Vega-Lite: A Grammar of Interactive Graphics

Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer

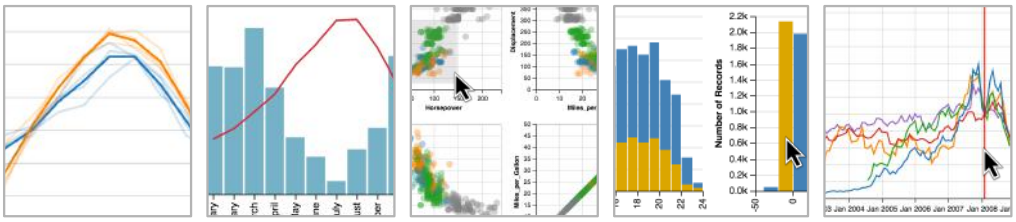

Fig. 1. Example visualizations authored with Vega-Lite. From left-to-right: layered line chart combining raw and average values, dual-axis layered bar and line chart, brushing and linking in a scatterplot matrix, layered cross-filtering, and an interactive index chart.

Abstract—We present Vega-Lite, a high-level grammar that enables rapid specification of *interactive* data visualizations. Vega-Lite combines a traditional grammar of graphics, providing visual encoding rules and a composition algebra for layered and multi-view displays, with a novel grammar of interaction. Users specify interactive semantics by composing *selections*. In Vega-Lite, a selection is an abstraction that defines input event processing, points of interest, and a predicate function for inclusion testing. Selections parameterize visual encodings by serving as input data, defining scale extents, or by driving conditional logic. The Vega-Lite compiler automatically synthesizes requisite data flow and event handling logic, which users can override for further customization. In contrast to existing reactive specifications, Vega-Lite selections decompose an interaction design into concise, enumerable semantic units. We evaluate Vega-Lite through a range of examples, demonstrating succinct specification of both customized interaction methods and common techniques such as panning, zooming, and linked selection.

Index Terms—Information visualization, interaction, systems, toolkits, declarative specification

1 Introduction

Grammars of graphics span a gamut of expressivity. Low-level grammars such as Protovis [3], D3 [4], and Vega [22] are useful for *explanatory* data visualization or as a basis for customized analysis tools, as their primitives offer fine-grained control. However, for *exploratory* visualization, higher-level grammars such as ggplot2 [27], and grammar-based systems such as Tableau (née Polaris [24]), are typically preferred as they favor conciseness over expressiveness. Analysts rapidly author partial specifications of visualizations; the grammar applies default values to resolve ambiguities, and synthesizes low-level details to produce visualizations.

High-level languages can also enable search and inference over the space of visualizations. For example, Wongsuphasawat et al. [30] introduced Vega-Lite to power the Voyager visualization browser. By providing a smaller surface area than the lower-level Vega language, Vega-Lite makes systematic enumeration and ranking of data transformations and visual encodings more tractable.

However, existing high-level languages provide limited support for interactivity. An analyst can, at most, enable a predefined set of common techniques (linked selections, panning & zooming, etc.) or parameterize their visualization with dynamic query widgets [21]. For custom, direct-manipulation interaction they must instead turn to imperative event handling callbacks. Recognizing that callbacks can be error-prone to author, and require complex static analysis to reason about, Satyanarayan et al. [23] recently formulated declarative interaction primitives for Vega. While these additions facilitate programmatic generation and retargeting of interactive visualizations, they remain low-level. Verbose specification impedes rapid authoring and hinders systematic exploration of alternative designs.

In this paper we extend Vega-Lite to enable concise, high-level specification of *interactive* data visualizations. To support expressive interaction methods, we first contribute an algebra to compose single-view Vega-Lite specifications into multi-view displays using *layer*, *concatenate*, *facet* and *repeat* operators. Vega-Lite's compiler infers how input data should be reused across constituent views, and whether scale domains should be unioned or remain independent.

Second, we contribute a high-level interaction grammar. With Vega-Lite, an interaction design is composed of *selections*: visual elements or data points that are chosen when input events occur. Selections parameterize visual encodings by serving as input data, defining scale extents, and providing *predicate* functions for testing or filtering items. For example, a rectangular "brush" is a common interaction technique for data visualization. In Vega-Lite, a brush is defined as a selection that holds two data points that correspond to its extents (e.g., captured when the mouse button is pressed and as it is dragged, respectively). Its predicate can be used to highlight visual elements that fall within the brushed region, and to materialize a dataset as input to other encodings. The selection can also serve as the scale domain for a secondary view, thereby constructing an overview + detail interaction.

For added expressivity, Vega-Lite provides a series of operators to *transform* a selection. Transforms can be triggered by input events as well, and manipulate selection points or predicate functions. For example, a *toggle* transform adds or removes a point from the selection, while a *project* transform modifies the predicate to define inclusion over specified data fields.

The Vega-Lite compiler synthesizes a low-level Vega specification [22] with the requisite data flow, and default event handling logic that a user can override. Through a range of examples, we demonstrate that Vega-Lite brings the advantages of high-level specification to interactive visualization. Common methods, including linked selection, panning, and zooming, as well as custom techniques (drawn from an established taxonomy [31]) can be concisely described. Moreover, selections, transformations, and their application to visual encodings decompose interaction into a parametric design space. We show how

- *Arvind Satyanarayan is with Stanford University. E-mail: arvindsatya@cs.stanford.edu.*
- *Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer are with the University of Washington. E-mails: {domoritz, kanitw, jheer}@uw.edu.*

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org.
Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx/

each of these parameters can be systematically varied to generate alternate interaction techniques for a given set of visual encodings. Such enumeration can be useful to explore alternative designs, and can aid higher-level reasoning about interaction — for example, recommending suitable interaction techniques as part of a design tool.

2 Related Work

Vega-Lite builds on prior work on grammars of graphics, visualization systems, and techniques for interactive selection and querying.

2.1 Grammar-Based Visual Encoding

Since the initial publication of Wilkinson's *The Grammar of Graphics* [29] in 1999, formal grammars for statistical graphics have grown increasingly popular as a way to succinctly specify visualizations. Wilkinson's work was quickly followed by the Stanford Polaris system [24], later commercialized as Tableau. Hadley Wickham's popular ggplot2 [27] and ggvis [20] packages implement variants of Wilkinson's model in the R statistical language. These tools eschew chart templates, which offer limited means of customization, in favor of combinatorial building blocks. Abstracting data models, graphical marks, visual encoding channels, scales and guides (i.e., axes and legends) yields a more expressive design space, and allows analysts to rapidly construct graphics for exploratory analysis [13]. Concise specification is achieved in part through ambiguity: users may omit details such as scale transforms (e.g., linear or log) or color palettes, which are then filled in using a rule-based system of smart defaults. More expressive lower-level (and thus more verbose) grammars, including those of Protovis [3], D3 [4], and Vega [22], have been widely used for creating explanatory and highly-customized graphics.

The design of Vega-Lite is heavily influenced by these works. Drawing from Wilkinson's grammar and Polaris/Tableau, Vega-Lite similarly represents basic plots using a set of encoding definitions that map data attributes to visual channels such as position, color, shape, and size, and may include common data transformations such as binning, aggregation, sorting, and filtering. Drawing from Vega, Vega-Lite uses a portable JSON (JavaScript Object Notation) syntax that permits generation from a variety of programming languages. Vega-Lite specifications are compiled to full Vega specifications, hence the expressive gamut of Vega-Lite is a strict subset of that of Vega. As we will later demonstrate, Vega-Lite sacrifices some expressiveness for dramatic gains in the conciseness and clarity of specification.

In terms of visual encoding, Vega-Lite differs most from other high-level grammars in its approach to multiple view displays. Each of these grammars supports faceting (or nesting) to construct trellis plots in which each cell similarly visualizes a different partition of the data. Both Wilkinson's grammar and Polaris/Tableau achieve this through a *table algebra* over data fields, which in turn determines spatial subdivisions. Tableau additionally supports the construction of multi-view dashboards via a different mechanism, with each view backed by a separate specification. In contrast, we contribute a *view algebra*: starting with unit specifications that define a single plot, Vega-Lite expresses composite views using operators for layering, horizontal or vertical concatenation, faceting, and parameterized repetition. When applicable, these operators will merge scale domains and properly align constituent views. Disparate views can also be combined into arbitrary dashboards, all within a unified algebraic model.

