Fisher, and M. Meyer. Iterating Between Tools to Create and Edit Visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 2016. doi: 10.1109/TVCG.2016.2598609

[8] E. Botoeva, D. Calvanese, B. Cogrel, and G. Xiao. Expressivity and complexity of mongodb queries. In *21st International Conference on Database Theory (ICDT 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[9] M. Brachmann, C. Bautista, S. Castelo, S. Feng, J. Freire, B. Glavic, O. Kennedy, H. Müeller, R. Rampin, W. Spoth, et al. Data debugging and exploration with vizier. In *Proceedings of the 2019 International Conference on Management of Data*, pp. 1877–1880. ACM, 2019.

[10] S. J. Chandler. Executions in the United States, 2018. Wolfram Data Repository.

[11] S. E. Chasins, E. L. Glassman, and J. Sunshine. Pl and hci: better together. *Communications of the ACM*, 64(8):98–106, 2021.

[12] Q. Chen, F. Sun, X. Xu, Z. Chen, J. Wang, and N. Cao. Vizlinter: A linter and fixer framework for data visualization. *IEEE transactions on visualization and computer graphics*, 2021.

[13] H. Choi, W. Choi, T. M. Quan, D. G. Hildebrand, H. Pfister, and W.-K. Jeong. Vivaldi: A domain-specific language for volume processing and visualization on distributed heterogeneous systems. *IEEE transactions on visualization and computer graphics*, 20(12):2407–2416, 2014.

[14] R. Chugh, B. Hempel, M. Spradlin, and J. Albers. Programmatic and Direct Manipulation, Together at Last. In *ACM SIGPLAN Notices*, vol. 51, pp. 341–354. ACM, 2016. doi: 10.1145/2908080

[15] K. Compton, B. Kybartas, and M. Mateas. Tracery: An Author-Focused Generative Text Tool. In *International Conference on Interactive Digital Storytelling*, pp. 154–161. Springer, 2015.

[16] M. Conlen, M. Vo, A. Tan, and J. Heer. Idyll Studio: A Structured Editor for Authoring Interactive & Data-Driven Articles. In *Symposium on User Interface Software and Technology*, 2021.

[17] M. Correll and J. Heer. Black hat visualization. In *Workshop on Dealing with Cognitive Biases in Visualisations (DECISIVe), IEEE VIS*, 2017.

[18] M. Correll, M. Li, G. Kindlmann, and C. Scheidegger. Looks Good to Me: Visualizations as Sanity Checks. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):830–839, 2018. doi: 10.1109/TVCG.2018.2864907

[19] J. de Jong. jsoneditor. `https://github.com/josdejong/jsoneditor`, 2021.

[20] Dhall. Design choices. `https://docs.dhall-lang.org/discussions/Design-choices.html`, 2021. Accessed Janurary 3, 2021.

[21] E. Dimara, S. Franconeri, C. Plaisant, A. Bezerianos, and P. Dragicevic. A task-based taxonomy of cognitive biases for information visualization. *IEEE Transactions on Visualization and Computer Graphics*, 2018. doi: 10.1109/TVCG.2018.2872577

[22] Q. Do, K. Campbell, E. Hine, D. Pham, A. Taylor, I. Howley, and D. W. Barowy. Evaluating prodirect manipulation in hour of code. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*, pp. 25–35, 2019.

[23] A. F. Donaldson, H. Evrard, A. Lascu, and P. Thomson. Automated testing of graphics shader compilers. *Proceedings of ACM Programming Languages*, 1(OOPSLA):93:1–93:29, Oct. 2017. doi: 10.1145/3133917

[24] P. Dragicevic, Y. Jansen, A. Sarma, M. Kay, and F. Chevalier. Increasing the transparency of research papers with explorable multiverse analyses. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, p. 65. ACM, 2019. doi: 10.1145/3290605.3300295

[25] I. Drosos, T. Barik, P. J. Guo, R. DeLine, and S. Gulwani. Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists. In *CHI '20: CHI Conference on Human Factors in Computing Systems*, pp. 1–12. ACM, 2020. doi: 10.1145/3313831.3376442

[26] D. J. Duke, R. Borgo, M. Wallace, and C. Runciman. Huge data but small programs: Visualization design via multiple embedded dsls. In *International Symposium on Practical Aspects of Declarative Languages*, pp. 31–45. Springer, 2009.

