Formal Semantics and Type System for Vega Data Transformations

Kristýna Petrlíková
Charles University
Prague, Czech Republic
auburn@kam.mff.cuni.cz

Tomas Petricek
Charles University
Prague, Czech Republic
tomas@tomasp.net

Abstract

Vega is a popular declarative language for creating interactive data visualizations. It supports reactive data transformations using its streaming dataflow architecture. Despite its widespread adoption, the exact semantics of Vega is subtle and poorly documented. This leads to incorrect or confusing visualizations and difficult-to-understand error messages.

This paper makes two contributions. First, we define a graph-based operational semantics, providing a precise model of the streaming dataflow architecture of Vega. Second, we present a type system for the core data transformation language of Vega, which can prevent a range of common errors. We show that our type system is sound with respect to the semantics.

While the dataflow architecture of Vega closely resembles well-studied models such as functional reactive programming and adaptive computation, there are important differences. The novelty of our work lies in making these precise and providing static analysis for such a reactive data visualization language. The result is a checker for Vega that can catch common real-world errors.

Keywords: Data Visualization, Dataflow, Functional Reactive Programming, Operational Semantics

1 Introduction

Vega [36] is a declarative language for creating data visualizations inspired by the grammar of graphics [42]. Vega makes it possible to define rich interactive visualizations, but the exact working of the system is complex and poorly documented.

First, Vega is built on top of JavaScript and inherits some of its subtle semantics. Moreover, the implementation uses many defaults that might appear unexpected or unintuitive. For example, the default output fields for the `stack` transform are "`y0`" and "`y1`", whereas for the `bin` transform they are "`a`" and "`b`" [39].

Second, Vega uses a streaming architecture [35], which makes understanding dataflow challenging. As a result, debugging Vega visualizations is a known challenge [11].

Debugging Vega Errors. Consider the visualization in Figure 1, which takes a collection of books and aggregates the data to show the average word count per book. The transformations are specified as follows:

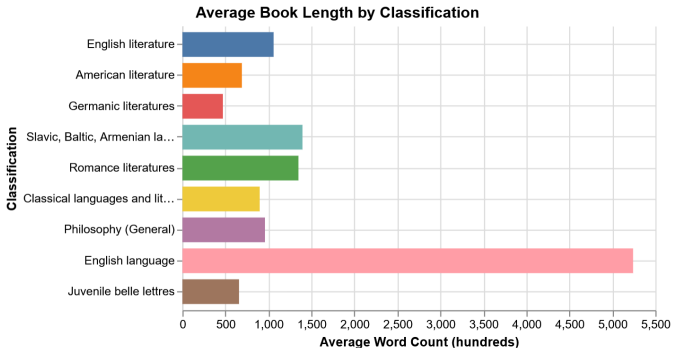


Figure 1. Vega visualization of average book lengths.

```
transform: [
  { type: "aggregate", groupby: ["classification"],
    ops: ["mean", "count"], fields: ["words", ""],
    as: ["avg_words", "book_count"] },
  { type: "filter", "expr": "datum.book_count >= 10" },
  { type: "formula", as: "avg_hundreds",
    expr: "datum.avg_words / 100" }
]
```

The example first uses the aggregate transform to group books by their classification and computes an average word count and a number of books in each group. It then adds the word count in hundreds as avg_hundreds and removes groups with fewer than 10 books.

Let's say that we accidentally type avgwords instead of avg_words in the formula transform:

```
{ type: "formula", as: "avg_hundreds",
  expr: "datum.avgwords / 100" }
```

Vega displays an empty visualization and reports a warning "Infinite extent for field avg_hundreds_start". The reason is that accessing a non-existent field in JavaScript returns undefined and undefined/100 evaluates to NaN. Vega thus tries to draw a chart where all bars have the same NaN length and fails to calculate the scale for the axis.

Now, let's say that we try to change the filter, after the aggregation, to look only at non-English books:

```
{ type: "filter", "expr": "datum.language != 'en'" }
```

Vega displays a visualization that includes all groups, counting all the books in the input dataset. The reason is that

```
{ $schema: "https://vega.github.io/schema/vega/v5.json",
  title: "Average Book Length by Classification",
  width: 400, height: 240,
  data: [ // Define data sources and transformations
   { name: "books", url: "data/books.json" },
   { name: "aggs", source: "books", transform: [ ... ] }
  ],
  scales: [ ... ], // Scales map data values to visual values
  axes: [ ... ], // Axes show scales in the visualization
  marks: [ ... ] // Define visual marks in the visualization
};
```

Figure 2. The structure of a Vega specification.

the earlier aggregate operation produces values that contain only the group key (classification) and newly computed aggregations (avg_words and book_count). The field language is no longer present and undefined != "en" evaluates to true.

The root cause of both issues is access to an invalid field, but they are exhibited in different ways that are easy to overlook. Moreover, the exact source of the error is not easy to trace because of the streaming dataflow architecture of Vega.

Understanding Vega Architecture. Vega makes it possible to create interactive data visualizations, in which user interface events such as moving the mouse cursor can cause items in the dataset to be added, updated or removed. To support this, the Vega runtime [35] builds a directed acyclic dataflow graph that represents dependencies between components. The nodes (called *operators* in Vega) can have an internal state. The state is used, e.g., by aggregation to keep track of values encountered so far. The source nodes (those without incoming edges) correspond to data sources or user interface events.

When a dataset is created or an event occurs, the source node creates a *changeset* containing a set of newly created, updated, and removed values. The changeset then *pulses* (propagates) along the graph in topological order and is transformed in various ways by the individual nodes. For example, the filter transform marks updated values that no longer satisfy the given condition as removed.

The streaming dataflow architecture makes error reporting more challenging. It also makes defining accurate semantics of Vega an interesting theoretical challenge.

Related Work. Vega is well-studied from the user-centric perspective [34, 36], but the streaming dataflow architecture of Vega has only been described in high-level terms [35]. The execution model is akin to event-driven functional reactive programming [8, 41] and adaptive computation [2], which have been studied in more depth.

The semantics of reactive programming has been described both in operational [12] and denotational style [9, 15]. Oeyen [24] presents a graph-based model, which is close to how the Vega runtime operates, but the model works with singular values, rather than updates to a dataset.

Verifying the properties of Vega visualizations has been done through linters [4, 20], but these focus mainly on correcting misleading visualizations. Vega errors can also be detected using debugging and profiling tools [11, 43]. Static checking of specifications is an under-explored area.

Contributions. We present a formal model and a type system for Vega data transformations. Our contributions are:

- We present the μVega calculus, which captures the essence of Vega data transformations (§3), and formally model its evaluation using a graph-based operational semantics based on changeset propagation (§5).
- We define a type system for μVega (§4) that ensures the correctness of Vega specifications, and shows its soundness with respect to our operational semantics (§6).
- We introduce a CLI-based Vega specification checker that can check a subset of real-world Vega specifications for common errors such as invalid field accesses (§7), and which accounts for Vega's unexpected defaults.

2 Vega Overview

A data visualization in Vega is defined by a JSON specification. Figure 2 shows the structure of the specification of the data visualization discussed in §1. Although the focus of this paper is on data transformations, we give a brief overview of the remaining components here. In the second part of this section, we discuss the properties of the runtime.

2.1 Specification

The most important parts of a Vega specification are sections that define the data sources and transformations (data), mapping of data to visual attributes (scale) and the visual elements of the chart (axes, marks).

Data Sources and Transformations. The data block in Figure 2 defines two datasets. The first, books, reads data from a specified JSON file. The second, aggs, is obtained from the source dataset books through a series of data *transformations* discussed in §1. Besides these, Vega offers a plethora of built-in transformations (total of 51 at the time of writing) that reshape and filter data, calculate new fields, or derive new data streams [39]. We discuss the different kinds of transformations in §3. It is worth noting that dataset definitions in Vega are processed sequentially and Vega reports an error if the source dataset has not yet been defined.

Scales and Axes. Vega separates the mapping of values from the data domain (e.g., classification, numbers of words) to the visual domain (pixels on screen) from the specification of visual elements. The former is done in the scales block.

Figure 3 shows two scale definitions for our running example. The yscale is a scale of type band, which defines mapping from a discrete domain (string values) to a continuous numerical range (pixels). The range of the scale sets

```
scales: [
  { name: "yscale", type: "band", range: "height",
    domain: { data: "aggs", field: "classification" } },
  { name: "xscale", type: "linear", range: "width",
    domain: { data: "aggs", field: "avg_hundreds" } }
],
axes: [
  { orient: "left", scale: "yscale", title: "Classification" },
  { orient: "bottom", scale: "xscale", title: "Avg Words" }
],
marks: [ {
  type: "rect", from: { data: "aggs" },
    encode: { enter: {
      x: { scale: "xscale", value: 0},
      x2:{ scale: "xscale", field: "avg_hundreds" },
      y: { scale: "yscale", field: "classification" },
      height: { scale: "yscale", band: true },
    } }
} ]
```

Figure 3. Scales, axes and marks for our running example.

the maximal value of the target domain to the height of the visualization. Finally, domain specifies the source of the data for the scale. This is the classification field of the data in the computed dataset named aggs. The xscale is a linear scale mapping a continuous range of values (word counts) to a numerical range defined by the width of the visualization. This time, the source uses the computed avg_hundreds field.

Axes are visual elements used for displaying information about scales. In Figure 3, we specify that axes should appear on the left and bottom, we define the title for each of the axes and we specify what scales they represent.

Marks. The content of a chart is defined in marks. Marks encode data from some data source as geometric elements such as rectangles. Vega includes many types of marks, ranging from lines, paths and rectangles to icons and text.