2.2 Specifying Interactions in Visualization Systems

Despite the central role of interaction in effective data visualization [13, 19], little work has been done to develop a grammar for specifying interaction techniques. Wilkinson's grammar includes no notion of interaction. Tableau supports common interaction techniques, but relies on mechanisms external to the visual encoding grammar. Early systems like GGobi [25] support common techniques as well, and provide imperative APIs for custom methods. However, such APIs make easy tasks needlessly complex, burdening developers with learning low-level execution details. More recent systems, including Protovis, D3, and VisDock [7], offer a typology of common techniques that can be applied to a visualization. Such top-down approaches, however, limit customization and composition. For example, D3's interactors encapsulate event processing, making it difficult to combine them if their events conflict (e.g., if dragging triggers brushing *and* panning).

The prior work perhaps most closely related to Vega-Lite is the Reactive Vega language [23]. Reactive Vega draws on Functional Reactive Programming techniques to formulate composable, declarative interaction primitives for data visualization. Reactive Vega models input events as continuous data streams. To succinctly define event streams of interest, Vega employs an *event selector* syntax, which Vega-Lite also uses for customized event logic. Event streams, in turn, drive dynamic variables called *signals*. Signals parameterize the remainder of the visualization specification, endowing it with reactive semantics. When a new event fires, it propagates to dependent signals; visual encodings that use them are automatically re-evaluated and re-rendered. This reactive approach is not only capable of expressing a diverse set of interactions [23], it is performant as well [22], with interactive performance at least twice as fast as the equivalent D3 program.

However, the resulting reactive specifications are low-level and verbose. Specifying common techniques can be time-consuming, requiring tens of lines of JSON, and it is difficult to know how to adapt techniques in pursuit of alternative designs. In contrast, Vega-Lite is a higher-level specification language, with primitives that decompose interaction design into a parametric space. Common methods require typically 1-2 lines of code, and design variations can be explored by systematically enumerating defined properties. Nevertheless, Reactive Vega provides a performant runtime and an "assembly language" to which Vega-Lite specifications are compiled.

2.3 Interactive Selection and Querying

Selection, often in the form of users clicking or lassoing visual items of interest, is a fundamental operation in user interfaces and has been well-studied in the context of data visualization. For example, in Snap-Together Visualization [17], multiple views are coordinated via "primary-" and "foreign-key actions," which propagate selected data tuples from one view to the others. Wilhelm [28] describes the need for such "indirect object manipulation" methods as an axiom of interactive data displays. Chen's compound brushing [6] provides a visual dataflow language for specifying a rich space of transformations of brush selections. More recently, Brunel [5] provides a special `#selection` data field that is dynamically populated with the elements a user interacts with, and can be used to link multiple views or filter input data. Similarly, RStudio's Shiny [21], an imperative web application layer, provides `brushedPoints` and `nearestPoints` functions which can be used throughout an R script to operate on selected elements.

Other systems have studied formally representing selections as data queries [28]. For example, brushing interactions in VQE [9] generate *extensional* queries that enumerate all items of interest; a form-based interface enables specification of *intensional* (declarative) queries. Individual point and brush selections in DEVise [15], known as *visual queries*, map to a declarative structure and are used to link together multiple views. With VIQING [18], rectangular "rubber band" selections are modeled as range extents, and views can be dropped on top of each other to join their underlying datasets. Heer et al. [12] demonstrate that by modeling a selection as a declarative query, interactive "query relaxation" can successively capture more items of interest.

Vega-Lite builds on this work by richly integrating an interactive *selection* abstraction with the primitives of visual encoding grammars. Vega-Lite selections are populated with one or more points of interest, in response to user interaction. Extensible *predicate* functions map selections to declarative queries, and allow a minimal set of "backing" points to represent the full space of selected points. Additional operators can transform a selection's predicate or backing points (e.g., offseting them to translate a brush selection or perform panning). Selections then parameterize visual encodings by serving as input data, defining scale extents, or using predicates to test or filter items. The end result is an enumerable, combinatorial design space of *interactive* statistical graphics, with concise specification of not only linking interactions, but panning, zooming, and custom techniques as well.

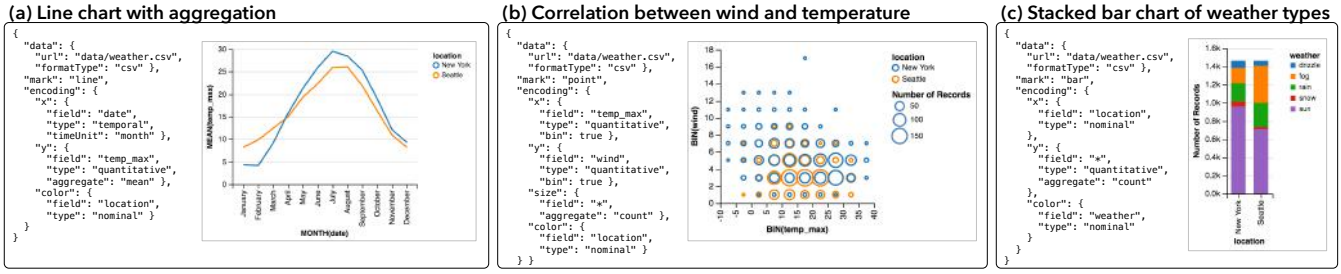

Fig. 2. Vega-Lite *unit* specifications visualizing weather data. These examples demonstrate varied mark types and data transformations.

3 The Vega-Lite Grammar of Graphics

Vega-Lite combines a grammar of graphics with a novel grammar of interaction. In this section, we describe Vega-Lite's basic visual encoding constructs and an algebra for view composition. In prior work, Wongsuphasawat et al. [30] introduced the simplest Vega-Lite specification — here referred to as a *unit* specification — that defines a single Cartesian plot with a specific mark type to encode data (e.g., bars, lines, plotting symbols). Given multiple unit plots, we introduce *layer*, *concat*, *facet*, and *repeat* operators to provide an algebra for constructing *composite* views. This algebra can express layered plots, trellis plots, and arbitrary multiple view displays. Each operator is responsible for combining or aligning underlying scales and axes as needed.

3.1 Unit Specification

A unit specification describes a single Cartesian plot, with a backing *data* set, a given *mark-type*, and a set of one or more *encoding* definitions for visual *channels* such as position (*x*, *y*), *color*, *size*, etc. Formally, a unit view consists of a four-tuple:

unit := (data, transforms, mark-type, encodings)

The *data* definition identifies a data source, a relational table consisting of records (rows) with named attributes (columns). This data table can be subject to a set of *transforms*, including filtering and adding derived fields via formulas. The *mark-type* specifies the geometric object used to visually encode the data records. Legal values include *bar*, *line*, *area*, *text*, *rule* for reference lines, and plotting symbols (*point* & *tick*). The *encodings* determine how data attributes map to the properties of visual marks. Formally, an encoding is a seven-tuple:

encoding := (channel, field, data-type, value, functions, scale, guide)

Available visual encoding *channels* include spatial position (*x*, *y*), *color*, *shape*, *size*, and *text*. An *order* channel controls sorting of stacked elements (e.g., for stacked bar charts and the layering order of line charts). A *path* order channel determines the sequence in which points of a line or area mark are connected to each other. A *detail* channel includes additional group-by fields in aggregate plots.

The *field* string denotes a data attribute to visualize, along with a given *data-type* (one of *nominal*, *ordinal*, *quantitative* or *temporal*). Alternatively, one can specify a constant literal *value* to serve as the data field. The data field can additionally be transformed using *functions* such as binning, aggregation (sum, average, etc.), and sorting.

An encoding may also specify properties of a *scale* that maps from the data domain to a visual range, and a *guide* (axis or legend) for visualizing the scale. If not specified, Vega-Lite will automatically populate default properties based on the *channel* and *data-type*. For *x* and *y* channels, either a linear scale (for quantitative data) or an ordinal scale (for ordinal and nominal data) is instantiated, along with an axis. For *color*, *size*, and *shape* channels, suitable palettes and legends are generated. For example, quantitative color encodings use a single-hue luminance ramp, while nominal color encodings use a categorical palette with varied hues. Our default assignments largely follow the model of prior systems [24, 30].