[27] T. Duplantis, I. Karth, M. Kreminski, A. M. Smith, and M. Mateas. A genre-specific game description language for game boy rpgs. In *2021 IEEE Conference on Games*, 2021.

[28] S. Erdweg, P. G. Giarrusso, and T. Rendel. Language composition untangled. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, pp. 1–8, 2012.

[29] K. Ferdowsifard, A. Ordookhanians, H. Peleg, S. Lerner, and N. Polikarpova. Small-step live programming by example. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, pp. 614–626, 2020.

[30] S. Few. *The Data Loom: Weaving Understanding by Thinking Critically and Scientifically with Data*. Analytics Press, 2019.

[31] M. Fowler. *Domain-specific languages*. Pearson Education, 2010.

[32] S. Gathani, P. Lim, and L. Battle. Debugging database queries: A survey of tools, techniques, and users. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pp. 1–16, 2020.

[33] Google. Types of charts & graphs in Google Sheets. `https://support.google.com/docs/answer/190718`, 2020. Accessed August 19, 2020.

[34] D. Gotz, W. Wang, A. T. Chen, and D. Borland. Visualization model validation via inline replication. *Information Visualization*, 18(4), 2019. doi: 10.1177/1473871618821747

[35] I. Grabska-Gradzińska, L. Nowak, W. Palacz, and E. Grabska. Application of graphs for story generation in video games. In *2021 Australasian Computer Science Week Multiconference*, pp. 1–6, 2021.

[36] L. Grammel, C. Bennett, M. Tory, and M.-A. D. Storey. A Survey of Visualization Construction User Interfaces. In *EuroVis (Short Papers)*, 2013. doi: 10.2312/PE.EuroVisShort.EuroVisShort2013.019-023

[37] L. Grammel, M. Tory, and M.-A. Storey. How Information Visualization Novices Construct Visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 2010. doi: 10.1109/TVCG.2010.164

[38] T. R. Green. Cognitive dimensions of notations. *People and computers V*, pp. 443–460, 1989.

[39] Y. Guo, C. Binnig, and T. Kraska. What you see is not what you get!: Detecting simpson's paradoxes during data exploration. In *ACM SIGMOD Workshop on Human-In-the-Loop Data Analytics (HILDA)*, pp. 2:1–2:5. ACM, 2017. doi: 10.1145/3077257.3077266

[40] J. Heer and M. Agrawala. Software design patterns for information visualization. *IEEE transactions on visualization and computer graphics*, 12(5):853–860, 2006.

[41] J. Heer, J. M. Hellerstein, and S. Kandel. Predictive Interaction for Data Transformation. In *CIDR*, 2015.

[42] B. Hempel and R. Chugh. Semi-Automated SVG Programming via Direct Manipulation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, pp. 379–390, 2016. doi: 10.1145/2984511.2984575

[43] B. Hempel, J. Lubin, and R. Chugh. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, pp. 281–292, 2019. doi: 10.1145/3332165.3347925

[44] B. Hempel, J. Lubin, G. Lu, and R. Chugh. Deuce: A Lightweight User Interface for Structured Editing. In *Proceedings of the 40th International Conference on Software Engineering*, pp. 654–664, 2018.

[45] J. Hoffswell, A. Borning, and J. Heer. Setcola: High-level constraints for graph layout. In *Computer Graphics Forum*, vol. 37, pp. 537–548. Wiley Online Library, 2018.

[46] J. Hoffswell, A. Satyanarayan, and J. Heer. Visual Debugging Techniques for Reactive Data Visualization. In *Computer Graphics Forum*, vol. 35, pp. 271–280. Wiley Online Library, 2016. doi: 10.1111/cgf.12903

[47] J. Hoffswell, A. Satyanarayan, and J. Heer. Augmenting Code with In Situ Visualizations to Aid Program Understanding. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, 2018. doi: 10.1145/3173574.3174106

[48] H. Hofmann, L. Follett, M. Majumder, and D. Cook. Graphical tests for power comparison of competing designs. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2441–2448, 2012. doi: 10.1109/TVCG.2012.230

[49] M. Hogräfer and H.-J. Schulz. *ReVize: A Library for Visualization Toolchaining with Vega-Lite*. Eurographics, 2019. ISSN: 2617-4855. doi: handle/10.2312/stag20191375

[50] A. K. Hopkins, M. Correll, and A. Satyanarayan. VisuaLint: Sketchy In Situ Annotations of Chart Construction Errors. In *Computer Graphics Forum*, 2020. doi: 10.1111/cgf.13975

[51] N. Hynes, D. Sculley, and M. Terry. The data linter: Lightweight, automated sanity checking for ml data sets. In *NIPS: Workshop on Systems for ML and Open Source Software*, 2017.

