https://doi.org/10.2352/ISSN.2470-1173.2017.1.VDA-386

Declarative Guide Creation

Joseph A. Cottam; CREST/Indiana University; Bloomigton, IN
Andrew Lumsdaine;CREST/Indiana University; Bloomigton, IN

Abstract

Correct guides, such as axes and legends, are an important part of creating an understandable visualization. Guides contextualize the other visuals by providing information about the source data and analysis process. Despite inherent ties to analysis already specified, most visualization programming libraries do reuse the existing specification. Automatic guide creation based on the analysis specification can be performed if the visualization program semantics are well defined and proper metadata is supplied. This paper presents high-level execution semantics for visualization-supporting analysis. These semantics are used with selected metadata to automatically construct guides. The Stencil visualization system includes an implementation of the presented guide system. Stencil is used to explore advantages, limitations and possible extensions to the proposed system. The principles presented can be applied to other visualization frameworks that include programmable analysis. Implementation of automatic guide creation simplifies the construction of visualizations, and can ultimately lead to higher quality visualizations.

Introduction

Reference marks provide the context required to interpret a visualization. Axes, labels and similar guides have distinct semantics from other marks in a visualization. Unfortunately, many visualization frameworks do not use these distinct semantics to assist in guide creation. In most circumstances, visualization frameworks allow custom analysis to be specified for a visualization. Unfortunately, tying that analysis to guides (even though they are inherently related) is not supported. Instead, proper coordination of guides to analysis is left to good programming practices. Escaping this system of manual guide construction requires useful semantics for analysis processes and the ability to manipulate analysis definitions. This paper describes such a system and an implementation of declarative reference mark construction.

Simple construction of reference marks is achieved by visualization applications (like Tableau [1] or Excel [25]) and widget based libraries (like the InfoVis Toolkit [15]) by limiting the types of producible visualizations. For each visualization type, appropriate reference mark generation is provided. This technique cannot be applied in general purpose visualization libraries because the produceable visualizations are impractically large. Some state-based frameworks enable automatically producing guides by synthesizing guides from the final data state before visualization is performed (e.g., Prefuse [18] and ggplot2 [29] to various degrees), but this is a limited case. Going beyond the final data state (e.g., to represent intermediate results) requires programmer discipline to ensure the custom created reference marks match the rest of the visualization.

A visualization system with an appropriate language and metadata has enough information to build guides. This paper describes sufficient language semantics and metadata to achieve guide creation in the general case. Our approach provides the ability to reason over multi-stage, stateful analysis using user-defined transformations to produce a variety of reference marks. Reference marks produced using the described techniques hold many desirable properties, including the potential to guarantee correctness. The system we describe has been implemented in the Stencil visualization system [12] to support declarative guide creation.

Declarative Guides

In general, reference marks can be divided into three classes: Annotations, Direct Guides and Summarization Guides. Annotations include subjective notes and comments; they relate a visualization to external entities through a process orthogonal to the visualization. By their nature, annotations cannot be automatically generated and are not dealt with further. In contrast, guides represent information inherent to the data and/or process of a visualization. *Direct* guides represent the mapping between input data (or its direct derivatives) and a visual effect. Axes and legends are prototypical direct guides. *Summarization* guides focus on analysis results (but may include input data supplementally). Trend lines and point labels are common summarization guides (point labels may use input data to produce the label text, but positioning is in line with summarization style). Guides of both types are (illustrated in Figure 1a) are treated in this paper.

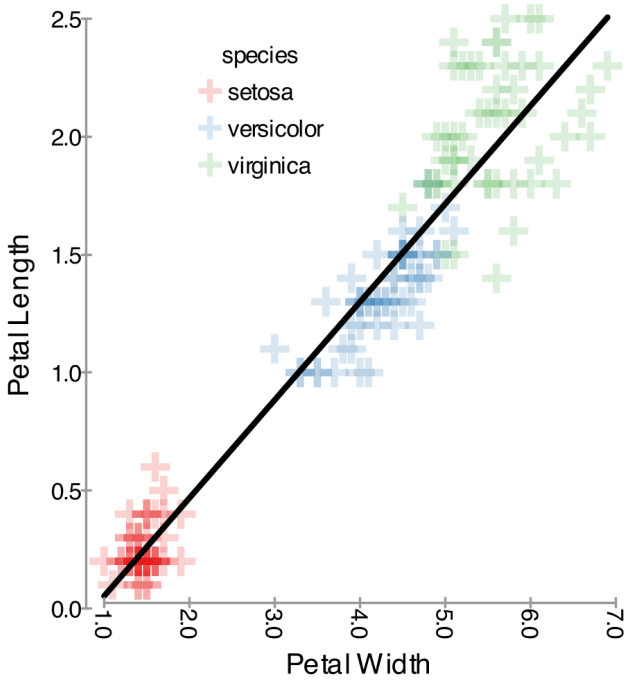
Declarative programming specifies *what* is to be accomplished, instead of *how* to accomplish it [20]. Declarative guide creation is achieved by having a guide request mechanism and a system that can derive relevant analysis and infer other features. Supporting derivation and inferential reasoning requires the ability to represent and reason over the data analysis program. For example, regardless of the analysis or data involved in mapping input data to the X-axis in Figure 2, the declaration on line 5 provides a suitable guide. The underlying system inspects and modifies the program to determine how to construct guides. This paper discusses the framework support required and the analysis and modifications performed to enable declarative guide creation.

Guide Systems

We consider the following characteristics and capabilities when discussing guide systems:

Complete: Reflects the complete analysis process. The guides represent the analysis used to create the visualization, independent of the data used. This includes taking transforms into account and communicating discontinuities in the analysis, if there is a plausible impact on analysis.

Consistent: Reflects data. The guides represent data in the visualization at the time of rendering. This does not limit the guides to just the data presented (e.g. axes may extend

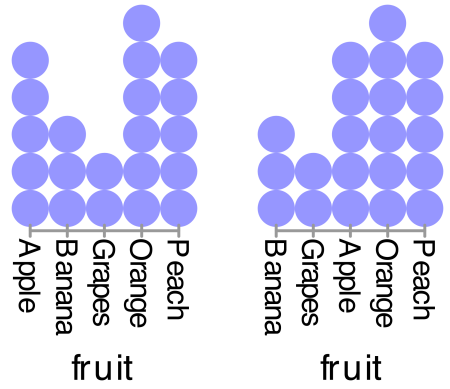


(a) Anderson's Flowers Cell

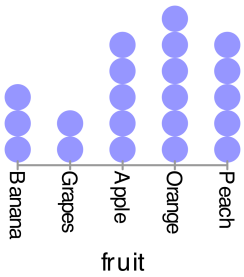
```
1  import BrewerPalettes
2  stream flowers(sepalL, petalW, sepalW, petalL, species, obs)
3
4  layer FlowerPlot
5  guide
6    trend from ID
7    legend[X:20,Y:-90] from FILL_COLOR
8    axis[sample:"Linear",guideLabel:"Petal Length"]
9      from Y
10   axis[sample:"Linear",guideLabel:"Petal Width"]
11     from X
12
13 from flowers
14    ID: obs
15    X:* Scale[0,100](petalL)
16    Y:* Scale[0,100](petalW) -> Mult(_,-1)
17    FILL_COLOR: BrewerColors(species) -> SetAlpha(50,_)
18    REGISTRATION: "CENTER"
19    SHAPE: "CROSS"
```

(b) Anderson's Flowers Program

Figure 1: A single cell part of the Anderson's flowers trellis visualization [6], augmented with a trend line. Guide declarations appear on lines 6-11. The legend in this program demonstrates some of the design support by adopting the characteristic shape (contrast with the legend shape in Figure 7).