Unit specifications are capable of expressing a variety of common, useful plots of both raw and aggregated data. Examples include bar charts, histograms, dot plots, scatter plots, line graphs, and area graphs. Our formal definitions are instantiated in a JSON (JavaScript Object Notation) syntax, as shown in Fig. 2.

3.2 View Composition Algebra

Given multiple *unit* specifications, *composite* views can be created using a set of composition operators. Here we describe the set of supported operators. We use the term *view* to refer to any Vega-Lite specification, whether it is a *unit* or *composite* specification.

3.2.1 Layer

The *layer* operator accepts multiple *unit* specifications to produce a view in which subsequent charts are plotted on top of each other. For example, a layered view could consist of one layer showing a histogram of a full data set, and another overlaying a histogram of a filtered subset (Fig. 11). The signature of the operator is:

layer([unit$_1$, unit$_2$, ...], resolve)

To create a layered view, we produce shared scales (if their types match) and merge guides by default. For example, we compute the union of the data domains for the *x* or *y* channel, for which we then generate a single scale. We believe this is a useful default for producing coherent and comparable layers. However, Vega-Lite can not enforce that a unioned domain is *semantically* meaningful. To prohibit layering of composite views with incongruent internal structures, the *layer* operator restricts its operands to be *unit* views.

To override the default behavior, users can specify strategies to *resolve* scales and guides using tuples of the form *(channel, scale|guide,*

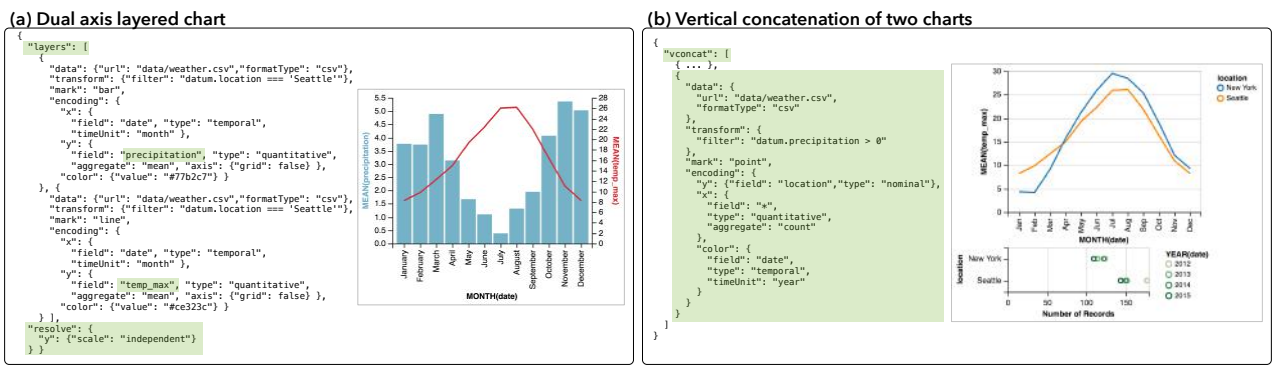

Fig. 3. (a) A dual axis chart that *layers* lines for temperature on top of bars for precipitation; each layer uses an independent y-scale. (b) The temperature line chart from Fig. 2(a) *concatenated* with rainy day counts in New York and Seattle; scales and guides for each plot are independent.

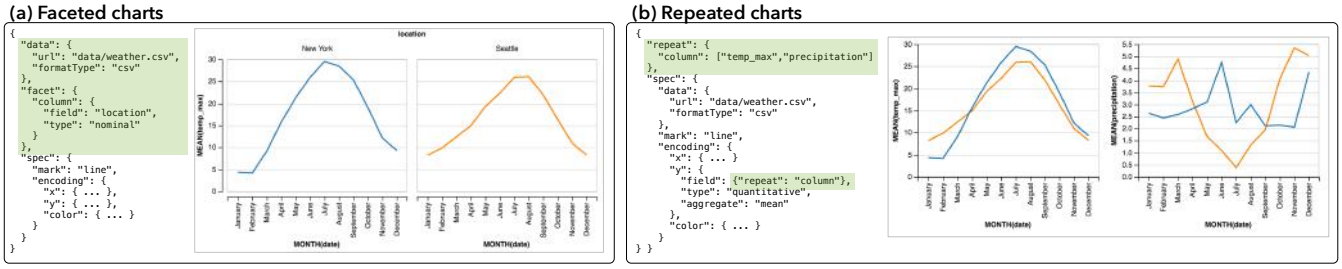

Fig. 4. (a) Weather data *faceted* by location; the y-axis is shared, and the underlying scale domains unioned, to enable easier comparison. (b) *Repetition* of different measures across columns; the *y* channel references the *column* template parameter to vary the encoding.

resolution), where *resolution* is one of *independent* or *union*. Independent scales and guides for each layer produce a dual-axis view, as shown in the layered plots in Fig. 3(a).

3.2.2 Concatenation

To place views side-by-side, Vega-Lite provides operators for horizontal and vertical concatenation. The signatures for these operators are:

$$hconcat([view_1, view_2, ...], resolve)$$
$$vconcat([view_1, view_2, ...], resolve)$$

If aligned spatial channels have matching data fields (e.g., the *y* channels in an *hconcat* use the same field), a shared scale and axis are used. Axis composition facilitates comparison across views and optimizes the underlying implementation. Fig. 3(b) concatenates the line chart from Fig. 2(a) with a dot plot, using independent scales.

3.2.3 Facet

While concatenation allows composition of arbitrary views, one often wants to set up multiple views in a parameterized fashion. The *facet* operator produces a trellis plot [1] by subsetting the data by the distinct values of a field. The signature of the facet operator is:

facet(channel, data, field, view, scale, axis, resolve)

The *channel* indicates if sub-plots should be laid out vertically (*row*) or horizontally (*column*). The given *data* source is partitioned using distinct values of the *field*. The *view* specification provides a template for the sub-plots, inheriting the backing *data* for each partition from the operator. The *scale* and *axis* parameters specify how sub-plots are positioned and labeled. Fig. 4(a) demonstrates faceting into columns.

To facilitate comparison, scales and guides for quantitative fields are shared by default. This ensures that each facet visualizes the same data domain. However, for ordinal scales we generate independent scales by default to avoid unnecessary inclusion of empty categories, akin to Polaris' *nest* operator. When faceting by fiscal quarter and visualizing per-month data in each cell, one likely wishes to see three months per quarter, not twelve months of which nine are empty. Users can override the default behavior via the *resolve* component.

3.2.4 Repeat

The *repeat* operator generates multiple plots, but unlike *facet* allows full replication of a data set in each cell. For example, *repeat* can be used to create a scatterplot matrix (SPLOM), where each cell shows a different 2D projection of the same data table. The signature is:

repeat(channel, values, scale, axis, view, resolve)

Similar to *facet*, the *channel* parameter indicates if plots should divide by *row* or *column*. Rather than partition data according to a field, this operator generates one plot for each entry in a list of *values*. Encodings within the repeated *view* specification can refer to this provided *value* to parameterize the plot[1]. By default, scales and axes are independent, but legends are shared when data fields coincide. Like *facet*, the *scale* and *axis* components allow users to override defaults for how sub-plots are positioned and labeled, while *resolve* controls resolution of scales and guides within the plots themselves.

[1]As the *repeat* operator requires parameterization of the inner view, it is not strictly algebraic. It is possible to achieve algebraic "purity" via explicit repeated concatenation or by reformulating the repeat operator (e.g., by including rewrite rules that apply to the inner view specification). However, we believe the current syntax to be more usable and concise than these alternatives.

3.3 Nested Views

Composition operators can be combined to create more complex nested views or dashboards, with the output of one operator serving as input to a subsequent operator. For instance, a layer of two unit views might be repeated, and then concatenated with a different unit view. The one exception is the *layer* operator, which, as previously noted, only accepts unit views to ensure consistent plots. For concision, two dimensional faceted or repeated layouts can be achieved by applying the operators to the *row* and *column* channels simultaneously. When faceting a composite view, only the dataset targeted by the operator is partitioned; any other datasets specified in sub-views are replicated.