Our example, shown in Figure 3, defines one set of marks whose type is rect. The data source, specified using from, is the computed dataset aggs. In Vega, the source for a mark can be any dataset, but also another mark. This feature, referred to as *reactive geometry*[32] makes it possible to specify new set of marks relative to existing previously defined marks, for example to add text labels to the visual display. The from definition can also specify faceting where a single dataset is split across multiple groups of marks.

Each mark has a set of encoding rules that map the backing data to the visual properties of the mark. There is a number of different encoding rules, applied in different stages of a mark's lifetime. The three basic encoding rules cover the cases when a mark is first created (enter), when data is updated (update) and when it is removed (exit). In our example, we only specify the enter rule for creating the marks.

Depending on the type of the mark, the encoding rule needs to define properties that are used for drawing the mark. In case of a horizontal rectangle, we define its range using a pair of X coordinates (x and x2), Y offset (y) and its height (height). The object that specifies the value, called *value reference* in Vega, can be a constant (value), reference to a field (field) as well as a signal or a color palette reference (not discussed here). Optionally, a specified scale can be used to transform domain values to the visual domain. Finally, the band field can be used to refer to the size of the band computed by the scale of type band.

Event Streams and Signals. Interactive data visualizations can be defined using event streams and signals. Event streams model primitive DOM events such as `click` or `mouseover`. More complex event streams may be formed by filtering or merging existing ones.

Signals are dynamic variables that can be used to make a value in the specification time-varying. They are defined by an initial value and a set of event handlers that update the signal's value. A signal with event handlers is reactive and re-evaluates as events occur, whereas one with only an initial value is non-reactive; the two correspond to reactive and non-reactive behaviors in E-FRP [41]. Signals cannot be used for dataset, signal, or scale names, which must be fixed in their definitions. Signals can also be composed to form complex dependency graphs. While there is no ordering requirement for signals (as opposed to dataset definitions), Vega requires the resulting graph to be acyclic.

We focus on modeling the streaming dataflow that propagates dataset updates through data transformations. Event streams and signals would be an interesting addition as they make Vega more dynamic, akin to functional reactive programming frameworks.

2.2 Runtime

After the specification has been parsed and validated using JSON Schema [5, 31], Vega constructs a dataflow graph based on the specification. It uses the graph to efficiently propagate updates in reaction to user input by traversing the graph.

Operators. The nodes of the graph are called *operators*. Each operator has a value, an optional update function, and a set of *parameters* whose changes trigger a value update. Parameters can be primitive values (which never change) or other operators.

For example, the formula transformation in an earlier example has parameters expr and as. If the expression in the expr string refers to a signal name, the graph node (operator) corresponding to the signal becomes a parameter of the expr expression. If the transform is part of a transform definition, then it includes a special parameter named *pulse* that refers to the previous transform in the pipeline (if it exists).

When an operator v depends on an operator u as a parameter, there exists a directed edge from u to v. There can

$$
\begin{array}{rrcl}
\text{Field names} & f & ::= & \texttt{String} \\
\text{Primitive values} & \mathsf{v} & ::= & b \in \texttt{Boolean} \mid i \in \texttt{Number} \mid s \in \texttt{String} \\
\text{Path access} & p & ::= & \mathit{datum} \mid p.f \\
\text{Simple expressions} & a & ::= & \mathsf{v} \mid p \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \mid a_1 \,/\, a_2 \mid \mathit{builtin}(e_1, \ldots, e_n) \\
\text{General expressions} & e & ::= & a \mid [e_1, \ldots, e_n] \mid \{f_1 \colon e_1, \ldots, f_n \colon e_n\} \\
\text{Aggregation operators} & \mathrm{op} & ::= & \mathsf{sum}\ p_{\text{in}}\ \mathsf{as}\ f_{\text{out}} \mid \mathsf{avg}\ p_{\text{in}}\ \mathsf{as}\ f_{\text{out}} \mid \mathsf{count}\ \mathsf{as}\ f_{\text{out}} \\
\text{Transforms} & t & ::= & \mathsf{agg}(p_1, \ldots, p_n, \mathrm{op}_1, \ldots, \mathrm{op}_k) \mid \mathsf{cross}(f_a, f_b) \mid \mathsf{formula}(e, f_{\text{out}}) \mid \mathsf{filter}(e) \\
\text{Dataset identifiers} & n & ::= & \texttt{String} \\
\text{Dataset definitions} & d & ::= & \mathsf{data}(n_{\text{out}}, n_{\text{in}}, t_1, t_2, \ldots, t_k) \\
\text{Specifications} & \mathrm{spec} & ::= & d_1, \ldots, d_k
\end{array}
$$

Figure 4. Syntax of μVega, which captures the essence of Vega.

be multiple edges between two nodes (e.g., when a single signal name is used in both expr and as), and there can even be self-loops (e.g., a self-incrementing signal that reacts to user clicks). The only important restriction is that, excluding self-loops, the graph must be acyclic, so that there exists a topological ordering on the operators.

Pulses. At runtime, Vega propagates changes to individual data tuples through the dataflow graph. When the source data changes, it propagates the corresponding change using a *pulse*. Pulses carry references to *changesets*, which in turn contain references to data tuples that were added, removed, or updated.

Upon receiving a pulse, an operator pulls current values from its parameters and calls its own update function. This updates the operator's value and returns a modified pulse which is then passed to successors. The propagation is orchestrated by a dataflow scheduler which ensures that the changes propagate in topological order (an operator is thus only evaluated once its dependencies have processed their pulses).

Mutation and Tracking Copies. During the processing of pulses, data tuples are announced as being added to, removed from, or modified in the source dataset. In case of modification events, the change is applied directly on the original data tuple, mutating it in-place.

Mutation is only a local measure as the mechanism would easily break if the same source tuples were modified throughout the whole graph. Thus, in a single data pipeline, all data tuples are modified in-place, but on the pipeline's boundaries, copies of original tuples are created, and updates to these copies are redirected using lookup tables. As the tuples can get almost arbitrarily modified throughout the graph, they receive unique numerical identifiers. The primary usage of identifiers is tracking observed tuples in transform operators by using the identifiers as keys in various lookup tables.

3 The Essence of Vega

We capture the essence of Vega data transformations using a small formal model that we call the μVega calculus. We discuss possible extensions to the calculus to model a larger portion of the language (most importantly, signals) in §5.5.

The syntax of the μVega calculus is shown in Figure 4. It models the fact that a specification defines new datasets computed from existing datasets through transformations, includes four basic transformations and models the restricted expression language used in Vega.

Specifications. A *specification* s consists of a sequence of *dataset* definitions that define new named datasets by transforming earlier datasets through a sequence of *transforms* t. We refer to the former as input datasets. We assume that those are defined outside of the formal model.

As an example, the following models a subset of the example discussed in §1. We have an input dataset *books* and we want to count the average number of words per book *classification* and report the average in hundreds:

```
data(books, aggs,
  agg(datum.classification,
    avg datum.words as avg_words, count as book_count),
  formula(round(avg_words / 100), avg_hundreds) )
```

The specification assumes an input dataset *books* and defines a new dataset *aggs*. It performs two transformations. First, it aggregates the data using *classification* as the key, counting the books and computing *avg_words* by averaging the number of *words*. Then, it uses the formula transformation to define a new field *avg_hundreds*, which is computed by dividing the number of words by 100 and rounding the result.

Transformations. Vega uses a streaming-based execution model. When processing a stream of input data, a transformation takes a single data tuple (datum) and outputs a number of updates to be propagated to the subsequent steps of the computation. The shape of the output data tuple depends on the kind of transformation. There are four common kinds:

1. those that *keep* the input shape (e.g. filtering)
2. those that *extend* the input shape with additional, newly defined fields (e.g. the formula transform)
3. those that *nest* the input shape (e.g. cross product)
4. those that *discard* the input shape and produce a different output shape (e.g. aggregation)

μVega includes one example of each kind; filter emits data for which the given expression e evaluates to true; formula evaluates the given expression and adds the result to the datum as a new field; cross produces a cross product of the data stream with itself, storing the two data points in a record as fields f_a and f_b. Finally, agg groups the input data according to a key consisting of the specified fields and computes the specified aggregation operations on each group. The resulting data tuples contain the keys, along with the computed aggregates stored in the output fields f_{out}.

We omit transforms that emit signals and transforms working with GeoJSON and trees. We also omit transforms that combine multiple data sources (e.g., lookup). Those are infrequent, but are theoretically interesting as they can introduce *glitches* (§5.5). Finally, we omit transformations where the resulting shape depends on the values in the input dataset (e.g., pivot) as those cannot be type-checked statically.

Expressions. Vega specifications can use a limited subset of JavaScript to specify computations inside transforms. Most notably, expressions cannot introduce new variables and can only access the current data tuple (datum) and its fields, perform numerical operations and call a limited number of built-in functions.

In μVega, primitives include Booleans, numbers and strings. An access to the data tuple or its (nested) field is modelled using a path accessor p. Simple expressions a can appear as arguments to numerical operators and include values, path accessors, numerical operators and calls to built-in functions. Expressions e can also construct arrays and records.

4 Type System

A well-typed μVega specification does not access undefined fields or datasets not defined previously. Our type system captures the idea formally, focusing on how data transformations change the shape of data tuples. Our system recognizes a number of primitive types, records and collections:

$$\begin{aligned} \text{primitive} &::= \text{boolean} \mid \text{number} \mid \text{string} \\ T &::= \text{primitive} \mid \{f_1 : T_1, \ldots, f_n : T_n\} \mid [T] \end{aligned}$$

We define the type system using a series of judgments for individual syntactic categories of μVega; §5 adds a graph-based operational semantics and §6 discusses soundness.

Expression judgment. The judgment $T \vdash e : T'$, defined in Figure 5, states that, upon receiving a datum of type T, the expression e outputs a new datum of type T'. In the standard notation, this translates to $\{\text{datum} : T\} \vdash e : \{\text{datum} : T'\}$.