(a) Rank-based Layout (b) Index-based Layout



(c) Wider Index-based Layout

```
1
2
3
4
5
6
7
8
9
10
11
12
13
```

```
stream survey(fruit)

layer plot
guide
  axis from X
from survey
   ID: Count()
   X:* Rank(fruit) -> Mult(5, rank)
/* X:* Index(fruit) -> Mult(5, index)*/
/* X:* Index(fruit) -> Mult(10, index)*/
   Y: Count(fruit) -> Mult(-5, count)
   REGISTRATION: "CENTER"
   FILL_COLOR: @Color{150,150,255}
```

(d) Stencil Program

Figure 2: A series of plots based on a simple survey of fruit preferences. The input data is presented as a list of fruit names. The visualization program is found in Figure 2d, with Figures 2a, 2b and 2c using lines 8, 9 and 10 respectively. Each time the analysis used to construct the X-axis changes, but the guide declaration in line 5 does not need to be updated.

beyond the actual data range to make a "nice" range), but tighter bounds are generally preferable.

Subordinate: Influenced by analysis, but not vice-versa. The analysis is the principle concern, guides support the interpretation of that analysis. Therefore, using the guide system should not modify the results of analysis.

Efficient: Places little pressure on runtime resources. A guide system that requires fewer resources to produce the same result is preferable to one requiring more.

Customizable: Permits data reformatting for presentation. Many aspects of a guide often need re-formatting. For example, a guide for the x-axis may need line weights adjusted to be less obtrusive or labels abbreviated to fit a space. Guide customization enhances the flexibility of an automatic system.

Simple: Requires little additional work to create. The less work

required to create a guide, the more likely it will be created. Similarly, if guides are easy to maintain and refine they will more likely be of higher quality. Declarative specification is one route to simplicity by supporting inference about the guides from the declaration context. Ideally, only a specification of the differences between analysis processes and the guide creation process should be required.

Redundancy-aware: Combines redundant encodings. Redundant encodings occur when two visual attributes are based on a single input attribute. In such circumstances a single guide presents both encodings.

Attribute-crossing: Represents multiple inputs in their combinations. It is common for two visual attributes to be varied based on different, but related, inputs. A guide that incorporates the cross-product of potential elements can aid in interpretation. In an attribute-crossing legend, the legend itself is a grid of regularly space examples.

Separation-supporting: Can use separate guides for distinct encodings of value subsets. Even though redundancy-aware and attribute-crossing guides are beneficial, separate guides should still be possible.

Design-sensitive: Reflects design decisions not related to data encoding. The guides produced should appear to "belong" with the data they apply to. Characteristic colors, shapes and sizes contribute to this cohesion. A guide system should be capable of appropriately employing the visual attribute constants used in a visualization program.

Though the above attributes are desirable, they are not all essential. Completeness and consistency are the basis of any useful guides, distinguish guides from an annotation. The other attributes provide means of reasoning about the guide system and effectively representing analysis details (e.g. per Wilkinson [30]).

These attributes can be placed into two major groups: analysis-focused and presentation-focused. Analysis focused attributes describe the process that goes into creating a guide, indirectly touching on how it is actually displayed. Analysis focused attributes are Complete, Consistent, Subordinate, Efficient, and Customizable. These attributes will be addressed in abstract terms in *Section: Analysis Semantics and Metadata*. Simplicity, Redundancy awareness, Attribute-crossing, Separation-supporting and Design-sensitivity are presentational-focused attributes. Independent of how guide contents are determined, these attributes affect how it can be displayed. They are discussed in *Section: Display*.

Related Work

Visualization creation has been examined on a number of occasions. Bertin discusses the design space of visual construction [4], while Tufte focuses on guidelines for effectively employing that design space [27, 28]. These works are concerned with the data and analysis types in relation to the properties of visual representation. Understanding these concerns is important in deciding how to represent data, including how reference marks should be used to contextualize that representation.

One difficulty in treating visualization programs as objects of analysis themselves is a lack of useful semantics in many frameworks. Many frameworks rely on the complex underlying semantics of the implementing language. Chi [11] provides general semantics, but insufficient details (for example, in operation ordering or memory handling) to afford the types of analysis required for guide creation. However, Chi's work demonstrates the value of even partial semantics.

Stencil Overview

Stencil is a coordination language for visualizations. This overview will describe the parts of Stencil used in this paper; line numbers are in reference to Figure 1 unless otherwise noted.

All Stencil operations are in response to incoming data. Data are represented as streams of tuples. A stream schema is required (line 2). Layers are the principle organization unit for both visuals and analysis (line 4). Layer declarations optionally include a type and always include at least one *from* block (line 13) that tie analysis to a declared stream. From blocks can include filters (see Figure 6b, line 12) and rules that bind layer attributes (left of the colon) to the analysis (right of the colon) (line 14). The dynamic binding (indicated with a colon-star line 15), indicates that periodically repeat the analysis and binding to reflect changes in stateful transformations. Analysis chains are composed of transformation operators. The post-fix composition operator $->$ allows multi-stage analysis (line 17). Most operators are loaded from provided modules. New operators can be defined with the *operator* keyword. Arguments are typically lists of values, though a prefixing @ sign allows for special syntax (see Figure 6b, line 18). The underscore is a shorthand for "the most recent value." In line 17, it refers to the color that the BrewerColros operator just produced.

Guides are declared as the layer they belong to (lines 6 to 11). Guide declaration details are discussed in Section Stencil.

Prefuse is a prototypical visualization library [18]. Guides are largely constructed by specifying a new analysis pathway that mirrors (and often partially repeats) the analysis used in visualization construction. Any type of guide can be created in this way, since reference marks are one type of analysis based visuals. However, guide correctness depends on programmer discipline. No support is given to ensure that the guide-determining analysis corresponds to the rest of the visualization. Prefuse does supply limited support for automatically deriving guides when two conditions are met: (1) the guide is an axis; (2) the last step in the analysis conforms to an interface that allows access to a descriptor object. When these two conditions are met, an axis can be automatically created based on a generated descriptor (called a ValuedRangeModel) that reflects the last step of the analysis. The process of creating such abstract descriptors is the basis of the work presented in this paper.

Protovis [5, 17] presents an alternative approach to creating visual guides. Guide support methods are built in to many Protovis-provided objects. For example, pv.scale has a 'ticks' method, glyph types have 'anchors', and the data iterator is pervasively availability. Anchors and the data iterator are discussed further in *Section: Comparison* as they can be used to approximate some of the functionality of the system in this paper.

In language-based approaches to visualization and analysis, The Grammar of Graphics [30] (GoG) provides the most complete discussion of reference marks. As here, GoG distinguishes direct and summarization guides (though using different vocabulary). The specification style for guides in GoG is declarative, mirroring the one called for in this paper. However, the *deriva-*

$\mapsto$ Operation counterpart or process correspondence
$\mathbb{G}$/$\mathbb{A}$ Process/operation chain (blackboard bold)
An An analysis operation
On An opaque analysis operation
Gn A guide operation
S/*V*/*M* Source/Visuals/Memory value sets
s/*v* Source/visual value

Figure 3: Description of notational elements.

tion of guides is not well described. It is unclear how to apply the concepts outside of the frameworks that directly implement GoG. One implementation of the concepts from GoG is ggplot2 [29]. ggplot2 provides concise production of many guide types. However, ggplot2 is limited by the data model of R, so it inherently handles only the last stage of analysis. Correctly tying guide labels to earlier stages of analysis or applying custom formatting requires programmer discipline.

Some considerations relevant to guide creation have been explored in detail. For example, ColorBrewer explores how to select and present color scales [10]. Proper selection of tick-marks on axes and selection/placement of point-labels is a perennial topic (examples include Talbot [26] and Luboschik [21]). Such work provides details on parts of guide creation, but does not integrate it within a larger guide framework.

This work builds on early work presented by Cottam and Lumsdaine [13]. The earlier work was limited to categorical axes and legends and required operator metadata that was sometimes situationally dependent. The system presented here expands support to include continuosly-valued data, allows chaining multiple categorical operations and supports summarization guides. The required operator metadata has also been reduced and simplified.