4 The Vega-Lite Grammar of Interaction

To support specification of interaction techniques, Vega-Lite extends the definition of unit specifications to also include a set of *selections*. Selections identify the set of points a user is interested in manipulating. In this section, we define the components of a selection, describe a series of transforms for modifying selections, and detail how selections can parameterize visual encodings to make them interactive.

4.1 Selection Components

We formally define a selection as an eight-tuple:

$$selection := (name, type, predicate, domain|range, event, init, transforms, resolve)$$

When an input *event* occurs, the selection is populated with *backing points* of interest. These points are the minimal set needed to identify all *selected points*. The selection *type* determines how many backing values are stored, and how the *predicate* function uses them to determine the set of selected points. Supported types include a single *point*, a *list* of points, or an *interval* of points.

A point selection is backed by a single datum, and its predicate tests for an exact match against properties of this datum. It can also function like a dynamic variable (or *signal* in Vega [23]), and can be invoked as such. For example, it can be referenced by name within a filter expression, or its values used directly for particular encoding channels. List selections, on the other hand, are backed by datasets into which points are inserted, modified or removed as events fire. Lists express discrete selections, as their predicates test for an exact match with at least one value in the backing dataset. The order of points in a list selection can be semantically meaningful, for example when a list selection serves as an ordinal scale domain. Fig. 5 illustrates how points are highlighted in a scatterplot using point and list selections.

Intervals are similar to list selections. They are backed by datasets, but their predicates determine whether an argument falls within the minimum and maximum extent defined by the backing points. Thus, they express continuous selections. The compiler automatically adds a rectangle mark, as shown in Fig. 6(a), to depict the selected interval. Users can customize the appearance of this mark via the `brush` keyword, or disable it altogether when defining the selection.

Predicate functions enable a minimal set of backing points to represent the full space of selected points. For example, with predicates,

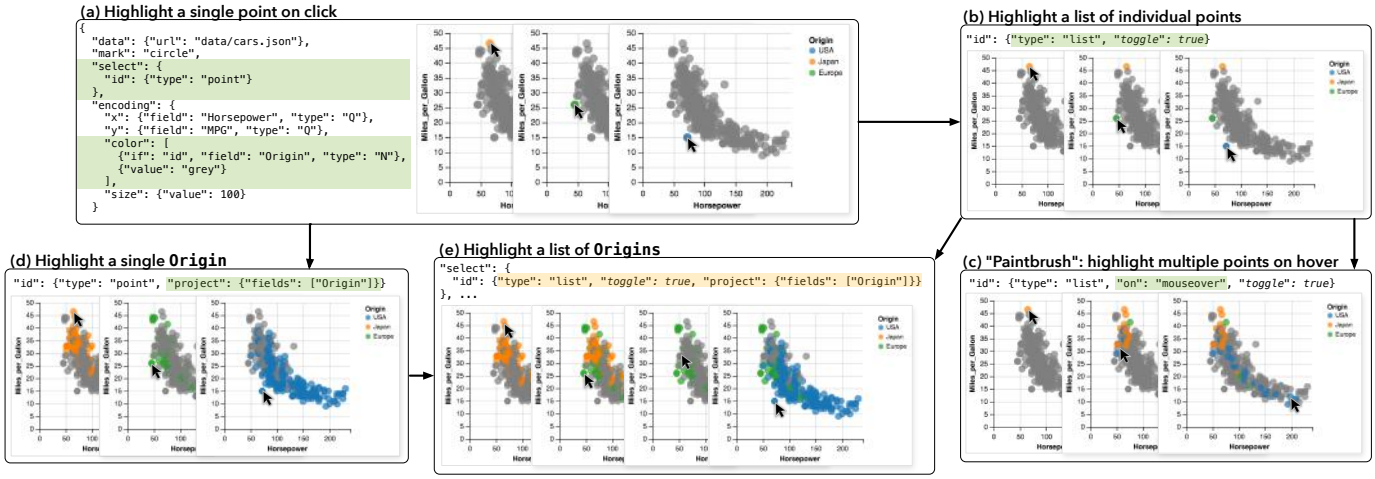

Fig. 5. (a) Adding a single *point* selection to parameterize the fill color of a scatterplot's circle mark. (b) Switching to a *list* selection, with the *toggle* transform automatically added (`true` enables default shift-click event handling). (c) Specifying a custom event trigger: the first point is selected on `mouseover` and subsequent points when the shift key is pressed (customizable via the *toggle* transform). (d) Using the *project* transform with a single-point selection to highlight all points with a matching `Origin`, and (e) combining it with a list selection to select multiple `Origins`.

an interval selection need only be backed by two points: the minimum and maximum values of the interval. While selection types provide default definitions, predicates can be customized to concisely specify an expressive space of selections. For example, a single point selection with a custom predicate of the form `datum.binned_price == selection.binned_price` is sufficient for selecting all data points that fall within a given bin.

By default, backing points lie in the data *domain*. For example, if the user clicks a mark instance, the underlying data tuple is added to the selection. If no tuple is available, event properties are passed through inverse scale transforms. For example, as the user moves their mouse within the data rectangle, the mouse position is inverted through the `x` and `y` scales and stored in the selection. Defining selections over data values, rather than visual properties, facilitates reuse across distinct views; each view may have different encodings specified, but are likely to share the same data domain. However, some interactions are inherently about manipulating visual properties — for example, interactively selecting the colors of a heatmap. For such cases, users can define selections over the visual *range* instead. When input events occur, visual elements or event properties are then stored.

The particular events that update a selection are determined by the platform a Vega-Lite specification is compiled on, and the input modalities it supports. By default we use mouse events on desktops, and touch events on mobile and tablet devices. A user can specify alternate events using Vega's event selector syntax [23]. For example, Fig. 5(c) demonstrates how `mouseover` events are used to populate a list selection. With the event selector syntax, multiple events are specified using a comma (e.g., `mousedown, mouseup` adds items to the selection when either event occurs). A sequence of events is denoted with the right-combinator. For example, `[mousedown, mouseup] > mousemove` selects all `mousemove` events that occur between a `mousedown` and a `mouseup` (otherwise known as "drag" events). Events can also be filtered using square brackets (e.g., `mousemove [event.pageY > 5]` for events at the top of the page) and throttled using braces (e.g., `mousemove{100ms}` populates a selection at most every 100 milliseconds).

Finally, selections can be *initialized* with specific backing points (we defer discussion of *transforms* and *resolve* to subsequent sections). Vega-Lite provides a built-in mechanism to initialize list and interval selections using the scales of the unit specification they are defined in. Doing so populates the selection with the given scales' domain or range, as appropriate for the selection, and parameterizes the scales to use the selection instead. By default, this occurs for the scales of the `x` and `y` channels, but alternate scales can be specified by the user. This step allows scale extents to be interactively manipulated, yet remain automatically initialized by the input data.

4.2 Selection Transforms

Analogous to data transforms, selection transforms manipulate the components of the selection they are applied to. For example, they may perform operations on the backing points, alter a selection's predicate function, or modify the input events that update the selection. We identify the following transforms as a minimal set to support both common and custom interaction techniques:

project(fields, channels): Alters a selection's predicate function to determine inclusion by matching only the given *fields*. Some fields, however, may be difficult for users to address directly (e.g., new fields introduced due to inline binning or aggregation transformations). For such cases, a list of *channels* may also be specified (e.g., `color`, `size`). Fig. 5(d, e) demonstrate how *project* can be used to select all points with matching `Origin` fields, for example. This transform is also used to restrict interval selections to a particular dimension (Fig. 6(c)) or to determine which scales initialize a selection.

toggle(event): This transform is automatically instantiated for uninitialized list selections. When the *event* occurs, the corresponding point is added or removed from a list selection's backing dataset. By default, the toggle *event* corresponds to the selection's event but with the shift key pressed. For example, in Fig. 5(b), additional points are added to the list selection on shift-click (where `click` is the default event for list selections). The selection in Fig. 5(c), however, specifies a custom `mouseover` event. Thus, additional points are inserted when the shift key is pressed and the mouse cursor hovers over a point.