[52] H. Kang and P. J. Guo. Omnicode: A novice-oriented live programming environment with always-on run-time value visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, pp. 737–745, 2017.

[53] M. B. Kery, D. Ren, F. Hohman, D. Moritz, K. Wongsuphasawat, and K. Patel. mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. ACM, 2020.

[54] Kibana. Query string query. `https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-query-string-query.html`, 2021. Accessed 1/3/2022.

[55] Kibana. Vega. `https://www.elastic.co/guide/en/kibana/current/vega.html`, 2021. Accessed 1/3/2022.

[56] N. W. Kim, E. Schweickart, Z. Liu, M. Dontcheva, W. Li, J. Popovic, and H. Pfister. Data-Driven Guides: Supporting Expressive Design for Information Graphics. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):491–500, 2016. doi: 10.1109/TVCG.2016.2598620

[57] W. Kim, B.-J. Choi, E.-K. Hong, S.-K. Kim, and D. Lee. A taxonomy of dirty data. *Data Mining and Knowledge Discovery*, 7(1):81–99, 2003. doi: 10.1023/A:1021564703268

[58] Y. Kim and J. Heer. Gemini: A grammar and recommender system for animatedtransitions in statistical graphics. *IEEE Transactions on Visualization and Computer Graphics*, 2021.

[59] Y. Kim, K. Wongsuphasawat, J. Hullman, and J. Heer. GraphScape: A Model for Automated Reasoning about Visualization Similarity and Sequencing. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pp. 2628–2638. ACM, 2017. doi: 10.1145/3025453.3025866

[60] G. Kindlmann, C. Chiw, N. Seltzer, L. Samuels, and J. Reppy. Diderot: a domain-specific language for portable parallel scientific visualization and image analysis. *IEEE transactions on visualization and computer graphics*, 22(1):867–876, 2015.

[61] G. Kindlmann and C. Scheidegger. An algebraic process for visualization design. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2181–2190, 2014. doi: 10.1109/TVCG.2014.2346325

[62] L. C. Klopfenstein, S. Delpriori, and A. Ricci. Adapting a conversational text generator for online chatbot messaging. In *International Conference on Internet Science*, pp. 87–99. Springer, 2018.

[63] A. Ko. tweet, November 2021. `https://twitter.com/amyjko/status/1458537839939895299` content:
Programming language dystopia:
· Syntax? JSON
· Semantics? Stale GitHub wiki
· Type checking? JSON schema validation
· Abstraction mechanism? Whatever you can fit in a string
· Autocomplete? Stack Overflow
· Debugging? Not even print statements.

[64] A. J. Ko and B. A. Myers. Citrus: a language and toolkit for simplifying the creation of structured editors for code and data. In *Proceedings of the 18th annual ACM symposium on User interface software and technology*, pp. 3–12, 2005.

[65] A. J. Ko and B. A. Myers. Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, pp. 387–396, 2006.

[66] Y. S. Kristiansen and S. Bruckner. Visception: An interactive visual framework for nested visualization design. *Computers & Graphics*, 92:13–27, 2020.

[67] Y. S. Kristiansen, L. Garrison, and S. Bruckner. Semantic snapping for guided multi-view visualization design. *IEEE Transactions on Visualization and Computer Graphics*, 2021.

[68] S. Lau, I. Drosos, J. M. Markel, and P. J. Guo. The design space of computational notebooks: An analysis of 60 systems in academia and industry. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 1–11. IEEE, 2020.

[69] D. J.-L. Lee, V. Setlur, M. Tory, K. G. Karahalios, and A. Parameswaran. Deconstructing categorization in visualization recommendation: A taxonomy and comparative study. *IEEE Transactions on Visualization and Computer Graphics*, 2021.