Expression typing: $T \vdash e : T'$

$$T \vdash b : \text{boolean} \quad \text{(T-eBool)}$$
$$T \vdash i : \text{number} \quad \text{(T-eNum)}$$
$$T \vdash s : \text{string} \quad \text{(T-eStr)}$$
$$T \vdash \text{datum} : T \quad \text{(T-eDatum)}$$

$$\frac{T \vdash p : \{\ldots, f : T', \ldots\}}{T \vdash p.f : T'} \quad \text{(T-eField)}$$

$$\frac{\oplus \in \{+, -, \times, /\} \quad T \vdash e_1 : \text{number} \quad T \vdash e_2 : \text{number}}{T \vdash e_1 \oplus e_2 : \text{number}} \quad \text{(T-eArithNum)}$$

$$\frac{T' \in \{\text{string}, \text{number}, \text{boolean}\} \quad T \vdash e_1 : \text{string} \quad T \vdash e_2 : T'}{T \vdash e_1 + e_2 : \text{string}} \quad \text{(T-eAddStrL)}$$

$$\frac{T' \in \{\text{number}, \text{boolean}\} \quad T \vdash e_1 : T' \quad T \vdash e_2 : \text{string}}{T \vdash e_1 + e_2 : \text{string}} \quad \text{(T-eAddStrR)}$$

$$\frac{\forall i \in 1..n \;.\; T \vdash e_i : T'}{T \vdash [e_1, \ldots, e_n] : [T']} \quad \text{(T-eArr)}$$

$$\frac{\forall i \in 1..n \;.\; T \vdash e_i : T_i}{T \vdash \{f_1 : e_1, \ldots, f_n : e_n\} : \{f_1 : T_1, \ldots, f_n : T_n\}} \quad \text{(T-eObj)}$$

$$\frac{\Sigma(\mathit{builtin}) = (T_1, \ldots, T_n) \Rightarrow T \quad \forall i \in 1..n \;.\; T \vdash e_i : T_i}{T \vdash \mathit{builtin}(e_1, \ldots, e_n) : T} \quad \text{(T-eBuiltin)}$$

Figure 5. Expression judgment $T \vdash e : T'$.

Vega expressions can access only the datum variable (and signals and builtins, which we omit). The T-eDatum and T-eField rules model access to the datum object and its fields. We match the semantics of JavaScript by allowing the use of + for string concatenation with an implicit conversion for numbers and booleans (T-eAddStrL and T-eAddStrR). Vega expressions can also create new collections (T-eArr) and objects (T-eObj) and call a range of primitive functions (T-eBuiltin). We assume their types are given in Σ.

Transform judgment. The judgment $T \vdash t : T'$ (Figure 6) states that, given a data tuple of shape T, the transform t produces a data tuple of shape T'. The formula transform (T-tFormula) computes a value of type T_{out} and adds it as a new field to the data tuple. We use the ∪ operator to denote the join of two data tuples on their keys, with the right-hand-side keys taking precedence. Cross product (T-tCross) creates a new data tuple that contains two values of the input type. The output type of the filter transform (T-tFilter) is the

Aggregation operator typing: $T \vdash \text{op} : T'$

$$\frac{T \vdash p_{\text{in}} : \text{number}}{T \vdash \text{sum } p_{\text{in}} \text{ as } f_{\text{out}} : \text{number}} \quad \text{(T-opSum)}$$

$$\frac{}{T \vdash \text{count as } f_{\text{out}} : \text{number}} \quad \text{(T-opCount)}$$

Transform typing: $T \vdash t : T'$

$$\text{name}(\text{datum}.f_1.\ldots.f_n) = f_1.\ldots.f_n$$
$$\text{out}(\text{sum } p_{\text{in}} \text{ as } f_{\text{out}}) = f_{\text{out}}$$
$$\text{out}(\text{count as } f_{\text{out}}) = f_{\text{out}}$$

$$\frac{T \vdash e : T'}{T \vdash \text{filter}(e) : T} \quad \text{(T-tFilter)}$$

$$\frac{T \vdash e : T_{\text{out}}}{T \vdash \text{formula}(e, f_{\text{out}}) : T \cup \{f_{\text{out}} : T_{\text{out}}\}} \quad \text{(T-tFormula)}$$

$$\frac{}{T \vdash \text{cross}(f_a, f_b) : \{f_a : T, f_b : T\}} \quad \text{(T-tCross)}$$

$$\frac{\begin{array}{c}\forall i \in 1..n \;.\; T \vdash p_i : T_i \quad \forall j \in 1..k \;.\; T \vdash \text{op}_j : T'_j \\ T_r = \{\text{name}(p_1) : T_1, \ldots, \text{name}(p_n) : T_n, \\ \text{out}(\text{op}_1) : T'_1, \ldots, \text{out}(\text{op}_k) : T'_k\}\end{array}}{T \vdash \text{agg}(p_1, \ldots, p_n, \text{op}_1, \ldots, \text{op}_k) : T_r} \quad \text{(T-tAgg)}$$

Dataset definition typing: $\Gamma \vdash d : T$

$$\frac{\forall i \in 1..k \;.\; T_{i-1} \vdash t_i : T_i \qquad \Gamma(n_{\text{in}}) = T_0}{\Gamma \vdash \text{data}(n_{\text{out}}, n_{\text{in}}, t_1, \ldots, t_k) : T_k} \quad \text{(T-dData)}$$

Specification typing: $\Gamma \vdash \text{spec} : \Gamma'$

$$\frac{\begin{array}{c}\forall i \in 1..n \;.\; \Gamma_{i-1} \vdash d_i : T_i \\ d_i = \text{data}(\_, n_i, \ldots) \qquad \Gamma_i = \Gamma_{i-1}, n_i : T_i\end{array}}{\Gamma_0 \vdash d_1, \ldots, d_n : \Gamma_n} \quad \text{(T-spec)}$$

Figure 6. Judgments that define the μVega type system.

same as the input type, but we keep the JavaScript semantics that allows the conditional to evaluate to a value of any type.

Finally, aggregation (T-tAgg) produces a tuple that combines the fields used as grouping keys with new fields defined by the aggregation operations (T-opSum and T-opCount). The names of the keys are generated using the name helper, by concatenating the names of the nested fields accessed. (In Vega, a field such as "a.b" is accessed using `datum["a.b"]` to avoid an ambiguity with nested field access.) The names of new fields are those specified in the aggregation operator, obtained by the out helper.

Dataset judgment. For dataset and specification judgments, we use a typing context Γ, which maps the dataset names to the type of the data tuples included in the dataset. (The type is the type of the individual elements in the dataset.)

The rule T-dData that defines the dataset judgment is shown in Figure 6. It specifies the type of data tuples for a newly defined output dataset named n_{out} that is computed by a series of transformations from the input dataset named $n_{\in}$. The sequence of types $T_0, \ldots, T_k$ starts from the type of the source and gradually transforms the type using the transform judgment.

Specification judgment. Finally, the judgment $\Gamma \vdash s : \Gamma'$ defined in Figure 6 states that a specification s transforms the initial dataset context Γ_0 to a newly defined dataset Γ_n. This is done by iteratively using the dataset judgment and adding the newly defined dataset to the previous typing context Γ_{i-1}. Note that the definition models the fact that dataset definitions are sequential and that datasets might be referenced only after their declaration.

5 Operational Semantics

Our model of the Vega runtime is mainly based on the high-level description of Reactive Vega [35], but it reflects some aspects of the implementation to show how real Vega transforms can be formalized using our model. The overall idea is based on propagating data tuple changes in a dataflow graph. The graph construction is akin to other reactive and live programming environments [24, 28], but it is then used to propagate changes (adding, removing or updating data tuples).

The graph structure is illustrated in Figure 7. Vertices in the graph correspond to data transformations and the edges represent how data flows between them.

As our ultimate goal is to statically check dataset field access, we restrict ourselves to dataflow graphs with only transformation operators (i.e. 'pulse' parameters). We discuss how our model can be extended to allow limited usage of user input in §5.5.

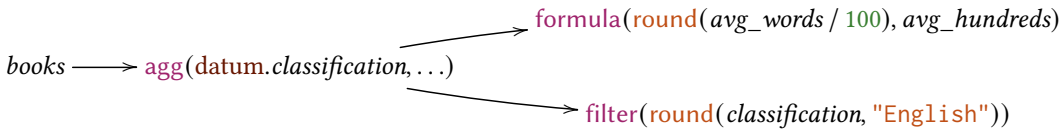
Formally, a μVega dataflow graph (V, E) consists of:

$$\begin{array}{ll} V ::= \{(l_1, t_1) \ldots\} & \textit{transform } t_i \textit{ with a unique label } l_i \\ E \subseteq \{(v_i, v_j), i, j \in V\} & \textit{data from } v_i \textit{ flows to } v_j \end{array}$$

To distinguish vertices that perform the same transformation in different parts of the dataflow graph, we annotate them with a unique label l_i.

5.1 Dataflow graph construction

The process of dataflow graph construction for a given μVega specification is defined in Figure 8. It is structured in terms of three functions for translating a transformation, dataset and a specification, respectively. Each of these functions takes




```
data(books, aggs, agg(datum.classification, avg datum.words as avg_words, count as book_count))
data(aggs, hundreds, formula(round(avg_words / 100), avg_hundreds))
data(aggs, english, filter(round(classification, "English")))
```

Figure 7. A μVega specification that defines three data transformations and a corresponding graph. Vertices are annotated with the transformations they perform and edges model data flow in the graph.

a partially built graph and produces an updated graph with newly added vertices and edges.

Transform builder. The buildT function is called with the graph constructed so far, a vertex v representing the data source and a transform. As illustrated in Figure 7, it constructs a new vertex annotated with the transformation and a fresh label and an edge connecting it to the source. The function then returns the modified graph and the new vertex.