Analysis Semantics and Metadata

The general process for automatic guide creation is to derive a subset of the analysis process that (1) reproduces the analysis being preformed (2) in a manner that does not interfere with that analysis and (3) executes the derived process over relevant data. This section provides the groundwork to more formally define what it means for an analysis process to be reproduced, how interference is avoided and what the relevant data are. From a high level, the analysis process is augmented with monitoring functions that can supply samples of the input data. These monitors are placed after the first operation that cannot be safely reproduced (referred to as 'opaque' operations) in an analysis chain. This process is represented abstractly in Figure 4. This section includes discussion of execution semantics and operator meta data that are sufficient to implement automatic guide creation. The formal description pursued in this section is to provide definitions that can be used independent of implementation. To simplify discussion, direct guides are used as the principle example, though the definitions given cover summarization guides as well. The notation for this section is summarized in Figure 3.

We treat a visualization program as a linear composition of data transformation operators. This representation corresponds to a single path through a data-flow network [2]. Memory is treated as an input parameter, so all operators are functions, making the composite an applicative transformation chain [19, 3]. As an applicative framework, control flow can be treated separately from the computations involved, similar to the treatment done for coordination languages [16]. In effect, the details of the transformation performed can be treated separately from the coordination of the transformation. The linear composition representation does makes transformations involving branches, merges and loops non-obvious in implementation. We assume that all such non-linear flow is either encapsulated in a single operator (branch and loop) or handled by proper treatment of the memory arguments (merge).

For convenience, all operands are represented as immutable tuples and all operators are assumed to take a list of such tuples as a batch of requests. Therefore, all operations have two arguments $(tuple*, M)$ and return two values $(tuple*, M')$. In practice, these batch request semantics can be achieved by wrapping a data-flow operator accepting $(tuple, M)$ in a for-each loop. The input and output lists are assumed to be the same length and each tuple in the results list corresponds to the input element at the same index. These call semantics allow data-state or data-flow style operations to be expressed [11] by modifying how memory states are treated between calls and the contents of the input list [14]. An analysis chain is given in Equation (1); in general, user-supplied analysis chains start with a read from some source.

$$\mathbb{A} : A5 \circ A4 \circ A3 \circ A2 \circ A1 \circ Read \tag{1}$$

These semantics enable the the guide properties from *Section: Declarative Guides* while remaining applicable to many programming styles.

Transformation operators have two pieces of metadata. First, all operators are expected to be able to provide a *counterpart* operator. Definition 1 formalizes a counterpart's behavior. Informally, if **A** creates a memory state and a result, then counterpart **G** computes the same tuple (1.3) but does not modify the memory state (1.2). Additionally **G** should be a function (1.1). When met, these conditions provide three important properties. First, by conditions 1 and 2, **G** may be executed as often as needed without changing the resulting guide, providing flexibility in scheduling its execution. Second, by condition 2, **G** does not interfere with **A** [22], making the resulting system Subordinate (as defined in *Section: Declarative Guides*). Finally, by condition 3, using **G** is the same as using **A** as it currently stands, which helps establish the Correctness and Consistency properties. A function CP is defined such that $\mathrm{CP}(\mathbf{A}) = \mathbf{G}$ such that **G** corresponds to **A**.

Definition 1 *$\boldsymbol{G} \mapsto \boldsymbol{A}$ if and only if, for all memory states M of **A** for all x in the range of **A** and when $\boldsymbol{G}(x, M) = (y1, M1)$ and $\boldsymbol{G}(x, M) = (y2, M2)$ then*

$$y1 = y2 \tag{1.1}$$
$$M = M1 = M2 \tag{1.2}$$
$$\boldsymbol{G}(x, M) = \boldsymbol{A}(x, M) \tag{1.3}$$

The second piece of metadata that all operators must provide is a categorization with respect to memory usage: Function, Reader, Reader/Write, and Opaque. The first three categories follow the standard definitions. Opaque operators are those operators that use memory but that cannot provide a counterpart operator. Operators that depend on external operations (network, keyboard, mouse, etc) or randomness are typically opaque. Such operations

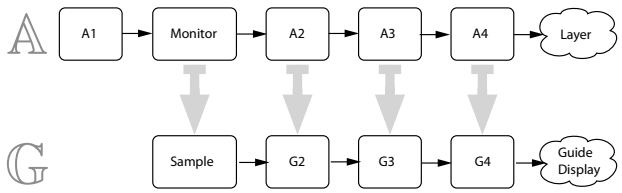


Figure 4: The guide system constructs $\mathbb{G}$ given a precursor to $\mathbb{A}$. Construction is accomplished by inserting a **Monitor** operation and transforming all operations from **Monitor** forward into counterpart operations. When executed, $\mathbb{G}$ passes its results to the guide display system.

limit the repeatable parts of an analysis, and thus determine how much of the analysis can be presented in a guide. From the standpoint of guides, the other memory usage styles only determine how difficult **G** is to construct given **A**.

Generation

With these execution semantics and metadata, it is possible to create guides that properly present visualization transformations. In general, the guide process, denoted $\mathbb{G}$, must produce a description of the input and result space of some analysis, denoted $\mathbb{A}$. This *guide descriptor* is interpreted by a display system to produce an axis, legend, etc. This leads to two general parts: a transformation and a display system. The transformations are slightly different for direct and summarization style guides. Because direct guides are more directly tied to analysis, they are treated first in in *Section: Direct Guides*. Changes required to create summarization guides are given in *Section: Summarization Guides*. The shared display system is described in *Section: Display*.

Direct Guides

A correct guide descriptor for a direct guide, like Prefuse's ValuedRangeModel [18], depends on input data and analysis. This descriptor generally includes a description of inputs to and outputs from some $\mathbb{G}$. Properly constructed, the outputs of the guide system should meaningfully contain the outputs of the original analysis (though they may extend beyond it for practical reasons). Furthermore, the inputs to the guide system should be a superset of the output of some intermediate analysis step. The purpose of the transformation presented here is to create a $\mathbb{G}$ that produces such a descriptor for a corresponding $\mathbb{A}$.

Definition 2 $\mathbb{G} \mapsto \mathbb{A}$ *if and only if*

$$\forall \boldsymbol{Gn} \in \mathbb{G} : \exists \boldsymbol{Ax} \in \mathbb{A} \text{ such that } \boldsymbol{Gn} \mapsto \boldsymbol{Ax} \tag{2.1}$$

$$\forall \boldsymbol{Gn}, \boldsymbol{Gm} \in \mathbb{G} \text{ and } \forall \boldsymbol{Ax}, \boldsymbol{Ay} \in \mathbb{A}$$
$$\text{if } \boldsymbol{Gn} \mapsto \boldsymbol{Ax} \text{ and } \boldsymbol{Gm} \mapsto \boldsymbol{Ay} \text{ then}$$
$$\boldsymbol{Gn} \circ \boldsymbol{Gm} \Rightarrow \boldsymbol{Ax} \circ \boldsymbol{Ay} \tag{2.2}$$

$$\exists \boldsymbol{Gn} \in \mathbb{G} \text{ where } \boldsymbol{Gn} \mapsto \boldsymbol{Ax}$$
$$\text{such that } \nexists \boldsymbol{Ax+1} \in \mathbb{A} \text{ where } \boldsymbol{Ax} \circ \boldsymbol{Ax+1} \tag{2.3}$$

$$|\mathbb{G}| > 0 \tag{2.4}$$

$$\boldsymbol{Gn} \mapsto \boldsymbol{Ax} \text{ and } \boldsymbol{Gn} \mapsto \boldsymbol{Ay} \Rightarrow \boldsymbol{Ax} = \boldsymbol{Ay} \tag{2.5}$$

Correspondence is expressed visually in Figure 4 and defined in Definition 2. Condition (2.1) indicates that $\mathbb{G}$ may only contain counterparts to operations in $\mathbb{A}$. Condition (2.2) ensures that operations in $\mathbb{G}$ appear in the same order and with the same dependencies as their counterparts in $\mathbb{A}$. Condition (2.3) indicates that $\mathbb{G}$ must include the end of $\mathbb{A}$. Conditions (2.2) and (2.3) combine to indicate that the guide need not reflect the full analysis process, but it must correspond to a tail of $\mathbb{A}$. The final two conditions stipulate that the guide system be non-empty and that analysis operators are uniquely represented.