[70] Y. Y. Lee, N. Chen, and R. E. Johnson. Drag-and-drop refactoring: Intuitive and efficient program transformation. In *2013 35th International Conference on Software Engineering (ICSE)*, pp. 23–32. IEEE, 2013.

[71] S. Lerner. Projection boxes: On-the-fly reconfigurable visualization for live programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pp. 1–7, 2020.

[72] T. Lieber, J. R. Brandt, and R. C. Miller. Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 2481–2490, 2014.

[73] Z. Liu, J. Thompson, A. Wilson, M. Dontcheva, J. Delorey, S. Grigg, B. Kerr, and J. Stasko. Data Illustrator: Augmenting Vector Design Tools with Lazy Data Binding for Expressive Visualization Authoring. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pp. 1–13. ACM, 2018. doi: 10.1145/3173574.3173697

[74] A. Lunzer and A. McNamara. It aint necessarily so: Checking charts for robustness. *IEEE VisWeek Poster Proceedings*, 2014.

[75] S. L'Yi, Q. Wang, F. Lekschas, and N. Gehlenborg. Gosling: A grammar-based toolkit for scalable and interactive genomics data visualization. *IEEE transactions on visualization and computer graphics*, 2022.

[76] J. Mackinlay, P. Hanrahan, and C. Stolte. Show Me: Automatic Presentation for Visual Analysis. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1137–1144, 2007.

[77] A. McNutt. On the potential of zines as a medium for visualization. In *2021 IEEE Visualization Conference (VIS)*, pp. 176–180. IEEE, 2021.

[78] A. McNutt, A. Crisan, and M. Correll. Divining Insights: Visual Analytics Through Cartomancy. *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing System*, 2020.

[79] A. McNutt and G. Kindlmann. Linting for Visualization: Towards a Practical Automated Visualization Guidance System. In *VisGuides: 2nd Workshop on the Creation, Curation, Critique and Conditioning of Principles and Guidelines in Visualization*, 2018.

[80] A. McNutt, G. Kindlmann, and M. Correll. Surfacing Visualization Mirages. *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 2020. doi: 10.1145/3313831.3376420

[81] A. M. McNutt. What are table cartograms good for anyway? an algebraic analysis. *Computer Graphics Forum*, 40(3):61–73, 2021. doi: 10.1111/cgf.14289

[82] A. M. McNutt and R. Chugh. Integrated visualization editing via parameterized declarative templates. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pp. 1–14, 2021.

[83] A. M. McNutt, L. Huang, and K. Koenig. Visualization for villainy. *alt.vis*, 2021.

[84] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[85] Microsoft. SandDance. `https://www.microsoft.com/en-us/research/project/sanddance/`, 2011.

[86] Microsoft. vscode-json-languageservice. `https://github.com/microsoft/vscode-json-languageservice`, 2021.

[87] D. Moritz, C. Wang, G. L. Nelson, H. Lin, A. M. Smith, B. Howe, and J. Heer. Formalizing Visualization Design Knowledge as Constraints: Actionable and Extensible Models in Draco. *IEEE Transactions on Visualization and Computer Graphics*, 2019. doi: 10.1109/TVCG.2018.2865240

[88] L. Muth. One chart, twelve charting libraries, 2016. `https://lisacharlottemuth.com/2016/05/17/one-chart-code/`.

[89] D. Norman. *The design of everyday things: Revised and expanded edition*. Basic books, 2013.

[90] C. Omar, D. Moon, A. Blinn, I. Voysey, N. Collins, and R. Chugh. Filling typed holes with live guis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 511–525, 2021.

[91] C. Omar, Y. S. Yoon, T. D. LaToza, and B. A. Myers. Active code completion. In *2012 34th International Conference on Software Engineering (ICSE)*, pp. 859–869. IEEE, 2012.

[92] D. Park, S. M. Drucker, R. Fernandez, and N. Elmqvist. Atom: A Grammar for Unit Visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 24(12):3032–3043, 2018. doi: 10.1109/TVCG.2017.2785807

[93] W. C. Payne, Y. Bergner, M. E. West, C. Charp, R. B. B. Shapiro, D. A. Szafir, E. V. Taylor, and K. DesPortes. danceon: Culturally responsive creative computing. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pp. 1–16, 2021.