Dataset builder. buildD constructs a new graph in steps $(V_1, E_1), \ldots, (V_k, E_k)$, by adding a vertex for each of the individual transformations using buildT. It operates on a lookup function that maps dataset names to corresponding vertices, using it to find the vertex corresponding to the input dataset n_{in} and adding a mapping for the output vertex n_{out} at the end. Note that $\text{lookup}(n_{\text{in}})$ is not defined if the input dataset does not appear earlier in the Vega specification.

Specification builder. Finally, buildS takes a sequence of dataset definitions. It constructs the graph iteratively using buildD to process individual dataset definitions, constructing a new graph and a lookup function that maps names of all datasets to the corresponding vertex in the graph.

μVega does not model primitive data sources such as files or URLs. We assume that the specification uses a set of pre-existent sources named $n_{\text{in}} \in \mathcal{S}$ and we construct special vertices representing the input nodes annotated with $(n_{\text{in}}, \text{input})$. An initial graph is constructed by the function $\text{build}(\mathcal{S}, \text{spec})$ as:

$$
\begin{aligned}
&\text{build}(\mathcal{S}, \text{spec}) = (V, E), \text{lookup} \quad \textit{where} \\
&\quad \text{lookup}_0 = \{(n_{\text{in}} \mapsto (n_{\text{in}}, \text{input})) \mid n_{\text{in}} \in \mathcal{S}\} \\
&\quad V_0, E_0 = \{(n_{\text{in}}, \text{input}) \mid n_{\text{in}} \in \mathcal{S}\}, \emptyset \\
&\quad (V, E), \text{lookup} = \text{buildS}_{(V_0,E_0),\text{lookup}_0}(\text{spec})
\end{aligned}
$$

5.2 Graph-based evaluation

The key aspect of the Reactive Vega implementation [35] that μVega aims to model accurately is the graph-based streaming propagation of updates. When a data tuple is added, removed, or modified at the data source, the execution engine propagates the update through the dataflow graph and

Transform builder buildT

$$
\begin{aligned}
&\text{buildT} : (\mathbb{V} \times \mathbb{E}, V, t) \to (\mathbb{V} \times \mathbb{E}, V) \\
&\text{buildT}_{(V,E),v}(t) \coloneqq (V \cup \{v'\}, E' \cup \{(v, v')\}), v' \\
&\quad \textit{where } v' = (l, t) \textit{ and } l \textit{ is a fresh label}
\end{aligned}
$$

Dataset builder buildD

$$
\begin{aligned}
&\text{buildD} : (\mathbb{V} \times \mathbb{E}, N \to V, d) \to (\mathbb{V} \times \mathbb{E}, N \to V) \\
&\text{buildD}_{(V_0,E_0),\text{lookup}}(d) \coloneqq (V_k, E_k), \text{lookup}', v_k \ \textit{where} \\
&\quad \text{data}(n_{\text{out}}, n_{\text{in}}, t_1, \ldots, t_k) = d \\
&\quad v_0 = \text{lookup}(n_{\text{in}}), \\
&\quad (V_i, E_i), v_i = \text{buildT}_{(V_{i-1},E_{i-1}),v_{i-1}}(t_i) \quad (\forall i \in 1..k) \\
&\quad \text{lookup}' = \text{lookup} \cup \{(n_{\text{out}}, v_k)\}
\end{aligned}
$$

Specification builder buildS

$$
\begin{aligned}
&\text{buildS} : (\mathbb{V} \times \mathbb{E}, N \to V, s) \to (\mathbb{V} \times \mathbb{E}, N \to V) \\
&\text{buildS}_{(V_0,E_0),\text{lookup}_0}(d_1, \ldots, d_k) \coloneqq (V_k, E_k), \text{lookup}_k \ \textit{where} \\
&\quad (V_i, E_i), \text{lookup}_i = \text{buildD}_{(V_{i-1},E_{i-1}),\text{lookup}_{i-1}}(d_i) \quad (\forall i \in 1..k)
\end{aligned}
$$

Figure 8. Builders that define data-flow graph construction for transforms, datasets and μVega specification.

recomputes values of all datasets that are derived from the source through a series of transformations.

Data tuple identity. In the implementation of Vega [30], data tuples are mutable objects with numerical identifiers stored under a special hidden key (symbol in JavaScript terminology). The implementation uses identifiers to propagate data changes to corresponding copies (e.g., the Relay transform) and to new objects containing the original tuples (e.g., TreeLinks transform). Moreover, identifiers are used to maintain various lookup tables in stateful transformations.

Our model also needs to uniquely identify data tuples. Although the definitions do not directly leverage identifiers in the same way as Vega, they are important in proving the soundness of our semantics. Furthermore, the existence

Data tuples: d

$$
\begin{array}{rrl}
\text{Traces} & R & ::= n \mid (R_1, R_2) \mid f_e(R) \mid \mathrm{agg}_{\mathbf{p},c,\mathbf{sum}}(R) \\
\text{Values} & \mathsf{v} & ::= b \in \mathtt{Boolean} \mid n \in \mathtt{Number} \mid s \in \mathtt{String} \\
 & & \mid \ [v_1, \ldots, v_n] \mid \{f_1 : v_1, \ldots, f_n : v_n\} \\
\text{Data tuples} & \mathsf{d} & ::= \{f_1 : v_1, \ldots, f_n : v_n, \mu : R\}
\end{array}
$$

Data changes: u

$\mathsf{add}(d)$	data tuple d has been created
$\mathsf{update}(d, d')$	replace data tuple d with d'; keep μ
$\mathsf{remove}(d)$	data tuple d has been removed
error	error caused by invalid field access

Figure 9. Definition of runtime data tuples and data updates.

of identifiers makes for semantics that can be extended to model more Vega transformations.

We address this by annotating each data tuple with a symbolic trace R, inspired by traces in provenance tracking [25]. As shown in Figure 9, a trace R can be a primitive number n, or a term that uniquely represents a transformation applied to a data tuple; (R_1, R_2) identifies a data tuple produced by joining two data tuples using cross, while $f_e(R)$ identifies a data tuple produced by applying formula which introduces the field f computed by the expression e. Finally, $\mathrm{agg}_{\mathbf{p},c,\mathbf{sum}}(R)$ denotes a group produced by aggregating tuples using the paths $\mathbf{p}, c, \mathbf{sum}$ as described in §5.3.

Values and changes. Figure 9 defines values, data tuples and data changes. A value v represents primitive values and values computed by Vega expressions. A data tuple is an object (record) with a trace field $\mu : R$. Choosing a special name μ corresponds to the fact that Vega stores identifiers under a symbol that cannot be accessed by a user. As such, traces are not present in our type system, but we assert their properties in § 6 . Note that traces are only needed at the top-level. There are three data changes (add, update, remove), and we also include the error change that is triggered when the evaluation fails due to invalid field access.

Expression evaluation. We assume a standard big-step evaluation relation $\Downarrow$ for Vega expressions. Recall that expressions do not contain variables, but may contain a reference to datum. We write $e[\mathrm{datum}/v]$ for substitution that replaces the datum with a value v. We write $e \Downarrow v$ if an expression e evaluates to v and $e \not\Downarrow$ if the expression cannot be evaluated, e.g., because it attempts to access a non-existent record field.

5.3 Transform evaluation

When a graph node receives a data change, it produces 0, 1 or more data changes that are sent to subsequent nodes. We first define the evaluation of each of our four transformations. There are two kinds of transformations. Stateful transformations need to maintain an internal state, associated with the graph node. Stateless transformations process data changes without keeping a state. The definitions of core transformations are given in Figure 10 and are written as:

$$
\begin{array}{rcll}
[\![t]\!]_{\mathrm{state}}(u) & = & \{u_1, \ldots, u_k\}, \mathrm{state}' & \text{(stateful transform)} \\
[\![t]\!](u) & = & \{u_1, \ldots, u_k\} & \text{(stateless transform)}
\end{array}
$$

For a given transform t, the semantic function receives a data update u (and optionally a transform-specific state) and produces a set of downstream data changes $u_1, \ldots, u_k$, possibly with an updated state′. Furthermore, every transformation propagates an error upon receiving it. Vertices labeled with input act as data sources and never receive any updates.

Formula. The formula transform (Figure 10) extends a data tuple v with a field f obtained by evaluating the expression e. An important aspect of the semantics is maintaining data tuple identity represented by a trace $v.\mu$. The helper operator $+_{f_e}$, which adds the field f also adds a new trace obtained from the original trace by wrapping it with $f_e(v.\mu)$. As a result, if the transformation receives multiple updates to a data tuple with the same identity, the downstream updates will also preserve the data tuple identity.

The formula rule evaluates the formula using $\Downarrow$. In case of an $\mathsf{update}(v, v_{\mathrm{new}})$, we reevaluate the previous value to produce a new downstream update. If the evaluation fails, the semantics produces an error.

Cross. The cross transform (Figure 10) needs to maintain a cross product of all received data tuples with themselves, i.e., a set $\{(d_1, d_2), d_1 \in D, d_2 \in D\}$. It uses the state to keep track of all values currently in the upstream dataset. When receiving an update, the transform outputs the new state along with the set of updates.

To construct downstream updates with stable identity, the semantics defines a helper $\times_{f_a, f_b}$. When it receives $\mathsf{add}(v)$, it computes cross product of v with every other element $v' \in$ state, yielding both $v \times v'$ and $v' \times v$ and also adding $v \times v$. Handling of remove and update works similarly.

Filter. The filter transform (Figure 10) produces a downstream dataset containing only upstream data tuples for which the specified condition holds. We assume a pair of predicates that model the JavaScript semantics of conditionals (truthy holds for non-zero numbers, non-empty strings and all object and array values).