Acquiring $\mathbf{G} \mapsto \mathbf{A}$ is the purpose of the counterpart metadata relationship given earlier. A constructive solution to building a **G** given an **A** is presented in *Section: Stencil*. Selecting a maximal tail of $\mathbb{A}$ for $\mathbb{G}$ is the purpose of the Opaque designation in the memory-related metadata. In short, no opaque operation can be included in any guide creation process.

Before transformation can be performed, two support operations need to be described: **Monitor** and **Sample**. **Monitor** implements the identity relation but also tracks information about what has been observed in its state ($\textbf{Monitor}(x, M_1) = (x, M_2)$ with M_1 not always equal to M_2). The memory state of **Monitor** is used by **Sample** to produce a set of sample of tuples. (Parameters to **Sample** and values it observes determine the type and contents of the sample). The **Monitor/Sample** pair is distinguished in that, even though they compute different functions, CP(**Monitor**) = **Sample** if the **Monitor** is the first element in a chain (otherwise CP(**Monitor**) = **Identity**). This special case simplifies transformation and supports multiple guides from the same analysis chain.

With the definition of process correspondence and the support operations, the transformation process is given in Equations (2)-(5) (assuming the initial equation from Equation (1)).

$$\mathbb{A} : A5 \circ A4 \circ A3 \circ O2 \circ A1 \circ Read \tag{2}$$

$$\mathbb{A} : A5 \circ A4 \circ A3 \circ Monitor \circ O2 \circ A1 \circ Read \tag{3}$$

$$\mathbb{A}' : A5 \circ A4 \circ A3 \circ Monitor \tag{4}$$

$$\mathbb{G} : G5 \circ G4 \circ G3 \circ Sample \tag{5}$$

The first step is to identify the opaque operations, yielding Equation (2). Next, a **Monitor** operator is inserted per Equation (3). By default, **Monitor** will be placed as early in the analysis chain as possible. However, **Monitor** may be validly placed anywhere after the last opaque operator, effecting a change in source space but otherwise leaving the guide process unchanged. After **Monitor** is placed, the guide system is only concerned with the analysis from **Monitor** forward. Therefore, $\mathbb{A}$ is trimmed per Equation (4). The operation chain is reduced is accordance with the *tail* and *contiguous* conditions Definition 2.

After trimming, each analysis operation is replaced with its counterpart guide operation using the CP relation. The resulting $\mathbb{G}$ is shown in Equation (5). Given the definitions of CP and restrictions on the placement of the **Monitor** operator, these transformations produce a guide process conforming to Definition 2.

With $\mathbb{G}$ constructed, it can be used to create a guide descriptor. By construction, the first operator in $\mathbb{G}$ is always **Sample**. Operator **Sample** is defined such that it which produces a list of values *S* in the source space. This source list is used for two purposes: as the source-space information presented in the guide and as the input to any transformations. Applied using the semantics described earlier for analysis, $\mathbb{G}$ will result in a list of results *V* in the same visual space as $\mathbb{A}$ (per the definition of process correspondence). The sets *S* and *V* can be matched pair-wise because of the

index correspondence rule given for invocation semantics. This yields a basic guide descriptor of the form $((s_1, v_1), (s_2, v_2)\ldots)$. This descriptor is sufficient to produce many guide types; however, it can be refined to provide support for operator discontinuities and multiple inputs mapping to the same output.

Operator discontinuities (such as divide by zero) can modify the interpretation of results and indicate potential problems in a visualization. It is possible that a discontinuity was avoided in original analysis, but discovered in guide creation. However, it is undesirable for the guide process to generate errors when the corresponding analysis did not. Working with sets of inputs and producing sets of results gives each **G** the opportunity to identify discontinuities. To handle these discontinuities the expected arguments to **G** are modified. Instead of taking a list of arguments, **G** is changed to a pair where the first element is a list of discontinuity warnings and the second element is the list of inputs as before. Operators that identify potential discontinuities can append relevant information to the warnings lists. Therefore, **Sample** must produce the pair $([], S)$. Other **G** operators produce a similar output. If a sample input actually strikes a discontinuity, the result is replaced with a sentinel *NoValue* to preserve the list semantics. All **G** operators must therefore recognize *NoValue* and simply echo it in that list position.

Ambiguous mappings occur when multiple source values map to the same visual value. Handling ambiguity in the sample/result mapping requires labeling the result more than once. Ambiguity identification does not require changes to the call semantics of operators. However, because it is fundamentally an analytical question it is best handled before the display system. The S and V pair process is therefore extended to include identification of duplicates of V. Such ambiguities are stored in the guide descriptor as sets of values, yielding a final guide descriptor of the form $(discont, (\{s+\}, v_1), (\{s+\}v_2)\ldots)$.

Summarization Guides

Summarization guides produce a descriptor compatible with that of direct guides. However, summarization guides primarily provide information about the result set. Scatter-plot trend lines are a common form of summarization guide.

The principle requirement of a summarization guide is identity-based access to the result space and a unique identity for each visual element. Acquiring an iterator of identities takes the place of **Sample** and collecting entries from the iterator is the whole of $\mathbb{G}$. The resulting descriptor has an id as the input value and all associated visual values as the results (the discontinuity warning list is guaranteed to be empty).

Display

Since the display system is framework dependent, we only discuss the framework independent requirements. The display portion of the guide system is responsible for the presentation of the guide descriptor generated from executing $\mathbb{G}$. Generally speaking, the source values become labels for the visual values. However, there are additional considerations.

The first consideration for the display system is to determine where guides are required. This may be achievable automatically, or specified (preferably declaratively). Exact placement of the **Monitor** operator may be part of this specification. Second, the display system support is required for presentation elements such as Redundancy-support and Design-sensitivity (see *Section: Declarative Guides*).

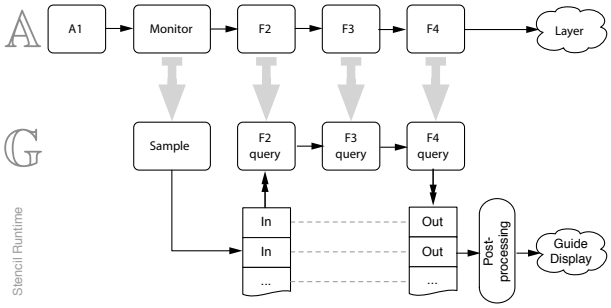


Figure 5: The guide process implemented by Stencil. The major distinctions are the inclusion of post-processing and maintenance of the input/output sets separate from analysis.

Analysis and Extensions

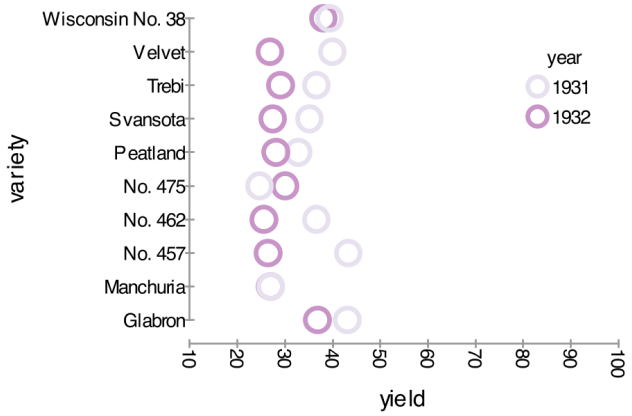
The presented system provides many of the desirable attributes given in *Section: Declarative Guides*. Provided with a suitable display system, the guides produced will be Complete, Consistent and Subordinate. The produced guide descriptor can also support Attribute-crosses and Redundancy-reporting, though display system support is required to take full advantage of the opportunities. Separation-support, Design-sensitivity and simplicity depend on implementation details of the display and guide requesting systems; but are not precluded by the preceding process.