translate(events, by): Offsets the spatial properties (or corresponding data fields) of backing points by an amount determined by the coordinates of the sequenced *events*. For example, on the desktop, drag events (`[mousedown, mouseup] > mousemove`) are used and the offset corresponds to the difference between where the `mousedown` and subsequent `mousemove` events occur. If no coordinates are available (e.g., as with keyboard events), an optional *by* argument should be specified. This transform respects the *project* transform as well, restricting movement to the specified dimensions. This transform is automatically instantiated for interval transforms, enabling movement of brushed regions (Fig. 6(b)) or panning of the visualization when scale extents initialize the selection (Fig. 7).

zoom(event, factor): Applies a scale factor, determined by the *event*, to the spatial properties (or corresponding data fields) of backing points. An optional *factor* should be specified, if it cannot be determined from the events (e.g., when the arrow keys are pressed).

nearest(): Computes a Voronoi decomposition, and augments the selection's event processing, such that the data value or visual element

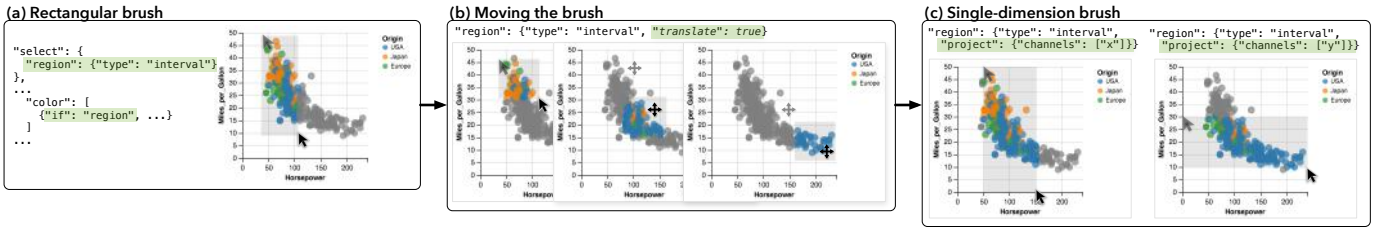

Fig. 6. (a) Adding a rectangular brush, as an *interval* selection, which can be (b) moved with the *translate* transform (automatically instantiated by the compiler) or (c) restricted to a single dimension with the *project* transform.

nearest the selection's triggering *event* is selected (approximating a Bubble Cursor [11]). Currently, the centroid of each mark instance is used to calculate the Voronoi diagram but we plan to extend this operator to account for boundary points as well (e.g., rectangle vertices).

Transforms can be composed. For example, the *toggle* and *nearest* transforms can be applied to a list selection to toggle the membership of a point nearest the user's mouse. Specifying a transform order is not necessary as the compilation step ensures commutativity. All transforms are first parsed, setting properties on an internal representation of a selection, before they are compiled to produce event handling and interaction logic. Moreover, additional transforms can be defined and registered with the system, and then invoked within the specification. In this way, the Vega-Lite language remains concise while ensuring extensibility for custom interactive behaviors.

4.3 Selection-Driven Visual Encodings

Once selections are defined, they parameterize visual encodings to make them interactive — visual encodings are automatically reevaluated as selections change. First, selections can be used to drive an `if-then-else` chain of logic within an encoding channel definition. Each data tuple participating in the encoding is evaluated against selection predicates in turn, and visual properties are set corresponding to the first branch that evaluates to `true`. For example, as shown in Fig. 5, the fill color of the scatterplot circles is determined by a data field if they fall within the `id` selection, or set to grey otherwise.

Next, selected points can be explicitly materialized and used as input data for other encodings within the specification. By default, this applies a selection's predicate against the data tuples (or visual elements) of the unit specification it is defined in. However, selections can also be materialized against arbitrary datasets; a *map* transform supports rewriting the predicate function in case of differing schemas. Using selections in this way enables linked interactions, including displaying tooltips or labels, and cross-filtering.

Besides serving as input data, a materialized selection can also define scale extents. Initializing a selection with scale extents offers a concise way of specifying this behavior within the same unit specification. For multi-view displays, selection names can be specified as the domain or range of a particular channel's scale. Doing so constructs

```
"select": {
  "region": {
    "type": "interval",
    "on": "[mousedown[event.shiftKey], mouseup] > mousemove"
  },
  "grid": {
    "type": "interval", "init": {"scales": true}, "zoom": true
    "translate": "[mousedown[!event.shiftKey], mouseup] > mousemove"
  }
}, ...
```

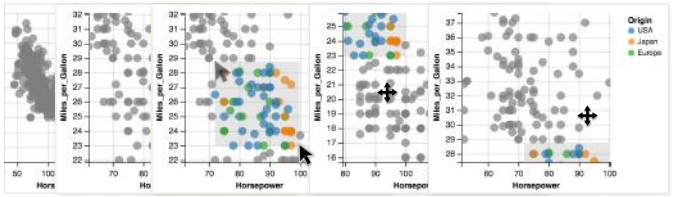

Fig. 7. Panning and zooming the scatterplot is achieved by first initializing a list selection with the *x* and *y* scale domain, and then applying *translate* and *zoom*. Alternate events are specified to prevent collision with the brushing interaction, previously defined in Fig. 6.

interactions that manipulate viewports, including panning & zooming (Fig. 7) and overview + detail (Fig. 9(a)).

In all three cases, selections can be composed using logical `OR`, `AND`, and `NOT` operators. As previously discussed, single-point selections offer an additional mechanism for parameterizing encodings. Properties of the backing point can be directly referenced within the specification, for example as part of a filter or compute expression, or to determine a visual encoding channel without the overhead of an `if-then-else` chain. For example, the position of the red rule in Fig. 9(b) is set to the `date` value of the `indexPt` selection.

4.4 Disambiguating Composite Selections

Selections are defined within unit specifications, providing a default context. For example, a selection's events are registered on the unit's mark instances, and materializing a selection applies its predicate against the unit's input data by default. When units are composed, however, selection definitions and applications become ambiguous.

Consider Fig. 8(a), which illustrates how a scatterplot matrix (SPLOM) is constructed by repeating a unit specification. To brush, we define an interval selection (`region`) within the unit, and use it to perform a linking operation by parameterizing the color of the circle marks. However, there are several ambiguities within this setup. Is there one `region` for the overall visualization, or one per cell? If the latter, which cell's `region` should be used? This ambiguity recurs when selections serve as input data or scale extents, and when selections share the same name across a layered or concatenated views.

Several strategies exist for resolving this ambiguity. By default, a *single* selection is created across all views. With our SPLOM example, this setting causes a single brush to be populated and shared across all cells. When the user brushes in a cell, points that fall within it are highlighted, and previous brushes are removed.

Users can specify an alternate ambiguity resolution when defining a selection. These schemes all construct one instance of the selection per view, and define which instances are used in determining inclusion. For example, setting a selection to resolve to *independent* creates one instance per view, and each unit uses only its own selection to determine inclusion. With our SPLOM example, this would produce the interaction shown in Fig. 8(b). Each cell would display its own brush, which would determine how only its points would be highlighted.

Selections can also be resolved to *union* or *intersect*. In these cases, all instances of a selection are considered in concert: a point falls within the overall selection if it is included in, respectively, at least one of the constituents or all of them. More concretely, with the SPLOM example, these settings would continue to produce one brush per cell, and points would highlight when they lie within at least one brush (*union*) or if they are within every brush (*intersect*) as shown in Fig. 8(c, d). We also support *union others* and *intersect others* resolutions, which function like their full counterparts except that a unit's own selection is not part of the inclusion determination. These latter methods support cross-filtering interactions, as in Figs. 10 & 11, where interactions within a view should not filter itself.