Changes add and remove are sent downstream if the condition holds. The handling of update needs to determine if the data tuple has been newly added (condition did not hold previously, but holds now), updated or removed. Note that the transform always emits a set of updates (possibly empty)

Formula transformation

$$v +_{f,e} v' = v \cup \{f = v', \mu = f_e(v.\mu)\}$$

$$\begin{aligned}
[\![\text{formula}_{e,f}]\!](\text{add}(v)) &= \{\text{add}(v +_{f,e} v_e)\} && (e[\text{datum}/v] \Downarrow v_e)\\
[\![\text{formula}_{e,f}]\!](\text{remove}(v)) &= \{\text{remove}(v +_{f,e} v_e)\} && (e[\text{datum}/v] \Downarrow v_e)\\
[\![\text{formula}_{e,f}]\!](\text{update}(v, v_{\text{new}})) &= \{\text{update}(v +_{f,e} v_e, v_{\text{new}} +_{f,e} v'_e)\} && (e[\text{datum}/v] \Downarrow v_e \wedge e[\text{datum}/v_{\text{new}}] \Downarrow v'_e)\\
[\![\text{formula}_{e,f}]\!](_) &= \{\text{error}\} && (\text{otherwise})
\end{aligned}$$

Cross transformation

$$v_a \times_{f_a,f_b} v_b = \{f_a = v_a, f_b = v_b, \mu = (v_a.\mu, v_b.\mu)\}$$

$$\begin{aligned}
[\![\text{cross}_{f_a,f_b}]\!]_{\text{st}}(\text{add}(v)) &= \{\text{add}(v \times_{f_a,f_b} v'), \forall v' \in \text{st} \cup \{v\}\} \cup \{\text{add}(v' \times_{f_a,f_b} v), \forall v' \in \text{st}\},\ \text{st} \cup \{v\}\\
[\![\text{cross}_{f_a,f_b}]\!]_{\text{st}}(\text{remove}(v)) &= \{\text{remove}(v \times_{f_a,f_b} v'), \forall v' \in \text{st}\} \cup \{\text{remove}(v' \times_{f_a,f_b} v), \forall v' \in \text{st} \setminus \{v\}\},\ \text{st} \setminus \{v\}\\
[\![\text{cross}_{f_a,f_b}]\!]_{\text{st}}(\text{update}(v, v_{\text{new}})) &= \{\text{update}(v \times_{f_a,f_b} v', v_{\text{new}} \times_{f_a,f_b} v'), \forall v' \in \text{st}\} \cup\\
&\quad \{\text{update}(v' \times_{f_a,f_b} v, v' \times_{f_a,f_b} v_{\text{new}}), \forall v' \in \text{st} \setminus \{v\}\}, (\text{st} \setminus \{v\}) \cup \{v_{\text{new}}\}\\
[\![\text{cross}_{f_a,f_b}]\!]_{\text{st}}(\text{error}) &= \{\text{error}\}, \text{st}
\end{aligned}$$

Filter transformation

$$\begin{aligned}
[\![\text{filter}_e]\!](\text{add}(v)) &= \{\text{add}(v)\} && (e[\text{datum}/v] \Downarrow v_e \wedge \text{truthy}(v_e))\\
[\![\text{filter}_e]\!](\text{add}(v)) &= \{\} && (e[\text{datum}/v] \Downarrow)\\
[\![\text{filter}_e]\!](\text{remove}(v)) &= \{\text{remove}(v)\} && (e[\text{datum}/v] \Downarrow v_e \wedge \text{truthy}(v_e))\\
[\![\text{filter}_e]\!](\text{remove}(v)) &= \{\} && (e[\text{datum}/v] \Downarrow)\\
[\![\text{filter}_e]\!](\text{update}(v, v_{\text{new}})) &= \{\text{update}(v, v_{\text{new}})\} && (e[\text{datum}/v] \Downarrow v_e \wedge \text{truthy}(v_e) \wedge e[\text{datum}/v_{\text{new}}] \Downarrow v'_e \wedge \text{truthy}(v'_e))\\
[\![\text{filter}_e]\!](\text{update}(v, v_{\text{new}})) &= \{\text{add}(v_{\text{new}})\} && (e[\text{datum}/v] \Downarrow v_e \wedge \text{falsy}(v_e) \wedge e[\text{datum}/v_{\text{new}}] \Downarrow v'_e \wedge \text{truthy}(v'_e))\\
[\![\text{filter}_e]\!](\text{update}(v, v_{\text{new}})) &= \{\text{remove}(v)\} && (e[\text{datum}/v] \Downarrow v_e \wedge \text{truthy}(v_e) \wedge e[\text{datum}/v_{\text{new}}] \Downarrow v'_e \wedge \text{falsy}(v'_e))\\
[\![\text{filter}_e]\!](\text{update}(v, v_{\text{new}})) &= \{\} && (e[\text{datum}/v] \Downarrow \wedge e[\text{datum}/v_{\text{new}}] \Downarrow)\\
[\![\text{filter}_e]\!](_) &= \{\text{error}\} && (\text{otherwise})
\end{aligned}$$

Aggregate transformation

$$\begin{aligned}
v +_{\text{c,p}} v' &= \{\text{c} = v.\text{c} + 1, \mathbf{p}_{\text{out}} = v.\mathbf{p}_{\text{out}} + v'.\mathbf{p}_{\text{in}}, \mu = v.\mu\}\\
v -_{\text{c,p}} v' &= \{\text{c} = v.\text{c} - 1, \mathbf{p}_{\text{out}} = v.\mathbf{p}_{\text{out}} - v'.\mathbf{p}_{\text{in}}, \mu = v.\mu\}\\
\text{newGroup}_{\text{groupby,count,p}}(v) &= \{\text{c} = 1, \text{f}_{\text{out}} = v.\text{f}_{\text{in}}, \mu = \text{agg}_{\text{groupby,count,f}}(v.\mu)\}\\
\text{groupOf}_{\mathbf{p}}(\text{st}, v) &= \text{st}(v.\mathbf{p})
\end{aligned}$$

$$\begin{aligned}
[\![\text{agg}_{\mathbf{p},\text{c},\mathbf{sum}}]\!]_{\text{st}}(\text{add}(v)) &= \{\text{add}(g)\}, \text{st} + (v.\text{f}, g) && (v.\mathbf{p} \notin \text{st}, g = \text{newGroup}_{\text{f,c},\mathbf{sum}}(v))\\
[\![\text{agg}_{\mathbf{p},\text{c},\mathbf{sum}}]\!]_{\text{st}}(\text{add}(v)) &= \{\text{update}(g, g')\}, \text{st} + (v.\mathbf{p}, g') && (v.\mathbf{p} \in \text{st}, g = \text{st}(v.\mathbf{p}), g' = g +_{\text{c},\mathbf{sum}} v)\\
[\![\text{agg}_{\mathbf{p},\text{c},\mathbf{sum}}]\!]_{\text{st}}(\text{remove}(v)) &= \{\text{update}(g, g')\}, \text{st} + (v.\text{f}, g') && (g = \text{st}(v.\mathbf{p}), g.\text{c} > 1, g' = g -_{\text{c},\mathbf{sum}} v)\\
[\![\text{agg}_{\mathbf{p},\text{c},\mathbf{sum}}]\!]_{\text{st}}(\text{remove}(v)) &= \{\text{remove}(g)\}, \text{st} - (v.\text{f}, g) && (g = \text{st}(v.\mathbf{p}), g.\text{c} = 1)\\
[\![\text{agg}_{\mathbf{p},\text{c},\mathbf{sum}}]\!]_{\text{st}}(\text{update}(v, v_{\text{new}})) &= \{\text{update}(g, g')\}, \text{st} + (v.\text{f}, g') && (v.\mathbf{p} = v_{\text{new}}.\mathbf{p}, g = \text{st}(v.\mathbf{p}), g' = g -_{\text{c},\mathbf{sum}} v +_{\text{c},\mathbf{sum}} v_{\text{new}})\\
[\![\text{agg}_{\mathbf{p},\text{c},\mathbf{sum}}]\!]_{\text{st}}(\text{update}(v, v_{\text{new}})) &= \{\text{remove}(g), \text{add}(g')\},\ \text{st} - (v.\text{f}, g) + (v_{\text{new}}.\text{f}, g') && (v.\mathbf{p} \neq v_{\text{new}}.\mathbf{p}, g = \text{st}(v.\mathbf{p}), g.\text{c} = 1,\ v_{\text{new}}.\mathbf{p} \notin \text{st}, g' = \text{newGroup}_{\text{p,c},\mathbf{sum}}(v_{\text{new}}))\\
[\![\text{agg}_{\mathbf{p},\text{c},\mathbf{sum}}]\!]_{\text{st}}(\text{update}(v, v_{\text{new}})) &= \{\text{remove}(g), \text{update}(g', g'')\},\ \text{st} - (v.\text{f}, g) + (v_{\text{new}}.\text{f}, g') && (v.\mathbf{p} \neq v_{\text{new}}.\mathbf{p}, g = \text{st}(v.\mathbf{p}), g.\text{c} = 1, v_{\text{new}}.\mathbf{p} \in \text{st},\ g' = \text{st}(v_{\text{new}}), g'' = g' +_{\text{c},\mathbf{sum}} v_{\text{new}})\\
[\![\text{agg}_{\mathbf{p},\text{c},\mathbf{sum}}]\!]_{\text{st}}(\text{update}(v, v_{\text{new}})) &= \{\text{update}(g, g''), \text{add}(g')\},\ \text{st} + (v.\text{f}, g'') + (v_{\text{new}}.\text{f}, g') && (v.\mathbf{p} \neq v_{\text{new}}.\mathbf{p}, g = \text{st}(v.\mathbf{p}), g.\text{c} > 1, v_{\text{new}}.\mathbf{p} \notin \text{st},\ g' = \text{newGroup}_{\text{p,c},\mathbf{sum}}(v_{\text{new}}), g'' = g -_{\text{c},\mathbf{sum}} v)\\
[\![\text{agg}_{\mathbf{p},\text{c},\mathbf{sum}}]\!]_{\text{st}}(\text{update}(v, v_{\text{new}})) &= \{\text{update}(g, g''), \text{update}(g', g''')\},\ \text{st} + (v.\text{f}, g'') + (v_{\text{new}}.\text{f}, g''') && (v.\mathbf{p} \neq v_{\text{new}}.\mathbf{p}, g = \text{st}(v.\mathbf{p}), g.\text{c} > 1, v_{\text{new}}.\mathbf{p} \in \text{st},\ g' = \text{st}(v_{\text{new}}), g'' = g -_{\text{c},\mathbf{sum}} v, g''' = g +_{\text{c},\mathbf{sum}} v_{\text{new}})\\
[\![\text{agg}_{\mathbf{p},\text{c},\mathbf{sum}}]\!]_{\text{st}}(_) &= \text{error}, \text{st} && (\text{otherwise})
\end{aligned}$$

Figure 10. Semantics of update propagation of core μVega transformations

if the evaluation succeeds. If the evaluation fails (or the upstream update is error, just like in other transformations), the transformation produces the error change.