[94] H. Peleg, R. Gabay, S. Itzhaky, and E. Yahav. Programming with a read-eval-synth loop. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.

[95] C. Penner. jet. `https://github.com/ChrisPenner/jet/`, 2021.

[96] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč. Foundations of JSON Schema. In *Proceedings of the 25th International Conference on World Wide Web*, 2016. doi: 10.1145/2872427.2883029

[97] X. Pu, M. Kay, S. M. Drucker, J. Heer, D. Moritz, and A. Satyanarayan. Special interest group on visualization grammars. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*, pp. 1–3, 2021.

[98] Z. Qu and J. Hullman. Keeping multiple views consistent: Constraints, validations, and exceptions in visualization authoring. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):468–477, 2017. doi: 10.1109/TVCG.2017.2744198

[99] P. Rautek, S. Bruckner, M. E. Gröller, and M. Hadwiger. Vislang: A system for interpreted domain-specific languages for scientific visualization. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2388–2396, 2014.

[100] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009.

[101] H. Romat, N. Henry Riche, K. Hinckley, B. Lee, C. Appert, E. Pietriga, and C. Collins. Activeink: (th) inking with data. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pp. 1–13, 2019. doi: 10.1145/3290605.3300272

[102] B. Saket, S. Huron, C. Perin, and A. Endert. Investigating Direct Manipulation of Graphical Encodings as a Method for User Interaction. *IEEE Transactions on Visualization and Computer Graphics*, 2019. doi: 10.1109/TVCG.2019.2934534

[103] B. Saket, L. Jiang, C. Perin, and A. Endert. Liger: Combining Interaction Paradigms for Visual Analysis. *arXiv*, p. arXiv:1907.08345, July 2019.

[104] B. Saket, H. Kim, E. T. Brown, and A. Endert. Visualization by Demonstration: An Interaction Paradigm for Visual Data Exploration. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):331–340, 2016. doi: 10.1109/TVCG.2016.259883

[105] A. Satyanarayan and J. Heer. Lyra: An Interactive Visualization Design Environment. In *Eurographics Conference on Visualization*, vol. 33, p. 10, 2014. doi: 10.1111/cgf.12391

[106] A. Satyanarayan, B. Lee, D. Ren, J. Heer, J. Stasko, J. Thompson, M. Brehmer, and Z. Liu. Critical Reflections on Visualization Authoring Systems. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):461–471, 2020. doi: 10.1109/TVCG.2019.2934281

[107] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics*, 2016. doi: 10.1109/TVCG.2016.2599030

[108] A. Satyanarayan, R. Russell, J. Hoffswell, and J. Heer. Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):659–668, 2016. doi: 10.1109/TVCG.2015.2467091

[109] A. Satyanarayan, K. Wongsuphasawat, and J. Heer. Declarative interaction design for data visualization. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*, pp. 669–678, 2014.

[110] A. Satyanarayan, K. Wongsuphasawat, and J. Heer. Declarative Interaction Design for Data Visualization. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*, pp. 669–678. ACM, 2014. doi: 10.1145/2642918.2647360

[111] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés. A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, 42(9):805–824, 2016. doi: 10.1109/TSE.2016.2532875

[112] C. Severance. Discovering javascript object notation. *Computer*, 45(4):6–8, 2012.

[113] L. Shen, X. Chen, R. Liu, H. Wang, and G. Ji. Domain-specific language techniques for visual computing: A comprehensive study. *Archives of Computational Methods in Engineering*, 28(4):3113–3134, 2021.

[114] B. Shneiderman. Direct manipulation for comprehensible, predictable and controllable user interfaces. In *Proceedings of the 2nd international conference on Intelligent user interfaces*, pp. 33–39, 1997.

[115] C. Stolte, D. Tang, and P. Hanrahan. Polaris: A System for Query, Analysis, and Visualization of Multidimensional Relational Databases. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):52–65, 2002.

[116] N. Tang, E. Wu, and G. Li. Towards democratizing relational data visualization. In *Proceedings of the 2019 International Conference on Management of Data*, pp. 2025–2030. ACM, 2019. doi: 10.1145/3299869.3314029

[117] W. Tao, X. Hou, A. Sah, L. Battle, R. Chang, and M. Stonebraker. Kyrix-s: Authoring scalable scatterplot visualizations of big data. *IEEE Transactions on Visualization and Computer Graphics*, 2020.