The ability to format data for presentation (i.e., supporting Projecting) can be approached in a variety of ways. A general solution is to use a post-processing step that operates on the guide descriptor before application of the display system. In this way, values can be reformatted in a presentation-friendly manner (e.g., limit decimal places, highlight outliers, etc). Post-processing can be used to support the Projecting property, but it jeopardizes Correctness and Consistency. When arbitrary calculations are possible, the correspondence between $\mathbb{G}$ and $\mathbb{A}$ is violated if the post-processor ignores its inputs. The alternative to post-processing is a proliferating special cases for specific reformations (this can be seen in part in R [23] with the multitude of optional formatting arguments to many plotting functions). This tradeoff needs to be carefully weighed: flexibility and parsimony vs. correctness and verbosity.

Stencil

The core of the described guide system has been implemented in the Stencil visualization system. Stencil is a declarative language for specifying visualizations [12]. Stencil supports basic guide descriptor creation (not including discontinuity or ambiguity reporting) and arbitrary post-processing.

Figures 6a and Figure 6b demonstrate direct guides: axis and legend. Lines 5-10 provide the guide declarations. Guides are declared as a section of a layer declaration, initiated with the keyword 'guide'. Each guide declaration has four parts: (1) type, (2) static parameters, (3) attribute selectors and (4) post-processing instructions. Attribute selectors are interpreted with respect to the current layer. Guides can inherit some graphic properties from their parent layer (the default fill in Stencil is solid black, but the guide inherits the clear center defined in line 18.



(a) Becker's Barley Cell

```
1  import BrewerPalettes
2  stream Barley(year, site, yield, variety)
3
4  layer BarleySite
5  guide
6    axis[X:0] from Y
7    axis[sample:"LINEAR", round: "T",
8           seed.min:10, seed.max:100] from X
9    legend[X:75, Y:-60] from PEN_COLOR
10
11 from Barley
12     filter(site =~ "University Farm")
13     ID: Concatenate(variety,year)
14     PEN_COLOR: BrewerColors["PuRd","BLACK"](year)
15     X:* Scale[0, 100, inMin:10, inMax:100](yield)
16     Y:* Rank(variety) -> Mult(-7,_) -> Add(-5,_)
17     REGISTRATION: "CENTER"
18     FILL_COLOR: @Color{CLEAR}
```

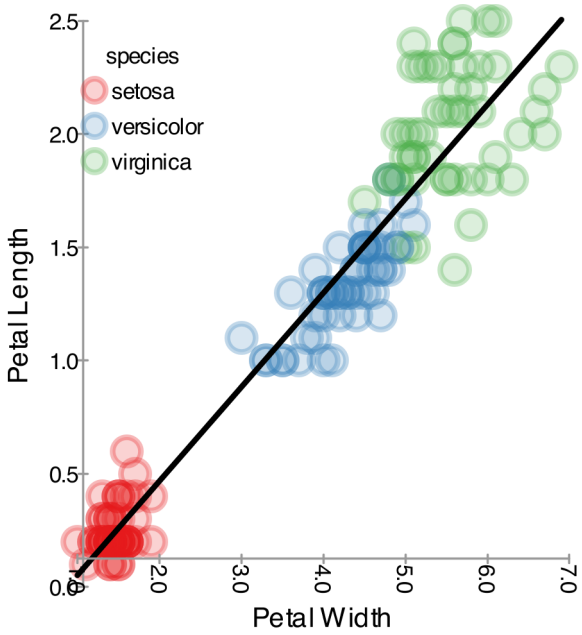
(b) Becker's Barley Program

Figure 6: A single cell of the Becker's Barley trellis. This example shows the directness of the automatic guide system and illustrates some weaknesses (discussed further later).

Part of the Crimean Rose visualization [8] and a program listing are given in Figure 9 (minus custom operator definitions). This visualization uses point-labels (a summarization guide) with post-processing and a direct guide to create the legend. The point labels by default use the ID as the text with positioning on the registration point of the associated glyph. Arbitrary post-processing enables custom formatting and positioning of labels; these effects are found on lines 7-10 of Figure 9b. An example of the trend-line construction and presentation can be seen in Figure 7.

Redundancy and Attribute-crosses are achieved by specifying two attributes in a guide definition. If the two attributes are based on the same input data, then a redundancy-encoding guide is created (like Figure 7). If the two attributes are based on separate data, then an attribute-crossing guide is created (see Figure 8).

The implementation presented is efficient in that it requires few additional operations, little memory and no additional iterations of the input data. There are at most two monitor operations inserted per guide (most guide types require just one). Monitoring a continuous input space is a bounds check/update while a categorical space employs a hash-table lookup/insert. In either case, monitoring only introduces a constant time overhead. Additional memory costs are: maintenance of the input space descriptor (at worst, linear in the data size but often constant) and storage of the

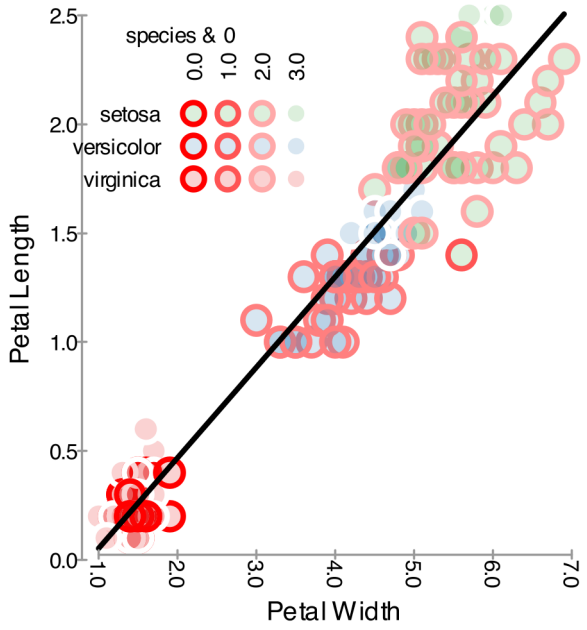


(a) Anderson's Flowers: Redundancy Encoding Guide

```
1  import BrewerPalettes
2  stream flowers(sepalL,petalW,sepalW,
3                 petalL,species,obs)
4
5  layer FlowerPlot
6  guide
7    trend from ID
8    legend[X:5, Y:-90] from FILL_COLOR,
9                       from PEN_COLOR
10   axis[sample:"Linear",guideLabel:"Petal Length"]
11     from Y
12   axis[sample:"Linear",guideLabel:"Petal Width"]
13     from X
14
15 from flowers
16     ID: obs
17     X:* Scale[0,100](petalL)
18     Y:* Scale[0,100](petalW) -> Mult(_,-1)
19     FILL_COLOR: BrewerColors(species)
20                 -> SetAlpha(50,_)
21     PEN_COLOR: BrewerColors(species)
22                 -> SetAlpha(80,_)
23     REGISTRATION: "CENTER"
```

(b) Stencil Program

Figure 7: Anderson's flowers with species encoded in both fill and pen-color. The redundancy encoding guide is defined in line 9. The definition is the same as that used for the redundancy guide in Figure 8, but since both fields are based on the same input the redundancy encoding style is used. Separate guides could also be requested by adding a second 'legend' request.