Aggregate. The agg transform (Figure 10) groups the input tuples by the values at paths $p_1, .., p_n$. For each of the groups, it computes its sum or count (we omit average, as it can be calculated by combining these two values) and outputs them in fields $\text{out}(\text{op}_1), \ldots, \text{out}(\text{op}_k)$. As it only makes sense to count groups at the root path, we assume there is at most one count operation present in $\text{op}_1, \ldots, \text{op}_k$. To save space, we introduce boldface for vector notation:

- $\mathbf{p}$ denotes $p_1, \ldots, p_n$
- $\mathbf{a} + \mathbf{b}$ denotes $a_1 + b_1, \ldots, a_n + b_n$ (assuming vectors $\mathbf{a}, \mathbf{b}$ of the same length n)
- $v.\mathbf{f}$ denotes either $\{v.\mathbf{f}_1, \ldots v.\mathbf{f}_n\}$ or its expansion $v.\mathbf{f}_1$, , $\ldots, v.\mathbf{f}_n$ if already present in another object
- $v.\mathbf{f} = \mathbf{f}'$ then means $v.f_1 = f_1', v.f_2 = f_2', \ldots, v.f_n = f_n'$
- Finally, assuming $\mathbf{f} = \text{op}_1, \ldots, \text{op}_k$, $\mathbf{f}_{\text{out}}$ denotes $\text{out}(f_1)$, $\text{out}(f_2), \ldots, \text{out}(f_n)$ (and similarly for $\mathbf{f}_{\text{in}}$).

Putting this all together, we define the semantics of agg using these three parameters:

- $\mathbf{p}$ (the paths to group the tuples by),
- count or c (name of the field representing the group size),
- and **sum** (the paths at which to compute a sum)

The state is a mapping from possible values at $\mathbf{p}$ to the currently produced nonempty groups. The group of a tuple v is then accessed via $\text{state}(v.\mathbf{p})$, and its existence is tested by $v.\mathbf{p} \in \text{state}$. Note that both of these actions result in error if some of the paths from $\mathbf{p}$ do not exist in v.

When agg receives $\text{add}(v)$, it creates a new group consisting only of the tuple itself (using the identity v for the new group), or it updates the statistics for an existing group. Similarly in $\text{remove}(v)$, we distinguish whether we should just update the target group or propagate a remove change. The cases in update are similar to those in filter. Either the group of the tuple has not changed, and so we just update the **sum**s accordingly. Otherwise, the group has changed, so we need to remove from one group and add to another. We consider 4 cases, handling the situations where the old group becomes empty or the new group does not exist yet.

This whole transformation has one subtle source of errors. If we somehow remove or update a nonexistent tuple v, none of the cases for remove or update respectively will match, and the evaluation instead falls-through to the error-propagation state. We address this issue in the graph evaluation algorithm, but it is important to realize that this is why tuple identities are useful in our model.

5.4 Graph evaluation

We define change propagation in a manner close to Vega, which means we propagate the changes with respect to a topological ordering $\prec$ on vertices. The only additional information we track in our model are the observed tuples in each node, which helps us prove safety. Let us proceed by defining the state and update logic of the graph.

Let $\text{cs} : V \to \text{DataChange}[]$ represent the list of unprocessed data changes in each vertex, and $\text{ls} : V \to S$ represent the local state for each vertex (if needed), maintaining the state of stateful transformations. Furthermore, let $\text{invals} : V \to R \to d$ represent the data tuples currently residing in the graph nodes. We define a step of reduction $\langle \text{cs}, \text{ls}, \text{invals} \rangle \longrightarrow \langle \text{cs}', \text{ls}', \text{invals}' \rangle$ where we process a single event in our graph and transform the current state into a new one.

1. Select and remove the first data change c from $\text{cs}(v)$ where v is the smallest w.r.t. $\prec$
   - If there are no more data changes, return the original cs and ls.
2. If $c = \text{add}(t)$:
   - If $t.\mu \in \text{invals}(v)$ then $\langle \text{cs}, \text{ls}, \text{invals} \rangle \not\longrightarrow$
   - otherwise: $\text{invals}' \leftarrow \text{invals}+(v, \text{invals}(v)+(t.\mu, t))$
3. If $c = \text{update}(t, t')$:
   - If $t.\mu \notin \text{invals}(v)$ or $\text{invals}(v)(t.\mu) \neq t$:
     - $\langle \text{cs}, \text{ls}, \text{invals} \rangle \not\longrightarrow$
   - otherwise: $\text{invals}' \leftarrow \text{invals}+(v, \text{invals}(v)+(t.\mu, t'))$
4. If $c = \text{remove}(t)$:
   - If $t.\mu \notin \text{invals}(v)$ or $\text{invals}(v)(t.\mu) \neq t$:
     - $\langle \text{cs}, \text{ls}, \text{invals} \rangle \not\longrightarrow$
   - otherwise $\text{invals}' \leftarrow \text{invals}+(v, \text{invals}(v)-(t.\mu, t))$
5. Let $[\![t, \text{ls}(v), c]\!] = C, \text{newState}$
6. $\text{ls}' \leftarrow \text{ls} + (v, \text{newState})$
7. $\text{cs}' \leftarrow \text{cs}$
8. For each $v' \in V$ such that there exists an edge (v, v'):
   - $\text{cs}'(v') \leftarrow \text{cs}'(v') \cup C$

We say $\langle \text{cs}_0, \text{ls}_0, \text{invals}_0 \rangle \longrightarrow^n \langle \text{cs}_n, \text{ls}_n, \text{invals}_n \rangle$ if there exists a sequence $\text{cs}_i, \text{ls}_i, \text{invals}_i$ such that $\forall i \in 0..n$, it holds that $\langle \text{cs}_i, \text{ls}_i, \text{invals}_i \rangle \longrightarrow \langle \text{cs}_{i+1}, \text{ls}_{i+1}, \text{invals}_{i+1} \rangle$.

5.5 Accuracy and Extensions

Two main omissions from our semantics, when compared with the description in §2.2 are multiple data sources and signals. We briefly discuss these here.

Multiple Data Sources. Because our semantics closely follows Vega's dataflow evaluation, adding multiple data sources does not introduce any difficulties. The only interesting situation would be transformations with multiple dependencies, which needs to avoid glitches [6], i.e., a situation where a change to one dependency is propagated first, temporarily resulting in an inconsistent state. In Vega, merging of data from multiple sources is handled by *materializing* the data (applying the changes to the source dataset), and *collecting* the materialized data in a single array. This makes sure that all the data is sent out at once and avoids glitches.

Signals. Signals, along with events, are a key concept of E-FRP, and they are the reason why it is hard to statically check Vega specifications. Indeed, they can be technically used anywhere except for identifiers, which must be determined statically. Despite this, we can still allow them in specifications when they are used to a certain extent.

A signal is defined by a small number of expressions (init/update/bind and event handlers). When the expressions are all strings (plain parameters), this allows for shape inference of signals in the same way as for the formula transform, since we can just combine the expressions together appropriately. Thus, signal definitions are not particularly problematic to analyze.

If we use signals to parameterize field or dataset names, we cannot guarantee correctness as the user can input any value into the signal at runtime. However, using signals in expressions poses no problems as long as we only need their shape to determine the shape of the expression.

6 Properties

Our key result is that the evaluation of well-typed μVega specifications on well-formed input never yields the error update. In the operational semantics, error is produced when an expression evaluation $\Downarrow$ fails (in formula or filter), when a path is not defined (in agg), or when the data change is not well-formed, i.e., a data update references a nonexistent tuple (in agg).

To show that this is the case, we first state the soundness of evaluation and show that well-typed specification has a unique derivation and a corresponding dataflow graph. We then prove the usual progress and preservation lemmas.

Assumption 1 (Evaluation soundness)**.** *We assume* $\Downarrow$ *such that if* $T \vdash e : T'$ *then for all values* v *(as defined in Figure 9), if* $\vdash v : T$ *then* $e[\text{datum}/v] \Downarrow v'$ *and* $\vdash v' : T'$.