5 The Vega-Lite Compiler

The Vega-Lite compiler ingests a JSON specification and outputs a lower-level Reactive Vega specification (also expressed as JSON). There are two main challenges when compiling Vega-Lite to Vega. First, there is no one-to-one correspondence between components of

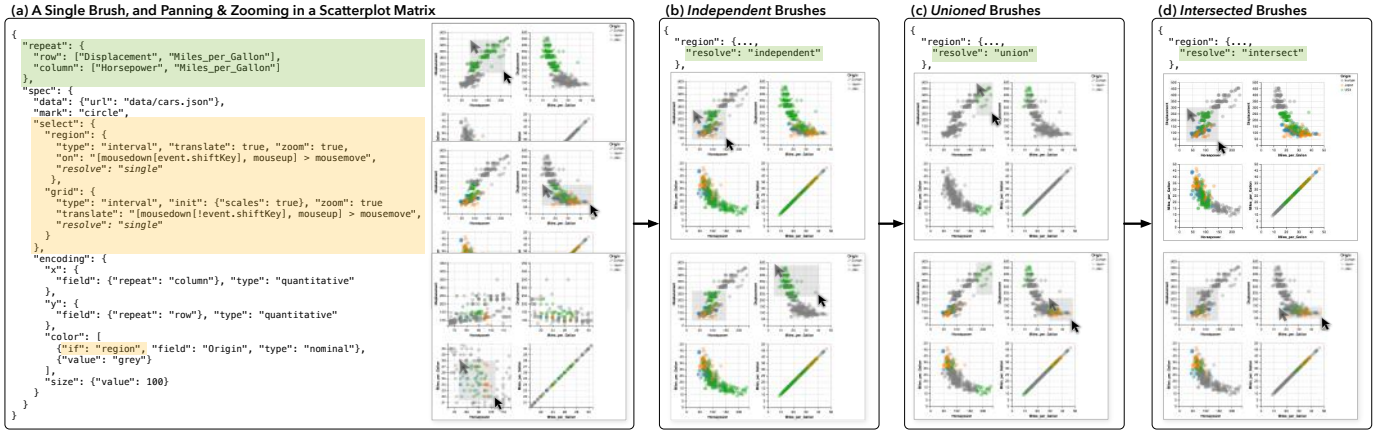

Fig. 8. (a) By adding a *repeat* operator, we compose the encoding and interactions from Fig. 7 into a scatterplot matrix. Users can brush, pan, and zoom within each cell, and the others update in response. By default, composite selections are resolved to a *single* global selection: brushing in a cell replaces previous brushes. However, the resolution scheme can be set to (b) *independent*, such that each cell uses its own brush; (c) *union*, such that points highlight if they fall in any brush; and (d) *intersect*, such that points highlight only when they are within all brushes.

the Vega-Lite and Vega specifications. For instance, the compiler has to synthesize a single Vega data source, with transforms for binning and aggregation, from multiple Vega-Lite encoding definitions. Conversely, for a single definition of a Vega-Lite selection, the compiler might generate multiple Vega signals, data sources, and even parameterize scale extents. Second, to facilitate rapid authoring of visualizations, Vega-Lite specifications omit lower-level details including scale types and the properties of the visual elements such as the font size. The compiler must resolve the resulting ambiguities.

To overcome these challenges, the compiler generates the output Vega specification in four phases: *parse* ingests and disambiguates the Vega-Lite specification; *build* creates the necessary internal representations to map between Vega-Lite and Vega primitives; *merge* optimizes this representation to remove redundancies; and finally, *assemble* compiles this representation into a Vega specification.

In the first step, the compiler parses a Vega-Lite specification to disambiguate it. It does so primarily by applying rules crafted to produce perceptually effective visualizations. For example, if the color channel is mapped to an nominal field, and the user has not specified a scale domain, a categorical color palette is inferred. If the color is mapped to a quantitative field, a sequential color palette is chosen instead.

Next, the compiler builds an internal representation of this unambiguous specification, consisting of a tree of *models*. Each model represents a unit or composite view produced by the algebraic operators described in §3, and stores a series of *components*. Components are data structures that loosely correspond to Vega primitives (such as data sources, scales, and marks) and provide a mapping to Vega-Lite primitives. Thus, they allow the compiler to bridge the gulf between the two levels of abstraction. For example, the `DataComponent` details how the dataset should be loaded (e.g., is it embedded directly in the specification, or should it be loaded from a URL, and in what format), which fields should be aggregated or binned, and what filters and derivation calculations should be performed.

In this step, compile-time selection transforms (those not parameterized by events) are applied to the requisite components. For example, the *project* transform overrides the `SelectionComponent`'s predicate function, while the *nearest* transform augments the `MarkComponent` with a Voronoi diagram. This phase also constructs a special `LayoutComponent` to calculate suitable spatial dimensions for views. This component emits Vega data sources and transforms to calculate a bottom-up layout at runtime.

Once the necessary components have been built, the compiler performs a bottom-up traversal of the model tree to merge redundant components. This step is critical for ensuring that the resultant Vega specification does not perform unnecessary computation that might hinder interactive performance. To determine whether components can be merged, the compiler serializes them using a hash code and compares components of the same type. For example, when a scatterplot matrix is specified using the *repeat* operator, merging ensures that we only produce one scale for each row and column rather than two scales per cell ($2N$ versus $2N^2$ scales). Merging may introduce additional components if doing so results in a more optimal representation. For example, if multiple units within a composite specification load data from the same URL, a new `DataComponent` is created to load the data and the units are updated to inherit from it instead. This step also unions scale domains and resolves `SelectionComponents`.

The final phase assembles the requisite Vega specification. `SelectionComponents`, in particular, produce signals to capture events and the necessary backing points, and list and intervals construct data sources as well to hold multiple points. Each run-time selection transform (i.e., those that are triggered by an event) generates signals as well, and may augment the selection's data source with data transformations. For example, the *translate* transform adds a signal to capture an "anchor" position, to determine where panning begins, and another to calculate a "delta" from the anchor. These two signals then feed transforms that offset the backing points stored in the selection's data source, thereby moving the brush or panning the scales.

6 Example Interactive Visualizations

Vega-Lite's design is motivated by two goals: to enable rapid yet expressive specification of interactive visualizations, and to do so with concise primitives that facilitate systematic enumeration and exploration of design variations. In this section, we demonstrate how these goals are addressed using a range of example interactive visualizations. To evaluate expressivity, we choose examples that cover Yi et al.'s [31] taxonomy of interaction methods. The taxonomy identifies seven categories of techniques: *select*, to mark items of interest; *explore* to examine subsets of the data; *connect* to highlight related items within and across views; *abstract/elaborate* to vary the level of detail; *reconfigure* to show different arrangements of the data; *filter* to show elements conditionally; and, *encode*, to change the visual representations used. To assess authoring speed, we compare our specifications against canonical Reactive Vega examples [22, 23, 26]. Where applicable, we also show how construction of our examples can be systematically varied to explore alternate points in the design space.

Select. Fig. 5(a) provides the full Vega-Lite specification for a scatterplot where users can mark individual points of interest. It includes the simplest definition of a selection — a name and type — and illustrates how the mark color is parameterized by `if-then-else` logic.

Modifying a single property, *type*, as in Fig. 5(b), allows users to mark multiple points (*toggle* is automatically instantiated by the compiler, but we explicitly specify it in the figure for clarity). We can instead add *project* (Fig. 5(d)) such that marking a single point of interest highlights all other points that share particular data values — a

Fig. 9. (a) An overview+detail visualization is constructed by concatenating two unit specifications, with a selection in the first one parameterizing the x scale domain in the second. (b) An index chart uses a point selection to renormalize data based on the index point nearest the mouse cursor.

connect-type interaction. Such changes to the specification are not mutually exclusive, and can be composed as shown in Fig. 5(e).

By using the *interval* type, users can mark items of interest within a continuous region. As shown in Fig. 6(a), the compiler automatically adds a rectangle mark to depict the selection, and instantiates *translate* to allow it to be repositioned (Fig. 6(b)). In this context, *project* restricts the interval to a single dimension (Fig. 6(c)).

These specifications are an order of magnitude more concise than their Vega counterparts. With Vega-Lite, users need only specify the semantics of their interaction and the compiler fills in appropriate default values. For example, by default, individual points are selected on click and multiple points on shift-click. Users can override these defaults, sometimes producing a qualitatively different user experience. For example, one can instead update selections on `mouseover` to produce a "paint brush" interaction, as in Fig. 5(c). In contrast, with Vega, users need to manually author all the components of an interaction technique, including determining whether event properties need to be passed through scale inversions, creating necessary backing data structures, and adding marks to represent a brush component.