(a) Anderson's Flowers: Attribute Cross Guide

```
PEN_COLOR: Round(petalW) -#> HeatScale(_)
```

(b) Modified line 18

Figure 8: Anderson's flowers with a petal width encoded in pen-color. Only the definition of pen color changed from the program show in Figure 7b but Stencil determines that a cross product is required since the fill and outline are based on different inputs.

visual elements of the guide representation (linear in the sample size produced). By monitoring the input data as it is loaded, the guide can be created with zero additional iterations of the data.

Stencil's implementation presents one way to satisfy the metadata and display systems requirements. In the display system, guides are requested with the keyword *guide* and followed by a type, construction parameters and a path to the requested analysis. Guide declarations may include post-processing statements. The post-processing statements admit any valid computations, and thus have the power and risks discussed in Section Analysis and Extensions (the input tokens to post-processing are the entries of the guide descriptor). The requested guide type and parameters determine the **Sample** and **Monitor** operators used. Default **Sample** placement is over-ridden with a special composition operator $-\#>$ in a targeted analysis.

Stencil's metadata system is rooted in the operator libraries, so metadata is specified by the library creator for use by Stencil sub-systems. All of the metadata used for guide generation is used by other sub-systems as well. The required metadata are provided in two ways. First, a memory category (Function, Opaque, etc.) is provided in a per-operator instance metadata object. Second, all operators in Stencil are required to provide a *query* operation that satisfies the counterpart requirements (from Definition 1). Since transformations are implemented as objects, this means a query method must be supplied. Many operators explicitly provide this method; however, it can be inferred using the metadata and Clone. For Function and Reader operators, query is an alias for the default transformer. Opaque operators by definition do provide a reasonable query operator. Reader/Writer operator can provide support query by performing a deep-clone and then calling the stateful transformation on the clone.

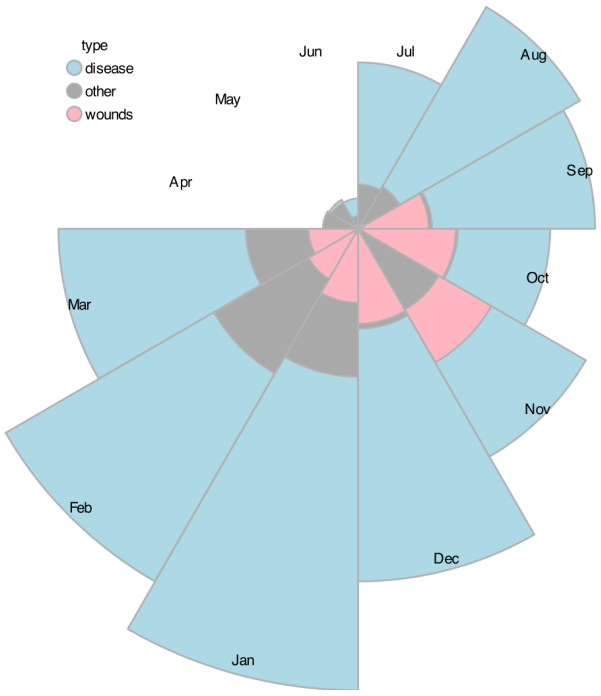
Stencil operators do not take the batch list arguments described in Section Generation. Therefore, the control flow is slightly is slightly modified (see Figure 5). In brief, the Stencil framework runs each sample through $\mathbb{G}$ individually, collating results at the end of the process. This is the principle reason why discontinuity reporting is not supported at this time. Full discontinuity reporting support could be achieved by adding a *discont* method that either has the call semantics for **G** described earlier but built using Stencil's query facets or simply performs discontinuity analysis. Concurrent support for both data-flow and data-state style operators has been explored in Stencil [14] and could be extended to support discontinuity reporting.

Scheduling of guide system execution has an important impact on overall system performance. The described system is suitable for both static and dynamic data systems. In static systems, the calculations can be deferred until all data is loaded. However, in dynamic data systems such a clear-cut schedule is not available. An optimal recalculation schedule will recalculate guides only when executing the guide system will change the result. Stateful Stencil operators provide a *stateID* access method to check for possible new results. StateID takes no arguments and returns a value associated with the internal state of the operator. If at any time $\mathbf{A}(xM) = (y, M')$ where $M \neq M'$ then StateID should change. (This definition differs from earlier discussions of StateID [13], but provides for the more general scheduling used in [14].) The Stencil runtime monitors StateID for all operators in $\mathbb{G}$ and its post-processing. Before scheduling $\mathbb{G}$ recalculation, the runtime queries for new StateIDs. If no StateID has changed, the re-calculation can be skipped. This yields generally effective scheduling, though it tends to over-approximate the need (some unnecessary recalculations are still performed).

The Stencil implementation currently restricts the placement of **Monitor** beyond the restrictions given in Section Generation. In a general Stencil analysis pathway, all prior computed values are available to all later computations. Executing $\mathbb{G}$ requires that all incoming values must be supplied by the sample. This implies that a **Monitor** operator acts as a partition in value accesses: no later operation may use values computed earlier than the monitor.

The presented implementation of the guide system supports all of the guide-system features presented in Section Declarative Guides, though some trade-offs are made. A basic guide can be declared in as few as four tokens (guide $< type >$ from $< att >$), with customization options available. Therefore, guide creation is simple by building on Stencil's declarative nature. Redundancy, attribute-crossing, separation support and design-sensitivity are all represented in various examples.

The guides are guaranteed to be complete (reflect the analysis), consistent (reflect the data) and subordinate, provided that post-processing is not used. Post-processing supports enables customizable guides, but makes it possible to violate these other properties. Use of post-processing does not necessarily invalidate any of these properties, but it must be used with care. For example, manipulating the labeling in a guide could be used to abbreviate ("December" from the data becomes "Dec" in the guide in Figure 9), a transformation that is consistent with the data. However, an alternative operator on line 7 could change "December"



(a) Nightingale's Crimean Rose

```
stream Deaths(date, type, count)

layer Rose["SLICE"]
guide
  legend[X: -75, Y: -50] from FILL_COLOR
  pointLabels from ID
    TEXT: MakeLabel(ID)
    (X,Y): MinRadius(50, X, Y, OUTERX, OUTERY)
    REGISTRATION: "CENTER"
    FONT: @Font{4}

from Deaths
   local(month, year) : ParseDate(date)
   ID: Concatenate(type, ":", month)
   FILL_COLOR: ColorBy(type)
   PEN: @Stroke{.5}
   PEN_COLOR: @Color{Gray70}
   HEIGHT:* Scale[min:0, max:250](count)
   Z: Mult(-1, count)
   (X,Y): (0,0)
   (START, END): Sub1(month) -> Partition(_)

operator ColorBy (t) -> (C)
   (t=~"wounds") => C:@Color{LightPink}
   (t=~"other") => C:@Color{DarkGray}
   (t=~"disease") => C:@Color{LightBlue}
```

(b) Crimean Rose Program

Figure 9: Recreation of Florence Nightingale's casualty causes in the Crimean war visualization. A Point-label guide with post-processing was employed. (Some operator definitions have been omitted for space).

into "June" just as easily, invalidating the consistency. A similar hazard is present with respect to the guides being subordinate if memory mutating operations are used in the guide system. There are no firm rules for when a manipulation invalidates the completeness, consistency or makes a guide non-subordinate. The Stencil design favors the flexibility of post-processing over the *guarantee* of completeness and consistency. However, all default constructions provide complete and consistent guides.

Comparison

The ability to construct guides in a framework supported, disciplined fashion ends up saving effort in the long run. This section compares the Stencil implementation of automatic guide generation to other frameworks. Stencil is used as the reference point because it provides the broadest support for the items discussed in *Section: Declarative Guides*. and generally provides compact ways. In several cases, Stencil provides only conditional or partial support because of conflicting requirements in the desirable attributes. For example, support for customization makes it possible to introduce inconsistency in Stencil. *Section: Stencil* discusses the conditions, trade-offs and omissions in detail.