Lemma 2 (Uniqueness of typing derivation)**.** *If* $\Gamma \vdash \text{spec} : \Gamma'$ *and* $\Gamma \vdash \text{spec} : \Gamma''$ *then* $\Gamma' = \Gamma''$. *Moreover the typing derivations for the two judgments are the same.*

Proof. Typing rules are deterministic. The only non-syntax-directed rules (T-eAritHNum, T-eAddStrL, T-eAddStrR), have non-overlapping assumptions. □

Lemma 3 (Graph existence)**.** *For any context* $\Gamma = n_{\text{init}} : T$, *if* $\Gamma \vdash \text{spec} : \Gamma'$ *then the dependency graph* $(V, E), \text{lookup} = \text{build}(\{n_{\text{init}}\}, (d_1, \ldots, d_n))$ *exists and is well-defined.*

Proof. Graph construction is only undefined if $\text{lookup}(n_\in)$ was undefined in buildD, but our type system ensures that datasets are defined before they are accessed. □

Note that we only consider case with a single external data source. This is a simplification without a loss of generality, because each of our transformations can only depend on a single data source. Now, the two results mean that, for each transform t in a well-typed specification, there is a unique corresponding typing derivation $T \vdash t : T'$ that assigns input and output types T and T' to the transform. Moreover, there is also a corresponding graph vertex:

$$\frac{\vdash v : T}{\vdash \text{add}(v) : T} \qquad \frac{\vdash v : T}{\vdash \text{remove}(v) : T} \qquad \frac{\vdash v : T \quad \vdash v' : T}{\vdash \text{update}(v, v') : T}$$

Figure 11. Typing of data updates

Lemma 4 (Correspondence)**.** *If* $n_{\text{init}} : T \vdash \text{spec} : \Gamma'$ *then* $(V, E), \text{lookup} = \text{build}(\{n_{\text{init}}\}, \text{spec})$ *and for every vertex* v, $v = (l, t) \in V$, *there is a unique corresponding typing derivation* $T \vdash t : T'$ *for the transform* t.

Proof. Follows from Lemma 2 and Lemma 3. □

The proof of the main soundness theorem will proceed by induction over $\longrightarrow$. The key step is to show that the semantics of individual transformation operations turns well-typed upstream data changes into well-typed downstream data changes. The typing of data changes is defined in Figure 11 and types the data passed in the changes. Note that the error change is not typable. For stateful transformations (cross, agg) we also need to ensure that values stored in the state are well-typed and that the received data changes are *well-formed.*

Definition 5 (Well-formed input)**.** An array of data changes $A = [c_1, \ldots, c_n]$ is said to be well-formed if

$$\langle \text{cs} \mid \text{ls}, \text{invals} \rangle \longrightarrow^n \langle \text{cs}_n \mid \text{ls}_n, \text{invals}_n \rangle$$

for a dataflow graph G with a single input vertex v_{init} with its changeset $\text{cs}(v_{\text{init}}) = A$ (otherwise, $\text{cs}(v) = []$), and $\text{ls}(v) = []$. The graph G along with $\langle \text{cs} \mid \text{ls}, \text{invals} \rangle$ is then referred to as a graph with well-formed input.

Definition 6 (Well-typed state)**.** Let $\text{cod}(f)$ denote the co-domain of a function f. Given a well-typed spec, for each dataflow graph vertex $v = (l, t)$ with its unique corresponding typing derivation $T \vdash t : T'$, we say that a state is well-typed, written $\vdash_t$ state ok:

- If $t =$ cross then $\vdash_t$ state ok if $\forall v \in \text{state} \,.\, \vdash v : T$.
- If $t =$ agg then $\vdash_t$ state ok if $\forall v \in \text{cod}(\text{state}) \,.\, \vdash v : T'$.

Lemma 7 (Typing progress)**.** *Let* $n_{\text{init}} : T \vdash \text{spec} : \Gamma'$, *and* $(V, E), \text{lookup} = \text{build}(n_{\text{init}}, \text{spec})$, *and* $\langle \text{cs} \mid \text{ls}, \text{invals} \rangle$ *be a graph with well-formed input. If* $\langle \text{cs} \mid \text{ls}, \text{invals} \rangle \longrightarrow^* \langle \text{cs}' \mid \text{ls}', \text{invals}' \rangle$, *then either* $\text{cs}' = \emptyset$ *or there exists* cs'', ls'' *and* invals'' *such that* $\langle \text{cs}', \text{ls}', \text{invals}' \rangle \longrightarrow \langle \text{cs}'' \mid \text{ls}'', \text{invals}'' \rangle$.

Proof. By induction on the length k of $\langle \text{cs} \mid \text{ls}, \text{invals} \rangle \longrightarrow^k \langle \text{cs}' \mid \text{ls}', \text{invals}' \rangle$.

- If k is less than the length of well-formed input, the well-formedness guarantees the existence of the next step.

- Otherwise, assume that cs' of the first $k \geq 1$ vertices (w.r.t. topological order) is empty. Both vertices correspond to the transformations in Figure 10. This means that we are currently processing changes coming from the k-th vertex to the $(k+1)$-th vertex. We can check by case analysis that if there was a bad change that would lead to "$\not\longrightarrow$" during evaluation (e.g. $\mathsf{add}(t)$ for an already observed vertex), we could trace the problematic change to the predecessor of the k-th vertex.

□

Lemma 8 (Typing preservation). *Assume a well-typed* spec *with a well-formed input,* $\langle \mathsf{cs} | \mathsf{ls}, \mathsf{invals} \rangle \longrightarrow^* \langle \mathsf{cs}' | \mathsf{ls}', \mathsf{invals}' \rangle$, *and* $\langle \mathsf{cs}' \mid \mathsf{ls}', \mathsf{invals}' \rangle \longrightarrow \langle \mathsf{cs}'' \mid \mathsf{ls}'', \mathsf{invals}'' \rangle$ *at the vertex* $v = (l, t)$ *with* $T \vdash t : T'$, $\vdash_t \mathsf{state}\ \mathsf{ok}$, *and the propagated change* $u \in \mathsf{cs}'(v)$, $\vdash u : T$ *resulting in* $[\![t]\!]_{\mathsf{state}}(u) = X, \mathsf{state}'$. *Then* $\vdash_t \mathsf{state}'\ \mathsf{ok}$ *and* $\forall u' \in X$ *it holds that* $\vdash u' : T'$, *and consequently* $\mathsf{error} \notin X$.

Proof. By case analysis on the transformation t. □

The soundness for μVega follows from progress and preservation in the standard way and states that, for any well-typed specification, we can construct a dataflow graph and, for any updates sent to the graph, the graph-based streaming evaluation will never emit the error update.

7 Vega Specification Checker

We have developed a proof-of-concept tool for checking invalid field access based on our type system. Besides following the type system, the tool also takes Vega's unexpected defaults into account. The tool, along with examples of interesting invalid specifications, is available online at https://github.com/kalivtrope/epsilon-lyrae/.

8 Related work

Vega Design and Implementation. Vega has been primarily described from the user-centric perspective [36] and the work has inspired a range of follow-up libraries that refine or simplify the Vega programming model including Altair [38] and Vega Lite [34]. Reactive Vega [35] describes the underlying streaming architecture, but a formal model of the semantics of Vega has not been previously described. Systems based on functional composition [29] may be more amenable to strong type checking.

Visualization grammars in practice. The use of Vega-like libraries in practice has been subject to research. Pu and Kay [33] point out that analysts sometimes make "hard-to-evaluate silent errors" also known as mirages [21]. Our type system does not check for semantic errors, but may be able to prevent silent errors of a more technical nature. Linters [4, 20] aim to recognize more conceptual design errors, albeit heuristically, while Draco [22] aims to limit the surface area for errors by using high-level constraints.

Profilers and debuggers also have their equivalents for data visualization. VegaProf [43] instruments Vega runtime to collect performance information—and exploring the performance implications of various design choices discussed in our semantics would be an interesting use of such tool. Debuggers have focused on helping the user understand data visualizations visually [11]; similar tools have been implemented using program slicing and tracing techniques [27].

Lu et al. [18] classify bugs in the implementations of data visualization libraries. A number of documented bugs could be prevented with the help of precise operational semantics.

Functional reactive programming. Vega is reactive and its transformations are akin to functional data processing. Research on functional reactive programming thus provides valuable references for defining the semantics of Vega.

Both the original FRP [8] and systems based on arrows [7, 40] are implemented using dataflow graphs, although arrows make this more explicit. The dataflow graph in our Vega semantics is static and so our system avoids problems with memory leaks in the context of FRP [3, 14].

The semantics of FRP system has been described in multiple ways, including a direct denotational style [9], using process categories [13] and ultrametric spaces [15].

Our approach is closer to the graph-based operational semantics defined by Oeyen et al. [24], which is based on two kinds of reduction rules—one kind for rewiring the dependency graph and another for propagating values. In our model, the graph is static and so we separate those more explicitly into graph construction and graph evaluation.

The semantics of arrow-based functional programming has also recently been formalized in Rocq [12] on arrows.

Adaptive and incremental computation. Finally, the Vega runtime can also be linked to self-adjusting, adaptive and incremental computation. In self-adjusting computation [1], the result is recomputed efficiently when an input changes without recomputing the whole program. The implementation is based on graphs and the change-propagation algorithm [2] is akin to our graph evaluation (§ 5.4). The key difference is that Vega only works with datasets, rather than arbitrary data structures and that the graph structure (in our model) remains static.

More generally, a range of incremental and adaptive systems use a form of dependency graph to propagate updates to inputs. These include distributed systems [23], and systems where propagation is on-demand [10, 26]. The need for precise semantics of such systems has been shown by recent work on Rocq formalization [16], which also shows the correctness of a number of optimizations.

9 Conclusion

Traditionally, formal semantics has been used to describe the behavior of elegant programming systems. Recent works

formalizing the behavior of x86 processors [37] or React [17, 19] show the importance of formally studying real-world messy systems. Our work is a contribution to this effort.

We presented a formal model of data transformations in the widely used Vega library. Our calculus accurately models evaluation as propagation of data updates through a dataflow graph. We then described a sound type system that can prevent invalid field accesses in Vega specifications. Although this is a seemingly simple error, it can have surprising consequences for Vega visualizations yielding, for example, empty visualizations and confusing warnings.