Explore & Encode. Vega-Lite's selections also enable accretive design of interactions. Consider our previous example of brushing a scatterplot. We can define an additional interval selection and initialize it using *scales* (Fig. 7). The compiler populates the selection with the x and y scale domains, parameterizes them to use it, and instantiates the *translate* and *zoom* transforms. Users can now brush, pan and zoom the scatterplot. However, the default definitions of the two interval selections collide: dragging produces a brush and pans the plot. This example illustrates that concise methods for overriding defaults can not only be useful (as in Fig. 5(c)) but also necessary. We override the default events that trigger the two interactions using Vega's event selector syntax [23]. As Fig. 7 shows, we specify that brushing only occurs when the user drags with the shift key pressed.

The Vega-Lite specification for panning and zooming is, once again, more succinct than the corresponding Vega example. However, it is more interesting to compare the latter against the output specification produced by the Vega-Lite compiler. The Vega example requires users

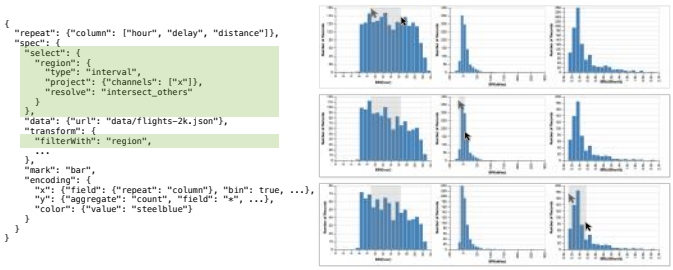

Fig. 10. An interval selection, resolved to *intersect others*, drives a cross filtering interaction. Brushing in one histogram filters and reaggregates the data in the others.

to manually specify their initial scale extents when defining the interaction. On the other hand, to enable data-driven initialization of interval selections, the Vega-Lite output calculates scale extents as part of a derived dataset in the output specification, with additional transformations to offset these calculations for the interaction. Such a construction is not idiomatic Vega, and would be unintuitive for users to construct manually. Thus, Vega-Lite's higher-level approach not only offers more rapid specification, but it can also enable interactions that a user may not realize are expressible with lower-level representations.

Moreover, by enabling this interaction through composable primitives (rather than a single, specific "pan and zoom" operator [4]), Vega-Lite also facilitates exploring related interactions in the design space. For example, using the *project* transform, we can author a separate selection for the x and y scales each, and selectively enable the *translate* and *zoom* transforms. While such a combination may not be desirable — panning only one scale while zooming the other — Vega-Lite's selections nevertheless allow us to systematically identify it as a possible design. Similarly, we could project over the color or size channels, thereby allowing users to interactively vary the mappings specified by these scales. For example, "panning" a heatmap's color legend to shift the data values considered high and low density. If the selections were defined over the visual *range*, users could instead shift the colors used in a sequential color scale. However, while such *encode*-type techniques are expressible in Vega-Lite, they do not currently produce valid Vega specifications as Vega does not yet support interactive legends. Thus, events do not fire over them.

Connect. We can wrap our previous example, from Fig. 7, in a *repeat* operator to construct a scatterplot matrix (SPLOM) as shown in Fig. 8. With no further modifications, all our previous interactions now work within each cell of the SPLOM and are synchronized across the others. For example, dragging pans not only the particular cell the user is in, but related cells along shared axes. Similarly, dragging with the shift key pressed produces a brush in the current cell, and highlights points across all cells that fall within it.

As its name suggests, the repeat operator creates one instance of the child specification for the given parameters. By default, to provide a consistent experience when moving from a unit to a composite specification, Vega-Lite creates a *single* instance of the selection that is populated and shared between all repeated instances (Fig. 8(a)). With the *resolve* property, users can specify alternate disambiguation methods including creating an independent brush for each cell, unioning the brushes, or intersecting them (Fig. 8(b, c, d) respectively). If selections are initialized by scales or parameterize them, only a single selection is supported for consistency with the composition algebra.

With this example, it is more instructive to compare the amount of effort required, with Vega-Lite and Vega, to move from a single interactive scatterplot to an interactive SPLOM. While the Vega specifications for the two are broadly similar, the latter requires an extra level of indirection to identify the specific cell a user is interacting in, and to ensure that the correct data values are used to determine inclusion

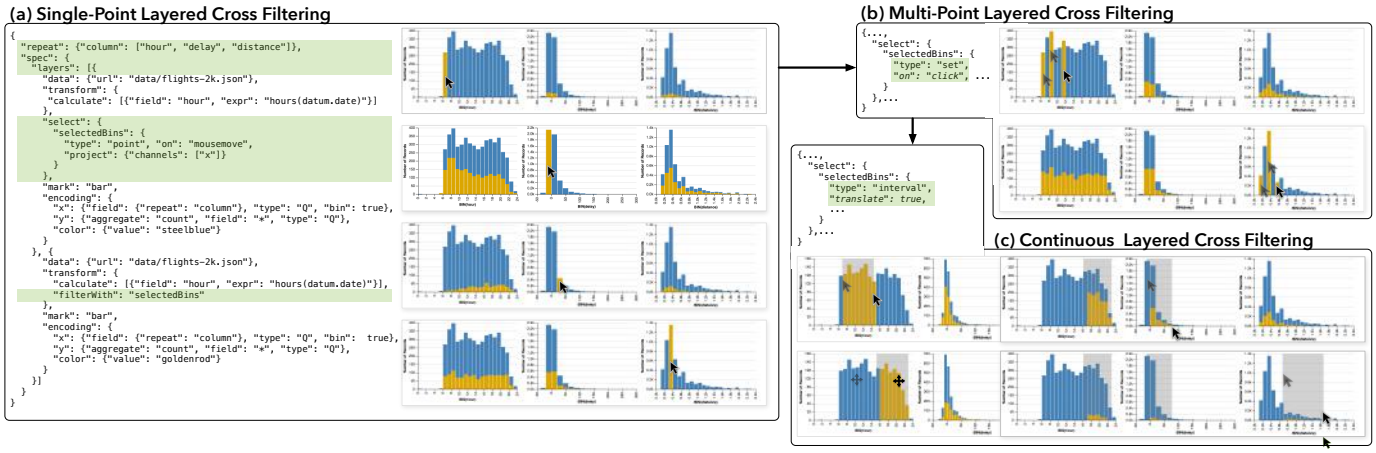

Fig. 11. Layered cross filtering interaction of binned histograms by (a) repeating a unit specification with a *point* selection that is materialized to serve as the input data for the second layer. When a user hovers over a bar in one histogram, bars in the others highlight to depict the distributions of the selected bin. By varying the selection *type*, users can (b) select multiple bins on shift-click or (c) brush a continuous interval.

within the brush. In Vega-Lite, this complexity is succinctly encapsulated by the *resolve* keyword which, as discussed, can be systematically varied to explore alternatives. Mimicing Vega-Lite's *union* and *intersect* behaviors is not trivial, and requires unidiomatic Vega once more. Users cannot simply duplicate the interaction logic for each cell manually, as the dimensions of the SPLOM are determined by data.

Abstract/Elaborate. Thus far, selections have parameterized scale extents through the initialization step. Previous examples have demonstrated how visualized data can be abstracted/elaborated via zooming. In Fig. 9(a), we show how a selection defined in one unit specification can be explicitly given as the scale domain of another in a concatenated display. Doing so creates an overview + detail interaction: brushing in the top (overview) chart displays only the brushed items at a higher resolution in the larger (detail) chart at the bottom.

Reconfigure. Fig. 9(b), uses a point selection to interactively normalize stock price time series data as the user moves their mouse across the chart. We apply the *nearest* transform, which calculates a Voronoi tessellation to accelerate the selection. By projecting the `date` field, the point selection represents both a single data value as well a set of values that share the selected `date`. Thus, we can reference the point selection directly, to position the red vertical rule, and also materialize it as part of the *lookup* data transform.

Filter. As selections provide a predicate function, it is trivial to use them to filter a dataset. Fig. 10, for example, presents a concise specification to enable filtering across three distinct binned histograms. It uses a *repeat* operator with a single-dimensional interval selection over the bins set to *intersect others*. The *filterWith* data transform applies the selection against the backing datasets such that only data values that fall within the selection are displayed. Thus, as the user brushes in one histogram, the datasets that drive each of the other two are filtered, the data values are re-aggregated, and the bars rise and fall. As with other interval selections, the Vega-Lite compiler automatically instantiates the *translate* transform, allowing users to drag brushes around rather than having to reselect them from scratch.