Figure 10 provides a capabilities inventory with regards to *Section: Declarative Guides*. Vega is a rich framework, with a well-defined guide system (expressed through the 'legend' and 'axis' components). However, the system does not include attribute crosses or multiple encodings and no examples were found that included them. Vega directly provides for conditional completeness (depending on how transforms are defined) and consistency. D3, Protovis and Prefuse, generally re-use their existing graphics capabilities to create guides, and thus use guide definitions have complexity comparable to the definition of the related analysis. This flexibility means that redundancy-aware, crossing or separate guides are supported to the degree that the programmer is willing to construct them. With the exception of some axes in Prefuse, neither completeness nor consistency are provided by these non-declarative graphics frameworks. ggplot2 provides a capable automatic guide system. Consistency is guaranteed if default labeling is used. However, completeness (e.g. representing the analysis) is not provided because only the last-produced value results are used for labeling, not the original inputs. Stencil implements the majority of the system described in this paper.

The remainder of this section examines the graphics frameworks in more detail. To compare verbosity, several visualizations were implemented in each framework, and the guide portions were isolated. For example, a Protovis guide description is given in Figure 11, this corresponds guides found in the Stencil program in Figure 6b. The number of tokens used to build the guides in each program were counted, the results are summarized in Figure 12. Stencil was used as the reference point because it was the most familiar and most capable system. However, Vega and ggplot2 compare favorably on many plots in terms of required tokens. This is not surprising since all three are declarative systems. Prefuse was omitted from this analysis because (1) its approach to analysis made it difficult to isolate the items that were dedicated to guides and (2) the general verbosity of the Java language made the counts performed absurdly large. Prefuse is also the oldest framework of the bunch, and thus not indicative of current practice.

For Vega, D3 and Protovis, source code was taken from

	Complete	Consistent	Subordinate	Efficient	Custom	Simple	Redundancy	Att. Cross	Separation	Design
Vega	?	Y	Y	Y	Y	?			Y	Y
D3				?	Y		?	?	?	
Protovis				?	Y		?	?	?	
Prefuse	-	-			Y		?	?	?	
ggplot2		?	Y	Y	?	-			Y	Y
Stencil	?	?	Y	Y	Y	?	-	-	Y	Y

Figure 10: Table comparing guide systems of various visualization systems. An ideal system would hold all the listed characteristics and capabilities. A Y indicates an attribute held by a system; a star (?) indicates a conditionally present attribute; a dash (-) indicates a partially present attribute; and a blank items indicates an attribute missing from the system.

```
1  /* X-ticks. */
2  vis.add(pv.Rule)
3      .data(x.ticks())
4      .left(function(d) 90 + Math.round(x(d)))
5      .bottom(-5)
6      .height(5)
7      .strokeStyle("#999")
8    .anchor("bottom").add(pv.Label);
9
10 /* A legend showing the year. */
11 vis.add(pv.Dot)
12     .extend(dot)
13     .data([{year:1931}, {year:1932}])
14     .left(function(d) 260 + this.index * 40)
15     .top(-8)
16   .anchor("right").add(pv.Label)
17     .text(function(d) d.year);
```

Figure 11: Protovis program fragment for producing guides on Becker's Barley [7]. Line 3 provides **Sample** by using *x*, an operator used for X-layout in the visualization. This explicit coordination is avoided in the declarative system. Axis formatting constitutes lines 3-8 and the legend for year coloring is in lines 11-17

their respective websites. ggplot2 examples were produced internally. When multiple versions were available for D3, the simplest found was used. In all cases, punctuation was considered a separator, and non-punctuation groups were considered tokens. Therefore, numbers, parameters, procedures and mathematical symbols are all tokens but parenthesis and dots are not. Strings were considered a single token, even if they contained multiple words. Operators that used math-like notation were considered tokens and separators ("3+4" is three tokens). Only tokens used *exclusively* for the guide creation were considered as tokens (so for D3 and Vega, when a 'scale' is created and used in the both data mapping and guide creation, tokens used to create that scale are not counted). Some programs included optional programmatic "furniture" (like the optional var keyword or function(x,y) {return ...} vs (x,y) => {...} syntax in javascript). When a clear shorter syntactic substitution was possible, the shorter substitution was used instead of the program found on the website.

Vega is a relatively new framework [24], with an emphasis on declarative construction. It is a JSON framework, and the Vega designers chose to rely heavily on named arguments (represented as dictionaries). Therefore, the majority items are actually encoded as name/value pairs. Vega is highly declarative and includes an extensive transformation library. Notably, Vega only admits pure functions. This means that much of the metadata requirements presented earlier for Stencil are removed (for example, pure functions can be their own counterparts, and there is no need for snapshots). Unfortunately, there are fewer examples available for Vega than other frameworks. There are several Vega-lite examples of corresponding plots, but Vega-lite is a separate product, one level higher up an abstraction hierarchy. In the examples found, Vega consistency matches Stencil in terms of token count, occasionally beating it as well. This demonstrates that the declarative approaches are (to some degree) comparable between the two systems. Subjectively, the Vega system is a touch less flexible (requiring greater verbosity when flexibility is required) but in many ways simpler than the Stencil system (in part due to the more functional nature of Vega). One place that Vega is clearly more complex is in working with partial analysis. In Stencil, a '-*¿' is used to indicate where the guide monitor should be placed. In Vega, a separate transformation is defined. The Vega technique requires no special syntax but is slightly more verbose and breaks up something that is otherwise logically a single unit.

D3 is the most popular javascript visualization library. D3 has different goals that Protovis, focusing on transforming data into DOM elements and re-using web-standards such as DOM elements and CSS [9]. Guide creation in D3 is generally done in the same style as any other part of the visualization, by mapping data to a visual representation. The difference is there are many convenience functions in D3 to produce that dataset, deriving from the original source data in many cases. This is still a separate definition, but less prone to errors that a constructing the guides completely separately. As the D3 library has evolved, support for guides has also expanded significantly (this paper reflects version 3). Figure 12 includes a comparison of the number of tokens dedicated to guide creation. In general, D3 guides were longer than their Protovis counterparts. Much of the additional length was caused by CSS selectors or needing to create essential SVG DOM furniture (such as 'g' tags). These are relatively simple elements, but are often repetitive and thus good targets for future abstraction. Some guide construction code reproduced source data (for example, the Becker's Barley legend hard-codes the years).

Visualization	Stencil	Protovis	Ratio	ggplot2	Ratio	D3	Ratio	Vega	Ratio
Bar Chart	44	55	4:5	34	5:4	59	2:3	28	3:2
Bckrs. Barley	33	64	1:2	34	1:1	70	1:3	49	2:3
Andrs. Irises	7	26	1:4	9	1:1	57	1:8	–	—
Scatterplot	15	69	1:5	29	1:2	81	1:5	15	1:1
Line Chart	14	60	1:5	20	3:4	53	4:15	13	1:1
Point Labels	5	14	1:3	7	1:1	21	1:4	28	2:11
Crimean	110	36	3:1	104	1:1	48	2:1	–	—
Crimean* **	(27)	36	(2:3)	104	(1:4)	48	(1:2)	–	—

Figure 12: Total tokens dedicated to the guide creation in selected examples and ratios of stencil vs other frameworks. The "Andrs. Irises" line is just the legend, not the axes (which were compared using scatter plot instead). ***The parenthesized values on this line exclude a Java function definition used only in the guide creation. The hight number of tokens is indicative of a need for a better ad-hoc bridge, such as the one D3 or Protovis provide.

Tieing guides to the data already present is tricky in D3, requiring manually transforming the dataset, then passing the transformed version to the guide creation code. This type of manual passing of data is avoided in a declarative system.