Acknowledgments. We thank Jiří Beneš, jury members of the Student Research Competition at ‹Programming› 2025, as well as the anonymous referees of the paper for their valuable feedback. The work has been supported by the Charles University grant PRIMUS/24/SCI/021 and by the Czech Ministry of Education, Youth and Sports grant ERC-CZ LL2325.

References

[1] Umut A Acar. 2005. *Self-adjusting computation.* Ph. D. Dissertation. Carnegie Mellon University.

[2] Umut A. Acar, Guy E. Blelloch, and Robert Harper. 2002. Adaptive functional programming. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon) *(POPL '02)*. Association for Computing Machinery, New York, NY, USA, 247–259. doi:10.1145/503272.503296

[3] Patrick Bahr. 2022. Modal FRP for all: Functional reactive programming without space leaks in Haskell. *Journal of Functional Programming* 32 (2022), e15. doi:10.1017/S0956796822000132

[4] Qing Chen, Fuling Sun, Xinyue Xu, Zui Chen, Jiazhe Wang, and Nan Cao. 2021. VizLinter: A Linter and Fixer Framework for Data Visualization. arXiv:2108.10299 [cs.HC] https://arxiv.org/abs/2108.10299

[5] JSON Schema contributors. 2025. JSON Schema. https://json-schema.org

[6] Gregory H. Cooper and Shriram Krishnamurthi. 2006. Embedding dynamic dataflow in a call-by-value language. In *Proceedings of the 15th European Conference on Programming Languages and Systems* (Vienna, Austria) *(ESOP'06)*. Springer-Verlag, Berlin, Heidelberg, 294–308. doi:10.1007/11693024_20

[7] Antony Courtney, Henrik Nilsson, and John Peterson. 2003. The Yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell* (Uppsala, Sweden) *(Haskell '03)*. Association for Computing Machinery, New York, NY, USA, 7–18. doi:10.1145/871895.871897

[8] Conal Elliott and Paul Hudak. 1997. Functional reactive animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming* (Amsterdam, The Netherlands) *(ICFP '97)*. Association for Computing Machinery, New York, NY, USA, 263–273. doi:10.1145/258948.258973

[9] Conal M. Elliott. 2009. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell* (Edinburgh, Scotland) *(Haskell '09)*. Association for Computing Machinery, New York, NY, USA, 25–36. doi:10.1145/1596638.1596643

[10] Matthew A. Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S. Foster. 2014. Adapton: composable, demand-driven incremental computation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 156–166. doi:10.1145/2594291.2594324

[11] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. 2016. Visual Debugging Techniques for Reactive Data Visualization. *Computer Graphics Forum (Proc. EuroVis)* (2016). doi:10.1111/cgf.12903

[12] Jordan Ischard, Frédéric Dabrowski, Jules Chouquet, and Frédéric Loulergue. 2025. A Mechanized Formalization of an FRP Language with Effects. In *Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing* (Catania International Airport, Catania, Italy) *(SAC '25)*. Association for Computing Machinery, New York, NY, USA, 1431–1439. doi:10.1145/3672608.3707907

[13] Wolfgang Jeltsch. 2014. An abstract categorical semantics for functional reactive programming with processes. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Programming Languages Meets Program Verification* (San Diego, California, USA) *(PLPV '14)*. Association for Computing Machinery, New York, NY, USA, 47–58. doi:10.1145/2541568.2541573

[14] Neelakantan R. Krishnaswami. 2013. Higher-order functional reactive programming without spacetime leaks. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 221–232. doi:10.1145/2500365.2500588

[15] Neelakantan R. Krishnaswami and Nick Benton. 2011. Ultrametric Semantics of Reactive Programs. In *2011 IEEE 26th Annual Symposium on Logic in Computer Science*. 257–266. doi:10.1109/LICS.2011.38

[16] Prashant Kumar, André Pacak, and Sebastian Erdweg. 2025. Incremental Computing by Differential Execution. In *39th European Conference on Object-Oriented Programming (ECOOP 2025) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 333)*, Jonathan Aldrich and Alexandra Silva (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 20:1–20:24. doi:10.4230/LIPIcs.ECOOP.2025.20

[17] Jay Lee, Joongwon Ahn, and Kwangkeun Yi. 2025. React-tRace: A Semantics for Understanding React Hooks: An Operational Semantics and a Visualizer for Clarifying React Hooks. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 289 (Oct. 2025), 28 pages. doi:10.1145/3763067

[18] Weiqi Lu, Yongqiang Tian, Xiaohan Zhong, Haoyang Ma, Zhenyang Xu, Shing-Chi Cheung, and Chengnian Sun. 2025. An Empirical Study of Bugs in Data Visualization Libraries. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE093 (June 2025), 24 pages. doi:10.1145/3729363

[19] Magnus Madsen, Ondřej Lhoták, and Frank Tip. 2020. A Semantics for the Essence of React. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 12:1–12:26. doi:10.4230/LIPIcs.ECOOP.2020.12

[20] Andrew McNutt and Gordon Kindlmann. 2018. 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*. 9.

[21] Andrew McNutt, Gordon Kindlmann, and Michael Correll. 2020. Surfacing Visualization Mirages. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–16. doi:10.1145/3313831.3376420

[22] Dominik Moritz, Chenglong Wang, Gregory Nelson, Halden Lin, Adam Smith, Bill Howe, and Jeffrey Heer. 2019. Formalizing Visualization Design Knowledge as Constraints: Actionable and Extensible Models in Draco. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2019). doi:10.1109/TVCG.2018.2865240

[23] Derek G. Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and Martin Abadi. 2016. Incremental, iterative data processing with timely dataflow. *Commun. ACM* 59, 10 (Sept. 2016), 75–83. doi:10.1145/2983551

[24] Bjarno Oeyen, Joeri De Koster, and Wolfgang De Meuter. 2023. A Graph-Based Formal Semantics of Reactive Programming from First Principles. In *Proceedings of the 24th ACM International Workshop on*

Formal Techniques for Java-like Programs (Berlin, Germany) *(FTfJP '22)*. Association for Computing Machinery, New York, NY, USA, 18–25. doi:10.1145/3611096.3611101

[25] Roly Perera, Umut A. Acar, James Cheney, and Paul Blain Levy. 2012. Functional programs that explain their work. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming* (Copenhagen, Denmark) *(ICFP '12)*. Association for Computing Machinery, New York, NY, USA, 365–376. doi:10.1145/2364527.2364579

[26] Roly Perera, Jeff Foster, and György Koch. 2005. A delta-driven execution model for semantic computing. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 63–71. doi:10.1145/1094855.1094871

[27] Roly Perera, Minh Nguyen, Tomas Petricek, and Meng Wang. 2022. Linked visualisations via Galois dependencies. *Proc. ACM Program. Lang.* 6, POPL, Article 7 (Jan. 2022), 29 pages. doi:10.1145/3498668

[28] Tomas Petricek. 2020. Foundations of a live data exploration environment. *The Art, Science, and Engineering of Programming* 4, 3 (02 2020), 8. doi:10.22152/programming-journal.org/2020/4/8

[29] Tomas Petricek. 2021. Composable Data Visualizations. *Journal of Functional Programming* 31 (2021), e13. doi:10.1017/S0956796821000046

[30] Vega Project and contributors. 2013. Vega at v5.30.0. https://github.com/vega/vega/tree/v5.30.0

[31] Vega Project and contributors. 2025. JSON schema for Vega and Vega-Lite. https://github.com/vega/schema

[32] Vega Project and contributors. 2025. Reactive geometry – Vega. https://vega.github.io/vega/docs/marks/#reactive-geometry

[33] Xiaoying Pu and Matthew Kay. 2023. How Data Analysts Use a Visualization Grammar in Practice. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 840, 22 pages. doi:10.1145/3544548.3580837

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

[35] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. 2016. Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2016). doi:10.1109/TVCG.2015.2467091

[36] Arvind Satyanarayan, Kanit Wongsuphasawat, and Jeffrey Heer. 2014. Declarative interaction design for data visualization. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology* (Honolulu, Hawaii, USA) *(UIST '14)*. Association for Computing Machinery, New York, NY, USA, 669–678. doi:10.1145/2642918.2647360

[37] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (July 2010), 89–97. doi:10.1145/1785414.1785443

[38] Jacob VanderPlas, Brian Granger, Jeffrey Heer, Dominik Moritz, Kanit Wongsuphasawat, Arvind Satyanarayan, Eitan Lees, Ilia Timofeev, Ben Welsh, and Scott Sievert. 2018. Altair: Interactive Statistical Visualizations for Python. *Journal of Open Source Software* 3, 32 (2018), 1057. doi:10.21105/joss.01057

[39] Vega Project. 2026. Vega Transforms Documentation. https://vega.github.io/vega/docs/transforms/. Accessed on 30 January, 2026.

[40] Zhanyong Wan and Paul Hudak. 2000. Functional reactive programming from first principles. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 242–252. doi:10.1145/349299.349331

[41] Zhanyong Wan, Walid Taha, and Paul Hudak. 2002. Event-Driven FRP. In *Practical Aspects of Declarative Languages*, Shriram Krishnamurthi and C. R. Ramakrishnan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 155–172.

[42] Leland Wilkinson. 2012. *The Grammar of Graphics*. Springer Berlin Heidelberg, Berlin, Heidelberg, 375–414. doi:10.1007/978-3-642-21551-3_13

[43] Junran Yang, Alex Bäuerle, Dominik Moritz, and Çağatay Demiralp. 2023. VegaProf: Profiling Vega Visualizations. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (San Francisco, CA, USA) *(UIST '23)*. Association for Computing Machinery, New York, NY, USA, Article 94, 11 pages. doi:10.1145/3586183.3606790