[118] J. Tran O'Leary, K. Lee, and N. Peek. A grammar of digital fabrication machines. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*, pp. 1–6, 2021.

[119] T. Tsandilas. Structgraphics: Flexible visualization design through data-agnostic and reusable graphical structures. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):315–325, 2020.

[120] A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[121] J. VanderPlas, B. E. Granger, J. Heer, D. Moritz, K. Wongsuphasawat, A. Satyanarayan, E. Lees, I. Timofeev, B. Welsh, and S. Sievert. Altair: Interactive Statistical Visualizations for Python. *Journal of Open Source Software*, 3(32):1057, 2018. doi: 10.21105/joss.01057

[122] Vega. Editor/IDE for Vega and Vega-Lite, 2020. https://vega.github.io/editor/.

[123] Vega. Vega-Lite Documentation, 2020. Accessed August 19, 2020.

[124] B. Victor. Drawing Dynamic Visualizations, May 2013.

[125] C. Wang, Y. Feng, R. Bodik, A. Cheung, and I. Dillig. Visualization by Example. *Proceedings of the ACM on Programming Languages*, 4(POPL):49:1–49:28, Dec. 2019. doi: 10.1145/3371117

[126] C. Wang, Y. Feng, R. Bodik, I. Dillig, A. Cheung, and A. J. Ko. Falx: Synthesis-powered visualization authoring. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pp. 1–15, 2021.

[127] Y. Wang. Deck.gl: Large-scale web-based visual analytics made easy. *arXiv preprint arXiv:1910.08865*, 2019.

[128] Webpack. Webpack. `https://webpack.js.org/`, 2022.

[129] B. Whitworth. Polite Computing. *Behaviour & Information Technology*, 2005. doi: 10.1080/01449290512331333700

[130] H. Wickham, D. Cook, H. Hofmann, and A. Buja. Graphical inference for infovis. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):973–979, 2010. doi: 10.1109/TVCG.2010.161

[131] Wikimedia. Extension: Graph/demo. `https://web.archive.org/web/20211027024118/https://www.mediawiki.org/wiki/Extension:Graph/Demo`, 2021. Accessed Janurary 3, 2021.

[132] L. Wilkinson. *The Grammar of Graphics*. Springer, 2013.

[133] K. Wongsuphasawat. Encodable: Configurable grammar for visualization components. *IEEE Transactions on Visualization and Computer Graphics*, 2020.

[134] K. Wongsuphasawat. Navigating the wide world of data visualization libraries, September 2020.

[135] K. Wongsuphasawat, D. Moritz, A. Anand, J. Mackinlay, B. Howe, and J. Heer. Voyager: Exploratory Analysis via Faceted Browsing of Visualization Recommendations. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):649–658, 2015. doi: 10.1109/TVCG.2015.2467191

[136] K. Wongsuphasawat, Z. Qu, D. Moritz, R. Chang, F. Ouk, A. Anand, J. Mackinlay, B. Howe, and J. Heer. Voyager 2: Augmenting Visual Analysis with Partial View Specifications. In *Proceedings of the 2017*

CHI Conference on Human Factors in Computing Systems, pp. 2648–2659. ACM, 2017. doi: 10.1145/3025453.3025768

[137] J. Wood, A. Kachkaev, and J. Dykes. Design Exposition with Literate Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):759–768, 2019. doi: 10.1109/TVCG.2018.2864836

[138] Y. Wu, J. M. Hellerstein, and A. Satyanarayan. B2: bridging code and interactive visualization in computational notebooks. In *UIST '20: The 33rd Annual ACM Symposium on User Interface Software and Technology*, pp. 152–165. ACM, 2020. doi: 10.1145/3379337.3415851

[139] C. Yick. GraphPad. `https://www.figma.com/community/widget/1027276088284051809`, 2021.

[140] Z. Q. Zhou and L. Sun. Metamorphic testing of driverless cars. *Communications of the ACM*, 62(3):61–67, Feb. 2019. doi: 10.1145/3241979

[141] J. Zong, D. Barnwal, R. Neogy, and A. Satyanarayan. Lyra 2: Designing interactive visualizations by demonstration. *IEEE Transactions on Visualization and Computer Graphics*, 2021.