The Protovis visualization framework supports many declarative constructs [5]. This includes some support for guides. For axes, for example, the data iterator and *Anchors* provide access to the input and result space respectively. Similar concepts exist for many Protovis objects (e.g., the 'ticks' method from pv.scale objects). However, guides themselves are not distinguished as separate entities; combining the various relevant components is done using standard analysis operators. Having input and output space information available provides near feature-parity with the described system and combined with the declarative basis, makes Protovis a natural target for comparison. Subjectively, the Stencil definitions were consistently more general than the Protovis ones. Like D3, some Protovis definitions explicitly encode information about the input data. Both Protovis and Stencil definitions included explicit guide positioning instructions, principally for legends. The shortest Protovis declarations occurred for categorical axes, where anchors and the data iterator provided all necessary information. Transformations on the input data or the sampling over continuous spaces were the most verbose parts of Protovis guide creation. In contrast, graphic design specification was the most verbose part of Stencil guides. This source of Stencil verbosity may be ameliorated by having guides inherit graphic attributes from the layers they are applied to. Protovis provides such inheritance through its scene graph mechanism.

ggplot2 is a popular visualization library for the R statical environment. ggplot2 automatically constructs many guides in an acceptable fashion. However, the data model of R is data-state based and ggplot2 operates on these state objects. This makes it easy for ggplot to create consistent guides (reflect the data), but since only analysis results are available it is more difficult to construct complete guides (reflecting the analysis) that look any earlier than the final analysis stage. A token-based comparison of ggplot2 to Stencil is also presented in Figure 12. The R's native handling of formulas helps ggplot2 use little syntactic furniture to automatically construct guides, generally resulting in little or no required code for basic definitions. However, non-standard guides quickly become cumbersome to construct, especially if the customization involves introducing data states not originally present in the analysis (such as abbreviated names). Furthermore, since custom labeling is based on row-matching in the data frame it is incumbent on the programmer to indicate a row that the results being presented actually depend on (thus, consistency is only *conditionally* supported in Figure 10). A zero-token solution for Stencil would require improving the default placement of the monitor operator to respect the need to cleanly divide the data dependencies. However, the default guide definitions themselves are trivial.

Future Work

Using the guide creation system in Stencil demonstrated the practicality of the approach. It also illuminated two shortcomings, both stemming from a need to re-state important information about guide attributes. The first, and more significant, is an insensitivity to sample spaces that are independent of the input data. This can be seen in Example 6b where the Scale operator's on line 15 has its max and min arguments manually copied and represented as 'seed' arguments in the guide declaration on line 8. This manual transfer is dictated by the fact that the provided **Monitor** and **Sample** operators (which together constitute the sample operator of *Section: Generation*) are based entirely on input data. However, the Scale operator in the analysis is parameterized to indicate an input space that exceeds the data presented. A workaround includes using shared constants, but this only hides the problem. Ideally, some means of communicating that an operator expands the input range would be provided. The earlier iteration of the guide system [13] allowed categorical operators to provide a sample instead of the dedicated seed/sampler pairs. This was removed from the current system to simplify the operator interface and transformations but a similar system could be re-introduced to allow for operators to determine the sample space. Alternatively, introducing a means for operators to communicate an expected sample space would achieve the same effect. The relative merits and implementation details of these techniques are not know.

The guide creation process described in *Section: Generation* relies on non-interference, and is thus simple, in many ways. A guide process that can modify memory could be used to perform some optimizations discussed in earlier work [13]. This requires a more complex set of semantics than was provided in *Section: Analysis Semantics and Metadata* and complicates reasoning about guide/analysis interaction by removing the Subordinate property. However, these same issues are shared by providing arbitrary post-processing. Investigating the effects of non-subordinate guides may provide insight into further useful semantics for visualization frameworks.

Conclusions

Guides provide essential support for the interpretation of a visualization. Existing library-based visualization software does not provide abstractions to support semantically-aware guide creation. Creating guides in an abstract fashion requires an encoding of the execution semantics of an analysis process and the semantic role that guides play in a visualization. We have presented a formalization of execution semantics and used that to develop a declarative guide creation process. The concepts of this paper have been implemented in the Stencil system, demonstrating that the concepts are sufficient and practical. We believe these creation concepts can be applied in other frameworks to improve guide creation and visualization frameworks in general.

References

[1] Tableau software: Business dashbaords. http://www.tableausoftware.com/business-dashboards, March 2008.

[2] G. Abram and L. Treinish. An extended data-flow architecture for data analysis and visualization. In *VIS '95: Proceedings of the 6th conference on Visualization '95*, page 263, Washington, DC, USA, 1995. IEEE Computer Society.

[3] J. Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. page 1977, 2007.

[4] J. Bertin. *Semiology of Graphics*. Reprinted by University of Wisconsin Press, 1967.

[5] M. Bostock and J. Heer. Protovis: A graphical toolkit for visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1121–1128, 2009.

[6] M. Bostock and J. Heer. Protovis: Anderson's flowers example. `http://vis.stanford.edu/protovis/ex/flowers.html`, March 2010.

[7] M. Bostock and J. Heer. Protovis: Beckery's barley example. `http://vis.stanford.edu/protovis/ex/barley.html`, March 2010.

[8] M. Bostock and J. Heer. Protovis: Nightingale's rose example. `http://vis.stanford.edu/protovis/ex/crimea-rose.html`, March 2010.

[9] M. Bostock, V. Ogievetsky, and J. Heer. D3: Data-driven documents. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2011.

[10] C. A. Brewer. Color use guidelines for data representation. In *Proceedings fo the Section on Statistical Graphics*, pages 55–60, Alexandria, VA, 1999. American Statistical Association.

[11] E. H. Chi. *A Framework for Visualizing Information (Human-Computer Interaction Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[12] J. A. Cottam and A. Lumsdaine. Stencil: A conceptual model for representation and interaction. *Information Visualisation, International Conference on*, 0:51–56, 2008.

[13] J. A. Cottam and A. Lumsdaine. Algebraic guide generation. In *IV '09: Proceedings of the 2009 13th International Conference Information Visualisation*, pages 68–73, Washington, DC, USA, 2009. IEEE Computer Society.

[14] J. A. Cottam and A. Lumsdaine. Automatic application of the data-state model in data-flow contexts. In *IV '10: Proceedings of the 2010 14th International Conference Information Visualisation*, Washington, DC, USA, 2010. IEEE Computer Society.

[15] J. Fekete. The InfoVis toolkit. In *Proceedings of the 10th IEEE Symposium on Information Visualization*, pages 167–174, Piscataway, New Jersey, 2004. IEEE Press.

[16] D. Gelernter and N. Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992.

[17] J. Heer and M. Bostock. Declarative language design for interactive visualization. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2010.

[18] J. Heer, S. K. Card, and J. A. Landay. Prefuse: A toolkit for interactive information visualization. In *Proceeding of the SIGCHI Conference on Human Factors in Computing Systems (CHI'05)*, pages 421–430, New York, NY, USA, 2005. ACM Press.

[19] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, 2004.

[20] J. Lloyd. Practical advantages of declarative programming. In *Joint Conference on Declarative Programming (GULP-PRODE'94)*, 1994.

[21] M. Luboschik, H. Schumann, and H. Cords. Particle-based labeling: Fast point-feature labeling without obscuring other visual features. *IEEE Transactions on Visualization and Computer Graphics (InfoVis'08*, pages 1237–1244, 2008.

[22] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Inf.*, 6:319–340, 1976.

[23] R Development Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria, 2005.

[24] 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.

[25] R. Scheck. *Create Dynamic Charts in Microsoft Office Excel 2007*. Microsoft Press, 2008.

[26] J. Talbot, S. Lin, and P. Hanrahan. An extension of wilkinson's algorithm for positioning tick labels on axes. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2010.

[27] E. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, Connecticut, 1983.

[28] E. Tufte. *Envisioning Information*. Graphics Press, Cheshire, Connecticut, 1990.

[29] H. Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer Publishing Company, Incorporated, 2nd edition, 2009.

[30] L. Wilkinson. *The Grammar of Graphics*. Springer-Verlag, New York, 2nd edition, 2005.