The *filterWith* data transform can also be used to materialize the selection as an input dataset for secondary views. For instance, one drawback of cross-filtering as in Fig. 10 is that users only see the selected values, and lose the context of the overall dataset. Instead of applying the selection back onto the input dataset, we can instead materialize it as an overlay (Fig. 11). Now, as the user brushes in one histogram, bars highlight to visualize the proportion of the overall distribution that falls within the brushed region(s). With this setup, it is necessary to change the selection's resolution to simply *intersect*, such that bars in the brushed plot also highlight during the interaction.

7 Discussion

The examples demonstrate that Vega-Lite specifications are more concise than those of the lower-level Vega language, and yet are sufficiently expressive to cover an interactive visualization taxonomy. Moreover, we have shown how primitives can be systematically enumerated to facilitate exploration of alternative designs. Nevertheless, we identify two classes of limitations that currently exist.

First, there are limitations that are a result of how our formal model has been reified in the current Vega-Lite implementation. In particular, components that are determined at compile-time cannot be interactively manipulated. For example, a selection cannot specify alternate fields to bin or aggregate over. Similarly, more complex selection types (e.g., lasso selections) cannot be expressed as the Vega-Lite system does not support arbitrary path marks. Such limitations can be addressed with future versions of Vega-Lite, or alternate systems that instantiate its grammar. For example, rather than a *compiler*, interactions could parameterize the entirety of a specification within a Vega-Lite *interpreter*.

The second class of limitations are inherent to the model itself. As a higher-level grammar, our model favors conciseness over expressivity. The available primitives ensure that common methods can be rapidly specified, with sufficient composition to enable more custom behaviors as well. However, highly specialized techniques, such as querying time-series data via relaxed selections [14], cannot be expressed by default. Fortunately, our formulation of selections, which decouple backing points from selected points via a predicate function, provide a useful abstraction for extending our base semantics with new, custom transforms. For example, the aforementioned technique could be encapsulated in a *relax* transform applicable to list selections.

While our selection abstraction supports *interactive* linking of marks, our view algebra does not yet provide means of *visually* linking marks across views (e.g., as in the Domino system [10]). Our view algebra might be extended with support for connecting corresponding marks. For example, points in repeated dot plots could be visually linked using line segments to produce a parallel coordinates display.

An early version of Vega-Lite is used to automatically recommend static plots as part of the Voyager browser [30]. Voyager leverages perceptual *effectiveness criteria* [2, 8, 16] to rank candidate visual encodings. One promising avenue for future work is to develop models and techniques to analogously recommend suitable interaction methods for given visualizations and underlying data types.

Vega-Lite is an open source system available at `http://vega.github.io/vega-lite/`. By offering a multi-view grammar of graphics tightly integrated with a grammar of interaction, Vega-Lite facilitates rapid exploration of design variations. Ultimately, we hope that it enables analysts to produce and modify interactive graphics with the same ease with which they currently construct static plots.

Acknowledgments

This work was supported by an SAP Stanford Graduate Fellowship, the Intel Big Data ISTC, the Moore Foundation, and DARPA XDATA.

REFERENCES

[1] R. A. Becker, W. S. Cleveland, and M.-J. Shyu. The visual design and control of trellis display. *Journal of computational and Graphical Statistics*, 5(2):123–155, 1996.

[2] J. Bertin. *Semiology of graphics: diagrams, networks, maps*. University of Wisconsin press, 1983.

[3] M. Bostock and J. Heer. Protovis: A graphical toolkit for visualization. *IEEE Trans. Visualization & Comp. Graphics*, 15(6):1121–1128, 2009.

[4] M. Bostock, V. Ogievetsky, and J. Heer. D3: Data-Driven Documents. *IEEE Trans. Visualization & Comp. Graphics*, 17(12):2301–2309, 2011.

[5] Brunel Visualization. `https://developer.ibm.com/open/brunel-visualization/`, June 2016.

[6] H. Chen. Compound brushing [dynamic data visualization]. In *Information Visualization, 2003. INFOVIS 2003. IEEE Symposium on*, pages 181–188. IEEE, 2003.

[7] J. Choi, D. G. Park, Y. L. Wong, E. Fisher, and N. Elmqvist. Visdock: A toolkit for cross-cutting interactions in visualization. *Visualization and Computer Graphics, IEEE Transactions on*, 21(9):1087–1100, 2015.

[8] W. S. Cleveland and R. McGill. Graphical perception: Theory, experimentation, and application to the development of graphical methods. *Journal of the American Statistical Association*, 79(387):531–554, 1984.

[9] M. Derthick, J. Kolojejchick, and S. F. Roth. An interactive visual query environment for exploring data. In *Proceedings of the 10th annual ACM symposium on User interface software and technology*, pages 189–198. ACM, 1997.

[10] S. Gratzl, N. Gehlenborg, A. Lex, H. Pfister, and M. Streit. Domino: Extracting, comparing, and manipulating subsets across multiple tabular datasets. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2023–2032, 2014.

[11] T. Grossman and R. Balakrishnan. The bubble cursor: Enhancing target acquisition by dynamic resizing of the cursor's activation area. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 281–290. ACM, 2005.

[12] J. Heer, M. Agrawala, and W. Willett. Generalized selection via interactive query relaxation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 959–968. ACM, 2008.

[13] J. Heer and B. Shneiderman. Interactive dynamics for visual analysis. *Communications of the ACM*, 55(4):45–54, 2012.

[14] C. Holz and S. Feiner. Relaxed selection techniques for querying time-series graphs. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, pages 213–222. ACM, 2009.

[15] M. Livny, R. Ramakrishnan, K. Beyer, G. Chen, D. Donjerkovic, S. Lawande, J. Myllymaki, and K. Wenger. Devise: integrated querying and visual exploration of large datasets. *ACM SIGMOD Record*, 26(2):301–312, 1997.

[16] J. Mackinlay. Automating the design of graphical presentations of relational information. *Acm Transactions On Graphics (Tog)*, 5(2):110–141, 1986.

[17] C. North and B. Shneiderman. Snap-together visualization: a user interface for coordinating visualizations via relational schemata. In *Proceedings of the working conference on Advanced visual interfaces*, pages 128–135. ACM, 2000.

[18] C. Olsten, M. Stonebraker, A. Aiken, and J. M. Hellerstein. Viqing: Visual interactive querying. In *Visual Languages, 1998. Proceedings. 1998 IEEE Symposium on*, pages 162–169. IEEE, 1998.

[19] W. A. Pike, J. Stasko, R. Chang, and T. A. O'Connell. The science of interaction. *Information Visualization*, 8(4):263–274, 2009.

[20] ggvis 0.4 overview. `http://ggvis.rstudio.com`, June 2016.

[21] Shiny by RStudio. `http://shiny.rstudio.com`, June 2016.

[22] A. Satyanarayan, R. Russell, J. Hoffswell, and J. Heer. Reactive Vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2016.

[23] 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*, pages 669–678. ACM, 2014.

[24] C. Stolte, D. Tang, and P. Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Trans. Visualization & Comp. Graphics*, 8(1):52–65, 2002.

[25] D. F. Swayne, D. T. Lang, A. Buja, and D. Cook. Ggobi: evolving from xgobi into an extensible framework for interactive data visualization. *Computational Statistics & Data Analysis*, 43(4):423–444, 2003.

[26] Online Vega Editor. `http://vega.github.io/vega-editor/`, June 2016.

[27] H. Wickham. A layered grammar of graphics. *Journal of Computational and Graphical Statistics*, 19(1):3–28, 2010.

[28] A. Wilhelm. User interaction at various levels of data displays. *Computational statistics & data analysis*, 43(4):471–494, 2003.

[29] L. Wilkinson. *The Grammar of Graphics*. Springer, 2 edition, 2005.

[30] K. Wongsuphasawat, D. Moritz, A. Anand, J. Mackinlay, B. Howe, and J. Heer. Voyager: Exploratory Analysis via Faceted Browsing of Visualization Recommendations. *IEEE Trans. Visualization & Comp. Graphics*, 2015.

[31] J. S. Yi, Y. ah Kang, J. T. Stasko, and J. A. Jacko. Toward a deeper understanding of the role of interaction in information visualization. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1224–1231, 2007.